Take it to the Limit: Peak Prediction-driven Resource Overcommitment in Datacenters

Noman Bashir
University of Massachusetts Amherst
nbashir@umass.edu

Nan Deng
Google
dengnan@google.com

Krzysztof Rzadca
Google and University of Warsaw
kmrz@google.com

David Irwin
University of Massachusetts Amherst
deirwin@umass.edu

Sree Kodak
Google
skodak@google.com

Rohit Jnagal
Google
jnagal@google.com

Abstract

To increase utilization, datacenter schedulers often overcommit resources where the sum of resources allocated to the tasks on a machine exceeds its physical capacity. Setting the right level of overcommitment is a challenging problem: low overcommitment leads to wasted resources, while high overcommitment leads to task performance degradation. In this paper, we take a first principles approach to designing and evaluating overcommit policies by asking a basic question: assuming complete knowledge of each task's future resource usage, what is the safest overcommit policy that yields the highest utilization? We call this policy the *peak oracle*. We then devise practical overcommit policies that mimic this peak oracle by predicting future machine resource usage. We simulate our overcommit policies using the recently-released Google cluster trace, and show that they result in higher utilization and less overcommit errors than policies based on per-task allocations. We also deploy these policies to machines inside Google's datacenters serving its internal production workload. We show that our overcommit policies increase these machines' usable CPU capacity by 10-16% compared to no overcommitment.

Keywords: Datacenters, resource management, overcommit

ACM Reference Format:
Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the Limit: Peak Prediction-driven Resource Overcommitment in Datacenters. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3447786.3456259

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456259

1 Introduction

An important goal of datacenters is minimizing their infrastructure and operating costs. Low utilization of allocated resources is one of the key impediments to achieving this goal, since it may require adding more physical resources to support increasing workload demands [4, 16]. Low utilization is common: as applications' resource usage varies over time, they typically request, and are allocated, enough resources to satisfy their *expected peak* resource demand, which may rarely occur. One way to increase resource utilization is to overcommit resources such that the sum of resources allocated to tasks on a machine exceeds its physical resources. Overcommitment is supported by many resource management platforms [5, 18, 19, 27–29, 44], and is evident in production datacenters [2, 50, 59].

The benefits of overcommitment come at the expense of a higher risk of task performance degradation or eviction [5, 44]. Without overcommitment, the sum of resources allocated to tasks on a machine is always less than its physical capacity. As long as the host operating system (OS) or the hypervisor isolates resources and enforces that tasks stay within their allocated resources' limits, they will never evict tasks or degrade their performance by depriving them of resources. In contrast, with overcommitment, even if all tasks are using less than their allocated resources, the sum of their desired resource usage may be greater than the machine's physical capacity. Thus, if the CPU is overcommitted, the OS may have to throttle some tasks by preventing them from using their allocated CPU shares, which can degrade their performance, e.g., by increasing the latency of users' requests. Similarly, if memory is overcommitted, the OS may have to suspend or kill some tasks. In these cases, overcommitment disrupts the infrastructure's workload and degrades its quality of service, which is normally expressed in terms of *Service Level Objectives*, or SLOs [36]. Thus, in designing overcommit policies, the task scheduler must carefully balance the benefits to platform utilization, which increases efficiency and reduces costs, with the risk of degrading task performance. Importantly, as long as the scheduler can maintain its SLOs, overcommit policies are effectively transparent to end-users, and can be deployed in production.

In this work, we take a first-principles approach to designing and evaluating overcommit policies. We argue that the overcommit policy is largely orthogonal to the scheduling policy. We formalize the overcommitment problem as the problem of predicting, on a per-machine basis, the free resource capacity at all times in the future. This future free capacity can then be used by the scheduler to decide if an additional task still fits on the machine. Given a workload and a machine's future free resource capacity, we then derive the best possible overcommit policy that guarantees no performance degradation. We call this the *peak oracle* policy, since it reduces to predicting a machine's peak resource usage in the future. Our peak oracle policy is *clairvoyant*: knowing the future load, it determines the remaining future free capacity on the machine. Our peak oracle maximizes utilization, which also maximizes savings, while guaranteeing no task evictions or performance degradation due to overcommitment. Any overcommit policy that achieves higher utilization than our peak oracle must eventually cause some task to not receive its allocated resources. Likewise, any policy yielding lower utilization can be improved to yield more savings, i.e., more resources available for other tasks. We use the peak oracle policy as the baseline to design and evaluate practical overcommit policies.

Our approach shifts the focus of designing an overcommit policy from optimally setting *per-task* resource usage *limits*, to instead accurately *predicting per-machine* peak resource usage. That is, a scheduler can overcommit resources on a machine as long as its future peak usage does not exceed its physical capacity. Since a machine's future peak depends on the resource usage of the tasks running on it, the future peak varies both over time and across machines, as does the static level of resource overcommitment in the classic sense, i.e., where a fixed fraction of resources allocated to tasks on a machine is permitted to exceed its physical capacity. Thus, in our policies, the level of overcommitment depends on a machine's utilization: a highly utilized machine is overcommitted less than an lowly utilized machine.

This paper's principal contribution is showing that overcommit policies that rely on per-machine predictions of peak resource usage are both efficient and readily applicable to the standard model of datacenter scheduling, e.g., used by Borg [59, 61], Omega [53], and Kubernetes [9, 19]. In contrast to our peak oracle, which serves only as our baseline, we design our remaining peak *predictors* to operate in a realistic environment that is: on-line, non-clairvoyant, and stochastic. Importantly, our predictors *require no modifications* to existing schedulers, and thus can support all their complex features, such as constraints on proximity and hardware architectures. Using both a simulator and a prototype deployed in production, we demonstrate that widely used static, limit-based overcommit polices are both more risky and less efficient than our usage-based predictors. Our experimental environment consists of ≈24,000 machines sampled from several geographically-distributed datacenters that mainly serve Google's end-user generated, revenue generating workloads.

To summarize, we make the following contributions:

- We propose an optimal clairvoyant overcommit algorithm that provides a best case scenario using future data. While impossible to implement in practice, we show — using large-scale data from Google datacenters — how this algorithm serves as a baseline for evaluating practical overcommit policies. By simulating these policies and the clairvoyant algorithm and comparing their results, we can evaluate which policy yields the best performance. This simulation-based methodology enables us to quickly evaluate overcommit policies without risking production workloads.
- For a realistic, non-clairvoyant setting, we propose multiple practical overcommit policies that rely on predicting a machine's future peak usage, and simulate them on the Google workload trace [59, 65]. We implement the best-performing policy and deploy it to ≈12,000 machines in multiple datacenters serving Google's internal workloads.
- To enable future work on designing overcommit policies, we publicly release our simulator's source code under a free and open source license.[1] The simulator uses the Google workload trace, and mimics Google's production environment in which Borg [59, 61] operates.

2 Motivation and Background

IIn this section, we provide background on the overcommitment problem and discuss the current overcommit policies used in production cluster schedulers capable of managing both batch workloads and service workloads with latency requirements [9, 59, 61]. We use the terminology introduced in our prior work on Borg [59, 61]. A *job* corresponds to a single execution of a batch workload, or a continuously-running service. A job is composed of one or more *tasks* that perform the actual processing. Each task runs on a single machine, but a single machine typically hosts dozens of tasks.

2.1 Datacenter Scheduling

Our work targets a datacenter software/hardware architecture of Borg [59, 61] or Kubernetes [19], which we claim is representative of the problems faced by all cloud providers. We assume that each task has a known limit on the maximum amount of resources it can use, e.g., 4 cores or 3GB of memory. These limits are either set manually by the job owner, or automatically by a service such as Autopilot [52]. During execution, if a task tries to use more resources than its assigned limit, the machine-level infrastructure, such as the OS, may throttle the CPU, reject the memory allocation, or evict tasks to ensure performance isolation across tasks [20, 30, 61, 67]. For CPU, the limits are usually soft, i.e., they are enforced only in the case of resource contention. In addition, not all tasks are equally important. Usually, there

[1] https://github.com/googleinterns/cluster-resource-forecast

are at least two classes of jobs: a batch class and a serving class [45, 52, 61]. Jobs from the serving class typically have well-defined SLOs [36] to ensure resources are available up to their allocated limit. The metrics used to define the SLOs, i.e., Service Level Indicators, depend on the type of resource and can vary. For CPU, a typical Service Level Indicator is the CPU scheduling latency, which is defined as the amount of time that tasks wait for free cycles in the OS's CPU scheduler while they are ready to run, e.g., not waiting for I/O.

Users submit their tasks to a software component called the *scheduler*, which manages a set of physical machines, called *cells*, and decides which tasks should run on which machine. Scheduling tasks involves two steps: 1) finding the set of candidate machines in the cell that have enough free resource capacity and satisfy other requirements imposed by each task, and 2) using some bin-packing algorithm [8, 13, 33, 63] to assign the tasks to machines. A simple and conservative approach to estimating a machine's free resource capacity is to directly use its tasks' resource limits and only select candidate machines for a new task such that its limit plus the sum of the running tasks' limits on a machine does not exceed the machine's physical capacity. Once a machine starts to run tasks with a total limit exceeding any resource's capacity, we say that the machine is overcommitted. Our work focuses on designing a system to overcommit machines by better estimating its free resources compared to the simple approach above. Thus, our work fits into the first step of task scheduling, and is orthogonal and complementary to the bin-packing problem. Our work is also orthogonal to handling resource contention, such as by task migration. The primary goal of our work is to *safely* overcommit a machine's resources while avoiding resource contention as much as possible.

2.2 Overcommit Overview

Schedulers must know the amount of free resources available on each machine to make scheduling decisions. The amount of free resources should reflect the available resources at all times in the future to avoid task performance degradation or eviction. As noted above, a conservative method for estimating free resource capacity is to take the difference between the physical capacity of the machine and the sum of limits of the tasks running on the machine. This approach protects against the worst-case scenario where all tasks use their maximum allocated resources at the same time. In practice, tasks rarely, if ever, use their maximum allocated resources. Since the number of tasks simultaneously executing on a machine is large, having all tasks use their peak resources at the same time is exceedingly rare. The maximum of the sum of tasks' resource usage is also consistently less than the sum of their allocated resources. This gap between actual usage and physical capacity is an opportunity for overcommitting resources without overloading the machine. We next investigate the factors that contribute to this opportunity.

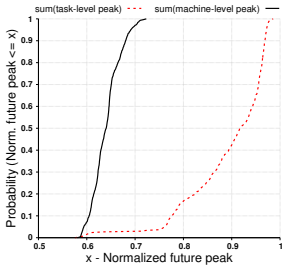


Figure 1. *CDF of cell-level future peak usage computed as aggregate of machine-level future peak (black) and task-level future peaks (red, dashed), normalized to cell's total limit.*

Usage to limit gap. The actual resources a task uses is often significantly lower than its resource limit, which is the upper bound of resources it can use. This usage-to-limit gap exists largely because reliably estimating a task's exact resource requirement is hard for *users*. Prior work has shown that the difference between the limit and usage can be significant [45, 52]. Autopilot [52] attempts to address this problem by setting the limits based on a task's observed historic usage. However, Autopilot's limits are conservative, e.g., the 98th percentile of resource usage over a certain time window with some additional margin for error. In addition, these limits must also cover the predicted peak usage over a long time horizon, e.g., a day, rather than follow a daily pattern. Limits also cannot be frequently updated, i.e., no more than a few changes a day are desirable, since increasing the limit may lead to a series of task evictions [52]. As a result, the usage-to-limit gap exists even after using complimentary approaches that specifically target reducing this gap. [52] reports an average usage-to-limit gap, which they call the relative *slack*, of 23%. [35] runs several benchmarks monitoring their CPU utilization with per-second resolution, and shows that it varies widely between 0% and 90% utilization. These benchmarks demonstrate that, despite their optimizations, long periods of low utilization remain.

Pooling effect. Ideally, tasks should set their resource limits close to their maximum future resource usage. Systems like Autopilot [52] use vertical scaling to automatically set the limit to match tasks' future peak. Assuming a task's future peak usage can be accurately predicted, schedulers can use these predictions to ensure that the sum of all tasks' limits does not exceed the physical capacity. However, because these predictions are made at the individual task level, doing so would overestimate the peak of aggregated resource usage of all tasks, i.e., the maximum of the sum of all tasks' usage, because tasks generally do not peak at the same time. Statistical multiplexing of peaks means that the maximum of the sum of tasks' resource usage is always less than or equal to the sum of maximum usage for each task.

$$\max\Big(\sum_{all\ tasks} (task's\ resource\ usage)\Big) \leq \sum_{all\ tasks} \max(task's\ resource\ usage),\ \forall\ t$$

To illustrate this effect, in Figure 1, we use the public Google cluster trace data [59] and plot the distribution of the sum of individual tasks' peak usage over time across all machines, and the distribution of the sum of machine-level peak usage. The distributions are plotted as cumulative distribution functions (CDFs) [12]. At the median, the sum of task-level peak usage is almost 50% higher than the machine-level peak usage. This indicates that even a perfect system, which always set tasks' resource limits equal to the tasks' peak resource usage, has room to safely overcommit machines by packing them with additional tasks whose total limit on each machine is higher than the machine's capacity. This statistical multiplexing of peaks provides an opportunity for overcommitment that is not exploited by existing works, such as Autopilot, that target setting limits for each task close to its actual usage.

Our approach explores this pooling effect, i.e., the temporal or probabilistic variability of utilization within a particular task. If a task never reaches a fraction, say 40%, of its allocated limit, the limit can be lowered by tools such as Autopilot. Our approach solves the related, but orthogonal, problem of a task that sometimes, e.g., 5% of time, reaches its limit, but usually operates at a much lower utilization.

3 Peak Oracle

We first establish the maximum free resource capacity that a machine can advertise to a scheduler without violating any tasks' resource limits, which we call the peak *oracle*. While we cannot implement this peak oracle in practice, since it requires perfect knowledge of the future, we use it as our baseline to evaluate practical peak predictors in Section 4.

3.1 Definition of the Oracle

Schedulers need to look at the available resources on a machine, not only at the present, but also in the future, when scheduling a new task on the machine. Placing a new task based only on the available resources at the time of scheduling can lead to resource shortages, since, in the future, existing tasks might use more resources. As a result, the machine's free capacity may drop below the new task's limit. There will be a resource shortage on the machine if, at the same time, the new task's resource demand exceeds the available free capacity. This can happen even if every task is operating well below its limit. Our *peak oracle* formalizes the reverse of this argument by assuming a *clairvoyant scheduler*, i.e., a scheduler that knows all tasks' future usage.

More formally, a *clairvoyant* scheduling algorithm computes the future usage $U(\mathcal{J}, t)$ (a time series) of any set of tasks $\mathcal{J}$ by simply adding the per-task usage $U_i(t)$. To simplify notation, we assume that if a task k completes at some time t' it has 0 usage after that time, i.e., $U_k(t) = 0$ for $t \geq t'$.

$$U(\mathcal{J}, t) = \sum_{i \in \mathcal{J}} U_i(t). \tag{1}$$

At any time τ, the *peak oracle* P_O computes the future *peak* usage as follows.

$$P_O(\mathcal{J}, \tau) = \max_{t \geq \tau} U(\mathcal{J}, t). \tag{2}$$

We denote as $P(\mathcal{J}, \tau)$ any *peak predictor*, i.e., a function that at time τ computes the predicted peak for a set of tasks $\mathcal{J}$.

A scheduling algorithm employs the peak oracle as follows. To decide whether a new task J, submitted (or released) at time r_j, fits on a machine, the peak oracle computes the future peak usage $P_O(\mathcal{J}_s \cup \{J\}, r_j)$ of the set of tasks already scheduled on this machine (denoted by $\mathcal{J}_s$) and the new task J. J is scheduled on a machine if and only if the future peak $P_O(\mathcal{J}_s \cup \{J\}, r_j)$ does not exceed the physical capacity of the machine, denoted as M. A scheduling algorithm is then *safe* if the total usage of tasks $\mathcal{J}_s$ scheduled on a machine never exceeds the machine's capacity, $U(\mathcal{J}_s, t) \leq M$.

We can easily prove, by contradiction, that a scheduler using the peak oracle is *safe*. Lets say at time t_0, a machine's resource usage exceeds its capacity, i.e., $U(\mathcal{J}_s, t_0) > M$. Assume the last task placed on the machine *before* time t_0 is released at time r_j. By definition, the future peak usage at time r_j for tasks $\mathcal{J}_s$ is greater than the machine's capacity: $P_O(\mathcal{J}, r_j) = \max_{t \geq r_j} U(\mathcal{J}, t) \geq U(\mathcal{J}, t_0) > M$. The scheduler should not schedule the task released at time r_j on the machine according to the oracle's output, contradicting the assumption that the task runs on the machine.

In addition, a greedy scheduler using the peak oracle is the *most efficient* scheduling policy among all safe policies. A non-greedy scheduler could refrain from placing a task on a machine even though the oracle predicts that there is sufficient space for it. Specifically, there is no safe scheduling policy that yields a utilization higher than the oracle by scheduling more tasks. To see why, lets assume that at time r_j, task J is released and the oracle rejects the task be assigned to a machine with capacity M, while another safe scheduling policy allows its placement. The only reason the oracle would reject task is when $P_O(\mathcal{J}_s \cup \{J\}, r_j) > M$. Thus, if the task is placed on the machine, at some time $t > r_j$, $U(\mathcal{J}_s, t) > M$, which makes the other policy unsafe, contradicting our previous assumption the policy is safe.

3.2 Measuring overcommit quality with oracle violations

Since a scheduling algorithm using the peak oracle is the most efficient safe policy, an effective approach to designing practical overcommit policies is mimicking the peak oracle by *predicting* the future peak usage given a set of tasks. That is, designing an algorithm that estimates the oracle value $P_O(\mathcal{J}, t)$ by an estimator $P(\mathcal{J}, t) = \widehat{P}_O(\mathcal{J}, t)$ using only data available at t: historical usage data $U_i(\tau)$ for $\tau < t$, or a task limit L_i, which is either defined by the user, or set by an automation tool such as Autopilot [52]. We call any algorithm that predicts the future peak usage for a set of tasks a *peak predictor*. Our peak oracle represents the most accurate and efficient peak predictor, since it results in the

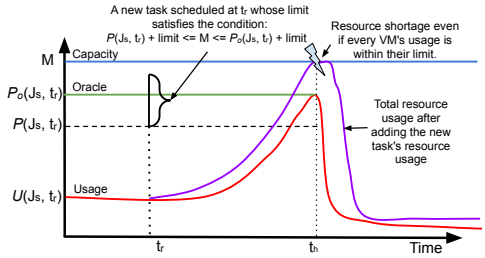


Figure 2. *Oracle violation at t_r may lead to scheduling a too-large task at t_r, and resource shortage at t_h.*

least unused capacity. In contrast, the most conservative peak predictor, which yields the most unused capacity and never overcommits, uses the sum of the limit of all running tasks on a machine, i.e., $P(\mathcal{J}, t) = \sum_i L_i$, as its prediction. Formally, we say a predictor overcommits if its predictions are ever lower than the sum of the limits, i.e., $P(\mathcal{J}, t) < \sum_i L_i$.

To evaluate a peak predictor, we compare it against the actual future peak usage. If a prediction is higher than the actual usage at time t, $P(\mathcal{J}, t) > P_O(\mathcal{J}, t)$, using it would be safe for the scheduler, since it would not result in a machine's resource demand exceeding its capacity, but it would be less efficient than the oracle as fewer tasks could potentially fit onto the machine. In contrast, if a peak predictor underestimates the future peak usage, more tasks could be placed onto a machine, which makes it unsafe.

We say there is a *violation of the oracle* when a peak predictor underestimates the future peak usage, i.e., at some moment t, $P(\mathcal{J}, t) < P_O(\mathcal{J}, t)$. The more overcommit violations that happen, and the greater the difference between the predicted and actual peak, the more likely some scheduling decision leads to resource exhaustion on the machine.

3.3 Estimating tasks' performance by oracle violations

We next use the hypothetical example from Figure 2 to show how an oracle violation can lead to a resource shortage. At time t_r, a new task J is submitted to the cluster, and the scheduler picks a machine, with capacity M, to see if it has enough resource to run task J. At the time, all tasks running on the machine, $\mathcal{J}_s$, have usage $U(\mathcal{J}_s, t_1)$. Looking into the future, these tasks' total usage then peaks to $P_O(\mathcal{J}_s, t_r)$ at time t_h. This means the machine has an oracle value of $P_O(\mathcal{J}_s, t_r)$ at time t_r based on the peak oracle's definition. Now lets say there is an oracle violation at time t_r, such that the predicted future peak usage is $P(\mathcal{J}_s, t_r) < P_O(\mathcal{J}_s, t_r)$. Let's say the new task J has a resource limit L_J, which satisfy the condition $P(\mathcal{J}_s, t_r) + L_J \leq M < P_O(\mathcal{J}_s, t_r) + L_J$. Because the infrastructure only ensures some level of resource availability for tasks up to their limit, the predicted peak can only increase up to L_J after considering task J, which gives us $P(\mathcal{J}_s \cup \{J\}, t_r) \leq P(\mathcal{J}_s, t_r) + L_J \leq M$. This allows the scheduler to schedule task J onto the machine because the scheduler *thinks* the machine has enough capacity to host all tasks including J. However, the actual free capacity left for task J in the future is $M - P_O(\mathcal{J}_s, t_r) < L_J$. This indicates that there is a time point when task J could maintain its usage within its limit, while every other task also runs within their resource limit, but still causes a resource shortage on the machine.

Cell No.	1	2	3	4	5
Approx machines ($\times 10^3$)	40	11	10.5	11	3.5
Approx tasks ($\times 10^6$)	14.8	12.8	9.4	81.3	3.7

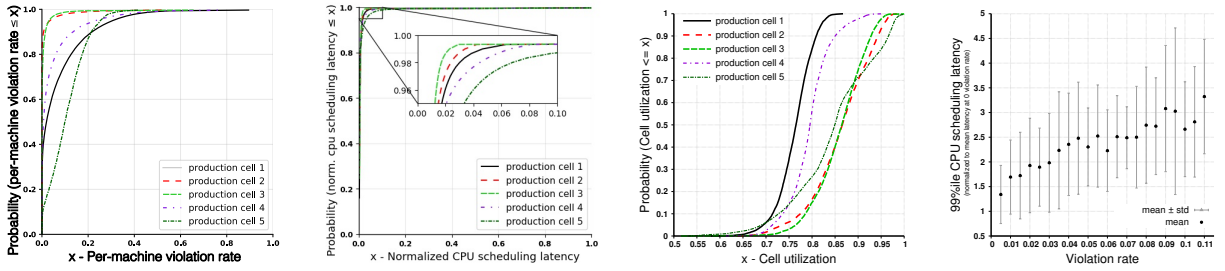
Table 1. *Performance correlation experiment statistics. Data covers a month-long period starting 2020-06-01.*

Getting rid of oracle violations is sufficient to avoid resource shortages as described above because it makes it impossible to satisfy the condition $P(\mathcal{J}_s, t_r) + L_J \leq M < P_O(\mathcal{J}_s, t_r) + L_J$ in the first place. On the other hand, as illustrated in Figure 2, a sequence of events must happen in a specific order to turn an oracle violation into an actual resource shortage: 1) the violation must happen; 2) the scheduler must schedule a task at time t_r when the violation happens; 3) the scheduled task's limit L_J must be higher than the remaining capacity $L_J > M - P_O(\mathcal{J}_s, t_r)$, i.e., the machine's capacity minus the true future peak usage; and 4) finally, at some future $t_h > t_r$, multiple tasks, including the newly scheduled one, must request resources at certain level at the same time.

Thus, a violation is not a sufficient condition for resource exhaustion, but with more violations, the scheduler gets more opportunities to schedule a task that fits these criteria, which might lead to resource exhaustion. Resource exhaustion can be exhibited in different ways depending on the resource, or the machine's policy of dealing with a resource shortage. For example, a shortage of CPUs could lead to high CPU scheduling latency, i.e., threads that are ready to run, but instead must spend time waiting for free CPU cycles. CPU scheduling (access) latency is particularly important, since it is one of the metrics that many schedulers use to establish CPU latency SLOs [3, 39].

To measure oracle violations, for each machine, we count the number of times a violation happens and normalize it to the total number of time instances. We call this ratio the *violation rate*. The higher the violation rate, the higher the risk to a machine. We next establish a link between the violation rate and our chosen QoS metric from above: the CPU scheduling (allocation) latency, i.e., the time a ready process must wait for CPU. For latency-sensitive serving tasks, this CPU scheduling latency should be low. The link between the violation rate and an established QoS metric is crucial, as it enables us to test our algorithms in simulation, where we can easily compute the violation rate, but where it is difficult to accurately simulate the effect on QoS.

We support this claim using two arguments based on a study of Borg, which employs a predictor that enables overcommit by linearly scaling down the running tasks' limit by a constant factor, similar to [5, 18, 27–29, 44]. Note that the peak predictor's specific implementation is irrelevant here



(a) Per-machine violation rates (b) CPU scheduling latency. (c) Per-cell utilization (d) 99%ile CPU scheduling latency

Figure 3. *Per machine oracle violation rate and performance.*

as the relationship between violations and performance is independent of the quality of prediction. Using data collected from Borg, on a macro level, we show that clusters with a higher average violation rate also have a higher average CPU scheduling latency. Later, on a per-machine level, we show a correlation between a machine's violation rate and its CPU scheduling latency. Our first argument enables us to assess the relative performance of predictors: if a predictor $P(\mathcal{J}, t)$ has a lower violation rate than another predictor $P'(\mathcal{J}, t)$, the former predictor $P(\mathcal{J}, t)$ should also result in better QoS in production under a similar workload. The second argument strengthens this link by showing marginal gains in QoS when improving the predictor.

We first measure oracle violation rates and QoS performance in 5 clusters of machines, called *cells* following the convention in our prior work [50, 59, 61]. These cells are not part of the Google cluster trace [64]. Each cell is managed by a separate scheduler that manages all machines within that cell. A median cell has 11k machines and processes 13M tasks in a month, but we also include smaller and larger cells for completeness, as shown in Table 1. We show CDFs over all machines in these cells, grouped by the cell. Figure 3(a) shows each cell's per-machine violation rate, while Figure 3(b) shows the distributions of tasks' CPU scheduling latency in each cell, normalized to a common constant.

Although there is no canonical way of ordering distributions, we can still see a general trend by comparing Figure 3(a) and Figure 3(b): cell 5 has the most machines with violations, and it indeed has the highest CPU scheduling latency at the tail when we zoom into Figure 3(b). Cells 2 and 3 have the lowest violation rate among the 5 cells, and they are also the two best performing cells. Cells 1 and 4 are in the middle, better than cell 5 but worse than cells 2 and 3, measured by violation rate and CPU scheduling latency.

However, one abnormality contradicts our thesis: cell 4 is clearly better than cell 1 in violation rate at every percentile, but its CPU scheduling latency is worse than cell 1. To dig deeper, we need to look at the cell-level CPU utilization distribution as shown in Figure 3(c), where each data point takes the total CPU usage of all running tasks in a cell at any given time and normalizes it to the cell's total CPU capacity. According to Figure 3(c), cell 1's utilization is lower than cell 4 at every percentile. The lower the utilization, the less likely it is that an oracle violation will lead to resource exhaustion as it requires specific events to happen in a certain order after a violation, as illustrated in Figure 2. This explains the better QoS of cell 1 compared to cell 4 despite its worse violation rate. Additionally, by analyzing the order of utilization in Figure 3(c) together with the QoS in Figure 3(b), we see that the average utilization alone is a poor predictor of QoS: cells 2 and 3 have the highest utilization, but also the lowest scheduling latency; cell 4 has the second-lowest utilization, but second-highest scheduling latency.

We next analyze the relationship between the violation rate and the CPU scheduling latency for individual machines. We take a random sample of 10,795 machines from the five cells. For each machine, we measure the violation rate averaged over the entire month, and the machine's tail (99%ile) CPU scheduling latency during the month, i.e., 99% of time, the CPU scheduling latency was lower than this value. We then normalize the scheduling latency by the average tail latency over machines that had no violations.

Figure 3(d) shows the data as an error-bar plot. We group the machines into buckets by violation rates, such that a bucket groups machines with violation rates $(x, x + 0.005]$. For each bucket we show the mean (represented by the black dot) and the standard deviation (represented by the error bars). We limit the x-axis range to the first bucket containing less than 50 machines. Although the CPU scheduling latency varies within each bucket due to confounding factors that are orthogonal to overcommit (such as NUMA locality and the network traffic) and the fact that violations do not necessarily result in resource exhaustion, there is still a clearly-visible trend: when a machine's violation rate increases, the 99%ile CPU scheduling latency also increases.

To quantify the correlation, we apply Spearman's rank correlation before and after grouping machines into buckets based on their violation rate and taking the mean. The correlation coefficients are 0.42 for the raw data and 0.95 for the mean data, respectively (p-values are smaller than 1e-100). If we fit a line to the mean latency of each bucket, the slope is 14.1. Given that the latency (the y-axis) is normalized to the mean with no violations, this slope suggests that with every percentage increase in violation rate, on average, the CPU

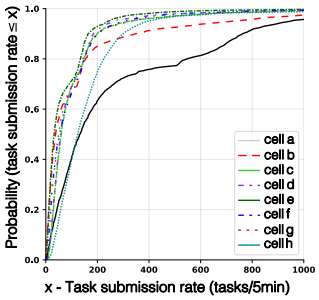


Figure 4. *CDF of number of tasks per 5 minute in Google trace v3 [59] cell a over first week period.*

scheduling latency increases by 14.1% compared to machines with no violation.

The correlation shown in Figure 3 helps us build a bridge between the performance of an overcommit policy in production and the oracle violation, which can be easily simulated offline as we show later. Such simulation-based discovery enables us to compare multiple overcommit policies and different configurations using only offline simulation before deploying anything into production. Considering a typical experiment using overcommit in production may take weeks or months assuming the experiment has not caused any disruption, evaluating overcommit policies using simulation is not only a safer way of testing ideas, but also greatly improves the velocity of development.

4 Practical Peak Predictors

While our peak oracle is optimal, it requires complete knowledge of the future, and is thus impossible to implement in practice. Of course, we can implement the peak oracle in simulation using historical data where we have such complete knowledge. We can then use our implementation to evaluate practical peak predictors as shown in Section 5.1.1. Our goal in designing a practical peak predictor is to approximate the oracle's behavior using only data available at the moment of prediction: per-task usage $U_i(t)$ and limits L_i.

Google's internal production environment puts additional practical constraints on the overcommit policy that we employ. A typical cluster has roughly 10,000 physical machines and is managed by a logically-centralized Borg scheduler. Borg periodically polls data from agents running on each machine, or Borglets [59], for their available resources. With a complete view of the cluster, the scheduling algorithm at Borg makes task placement decisions.

Running the peak predictor in the centralized scheduler is prohibitively expensive considering that it manages tens of thousands of machines, each running hundreds of tasks, while also maintaining a tight SLO on scheduling latency. Combined with a high arrival rate of new tasks as shown in Figure 4, the centralized Borg scheduler does not have the luxury of running a sophisticated prediction algorithm when making scheduling decisions. Thus, we choose to implement the policies in Borglets on *each* machine. We thus run the predictor in parallel and independently on individual machines and periodically update the prediction to the centralized scheduler. However, because each machine's Borglet shares its CPU and memory with production tasks, it should use as few resources as possible. In addition, the peak predictor should not depend on services running on other machines to ensure it does not cause the entire cluster to fail due to a single service's failure. Thus, our predictors must be lightweight, in both CPU and memory footprint, and should be fast to compute, so as to should respond within the polling frequency of the central scheduler. To maintain a light CPU workload, the predictors should almost always prefer a simple algorithm over a complicated one. In addition, to reduce the memory footprint, the node agent should not have to store too much information about the tasks. Thus, for each task, we only maintain a moving window to store the most recent samples; we denote the window size by *max_num_samples*.

Newly launched tasks would normally need a warm-up period before exhibiting a stable resource usage pattern that a peak predictor can use. To accommodate this, all the predictors we designed take another parameter, *min_num_samples*. For tasks with less than *min_num_samples* samples, the predictor considers only the task's limit as its usage when making predictions. In other words, the predictions are only made for the tasks with more than *min_num_samples* samples, which is then adjusted by adding the limit of tasks with less than *min_num_samples*.

We next outline peak predictors from prior work as additional baselines, as well as propose new peak predictors.

Borg-default Predictor (*borg-default*). This predictor is similar to the default overcommit policy used by Borg. It overcommits CPU resources by a fixed ratio [2]. This predictor computes the sum of tasks' limits and then takes a certain fraction ϕ as the estimate of future peak usage: $P_L(\mathcal{J}, t) = \phi \sum_{i \in \mathcal{J}} L_i$. $\phi = 1.0$ corresponds to no overcommit, as each task's usage is capped by its limit, $U_i(t) < L_i$. This policy has been widely adopted by many other systems in practice due to its simplicity [5, 18, 27–29, 44].

Resource Central-like Predictor (*RC-like*). This predictor is motivated by the overcommit policy from Microsoft's Resource Central, which predicts the machine-level resource usage peak as a sum of a percentile of each individual task's resource usage [14]. This predictor takes a percentile of each individual task's resource usage and returns the sum of the percentiles, $P_{pk}(\mathcal{J}, t) = \sum_{i \in \mathcal{J}} perc_k(U_i)$, where k denotes the percentile that is used, e.g., P_{p95} uses the 95%ile .

N-sigma Predictor. The central limit theorem states that the sum of independent, identically distributed (i.i.d.) random variables tends to a Gaussian distribution. However, these assumptions may not apply for random variables modeling the tasks' resource usage in a datacenter. For instance, the resource usage for tasks within a single job may be

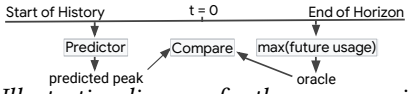


Figure 5. *Illustrative diagram for the overcommit simulator.*

correlated, as a job is driven by a load balancer. In addition, tasks from different jobs may exhibit different distributions [33]. However, as demonstrated in [17, 33], Gaussian approximation of the total load of a machine matches the actual distribution well ([33] used this approximation for bin-packing, i.e., no dynamic task arrivals and departures). Thus, our *N-sigma* predictor $P_{N\sigma}(\mathcal{J},t)$ leverages this insight by computing the mean total usage $\overline{U(\mathcal{J},t)}$ and its standard deviation $std(U(\mathcal{J},t))$ and then returns $P_{N\sigma}(\mathcal{J},t) = \overline{U(\mathcal{J},t)} + N \times std(U(\mathcal{J},t))$. Assuming the Gaussian approximation of load is correct, $P_{2\sigma}(\mathcal{J},t)$ corresponds to the 95%ile of the distribution, and $P_{3\sigma}(\mathcal{J},t)$ corresponds to the 99%ile.

Max Peak Across Predictors. This algorithm takes the output of the algorithms above as its input. Given a set of algorithms that each individually estimate a machine's future peak usage, this predictor estimates the peak usage as the maximum of the estimate across all the algorithms.

5 Evaluation in Simulation

We evaluate our overcommit policies in two different ways. First, in simulation, we compare our policies to the peak oracle by comparing the performance of each policy with the peak oracle. We also use the simulator to tune each policy's parameters. Second, in Section 6, we implement and deploy our policy on a representative fraction of machines managed by Borg. This deployment enables us to compare QoS metrics between machines using our policy and the control group, as well as to quantify our policy's savings, i.e., how much denser our policy packs tasks compared to the control group.

5.1 Evaluation Setup

Below, we describe the design of our simulator and its accuracy in mimicking the production environment, as well as present our key evaluation metrics.

5.1.1 Overcommit simulator. Our overcommit simulator provides a framework for experimenting with different overcommit policies by simulating peak predictors running independently on each simulated machine. The simulator is designed to closely match the *machine-level component* of the Borg scheduler. We decided not to simulate the scheduling algorithms, as these algorithms are complex and also dependent on feedback loops between jobs and the scheduler.

Our simulator is available under an open-source license to enable the development and evaluation of new prediction algorithms. Our simulator facilitates integrating new predictors and standardizes their interface to ease their later conversion to a production environment. Our overcommit simulator has two key goals: 1) compare the peak oracle with a predictor; and 2) facilitate testing over a variety of scheduling scenarios.

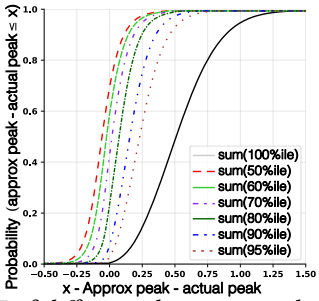
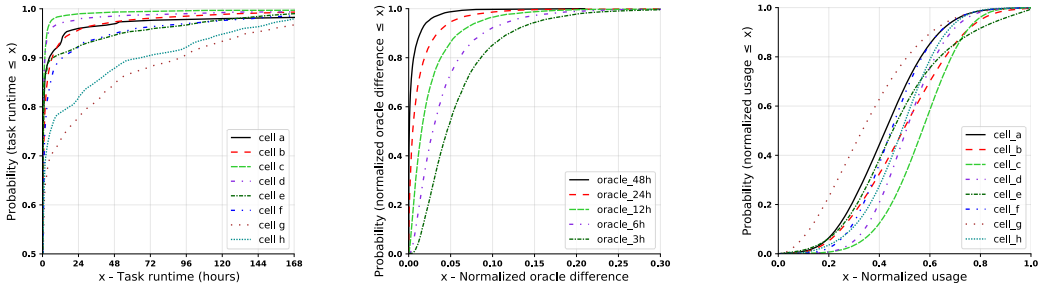


Figure 6. *CDF of difference between machine-level peak as estimated by sum of nth percentile usage (approx peak) and the actual machine-level peak.*

1) Oracle vs. Predictions. Figure 5 shows the one-line diagram of the simulator. Machines are simulated independently. For each machine, the input of the simulator is the set of time-series $\{U_i[t]\}$ specifying the complete resource usage of each task i scheduled on the machine during the simulated time period, such that $U_i[t] = 0$ before the task arrives on the machine and after it completes. For each instance in time τ, the simulator sends the simulated predictor algorithm the historic usage $U_i[t], t < \tau$ and gets the predicted peak. The simulator also computes the *peak oracle* using the future usage $U_i[t], t \geq \tau$. The predicted peak and *oracle* are compared to evaluate the accuracy of the prediction algorithm.

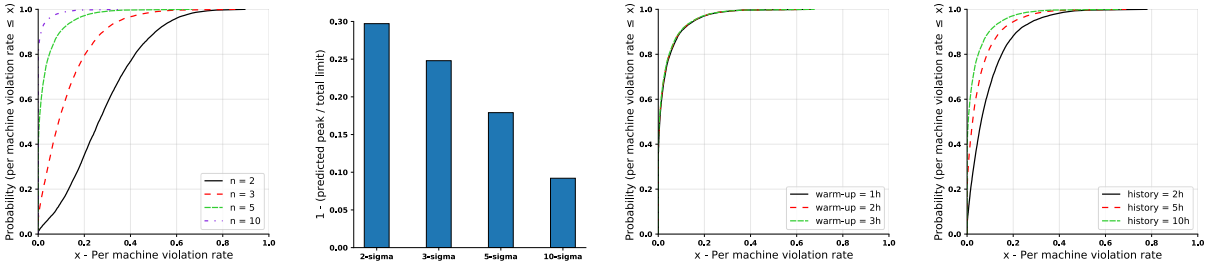
2) Simulation Scenarios. The simulator implements a configurable data processing pipeline to enable simulating a wide variety of realistic scenarios, i.e., task usage time series $U_i[t]$ allocated on each simulated machine. As its default, the simulator takes tasks' resource usage and machine placement from the Google cluster trace v3 [59, 65]. We also provide the user with the following set of optional functions: filter VMs based on time, priority, scheduling class, and alloc configurations; choose the metric a user wants to use for predicting the peak resource usage; enable generating more data than the public trace by shifting measurements in time; and choose the type of scheduler they want to use for their simulations. Users can also store and load intermediate data after each step to reduce the simulation's computation costs.

5.1.2 Simulation settings. We use the latest (v3) version of the Google cluster trace [59]. Most of the results and analysis are performed using the first week data from cell a. Section 5.5 shows results on other weeks and from other cells. We keep only the top level tasks (*alloc collection id* is 0). Tasks are not filtered by priority. As our infrastructure costs are driven largely by serving jobs, in our simulations we only consider latency sensitive tasks from the trace, which corresponds to scheduling classes 2 and 3. The trace provides the CPU usage distribution over the 5 minute interval and not the machine-level peak usage. However, internally, Borg stores machine-level peak usage, which we use as ground-truth to estimate the machine-level peak using task level usage information. As shown in Figure 6, if we estimate the machine-level peak as the sum of task level peaks from the CPU usage distribution, we significantly overestimate the



(a) Task runtime (b) Oracles with different horizons (c) Usage-to-limit ratio

Figure 7. *CDFs for (a) task runtime, (b) difference between oracles of different horizons normalized to 72h oracle, and (c) 5-minute task usage to its current limit. The CDF is computed over all 5-minute intervals for all tasks of cell a during the first week.*



(a) Effect of n (2h warm-up, 10h history) (b) Effect of n on savings (2h warm-up, 10h history) (c) Effect of warm-up (n=5, 10h history) (d) Effect of history (n=5, 2h warm-up)

Figure 8. *CDFs of per machine violation rate for* N-sigma *predictor under different parameters: a) n, c) warm-up period, and d) history. The effect of n on average cell-level savings shown in (b).*

machine-level peak since all tasks do not peak at the same time. We conservatively choose the 90%ile usage since it is greater than the actual peak for more than 95% of the time. Additionally, we cap each task's usage to its limit, as if there is resource contention; our scheduler also applies such capping. Finally, we do not modify the placement decisions of the Borg scheduler from the trace.

5.1.3 Evaluation metrics.

We use three metrics to evaluate overcommit policies in simulation: savings, violation rate, and violation severity. The savings metric is used to evaluate the benefit of an overcommit policy, while the other two measure risk. Unless otherwise stated, metrics are calculated for each machine averaged over the length of the simulated period. We mostly focus on cumulative distribution functions (CDFs) of these metrics over all machines.

Savings Ratio: For each 5-minute interval t, we compute the total limit of the tasks currently executing on a machine, $L(t) = \sum_{i \in \mathcal{J}} L_i(t)$. We then compare the predictor's output with the limit by calculating the relative difference, $(L(t) - P(\mathcal{J}, t))/L(t)$. Because the difference between the total limit, $L(t)$, and the predicted future peak, $P(\mathcal{J}, t)$, yields the amount of additional usable capacity for running tasks that is unavailable without overcommitting, we can translate the savings ratio directly to the additional capacity created by an overcommit policy, which represents the primary benefit of overcommitting resources.

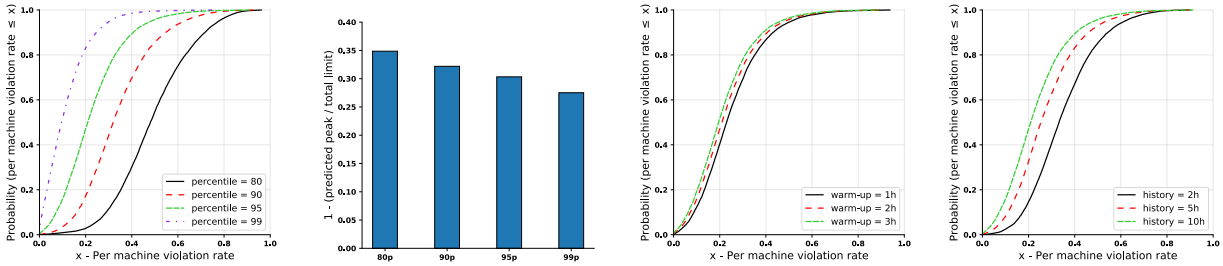
Violation Rate: The total number of oracle violations (Section 3.2): $|\{t : P(\mathcal{J}, t) < P_O(\mathcal{J}, t)\}|$. We normalize this total number of oracle violations by the length of the trace, i.e., the number of distinct time instances t.

Violation Severity: If, for an interval t, the peak prediction $P(\mathcal{J}, t)$ is lower than the peak oracle $P_O((J, t)$, the relative difference between the prediction and the oracle, normalized by the oracle value (and 0 if there is no oracle violation): $\max(0, P_O(\mathcal{J}, t) - P(\mathcal{J}, t))/P_O(\mathcal{J}, t)$.

5.2 Configuring peak oracle and *borg-default* predictors

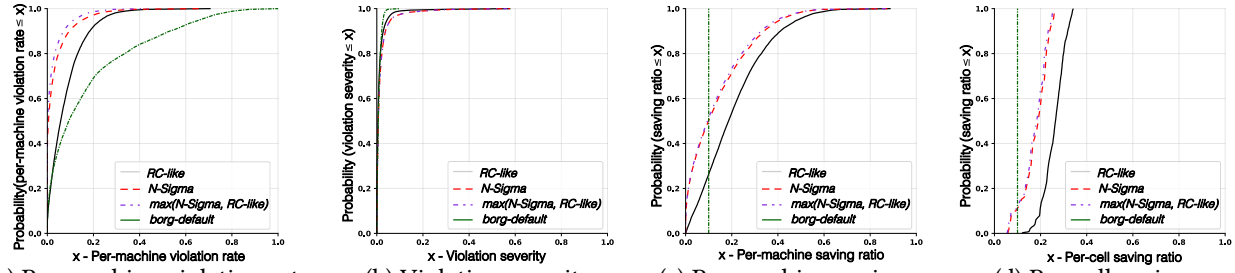
We first perform an exploratory analysis of the Google trace data to determine 1) how far in the future the peak oracle should look into (the oracle horizon) and 2) at what fraction of the total limit the *borg-default* predictor should be set.

Figure 7(a) shows the CDF of task runtime. As shown, there is a significant variation between task runtime across cells. For example, cell c has more than 98% of the tasks that are less than 24hrs, while cell g has only 75% of the tasks are less than 24hrs. The runtime of tasks is important in determining the right forecast horizon for the *peak oracle*; this forecast horizon specifies how far an oracle can see into the future. The *peak oracle* at t provides the peak future usage over the horizon for all the tasks present at t. Since all the tasks in the first week's trace are less than 168 hours, an oracle with a horizon of 168 hours will capture the true peak for all the tasks. However, there is a trade-off between accuracy and computational cost, since running an oracle with a long horizon is computationally expensive. Figure 7(b) shows how shorter-horizon oracles (3h-48h) compare to a



(a) Effect of percentile (2h warm-up, 10h history) (b) Effect of %ile on savings (2h warm-up, 10h history) (c) Effect of warm-up (95%ile, 10h history) (d) Effect of history (95%ile, 2h warm-up)

Figure 9. *CDFs of per machine violation rate for* RC-like *predictor under different parameters: a) percentile, c) warm-up period, and d) history. The effect of percentile on average cell-level savings shown in (b).*



(a) Per-machine violation rate (b) Violation severity (c) Per-machine savings (d) Per-cell savings

Figure 10. *CDFs of per-machine violation rate (a), violation severity (b), per-machine savings (c), and cell-level savings (d) for different predictors over one week period for cell a. Per-machine savings show the distribution of average difference between machine-level limit and machine-level peak, normalized to machine-level limit. Cell-level savings show the distribution over all 5-minute periods of the difference between cell-level limit and cell-level peak, normalized to cell-level limit.*

longer-horizon oracle of 72 hours. As the oracle horizon increases, the difference between the predicted peak decreases. For a 24-hour oracle, we are less than 5% lower than the 72-hour oracle peak for more than 95% of tasks. This occurs because the majority of tasks in the first week's trace are shorter than 24 hours. Further, even some of the longer jobs show daily periodic behavior, which means that the maximum peak of their lifetime will occur in a 24-hour period. Due to these reasons, we argue that a 24-hour oracle is sufficient, and choose it as the default horizon for the oracle in all our subsequent experiments.

Figure 7(c) shows the distribution of resource usage to limit ratio for each task at each time for all cells. As shown, all cells show highly consistent behavior with the 95%ile usage-to-limit ratio for all the cells being less than 0.9. We set the static fraction for the *borg-default* predictor, ϕ, to be 0.9, as 10% of resources are not utilized 95% of the times.

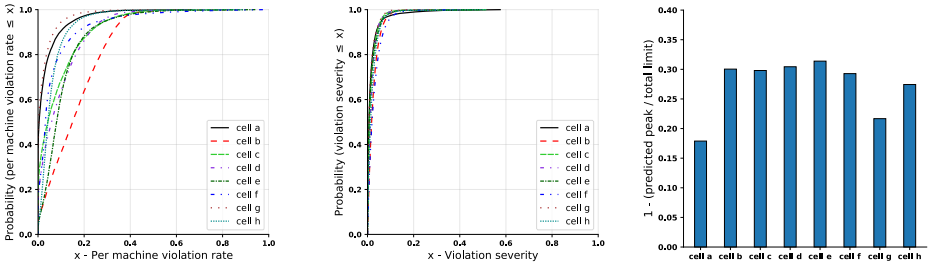
5.3 Configuring peak predictors

Figure 8 shows how the configuration parameters for the *N-sigma* predictors affect the per-machine violation rates and savings. For the *N-sigma* predictor, the per-machine violation rate decreases as we increase n while keeping the warm-up time and the amount of history constant (Figure 8(a)). This is expected since at high values of n the predicted peak approaches the limit, which leads to lower violation rates. However, there is a trade-off between the violation rate and the savings. Figure 8(b) shows that as n increases, the savings reduce. Thus, n is chosen such that it generates the maximum savings given the violation rate does not violate SLOs.

Figure 8(c) and 8(d) show the effect of warm-up period and history on per-machine violation rate, respectively. The warm-up period does not affect the violation rate significantly. This occurs because even 1h is a long period for the task usage to reach a steady state. In contrast, the length of history for each task has a more pronounced impact on the violation rate, as longer history better captures the long-term behavior, which is more indicative of future peak usage.

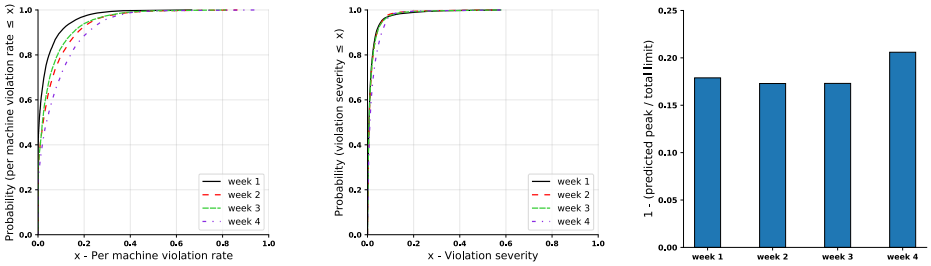
We perform a similar comparison for the *RC-like* predictor (shown in Figure 9). As we increase the percentile usage for each task, the violation rate decreases (Figure 9(a)) but the savings also decrease (Figure 9(b)). This is expected since at high values of p the predicted peak approaches the limit, which leads to lower violation rates that, in turn, reduce savings. The effect of the warm-up period and history on the per-task predictor is very similar to the *N-sigma* predictor, as expected. The warm-up period has only a marginal effect on the violation rate (Figure 9(c)), while history has a more significant impact on the violation rate (Figure 9(d)).

Based on this analysis, we pick $n = 5$ for the *N-sigma* predictor and *99%ile* for the *RC-like* predictor, as these values yield acceptable violation rates with good savings. Increasing the history decreases the violation rate, but the memory footprint of the prediction algorithm also increases. Therefore, we keep only 10 hours of history for each task. The warm-up period does not have a significant impact; we set



(a) Per-machine violation rate (b) Violation severity (c) Savings

Figure 11. *Performance of max predictors across three metrics: per-machine violation rate (a), violation severity (b), and savings (c) over one week period across all cells. Max predictor take max over* N-sigma *predictor (n = 5, 2h warm-up, 10h history) and* RC-like *predictor (99%ile, 2h warm-up, 10h history)*



(a) Per-machine violation rate (b) Violation severity (c) Savings

Figure 12. *Performance of max predictors across three metrics: per-machine violation rate (a), violation severity (b), and savings (c) for cell a over all weeks. Max predictor take max over* N-sigma *predictor (n = 5, 2h warm-up, 10h history) and* RC-like *predictor (99%ile, 2h warm-up, 10h history)*

the warm-up period as 2 hours. Unless stated otherwise, we use these configuration parameters for the two predictors in all subsequent experiments.

5.4 Peak predictor performance

Figure 10 compares the performance of different predictors using per-machine violation rate, violation severity, and savings (per-machine and cell-level). The *borg-default* predictor and *RC-like* predictor have the worst per-machine violation rates (Figure 10(a)). The *borg-default* predictor ignores the characteristics of the workload on a given machine. While it works for some of the machines, e.g., 0 violation rate for 20% of the machines, this static approach performs poorly for a large set of machines, e.g., 30% violation rate for 20% of the machines. While the workloads are different, our production environment that uses a version of the *borg-default* predictor yields similar performance (Figure 3(a)). The task-level resource usage generally shows considerable variation and, therefore, the *RC-like* predictor does not yield high performance. On the other hand, the *N-sigma* predictor, which uses the machine-level aggregate resource usage performs much better. Our max predictor, built on top of per-task and *N-sigma* predictor, yields the best performance, since no single predictor is be best suited for all the machines at all times.

Figure 10(b) shows the violation severity distribution for different predictors. Since the static predictor sets its overcommit limit 10% below the resource limit, the maximum degree of violation it can observe is 10%. However, not all tasks use resources up to their resource limit and therefore the static predictor shows the best performance, as the 99.99%ile violation severity is less than 0.05. The *N-sigma* and *RC-like* predictors also show similar performance. This demonstrates that while our predictors decrease violation rates, the magnitude of violations still remains comparable.

Figure 10(c) and 10(d) show the savings ratio at the machine level and cell level, respectively. Since the static predictor overcommits by a constant factor for all machines, it has a fixed savings of 10% at both the machine level and the cell level. The *RC-like* predictors generates the highest savings since it often violates the oracle by predicting a lower peak. In contrast, the *N-sigma* predictor conservatively predicts the peak and thus generates lower savings. Our max predictor yields slightly better performance than the *N-sigma* predictor. At the machine level, we see considerable variations in savings as the load on different machines varies significantly. The aggregate cell-level load is more stable, which results in a narrow range for savings distribution.

5.5 Behavior in time and across cells

Figure 11 shows the max predictor evaluated on all the cells. All cells, except b, show comparable performance with cell a (Figure 11(a)). The analysis of the utilization data shows that cell b has, at the median, the lowest per-machine utilization standard deviation. This low variation leads the *N-sigma* to predict lower peaks. Consequently, the *RC-like* algorithm becomes the guarding factor and always determines the peak. Looking at the violation rates for all predictors, Figure 10(a), we find that the cell b's violation rate is similar to *RC-like* predictor. Violation severity is similar in all cells, Figure 11(b) The savings for other cells is also almost always greater

than for cell a (Figure 11(c)). These results confirm that our predictors support a variety of workload profiles. Figure 12 shows our max predictor evaluated on all four weeks of cell a, including the week 1 results from Figure 10. The performance is consistent with the performance in the first week.

6 Evaluation in Borg

Starting from around 2016, Borg has been using a limit-based peak predictor (*borg-default*) for scheduling latency sensitive tasks. The predictor has been tuned over the years for Google's workload and has not caused any major incidents or user-visible performance degradation in recent years. Taking this fine-tuned *borg-default* peak predictor as our baseline, we evaluate the max predictor both in terms of its benefit, as measured by its savings, and risk, as measured by its CPU scheduling latency. In addition, we also show the per-machine violation rate and violation severity of both predictors to further validate the methodology of using metrics derived from simulation to infer the risks of deployment. However, note that the primary goal of overcommitting resources is to increase usable capacity for scheduling tasks without significantly sacrificing performance. For this reason, we tuned our max predictor in simulation to match the risk profile of our *borg-default* peak predictor used by Borg in terms of its violation rate and severity.

6.1 Production setup

We perform a one-month-long A/B experiment on a random sample of 24, 000 machines representative of Google's infrastructure. We sample these machines from at least a dozen clusters, where a typical cluster has roughly 11,000 machines. We place half the machines in our sample, or ≈12,000, in the experimental group where we deploy our max peak predictor, while we place the other half in the control group running *borg-default* policy that has been tuned in production for years. A similar limit-based static predictor has also been widely adopted by other platforms [5, 18, 27–29, 44]. We configure max predictor as the maximum over n-sigma predictor ($n = 3$, 2h warm-up, 10h history) and RC-like predictor (80%ile, 2h warm-up, 10h history). For our production evaluation, we consider only latency-sensitive tasks that mostly serve end-user generated, revenue-generating requests.

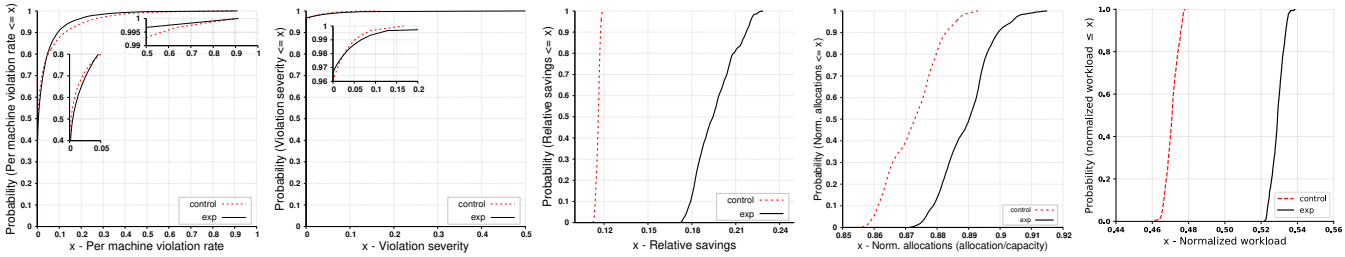
6.2 Predictor performance

Figure 13 compares the performance of our max predictor with the default predictor used by Borg using the per-machine violation rate, violation severity, and savings. The experimental group has a slightly better per-machine violation rate performance than the control group (Figure 13(a)). This result is by design, since the *borg-default* predictor running on the control group has been configured to have a risk profile, i.e., violation rate, that is acceptable in production. Thus, the goal is not to improve the risk profile, but instead to increase savings, while maintaining a similar risk profile. Figure 13(b) shows the violation severity distribution for both set of machines. As shown, both predictors have a similar risk profile such that the 98th percentile performance for both groups is comparable.

Comparison with simulation results. Comparing the violation rate distribution in Figure 11(a) and Figure 13(a), the risk profile of the max predictor running in the experimental group is similar to the results observed in simulation. This demonstrates the stability of the predictor, as it naturally adapts to the underlying workload. On the other hand, our *borg-default* predictor behaves significantly better (Figure 13(a), dotted curve) in control group than the simplified version in simulation (Figure 10(a), green curve). This difference is due to the tuning of *borg-default* in production over many years that has added many levels of ad hoc safety checks to reduce the likelihood of a violation. We did not attempt to replicate these safety checks in Section 5 , since they are not well-defined or well-documented. Even given these safety checks, the *borg-default* predictor still often has significant violations, as shown in Figure 3 for cell 1 and cell 5. Since the machines in the control and experimental group are chosen from multiple cells, their aggregate performance is better than the worst case performance of any cell.

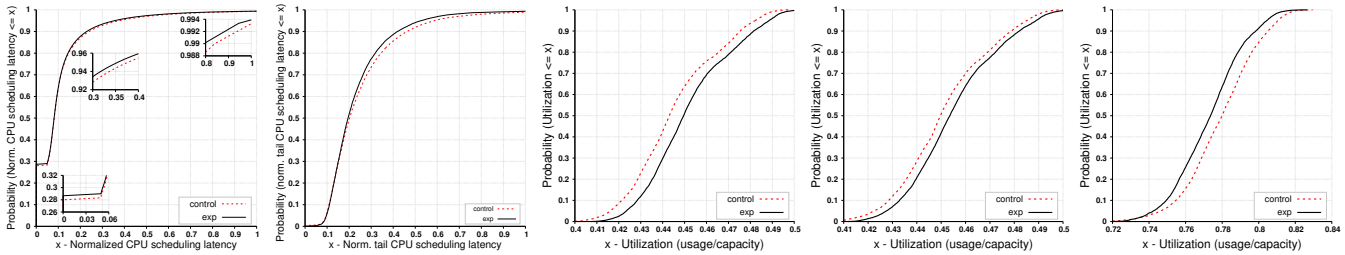
Savings and workload increase. Figure 13(c) shows the distributions of overall savings for both groups. Savings, calculated as in Section 5.1.3, are the difference between the total limit of all running tasks in the group and the sum of the predicted future peak of every machine, normalized by the total limit of all running tasks, and aggregated over time instances and machines. The difference directly tells us how much additional usable capacity is created by the overcommit policy compared with no overcommit, i.e., predicted future peak of a machine is the total limit of all running tasks on the machine. We then normalize the difference to the total limit of running tasks because it is the metric we used to guide our future capacity expansion for datacenters, similar to [11]. Both groups generate more than 10% of the additional capacity, while the max predictor consistently generates more savings that is higher than 16%. Again, the savings directly translate into usable capacity, which reduces the purchase of capacity in the future order, and hence lowers CapEx. The additional usable capacity in the experimental group invites more workload than the control group: the increase is ≈2% if we measure it by the total limit of running tasks at a given moment normalized by the total physical capacity (Figure 13(d)), and ≈6% if we measure it by actual usage normalized by the total physical capacity (Figure 13(e)). The additional workload then translates to either higher revenue if datacenters host 3rd party jobs, or, as in our case of an internal workload, increased efficiency.

Improvement in performance. Figure 14(a) shows the CDF of the CPU scheduling latency of each task measured from both the experiment and control group. The experimental group yields lower CPU scheduling latency, i.e., better



(a) Violation rate **(b)** Violation severity **(c)** Relative savings **(d)** Total allocations **(e)** Total workload

Figure 13. *Production experiment results. Performance of* borg-default *and max predictor across three metrics: per-machine violation rate (a), violation severity (b), relative savings (c), total allocations (d), and total workload (e) for production machines over 32 days.*



(a) Per-task CPU scheduling latency **(b)** 90%ile per-machine CPU scheduling latency **(c)** 50%ile machine utilization **(d)** Average machine utilization **(e)** 99%ile machine utilization

Figure 14. *Performance improvement in production. Normalized CPU scheduling latency CDF (a), 90%ile per-machine CPU scheduling latency (b), 50%ile machine utilization (c), average machine utilization (d), and 99%ile machine utilization (e) for production machines over 32 days.*

QoS, at all percentiles. For example, at the 90th %-ile, the latency is reduced by 5%. This should not come as a surprise after seeing better violation rates in Figure 13(a) and knowing how it relates to the performance metric described in Section 3.3. Looking further, the tail CPU scheduling latency per machine, as defined by the 90th percentile over the whole month for each machine (Figure 14(b)), tells us a similar story: the experimental group consistently outperforms the control group at almost every percentile. It is counter-intuitive to see the experimental group yield both higher workload (Figure 13(d)(e)) *and* better performance (Figure 14(a)(b)), since conventional wisdom tells us that systems tend to perform poorly as their workload intensity increases. However, this inconsistency can be resolved if we take a look at how the workload is distributed across machines in each group. At each instance in time, an average machine in the experimental group has higher utilization compared with an average machine in the control group, if we look at the median and average machine utilization (Figure 14(c)(d)), hence the increase of the overall workload as seen in Figure 13(d)(e). However, at the tail, the hottest machines in the experimental group are actually *less* utilized compared with machines in control group (Figure 14(e)), hence the higher QoS at the tail. In the end, the experimental group gives us a more balanced workload across machines providing better and more consistent performance.

We emphasize that our principal goal is not to improve performance, but rather to increase the usable capacity for scheduling tasks while maintaining an *acceptable* performance. We are pleased to see the max predictor having lower violation rates, and hence better task performance, while yielding a higher savings. This result suggests that the max predictor could be further tuned – to yield similar QoS to our current, *borg-default* policy, but with even higher savings.

7 Related Work

The key new elements of our approach relative to prior work are that 1) we do not change the cluster scheduling algorithm; and 2) we oversubscribe serving tasks with other serving tasks. Avoiding modifications to the scheduling algorithm is important, as these tend to be extremely complex in production systems; this complexity follows from various placement constraints, e.g., affinities and anti-affinities, resilience constraints [21, 66]. Additionally, this clear separation of concerns follows the 'relying on insights' architecture of merging machine learning into resource managers [6]. Thus, our work can be directly incorporated into production schedulers like Borg [59, 61], Omega [53], or Kubernetes [9, 19].

Workload balance: As shown in Section 6.2, our approach makes machines naturally adapt their workload resulting in a more balanced workload distribution, which has been considered as an important objective for datacenters [41, 45].

Memory overcommit: In addition to applying the methodology introduced in this paper, which focuses on statistical

multiplexing of CPUs, memory can be further overcommitted using technologies like page compression [42, 62], local or remote swap devices [26, 43, 46, 57].

Scheduling: Prior work also focuses on improving efficiency through better scheduling [7, 22, 24, 25, 31]. In principle, our approach is orthogonal to cluster schedulers: it can be used on top of any of these approaches to either pre-filter the set of available machines, or to manage node-level task queues, as in Apollo [7]. Our work is closely related to bin-packing: the peak predictor is essentially a test of whether an item fits into a bin. We refer to [13] for a recent overview of theoretical results in bin packing under chance constraints. [33] proposes an algorithm analogous to our *N-sigma* predictor, but is only evaluated for a classic stochastic bin packing (off-line, no arrivals/departures). Some production schedulers do not overcommit (e.g. TWINE [58]). FIRM [49] takes a different approach: it assumes that machine resources are unlimited; and if an action leads to oversubscribing of a resource, then it is replaced by a scale-out operation.

Overcommit for batch jobs: [35] requires changes in the cluster scheduler and annotations of the workload; this would be difficult to enforce in a production system used by thousands of engineers. Morpheus [37] targets periodic batch jobs: based on previous executions of a job, the scheduler derives an implicit deadline, i.e., an implicit SLO, and then combines period reservations with dynamic re-adjustment to fulfill this deadline. Our approach targets serving jobs with explicit SLOs, i.e., CPU scheduling latency, and, importantly does not require changing the scheduler. ROSE [56] proposes a distributed overcommit mechanism for centralized batch schedulers like YARN [60].

Overcommiting serving jobs with batch jobs: Our approach overcommits serving jobs, since they are the primary driver in determining our fleet's capacity. The remaining capacity is filled with best-effort jobs [11]. Prior work addresses choosing data-intensive, batch workloads to complement serving workloads [30, 38, 68]. Scavenger [34] dynamically adjusts batch workload limits to avoid interference with serving workloads, and Rythm [69] constructs finer-grained models of serving jobs to optimize the impact on latency. [1] proposes *harvest VMs* offering SLOs for the unallocated capacity. This approach is complementary to overcommit that increases utilization of the allocated capacity.

Per-task/VM usage predictions: Our peak predictor estimates the total usage of a machine, rather than estimating the usage of individual tasks or VMs. Our approach is complementary to Autopilot [52], which predicts per-task limits, and thus operates over a long-term forecast horizon, i.e., the task's duration (see Section 2.2, Figure 1). PRESS [23] and CloudScale [54] predict VM CPU usage over a short horizon (1 minute) to dynamically adjust resource limits and drive VM migrations, while AGILE [47] predicts VM pool load over a 2-minute horizon to horizontally scale the pool before load spikes. Our predictors target, and are evaluated against, significantly longer forecast horizons. [15] shows a CPU overcommit algorithm that relies on predicting high percentiles of utilization for individual VMs when marking a physical machine as eligible for overcommit, which is reminiscent of our percentile predictor. Yet, they further restrict the overcommit ratio to a certain static fraction that is 15%-25% of the limit. Their work also only focuses on non-production workloads, while ours explicitly targets serving, latency-sensitive tasks. Resource prediction in general can be used in domains outside overcommit. For example, [48] predicts CPU usage for a particular class of VMs that host database servers to aid in finding long periods of low CPU utilization when backups can be performed. Additionally, [10, 32, 40, 51, 55] outline different prediction algorithms for resource management in datacenters. These prediction algorithms are generally not lightweight and, thus, are not suitable for our setup.

Capacity planning: Datacenter owners must make long-term decisions on how much additional physical capacity they need to provide to their users. Such decisions are made by analyzing both datacenters' utilization in the past, and their projections of expected future resource demand. [11] predicts for a whole cluster and for longer-term periods, e.g., 6 months, the total slack between utilization and demand, which is related to our usage-to-limit gap. Their approach is complementary to ours: once the utilization increases in the short-term through overcommit, it will eventually be reflected in long-term capacity planning so that datacenter owners need to buy fewer machines to satisfy the same amount of demand.

8 Conclusion

Our work designs overcommit policies using a peak oracle baseline policy, which is a clairvoyant policy that computes the available capacity on a machine by analyzing its future resource usage. Using this peak oracle as a baseline, we then propose a methodology that leverages offline simulation to estimate the performance impact of any arbitrary overcommit policy. The oracle violations we derive from simulation correlate well with an actual QoS metric used in practice, i.e., CPU scheduling latency. We then propose practical overcommit policies that are lightweight and applicable in real-world production systems. By comparing the oracle violations and savings of these policies, we find that our max predictor policy is less risky and more efficient than existing static, limit-based overcommit policies, which are currently widely used in practice. Finally, we deploy our overcommit policy into Borg scheduler, and show that its results are consistent with our simulation.

Acknowledgements: We want to thank John Wilkes for his review and feedback on the early draft, Dawid Piątek for helpful discussions on data analysis, and Weici Hu who helped us to form an early version of proofs around the oracle. We also thank the anonymous reviewers, and our shepherd, Kang Chen, for their helpful comments.

References

[1] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] G. Amvrosiadis, J.W. Park, G.R. Ganger, G.A. Gibson, E. Baseman, and N. DeBardeleben. 2018. On the Diversity of Cluster Workloads and its Impact on Research Results. In *2018 USENIX Annual Technical Conference (ATC)*.

[3] E. Asyabi, SA. SanaeeKohroudi, M. Sharifi, and A. Bestavros. 2018. TerrierTail: mitigating tail latency of cloud virtual machines. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[4] L.A. Barroso, J. Clidaras, and U. Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.*

[5] S.A. Baset, L. Wang, and C. Tang. 2012. Towards an Understanding of Oversubscription in Cloud. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*. USENIX Association.

[6] R. Bianchini, M. Fontoura, E. Cortez, A. Bonde, A. Muzio, AM. Constantin, T. Moscibroda, G. Magalhaes, G. Bablani, and M. Russinovich. 2020. Toward ML-centric cloud platforms. *Commun. ACM* (2020).

[7] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[8] D. Breitgand and A. Epstein. 2012. Improving Consolidation of Virtual Machines with Risk-aware Bandwidth Oversubscription in Compute Clouds. In *INFOCOM*. IEEE.

[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* (2016).

[10] R.N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. 2014. Workload Prediction Using ARIMA Model and its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing* (2014).

[11] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. 2014. Long-term SLOs for Reclaimed Cloud Computing Resources. In *ACM Symposium on Cloud Computing (SoCC)*.

[12] CDF accessed 2020-10. Cumulative Distribution Function. https://en.wikipedia.org/wiki/Cumulative_distribution_function.

[13] M.C. Cohen, P.W. Keller, V. Mirrokni, and M. Zadimoghaddam. 2019. Overcommitment in Cloud Services: Bin Packing with Chance Constraints. *Management Science* (2019).

[14] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*.

[15] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[16] N. Deng, C. Stewart, D. Gmach, M. Arlitt, and J. Kelley. 2012. Adaptive Green Hosting. In *International Conference on Autonomic Computing (ICAC '12)*. ACM.

[17] N. Deng, Z. Xu, C. Stewart, and X. Wnag. 2015. Tell-tale Tails: Decomposing Response Times for Live Internet Services. In *International Green and Sustainable Computing Conference (IGSC)*.

[18] Apache Software Foundation. accessed 2020-10. Mesos: Oversubscription. http://mesos.apache.org/documentation/latest/oversubscription/.

[19] Cloud Native Computing Foundation. accessed 2020-10. Kubernetes. http://k8s.io.

[20] The Linux Foundation. 2020. cgroups(7) Linux User's Manual. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[21] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of European Conference on Computer Systems (EuroSys '20)*.

[22] I. Gog, M. Schwarzkopf, A. Gleave, R. NM Watson, and S. Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[23] Z. Gong, X. Gu, and J. Wilkes. 2010. Press: Predictive Elastic Resource Scaling for Cloud Systems. In *International Conference on Network and Service Management*. IEEE.

[24] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. 2016. Altruistic Scheduling in Multi-resource Clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[25] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[26] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K.G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. In *USENIX Conference on Networked Systems Design and Implementation (USENIX NSDI'17)*.

[27] Red Hat. accessed 2020-10. Open Shift: Overcommit. https://docs.openshift.com/container-platform/3.11/admin_guide/overcommit.html.

[28] VMWare Inc. accessed 2020-10. DRS Additional Option: CPU Over-Commitment. https://vspherecentral.vmware.com/t/vsphere-resources-and-availability/drs-additional-option-cpu-over-commitment/.

[29] VMWare Inc. accessed 2020-10. VMware vSphere: Memory Overcommitment. https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-895D25BA-3929-495A-825B-D2A468741682.html.

[30] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, et al. 2018. Perfiso: Performance Isolation for Commercial Latency-sensitive Services. In *USENIX Annual Technical Conference ATC)*.

[31] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM Symposium on Operating Systems Principles (SOSP'09)*.

[32] S. Islam, J. Keung, K. Lee, and A. Liu. 2012. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Computer Systems* (2012).

[33] P. Janus and K. Rzadca. 2017. Slo-aware Colocation of Data Center Tasks Based on Instantaneous Processor Requirements. In *ACM Symposium on Cloud Computing (SoCC'17)*.

[34] S.A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi. 2019. Scavenger: A Black-box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *ACM Symposium on Cloud Computing (SoCC'19)*.

[35] T. Jin, Z. Cai, B. Li, C. Zheng, G. Jiang, and J. Cheng. 2020. Improving Resource Utilization by Timely Fine-Grained Scheduling. In *Proceedings of European Conference on Computer Systems (EuroSys '20)*. ACM.

[36] C. Jones, J. Wilkes, N. Murphy, C. Smith, and B. Beyer. 2016. *Service Level Objectives.* https://landing.google.com/sre/book.html

[37] S.A. Jyothi, C. Curino, I. Menache, S.M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[38] K. Kambatla, V. Yarlagadda, I. Goiri, and A. Grama. 2020. Optimistic Scheduling with Service Guarantees. *J. Parallel and Distrib. Comput.* (2020).

[39] A. Kangarlou, S. Gamage, R.R. Kompella, and D. Xu. 2010. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*.

[40] A. Khan, X. Yan, S. Tao, and N. Anerousis. 2012. Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach. In *IEEE Network Operations and Management Symposium*. IEEE.

[41] C. Kilcioglu, J.M. Rao, A. Kannan, and R.P. McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *Proceedings of International Conference on World Wide Web (WWW'17)*.

[42] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K.A. Yurtsever, Y. Zhao, and P. Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM.

[43] S. Liang, R. Noronha, and D.K. Panda. 2005. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *IEEE International Conference on Cluster Computing (ICCC'05)*.

[44] Google LLC. accessed 2020-10. Google Cloud Compute Engine: Overcommitting CPUs on Sole-tenant VMs. https://cloud.google.com/compute/docs/nodes/overcommitting-cpus-sole-tenant-vms.

[45] C. Lu, K. Ye, G. Xu, CZ. Xu, and T. Bai. 2017. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *IEEE International Conference on Big Data (Big Data)*. IEEE.

[46] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. 2003. Nswap: A Network Swapping Module for Linux Clusters. In *Euro-Par Parallel Processing*. Springer.

[47] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *International Conference on Autonomic Computing (ICAC'13)*.

[48] O. Poppe, T. Amuneke, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, et al. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. *arXiv preprint*.

[49] H. Qiu, S.S. Banerjee, S. Jha, Z.T. Kalbarczyk, and R.K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[50] C. Reiss, A. Tumanov, G.R. Ganger, R.H. Katz, and M.A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *ACM Symposium on Cloud Computing (SoCC '12)*.

[51] N. Roy, A. Dubey, and A. Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *International Conference on Cloud Computing*. IEEE.

[52] K. Rzadca, P. Findeisen, J. Świderski, P. Zych, P. Broniek, J. Kusmierek, P.K. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes. 2020. Autopilot: Workload Autoscaling at Google Scale. In *Proceedings of European Conference on Computer Systems (EuroSys'20)*.

[53] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of European Conference on Computer Systems (EuroSys'13)*.

[54] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*.

[55] X. Sun, N. Ansari, and R. Wang. 2016. Optimizing Resource Utilization of a Data Center. *IEEE Communications Surveys & Tutorials* (2016).

[56] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li. 2018. ROSE: Cluster Resource Scheduling via Speculative Over-Subscription. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[57] Andrew S. Tanenbaum. 1992. *Modern Operating Systems*. Prentice-Hall, Inc., USA.

[58] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng, et al. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[59] M. Tirmazi, A. Barker, N. Deng, M.E. Haque, Z.G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. 2020. Borg: The Next Generation. In *Proceedings of European Conference on Computer Systems (EuroSys '20)*. ACM.

[60] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing (SoCC)*. ACM.

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

[62] C.A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[63] M. Wang, X. Meng, and L. Zhang. 2011. Consolidating Virtual Machines with Dynamic Bandwidth Demand in Data Centers. In *International Conference on Computer Communications (INFOCOM)*. IEEE.

[64] J. Wilkes. 2019. *Google cluster-usage traces v3*. Technical report at https://github.com/google/cluster-data. Google, Mountain View, CA, USA.

[65] J. Wilkes. 2020. *Google Cluster Usage Traces v3*. Technical report at https://github.com/google/cluster-data. Google.

[66] E. Zhai, R. Chen, D.I. Wolinsky, and B. Ford. 2014. Heading Off Correlated Failures Through Independence-as-a-Service. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[67] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. 2013. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of European Conference on Computer Systems (EuroSys'13)*. ACM.

[68] Y. Zhang, G. Prekas, G.M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini. 2016. History-based Harvesting of Spare Cycles and Storage in Large-scale Datacenters. In *Symposium on Operating Systems Design and Implementation (OSDI)*. ACM.

[69] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao. 2020. Rhythm: Component-Distinguishable Workload Deployment in Datacenters. In *Proceedings of European Conference on Computer Systems (EuroSys)*. ACM.

A Artifact Appendix

A.1 Abstract

Our work designs and evaluates practical overcommit policies using, as a baseline, an optimal clairvoyant overcommit algorithm called *peak-oracle*, which is impossible to implement in practice. By simulating these practical policies and *peak-oracle* and comparing their results, we evaluate which policy yields the best performance. This simulation-based methodology enables us to quickly evaluate overcommit policies without risking production workloads. While we deploy and evaluate the best-performing policy in production (Section 6), our artifact focuses on the evaluation of policies in simulation (Section 5) using our overcommit simulator.

Our simulator design focuses on extensiblity and flexibility. Users can easily write a new prediction algorithm. Also, the simulation configuration can be fully described as a protobuf message, allowing users to decide how to preprocess the data before running the algorithms. Because the configuration can be fully described using protobuf message, they can be reproduced precisely with the exact configuration. Also, the simulator pipeline allows users to run multiple algorithms in parallel. To enable future work on designing overcommit policies, we publicly release our simulator's source code under a free and open source license.

A.2 Artifact Check-list

- **Dataset:** Google workload trace v3 [59**?**].
- **Overcommit policies:** The artifact implements all peak predictors presented in the paper: *peak-oracle*, *borg-default*, *RC-like*, *N-sigma*, and *max(predictors)*.
- **Metrics:** The evaluation metrics include violation rate, violation severity, per-machine savings, and per-cell savings. Metric definitions can be found in the paper and artifact document on GitHub.
- **Experiments:** Our artifact runs experiments to evaluate the performance of: overcommit and autopilot oracles (**Figure 1**), *peak-oracle* at different horizons (**Figure 7(b)**), predictors with different parameters (**Figure 8**, **9**), predictors in comparison with each other (**Figure 10**), and best-performing predictor for all cells and over multiple weeks (**Figure 11**, **12**).
- **Hardware requirements:** A practical environment for our experiments is a compute cluster or a cloud platform that provides 50+ compute nodes and supports distributed processing back-ends such as Google Cloud Datalow, Apache Spark, or Apache Flink.
- **Expected experiment run time:** Our simulations take roughly 10,000 vCPU hours and the actual run time would vary depending on the level of parallelism, e.g. the expected run time is a half-hour with 10k cores.
- **Expected experiment cost:** Approximately $1500 on Google Cloud Platform (GCP).
- **Public link:** https://github.com/googleinterns/cluster-resource-forecast/
- **License:** Apache 2.0.

A.3 Description

Our simulator is written in Python using Apache Beam (https://beam.apache.org/). The simulator uses the Google workload trace [59**?**], and mimics Google's production environment in which Borg [59, 61] operates. A Python script, named `simulator-setup-walkthrough`, develops a Beam pipeline to convert the Google workload trace to the desired format outlined in `table_schema.json`. Another Python script, named `fortune-teller-setup-walkthrough`, builds a Beam pipeline to configure simulation scenarios and run various predictors (see Section A.5 for further details). The heart of the simulator is in `simulator/fortune_teller.py`, which implements the logic to run *peak-oracle* and other practical predictors. The configurations for the simulator are defined using protobuf's text format. The format for the configuration files is described in the `simulator/config.proto`.

A.3.1 How to Access. Our artifact is available at: https://github.com/googleinterns/cluster-resource-forecast/.

A.3.2 Software Dependencies. Our simulator defines data processing pipelines using Apache Beam. Beam pipelines can be run on selected distributed processing back-ends that include Apache Flink, Apache Spark, and Google Cloud Dataflow. Our simulator should run on any cloud platform or compute cluster, with slight modifications, that supports the aforementioned processing back-ends. However, we have only tested the simulator with Dataflow on GCP. When running on GCP, the simulator uses the Compute Engine, BigQuery (BQ), Google Cloud Storage, and Dataflow APIs. We plan to test the simulator on other cloud platforms and the university-owned compute clusters in the future.

A.4 Installation

Instructions for setting up the environment are available at https://github.com/googleinterns/cluster-resource-forecast/docs/installation/. The instructions for setting up the GCP project and enabling the required APIs are available under `/docs/installation/gcp-project-config`. The instructions for creating a Google Compute Engine virtual machine (VM) instance with required permissions are available under `/docs/installation/gcp-vm-config`. Finally, the instructions for setting up a Python virtual environment with required dependencies for the simulator are available under `/docs/installation/simulator-config`.

A.5 Experiment Workflow

There are three major steps for evaluating the performance of a predictor following the data pre-processing, simulation, and data post-processing framework: Join the table, then run the simulator on the joined table, and data analysis. The first two steps can be merged into a single beam pipeline, technically. But because the joining tables take lots of time

and resources, and its results can be reused, we choose to put it into a separate pipeline so that users can run it once and not worry about it later.

Data pre-processing: The Google workload trace provides event and usage information for instances in two separate tables: `InstanceEvents` table and `InstanceUsage` table. Our simulator expects this information in a single table. The instructions for joining Google trace tables are available under `docs/experiments/joining-tables`. The output is saved to a BQ table and is used as an input for the next step.

Simulation: Our simulator enables user to configure various simulation scenarios and run multiple predictors in parallel. The instructions for configuring the simulation scenarios are available under `docs/experiments/simulation-scenarios`. The instructions for configuring and running various peak predictors are available under `docs/experiments/running-predictors`. Each predictor saves its output to a BQ table, which contain the per-machine usage, sum of limits for all tasks running on each machine, and predicted peak for each machine, along with machine-specific information.

Data post-processing: The output BQ tables from simulations are processed in a Colab notebook to compute evaluation metrics and visualize the results. The instructions to using the Colab notebook and notebook's link are available under `/docs/visualization/`.

The aforementioned steps describe the generic framework for evaluating the performance of a given predictor. To reproduce the results in the paper, we provide bash scripts, under `scripts/`, that allow users to reproduce all the simulation-based results from the paper. The instructions for reproducing simulation results from the paper are available under `/docs/reproduce-results/eurosys21/`

A.6 Evaluation and Expected Results

The above instructions reproduce the results for **Figure 1**, **Figure 7(b)**, **Figure 8**, **Figure 9**, **Figure 10**, **Figure 11**, and **Figure12**. There should not be any deviation from the figures in the paper if all the experiments are configured and run properly as described in the documentation.

A.7 Experiment Customization

One of the key goals for our simulator is to enable future work on designing overcommit policies. We enable users to customize the parameters for the existing overcommit policies as well as contribute entirely new predictors to the simulator. The instruction for customizing the existing predictors are available under `docs/customize-predictors`. Our simulator facilitates integrating new predictors and standardizes their interface to ease their later conversion to a production environment. We implement supporting classes in `simulator/predictor.py` that define the interfaces to implement new peak predictors. Users can add any data-driven, machine learning-based predictors as long as they use the specified interfaces. The instructions for contributing new predictors are available under `docs/contribute`.

A.8 Artifact Limitations

As outlined in Section A.1, we evaluate our overcommitment policies using our open-source overcommit simulator and inside our production environment at Google (managed by Borg scheduler). Section 6 evaluates the best-performing policy in our production environment. Due to numerous logistical, technical, and legal reasons, we cannot provide access to the internal production environment to reproduce production evaluation results. Therefore, this artifact focuses on reproducing simulation results presented in Section 5 using our open-source overcommit simulator.

Another limitation of the artifact is that the evaluations are very time consuming for two key reasons. First, the data preprocessing step performs sorting of 100s of GBs of data, which is computationally expensive and cannot be parallelized. However, the simulation step itself can be very fast as it allows the user to run multiple predictors in parallel. Second, GCP supports Cloud Dataflow on only four of the GCP regions, i.e. `us-east1`, `us-east4`, `us-west1`, `us-central1`. This limits the number of Beam pipelines running in parallel.

We would also like to specify that during the artifact evaluation only parts of the simulation results presented in the paper were reproduced. The evaluations were limited, both in time span of data input and number of configurations, due to resource and time constraints. However, we provide the complete instructions and scripts to reproduce all the simulation results presented in the paper. Independent reviewers are welcomed to evaluate our artifact.

A.9 Maintenance and Extensions

The current implementation of the simulator only focuses on machine-level peak predictions and ignores the impact of predicted peak on the scheduling decisions made by the scheduler. This simplification does not have any serious drawbacks as the results obtained in simulation closely match the production environment results (Section 6.2). However, incorporating the scheduling component into the simulator will not only increase the accuracy of current simulations, but also enable evaluating scheduling policies with or without overcommit policies. We plan to enhance the capability of the simulator and invite the open-source community to contribute to this development.

Finally, the simulator was implemented by the first author, Noman Bashir, while working as an intern at Google. The development was facilitated by the continuous guidance and support from Nan Deng. Both of the authors intend to actively maintain the simulator to facilitate the evaluation of existing predictors and the contribution of new predictors.

A.10 AE Methodology

Submission, reviewing, and badging methodology:
https://sysartifacts.github.io/eurosys2021/