Unlocking unallocated cloud capacity for long, uninterruptible workloads

Anup Agarwal[†], Shadi Noghabi[‡], Íñigo Goiri[§], Srinivasan Seshan[†], Anirudh Badam[‡]
[†]*Carnegie Mellon University,* [§]*Azure Systems Research,* [‡]*Microsoft Research*

Abstract

Cloud providers auction off unallocated resources at a low cost to avoid keeping hardware idle. One such mechanism is Harvest VMs (HVMs). These VMs grow and shrink as the unallocated resources in a server change. While HVMs are larger in size and less prone to eviction compared to other low-cost VMs, their resource variations severely slow down long-running, uninterruptible (hard to checkpoint/migrate) workloads. We characterize HVMs from a major cloud provider and discover large spatial variations in their stability and resources. We leverage this diversity by predicting which HVMs will be stable enough to run tasks without preemptions. We use the predictions to inform scheduling and resource acquisition decisions. Our evaluation with real workloads shows that we can reduce mean and tail (90^{th} percentile) job completion times by 27% and 44% respectively, at 75% lower cost than regular VMs.

1 Introduction

Motivation. Failure to monetize idle hardware in cloud deployments is a huge opportunity cost for cloud providers. Providers typically provision hardware for peak demand, with little over-subscription, to deliver an illusion of elastic resources with strong isolation and performance guarantees. Due to variations in demand, 25–30% hardware sits idle [3]. Many proposals try to address this problem, including better resource packing using abstractions like FaaS (Function-as-a Service), and auctioning unallocated capacity (unreliably) using Spot or Burstable VMs [2, 18, 49, 74]. A latest advancement towards isolating and exposing unallocated resources is Harvest Virtual Machines [3].

Harvest VMs (HVMs) are variable-sized VMs co-located with other regular (on-demand, high-priority) VMs. When a new regular VM is allocated to a server, the HVM shrinks in capacity, and when a regular VM finishes, it grows. This agility allows HVMs to gather 2.5–7.5× more resources compared to other low-priority low-cost fixed-sized VMs (e.g., Spot VMs) at lower eviction rates (§2.1, [3]). This creates a new opportunity and a new challenge.

Large harvested capacity overcomes a major capacity bottleneck [21, 60] allowing many large-scale applications [15, 16, 34, 47, 67, 68, 81, 89] in the financial, scientific, genomics, energy and meteorology sectors to run economically in the cloud. However, HVM's resource variations can significantly slow down these long, uninterruptible (hard to checkpoint/migrate) applications due to preemptions (§2.3).

Prior efforts try to mask such overheads using scheduling, resource acquisition, and load-balancing techniques (§5). Unfortunately, these efforts do not fit well for the combination of HVMs and long, uninterruptible workloads. They either address only VM evictions (not resource variations exhibited by HVMs), or rely on the unique properties of Spot markets. On the workload side, they often use a combination of checkpointing, migration, replication, or application level changes. These are prohibitive or impractical as uninterruptible workloads have large working sets, run at large scale, and rely on many domain-specific libraries and frameworks with complex state stored in memory (§2.2).

Our work. We seek to answer: *"How can we best use HVMs to run long, uninterruptible workloads?"* We begin by characterizing HVMs and a collection of long, uninterruptible production workloads from a major cloud provider. We find large spatial diversity in the stability and resources of HVMs, i.e., some HVMs are more stable (change less often) or get more resources than others. Simultaneously, we observe large diversity in the runtimes of tasks in our workloads.

We leverage our observations in two ways to build SLACKSCHED. First, we build a scheduling component that avoids preemptions by better matching tasks to HVMs, i.e., runs longer tasks on more stable HVMs and vice versa. Second, we build a resource acquisition component that improves overall stability of the HVM pool by retaining relatively stable HVMs and continuously de-allocating unstable HVMs.

In building SLACKSCHED, a key technical challenge is identifying which HVMs are going to remain stable in the future. Resource variations in HVMs can depend on a number of factors which are hard to predict or control (e.g., the arrivals, lifetimes, and placement of regular VMs). We work around this using our insight that the distribution of time between HVM resource changes is relatively stationary over time. We use this to estimate when new resource changes are likely to occur and match tasks to HVMs that are likely to not change during task lifetimes.

We implement our scheduler as a pluggable component of YARN [5], a popular cluster orchestrator, and the acquisition component as a module that manages resource negotiation between the cloud provider and YARN. We evaluate SLACKSCHED under a variety of production workloads, operating conditions, and HVM environments considering

HVM traces collected from multiple regions and time periods. We find that SLACKSCHED reduces mean and tail (90^{th} percentile) job completion times by 27% and 44%, respectively.

We note that our system does not make any assumptions about, nor is reliant on, the cloud provider's allocation policy. The diversity of unallocated resources is fundamentally tied to the diversity in regular VM workloads. As evidence, we consider future sources of resource variability, e.g., if unallocated resources change in capacity based on variations in power supply to the data center due to renewable energy sources [13, 23], in addition to variations due to regular VM arrival/departure. We find SLACKSCHED has similar performance in this new environment.

Summary. We make the following contributions:

- We characterize the behavior of real-world production HVMs and long, uninterruptible cloud workloads (§2).
- We build a practical method to estimate future variations in HVMs (§3.1.1, §3.1.2).
- We design and implement SLACKSCHED, a system that enables the use of HVMs for large-scale, long-running, uninterruptible workloads (§3, §3.3).
- We show that our scheduling and resource acquisition decisions are effective in mitigating the overheads of HVMs for varied workloads and environments (§4).

2 Characterization & Motivation

We first characterize HVMs (§2.1) and long-running uninterruptible workloads (§2.2). Then, we focus on the overheads of running these workloads on HVMs (§2.3). Our characterization reveals two opportunities that motivate our design (§2.4). We detail in §5 and Appendix B.1 why past efforts at managing resource variability and building reliable infrastructure out of unreliable services are ineffective for the combination of uninterruptible workloads and Harvest VMs.

2.1 Harvest VMs

Background. HVMs dynamically expand and contract to leverage the unallocated resources left by regular VMs. As more (or fewer) on-demand VMs are placed on a server, an HVM will shrink (or grow) its core count. We focus on HVMs that harvest CPU cores but our work can be leveraged when harvesting other resources (e.g., memory [32] and storage). For Spot VMs to expose the same capacity as HVMs, one needs to provision more and/or larger size Spot VMs. This significantly increases the number of evictions to handle and the management overheads (e.g., more copies of the OS).

HVMs are configured with a minimum size (e.g., {2,4,8} CPU cores and {16GB, 32GB, 64GB} of memory). If an HVM needs to shrink below its minimum size (e.g., because of on-demand VMs), it will be evicted. HVMs are overall cheaper in price than both Spot and on-demand VMs. Today, HVM's minimum size is charged at the Spot VM discount (e.g., 48% to 88% cheaper than regular VMs [87]), and each harvested core has a further discounted price.

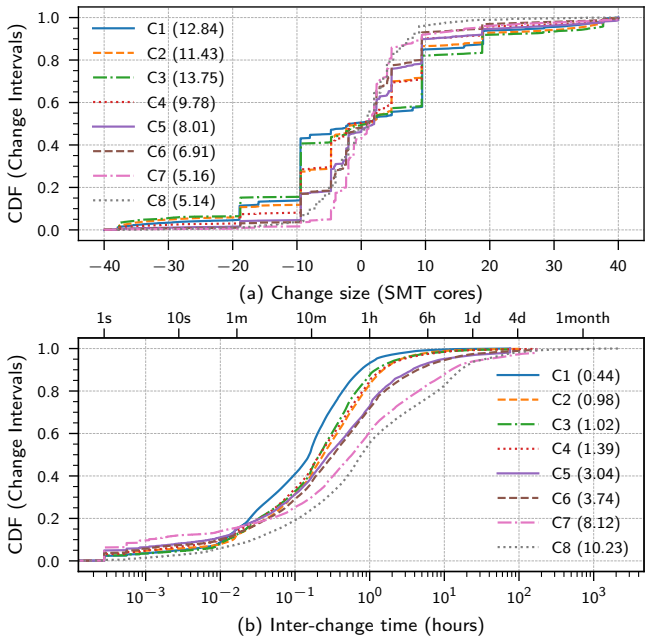


Figure 1: Resource variation in HVMs across clusters. The parenthesis in the legend lists (a) the mean magnitude of change size and (b) mean inter-change time (hrs). The clusters are sorted in increasing order of the mean inter-change time (C1–C8).

Overview. Prior work has studied properties of HVMs at an aggregate level [3, 87]. However, to understand how broader workloads might be impacted, we analyse HVMs at an individual level and answer: (1) *stability of HVMs:* how often and by how much do HVMs change?[1] (2) *spatial and temporal diversity of HVMs:* how do different HVMs compare and how do they change over time? (3) *impact of workloads:* how do the runtimes of long, uninterruptible workloads compare to the resource variations of HVMs? With this goal, we study HVM traces (from March 2019 and August 2021) for 8 clusters across 5 regions of a major cloud provider.

Stability. We measure the resource changes in terms of *size:* number of added/removed cores, and *frequency:* time between two consecutive changes (inter-change time or change interval). We count HVM evictions as a resource change to size 0.[2] We observe that different clusters witness different amounts of activity from regular VMs, so we order the clusters, with lower activity clusters on the bottom (C7–C8 in Figure 1).

Size of changes. Figure 1(a) shows the size of changes in HVM resources across clusters. Positive changes signify resource growths and negative changes signify shrink events. The mean magnitude of change is between ≈ 5 and 13 SMT cores (simultaneous multi-threaded cores or hardware threads) for different clusters. These are large variations, given the typ-

[1]While this aspect has been considered in [87], it was in the context of short-running FaaS workloads and considered only a single cluster. Hence, we revisit it in the context of our target workloads for more clusters.

[2]We do not separately study HVM evictions as these occur rarely relative to task durations in our workloads ([3], §2.2).

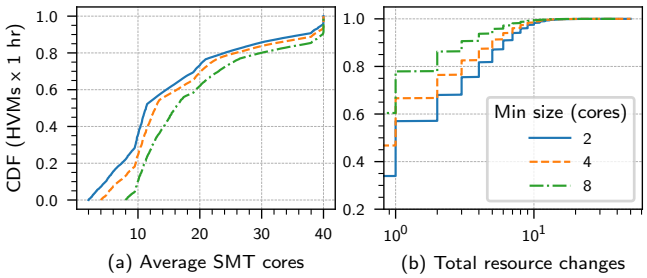


Figure 2: Spatial Variation — HVMs on different servers differ in amount of resources harvested and stability.

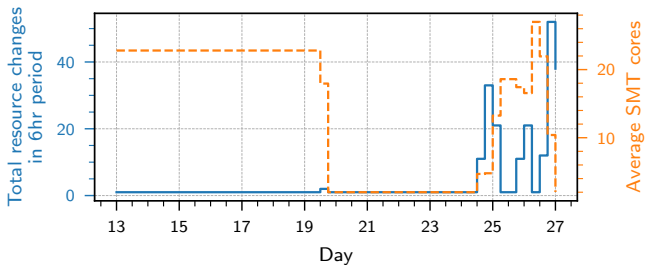


Figure 3: Temporal Variation in a single HVM.

ical minimum size of 2–8 SMT cores for an HVM and given that 1–2 cores is a popular regular VM size [25, 41].[3] Such large variations can have significant performance implications on applications (both positive and negative).

Frequency of changes. Figure 1(b) shows that the inter-change times have high variance and long tails. For higher activity clusters (C1–C6), we see 93–72% of changes within an hour, a mean inter-change time of 26–224 minutes, and a 95^{th} percentile of 1.2–11.2 hours. For the lower activity clusters (C7–C8), we see 61–55% changes within one hour, a mean of 8–10 hours, and 95^{th} percentile of a few days.

There are long time intervals without any resource changes (e.g., multiple days) as well as short intervals (e.g., 84% within one hour, 40% within 10 minutes, 16% within 1 minute for cluster C2). Ideally, we want to get the most out of long intervals without resource changes while coping with short change intervals. To this end, we analyse how the change intervals are distributed across space (HVMs) and time. This helps understand if there are periods of high activity or if changes are spread spatially. We show this analysis for a high activity cluster (C2). Other clusters exhibit similar trends but differ in the frequency and magnitude of variations.

Spatial diversity. The behavior of Spot and on-demand VMs is determined by the VM configuration and the region. However, HVMs with the same configuration (e.g., minimum size) can behave differently depending on the server they land on (even within the same region). This diversity is directly tied to competing VMs on the server as an HVM shrinks when new competing VMs are allocated and grows when competing VMs are deallocated.

For each HVM, for each 1-hour time window, we measure the harvested cores (time-averaged over the 1 hour) and stability (number of changes), shown in Figure 2. HVMs with a minimum size of 2, 4, and 8 get an average of 15, 17, and 20 cores respectively, which is 2.5–7.5× more resources than the minimum size. At the same time, the top 10% HVMs get a minimum of 36, 38, and 39 total cores while the bottom 10% get at most 3, 3, and 2 additional cores beyond the minimum size. Given that the additional harvested cores have an additional cost, HVMs will also exhibit a wide cost diversity.

All HVM minimum sizes show similar trends in terms of stability. Figure 2(b) shows that larger minimum size HVMs tend to be slightly more stable than smaller ones on average. However, a specific larger minimum size instance is not guaranteed to be stable. The worst 10% HVMs can witness tens of changes in an hour while the relatively stable 50% HVMs witness up to one change in an hour.

Temporal variation. The stability and resources of a particular HVM can also change over time (e.g., a stable HVM can start changing resources frequently). Figure 3 shows an example HVM over a 15 day period (aggregated over 6h windows to gauge longer term stability). For the first ≈ 4 days the HVM is very stable with over 40 harvested cores. However, towards the end, the HVM witnesses large number of resource variations.

2.2 Target Workloads

We focus on long-running and uninterruptible (hard to checkpoint or migrate) workloads. Many applications fall into this category, such as workloads in genomics, oil and gas, weather & financial simulations, geo-spatial workloads, and many scientific computing tasks [15, 16, 34, 47, 67, 68, 81, 89]. The market for these workloads is worth tens of billions of dollars with all major cloud providers pushing towards bringing them to cloud environments [35, 45–47, 59, 67, 68].

These workloads are often run at large scale. Thus, currently they are predominantly run in on-premise clusters that are perceived to be cheaper (compared to regular VMs) and more reliable (compared to Spot VMs, due to evictions). HVMs with their high resource availability and lower eviction rates pave the way for *economically* and *reliably* running these workloads in cloud environments. However, tasks in these workloads tend to be long relative to the typical change intervals of HVMs, hence a single task may see multiple resource variations leading to thrashing/preemptions. These workloads often use domain-specific libraries in containerized environments with large working sets [19, 34], making checkpointing entire containers prohibitive and making tailored checkpoints impractical due to the domain-specific nature of the code coupled with a rich ecosystem where new libraries are continuously added. Further, users of these applications are typically reluctant to modify applications [30, 61].

For concreteness, we study two large-scale production ap-

[3] A VM advertised with 2 cores may be mapped to a fractional amount of SMT cores, e.g., 1.5 or 2.5 SMT cores, depending on the VM's oversubscription or headroom.

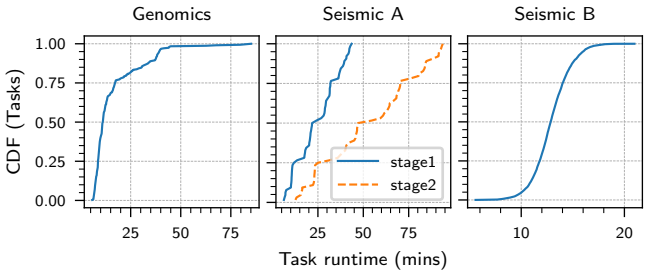


Figure 4: Analysis of task runtimes for two workloads (Genomics analysis and Seismic simulations).

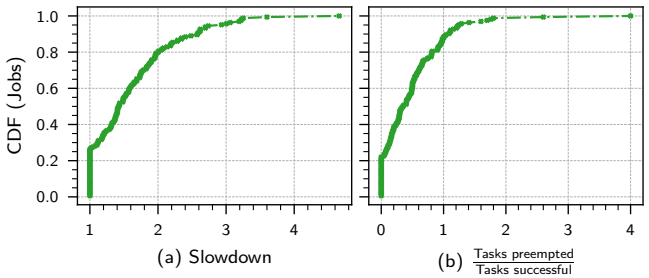


Figure 5: Impact of HVM resource variations on workloads. (a) HVM resource variations slow down jobs. Geometric mean slowdown is 1.5× with some jobs even slowing down by more than 4×. (b) HVM resource shrinks cause the preemption of a third of all launched tasks, i.e., for each successful task, there are 0.5 failed tasks on average. The same task might be preempted multiple times causing preemptions per successful task to grow beyond 1.

plications from a major cloud provider: (1) an application in the oil and gas domain that performs seismic imaging simulations [15, 81, 89], and (2) a genomics application that analyses genetic material to identify various properties (e.g., disease susceptibility for humans and physical traits of plants) [34].

To understand the impact of HVMs on these workloads, we contrast the HVM resource change intervals (representing supply variability) with the task execution times of these workloads (representing demand requirements). Figure 4 shows that tasks in both example applications are long, with a median of tens of minutes, and tails of over 75 minutes. Task distributions for different applications have various shapes (e.g., Uniform and Gaussian for Seismic A & B, and Bounded Pareto for Genomics). Task runtimes range from a few minutes to an hour either within a single job or across different jobs (for Seismic B, not shown). In other words, there is a sizable overlap in the range of the task runtimes and the range of HVM inter-change times. This means that tasks in these workloads are likely to witness a few resource change events during their execution on typical Harvest VMs.

While Gaussian-distributed runtimes are common in typical cloud workloads, Pareto and Uniform distributions show up because of concurrent tasks being heterogeneous. In the genomics applications, some tasks do the actual analysis while others do verification or data transformation. In the seismic simulations, different tasks perform imaging at different resolution or area/volume depending on the analysis requirements.

2.3 Running Workloads on HVMs

The unpredictable and arbitrary resource changes, especially shrink events, can cause tasks to slow down, thrash (for memory harvesting VMs [32]), or get preempted altogether. This leads to execution time overheads and resource wastage since preempted tasks need to be restarted from scratch (under uninterruptible workloads) wasting any previous progress. We measure these overheads comparing HVMs to on-demand VMs running a mix of our target workloads. We use trace-driven simulation for this analysis (methodology in §4).

Figure 5(a) shows the *slowdown* of jobs running on HVMs. We define job slowdown as:

$$\text{Slowdown(Job)} = \frac{\text{ExecutionTime(Job) on HVMs}}{\text{ExecutionTime(Job) on regular VMs}}$$

When running on out-of-the-box HVMs, the geometric mean job slowdown is 1.5× while some jobs are delayed by more than 4×. Such high slowdowns make HVMs impractical for many scenarios. This slowdown is caused by tasks being preempted when the HVM shrinks. For instance a task that needs 4 cores is preempted when the HVM shrinks to 2 cores (or when the HVM evicts, i.e., shrink to size 0). Figure 5(b) shows the distribution of tasks preempted across jobs. Around a third of all tasks fail which leads to an additional resource consumption of ≈ 20%. This directly translates into 20% more cost. SLACKSCHED's goal is to *make the execution of long uninterruptible workloads on HVMs similar to their execution on regular VMs.*

2.4 Opportunities for Improvement

To improve the efficiency of our target workload on HVMs, we build on the following observations:

- There is a large diversity in both task runtimes (within a job or across jobs) and HVM inter-change times and they have significant overlap in their ranges. This provides an opportunity to match long tasks to more stable HVMs and short tasks to unstable HVMs, thus allowing efficient use of HVMs by minimizing preemptions and reducing costs.
- Some HVMs are more stable than others and the stability of a HVM can change over time. There is an opportunity to improve workload execution times by acquiring and retaining a higher fraction of instantaneously stable HVMs.

These two implications motivate our design for two HVM-tailored components: (1) *Scheduler* that matches tasks to HVMs and (2) *Acquirer* that continuously maintains a relatively stable mix of HVMs. Leveraging these insights is not straightforward as the behavior of HVMs depends on multiple unknown factors (e.g., regular VM arrivals and departures).

3 SLACKSCHED Design

SLACKSCHED manages application execution on HVM clusters rented by a cloud user. Many applications running in the cloud run on top of cluster orchestrators like YARN, SLURM,

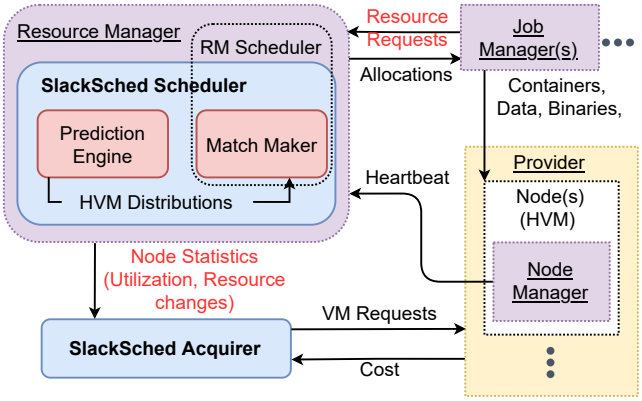


Figure 6: SLACKSCHED architecture with new components in blue and modifications shown in red and red.

Mesos, and others [5, 48, 53, 73], or their managed variants (e.g., Azure Batch [14], AWS Batch [9], GCP Batch [36], AKS [11], AWS EKS [55], and GKE [54]). SLACKSCHED works within such orchestrators, replacing key components to support more efficient use of HVMs. Cloud users can install modified version of orchestrator and/or providers can incorporate SLACKSCHED within managed variants of the orchestrators. Since SLACKSCHED mostly improves execution of individual jobs (§4) (in addition to improving aggregate job execution), providers can use SLACKSCHED to even manage workloads from different cloud users.

Figure 6 shows the architecture of SLACKSCHED and the coordination of three main entities (shown in purple). (1) A per-node *Node Manager*, running on each node in the cluster, that is responsible for reporting the health of the node and running tasks on the node. In the case of HVMs, this is also responsible for conveying instantaneous resource availability on the HVM.[4] (2) A per-job *Job Manager* (or "Application Master" in YARN terminology) that is responsible for managing the progress of the tasks in a single job. (3) A cluster-wide *Resource Manager (RM)* that receives requests for resources from the Job Managers, schedules and maps these requests to nodes, and conveys resource allocations to the job managers. Then, the Job Managers launch containerized programs on nodes based on the allocations. Additionally, we assume that Job Managers annotate their requests with resource requirements and task runtimes. These can be estimated based on input parameters and the type of computation (similar to [25, 50]). The Genomics and Seismic workloads that we consider already have resource requirement annotations. We measure sensitivity to errors in runtime estimates in §4.2.1.

SLACKSCHED consists of two key components (shown in blue): the *Scheduler* (§3.1) and the *Acquirer* (§3.2). The *Scheduler* extends the default orchestrator scheduler and exploits the diversity in task runtimes and HVM resource variations to match tasks to HVMs. The *Acquirer* interacts with the Provider and the Resource Manager to acquire and maintain a set of HVMs that are low cost, harvest more resources, and are stable enough for the workload.

[4]The hypervisor exposes the same number of logical cores to HVMs and only changes their mapping to physical cores at runtime. User programs can query the hypervisor for the core assignment anytime.

3.1 SLACKSCHED Scheduler

Typically, cluster schedulers decide on (1) the order of jobs, (2) the order of tasks within a job, and (3) the placement of tasks onto nodes. SLACKSCHED only affects the third decision and uses the default orchestrator mechanisms for the first two. The SLACKSCHED Scheduler tries to minimize preemptions and improve completion times by intelligently matching tasks to HVMs. Based on the task duration and the stability of HVMs, the Scheduler assigns longer tasks to more stable HVMs (predicted to maintain their resources for a longer time) and shorter tasks to less stable HVMs. Our design splits the operation into: (1) the *Prediction Engine* (§3.1.1) that predicts the stability of each HVM in the future, and (2) the *Match Maker* (§3.1.2) that matches tasks to HVMs based on task duration and HVM stability.

3.1.1 Prediction Engine

Challenges. For match making, we need fine granularity models that predict the resource availability of an individual HVM, e.g., "when will an HVM change its resources?" or "how many resources will it harvest?". This is very challenging since the resource availability of individual HVMs depends on several unknown and uncontrollable factors such as: (1) the arrivals and lifetimes of on-demand VMs, (2) the placement/VM allocation policy of the provider, and (3) the requested configuration of the HVM (e.g., minimum size). Even the cloud provider does not have full future knowledge, especially about when the on-demand VMs will come and go.

Prior efforts at modeling HVM resource availability [3] are not sufficient since they only make predictions at an aggregate level, rather than an individual level. For instance, they provide estimates such as "X% of HVMs will survive in the next hour", or "HVMs will expose on average Y number of cores in a specified time window". In addition, point prediction approaches similar to [25, 43, 78, 88], which use various machine learning models including SVMs, CNNs, and LSTMs [62, 75] to model resource availability, are not a good fit for HVM environments for two main reasons: (1) similar historic resource variations may provide widely different behaviors in the future, which makes point estimates inaccurate; (2) HVMs depict large skews in their behavior (§2.1) which makes the use of computational methods such as machine learning hard.

Key insights. Instead of making point estimations, we take an alternate approach of distribution-based predictions and conditional probabilities. This was inspired by prior work on task scheduling with unknown runtimes [63, 69, 83]. These make probabilistic estimates instead of exact predictions, leveraging the fact that the distribution of task runtimes is known

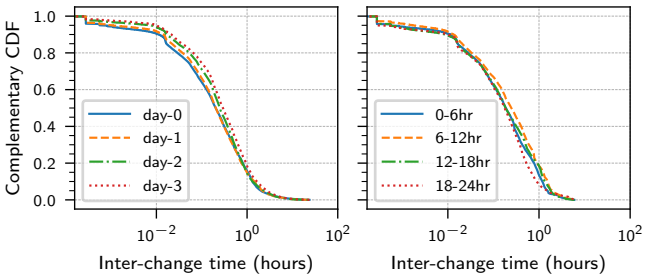


Figure 7: The inter-change-time distribution varies only slightly at different time scales. The legend lists the time period within the trace for which the CDF was computed. The historical inter-change time distribution is a good estimate of the future distribution.

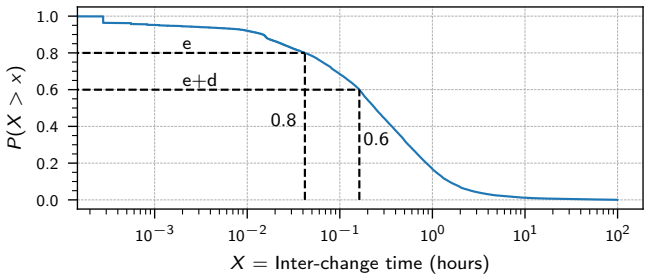


Figure 8: We use the inter-change time distribution for computing the probability that a resource change will occur during a task's lifetime. Say e time has elapsed since the HVM has seen a resource change, and d is the duration of the task. Then, from the complementary CDF, we get: $P(X \leq e+d|X > e) = 1 - P(X > e+d|X > e) = 1 - \frac{P(X>e+d)}{P(X>e)} = 1 - \frac{0.6}{0.8}$

even though the exact runtime is unknown. They proceed by computing the expected remaining runtime by conditioning on the task's progress or age. For instance, given a task has been running for 1 hour, what is the probability that it will run for 30 more minutes.

Similar to prior work, we condition on the inter-change time distribution to estimate the remaining time until the next change for a specific HVM. *We find that the inter-change time distribution is relatively stationary over time. Thus, even though the exact times when resource changes will occur is unknown, the distribution of time between changes is known.* Figure 7 shows the inter-change time distribution at multiple time scales. There are little changes in the distribution over multiple hours/days. As validated in Section 4.5, this approach generalizes to other future sources of variability in unallocated capacity, such as using renewable energy to power a data center, an emerging approach for sustainable cloud computing [12, 76, 77].

Workflow. The *Prediction Engine* maintains a rolling snapshot of the inter-change time distribution in the past D days ($D = 1$ in our implementation) for estimating the *completion probability*, i.e., the probability that a task will complete successfully on an HVM without getting preempted. We use completion probability as a proxy for HVM stability as it allows us to rank if an HVM is stable enough for a task.

We approximate the completion probability for a task and HVM as the probability that the HVM will not shrink during the lifetime of the given task. In reality, tasks can complete successfully despite witnessing resource shrinks, e.g., if the node is underutilized and has enough resources even after shrinking. However, since we only use the completion probability to obtain a relative ranking of nodes, its absolute value is of little consequence. This approximation also allows robustness to inaccuracies in task runtime estimates. Future work can consider predicting shrink size for more accurate estimation of completion probability and potentially better job execution on HVMs.

We compute completion probability in two steps. First, we estimate the likelihood that there will be a resource change event during the lifetime of a task on a particular HVM. Second, we estimate the likelihood that this resource change event will be a shrink event. Finally, the Match Maker combines these likelihoods (§3.1.2). Note, there are other ways to compute completion probability (Appendix B.2).

Likelihood of resource change. To estimate the probability of a change during the lifetime of a task, we condition on the inter-change time distribution (Figure 8). Given that e time has elapsed since the last resource change event, we compute the probability that the next change event will occur within d time from now, where d is the duration of the task.

Likelihood of shrink. To compute the probability for the next event to be a shrink, we perform an n-gram (bigram) analysis [17] on the sequence of resource changes. Specifically, we look at a historical sequence of resource changes and calculate how often a shrink occurs after a growth and how often a shrink occurs after another shrink. We use this to compute the probability that the next event will be a shrink given that the last one was a growth (or shrink) event. *We find that such shrink probabilities are also relatively stationary over time.*

3.1.2 Match Maker

To minimize task preemption likelihood, the *Match Maker* places tasks on HVMs with a high completion probability for the task. When no HVMs with available resources yield a high completion probability, the *Match Maker* may wait for occupied HVMs to free up resources ("delayed scheduling"). We now describe how we calculate the completion probability, perform delay scheduling, and our matching logic.

Completion probability. The *Prediction Engine* maintains the inter-change-time and shrinking probability distributions (§3.1.1). The *Match Maker* uses these distributions to compute the completion probability as follows:

$$
\begin{aligned}
P_c(x,e,d) &= 1 - P\left[\begin{matrix}\text{shrink occurs during}\\ \text{task lifetime}\end{matrix}\right] \\
&= 1 - P\left[\begin{matrix}\text{resource change}\\ \text{is shrink}\end{matrix} \wedge \begin{matrix}\text{change occurs}\\ \text{during task lifetime}\end{matrix}\right] \\
&= 1 - P_s(x) \cdot (1 - P(X > e+d|X > e)) \qquad (1)
\end{aligned}
$$

Table 1 shows the definition for each symbol. Recall that tasks are annotated with their estimated runtimes (§3).

Delayed scheduling. In certain situations, waiting for a more stable node (i.e., with better completion probability) might be a better choice than picking from the currently available resources. For example, rather than launching a task on a node with low completion probability, it might be better to wait for old tasks to complete on a node that is relatively stable but may be fully occupied with previous tasks. This implies a non–work–conserving schedule where the scheduling of tasks may be delayed even when resources are available.

It is challenging to decide whether to wait or run with current the best resource without future knowledge. A strawman approach is to launch tasks on an HVM only when it provides a threshold completion probability. However, this may cause the scheduler to delay indefinitely. To tackle this, we formulate the cost and benefit of waiting by estimating the expected completion time obtained from various decision choices. We derive expected completion time as follows:

$$
\begin{aligned}
&E_{wc}(x,a,e,d)\\
&= E\begin{bmatrix}\text{time spent waiting}\\ \text{for node to free up}\end{bmatrix} + E\begin{bmatrix}\text{time for completion}\\ \text{and preemptions}\end{bmatrix}\\
&= a + p\cdot d\\
&\qquad + (1-p)\cdot\left(E\begin{bmatrix}\text{time wasted}\\ \text{to preemption}\end{bmatrix} + E\begin{bmatrix}\text{time after}\\ \text{restart}\end{bmatrix}\right)\\
&= a + p\cdot d + (1-p)\cdot(w + E_{wc}(g,0,0,d)) \qquad (2)\\
&e' = e + a\\
&p = P_c(x,e',d)\\
&w = E\left(X - e' \mid (X > e') \wedge (X < e' + d)\right)
\end{aligned}
$$

This equation incorporates when will the node free up resources (a), the duration of the task (d) if it completes succesfully (with probability p), time potentially wasted (w) in case the task fails (with probability $1-p$), and the cost of rescheduling the task ($E_{wc}(g,0,0,d)$). Note that this is the worst-case expected completion time since we reschedule a task on the worst-case node that has just started (has not remained stable at all, i.e., $e = 0$). In reality, after preemption, the task may be started on a stable node. Since we know the estimated duration of tasks, their resource requirements, and how long they have been running, we can deterministically compute when the node will have enough resources to run a given task (a). The time wasted (w) is the expected time between when the task starts and when the node shrinks, given that the node does shrink during the lifetime of the task.

Workflow. The *Match Maker* uses the above formulation (Equation 2) to schedule the task on the node which gives the best expected completion time. If this node does not have resources at the moment, then it waits and reconsiders the decision at the next scheduling iteration. This automatically includes both preferring nodes with higher completion probability and considering to wait for nodes to free up. For completeness, we list the pseudo-code for the *Match Maker* in Algorithm 1 in the Appendix.

Symbol	Interpretation
P_c	Probability of completion
X	Inter-change time distribution
$x \in \{g,s\}$	Denotes whether last change event was growth or shrink. Default is growth for a newly started node
e	Time elapsed since last resource change event
d	Remaining duration of the task under consideration
P_s	Probability that the next resource change event is a shrink
E_{wc}	Expected worst case completion time of the task
a	Time between now and when the node will have enough resources to run the task under consideration. If the node currently has enough resources, then $a = 0$
w	Expected time wasted due to a shrink occurring before task completion

Table 1: Symbol definitions.

3.2 SLACKSCHED Acquirer

Challenges. Ideally, we want to maintain a set of HVMs whose stability matches the runtime of the tasks. However, in addition to task runtime information, this requires full knowledge on: (1) how unallocated resources are distributed across servers in a data center, and (2) how stable these resources are at any point in time. As a user (or cluster orchestrator), this information is not available or possible to model. Even as the provider, these metrics are only instantaneously available while HVM patterns may change over time (Figure 3).

Approach. We use a simple "exploration-exploitation" strategy to navigate the set of potential HVMs that the provider can offer to maintain a stable pool of HVMs. The Acquirer starts with a random mix of HVMs and periodically identifies the *worst* HVMs and decommissions them, gradually converging to a more stable pool of HVMs. The Acquirer defines "worst" as the most recently changed HVMs. This simple strategy works when there the HVM pool is unstable relative to the HVMs that can be returned by the provider (4.3).

In our implementation, the Acquirer runs hourly and deallocates 10% of allocated HVMs. Concurrently, it requests an equal number of new HVMs from the provider to maintain the same amount of cluster resources.[5] To avoid getting a HVM on the same server as the one just deallocated, the Acquirer first requests new HVMs and then deallocates the unwanted ones. To avoid task preemptions, it *gracefully decommissions* HVMs before deallocating them. The scheduler stops sending new tasks to the decommissioning HVMs and once all running tasks complete, the Acquirer deallocates them.

To maintain a target set of resources in the midst of load variations, the Acquirer works with a scaling policy that maintains the cluster utilization between a lower and upper bound (i.e., 60 to 80%). It requests or deallocates VMs whenever the utilization falls outside of this target range. When scaling-in, we decommission and deallocate the worst HVM first. The

[5] Currently, we consider mixes with only HVMs. One can consider a mix of both HVMs and regular VMs.

Acquirer can work with different scaling policies [8, 10], and the scaling can be decoupled from the intentional deallocations/allocations. In our implementation, scaling is triggered every hour coupled with the Acquirer. Scaling also allows maintaining resources as HVMs grow/shrink.[6] Algorithm 2 in the Appendix shows the pseudo-code for the Acquirer.

We studied different choices for the Acquirer's trigger period. We found that trigger periods in half to two hours range yield similar performance as 1 hour. Longer periods can lead to a stale cluster (with unstable HVMs) and shorter periods can operate on limited historical information (especially for HVMs allocated in the previous trigger) leading to erroneously deallocating potentially stable HVMs. In general, the trigger period should be chosen based on the range of HVM inter-change times and task runtimes.

3.3 Implementation

We implement SLACKSCHED within YARN [5] version 3.3.0. This includes the changes by [3] that make YARN aware of HVM resource variations. We added an Acquirer module that interfaces with the provider's VM request API and monitors statistics about allocated nodes. We updated Job Managers to convey task runtime estimates as part of Resource Requests and updated the matching logic to use our design (§3.1). These changes (highlighted in Figure 6) preserve compatibility with other scheduling features, e.g., reservations [27], delay scheduling [86], affinity [4], etc. Specifically for multi-dimensional packing [38], the affinity between tasks and nodes can be defined as a combination of resource match and stability match. We believe that our implementation can be easily ported to other cluster managers.

4 Evaluation

We evaluate SLACKSCHED to answer the following questions:

1. How much benefit do we gain solely from scheduling under different HVM resource profiles? (§4.2)
2. How robust is our design to various workload characteristics, operating conditions and estimation errors? (§4.2.1)
3. How much do we benefit from resource acquisition? (§4.3)
4. How does SLACKSCHED handle time varying arrival rate and compare with other VM types? (§4.4)
5. How does SLACKSCHED compare to prior techniques for addressing VM evictions? (Appendix B.1 and §5).
6. What happens under future and more extreme sources of resource variability (e.g., as a result of using renewable energy)? (§4.5)

4.1 Methodology

Setup. Since HVMs are highly variable and our method is probabilistic, to reach any conclusive results, we needed to run each experiment at a large scale: ≈ 50 jobs, where each job lasts hours and requires 100s of regular VMs. However, running such long and large-scale experiments was impractical on a real testbed. Thus, we evaluated our system using Hadoop's discrete time simulator [1, 6]. This simulator accurately mimics real-world setups, with only $\approx 1.3\%$ error on completion times and $\approx 1.5\%$ error on resource utilization [24]. The simulator runs actual YARN Resource Manager [5] code and only simulates the Node Managers, Job Managers, and clock and communication layers. Using a simulator also allows us to maintain the exact same trace of HVM availability, ensuring fair A/B testing across experiments.

Resource traces. We use two sets of production resource traces from Microsoft Azure: (1) *HVM traces:* time series of HVM resource availability that includes the time and sizes of growth/shrink events for each HVM; (2) *On-demand VM traces:* VM arrival times, lifetimes, and placement decisions made by the provider's production allocator. Our dataset includes traces from 8 clusters ($700-2000$ servers each) across 5 regions from two time periods (March 2019 and August 2021). For our experiments, we randomly select 64 HVMs for each cluster. This translates to 160–480 regular VMs in terms of resources as each HVM gets 2.5–7.5$\times$ more resources than their minimum size (§2.1).

Extensions to the simulator. To ensure different scheduling schemes witness the same HVM changes, we extend the simulator to replay HVM traces. When an HVM shrinks, containers are killed (in increasing order of their start time) until the available resources on the node exceed or equal the used resources on the node. When an HVM grows, new containers may be allocated to the node. These are the default Resource Manager behaviors. In other words, a task does not benefit from cores beyond what it asked for and is killed as soon as it gets fewer than its requested cores.

In addition, for resource acquisition experiments (that continually request new HVMs), we want every experiment to receive the same HVMs when requesting the same configuration at the same time. To ensure this, we replay the on-demand VM allocation traces to reconstruct the state of the unallocated resources. Alongside, we add a simulated HVM allocator to place HVMs on servers with unallocated resources. We study different allocation policies including random, load-balancing, and packing. Note that we use the simulated HVM allocator only for the resource acquisition experiments (§4.3, §4.4).

Workloads. Our traces are based on a collection of two workload categories running in a production environment: Seismic and Genomics [16, 22, 34, 58, 59]. We log their execution to build distributions of task runtimes, number of tasks, and resource requirements. We sample from these distributions to build traces of jobs/tasks. We model job arrival as a Poisson process (similar to [38–40, 56]) and study a variety of mean inter-arrival times.

Schemes. For scheduling, we compare SLACKSCHED (4) against the three schemes (1-3 below):

1. *CapacityScheduler* (or *CapS*): The default scheduler in

[6] Due to growth/shrinks/evictions, instantaneous cluster utilization can fall outside our target range in the time between two scaling triggers.

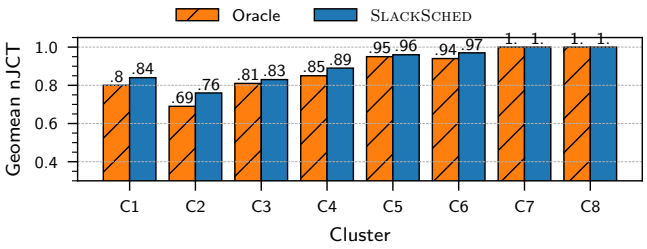


Figure 9: Normalized job completion time (nJCT) relative to *CapacityScheduler* across HVM traces from different clusters. SLACKSCHED improves JCT across different clusters.

YARN. Considers resource requirements (cores and memory) of tasks to select nodes [7].

2. *Oracle*: Uses future knowledge to schedule a task on an HVM only if it is guaranteed to complete before any future resource shrinks, and skips allocation otherwise. Skipping implicitly delegates scheduling to other nodes or later times. This provides an upper-bound for SLACKSCHED.
3. SLACKSCHEDNODELAY (or SSNODELAY): Only uses current resources to make decisions and does not wait for nodes to free up resources. It places tasks on nodes with the highest completion probability.
4. SLACKSCHED (or SLACKS): Uses expected completion time to match tasks to nodes and can wait for nodes to free up resources (§3).

Oracle is a good proxy for job completion time on a regular VM cluster that has same total resources as the HVM cluster. This is because *Oracle* does not cause any task preemptions and the clusters are sized to ensure little to no task queuing.

Metrics. We study job completion time (JCT), normalized job completion time (nJCT), and dollar cost, where,

$$\text{nJCT(Job)} = \frac{\text{JCT(Job) with scheme}}{\text{JCT(Job) with baseline}}$$

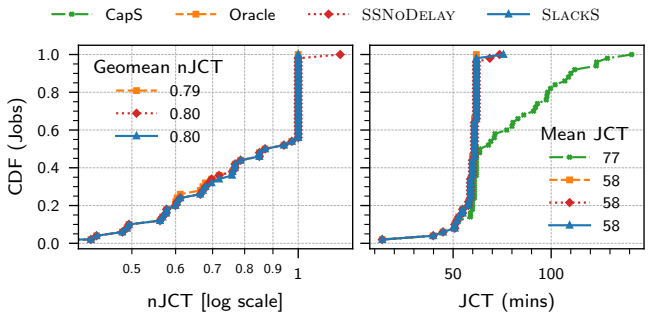
JCT distributions show absolute and tail completion times, while the normalization allows us to study the impact on jobs with different runtimes. A smaller nJCT i.e., nJCT $\in (0,1)$ is better, while nJCT $\in (1,\infty)$ is worse. nJCT of 0.7 translates to $(1-0.7)*100 = 30\%$ reduction in JCT (cf. [40]), and $1/0.7 = 1.43\times$ factor of improvement or speedup (cf. [39]). For computing nJCT, we use baseline as *CapacityScheduler* unless mentioned otherwise.

For aggregating across jobs, we study (a) mean reduction in JCT $(1 - \mathit{GeometricMean}(\mathit{nJCT}))$, (b) reduction in mean JCT $(1 - \frac{\text{mean JCT with scheme}}{\text{mean JCT with baseline}})$, (c) reduction in tail (90^{th} percentile or p90) JCT.

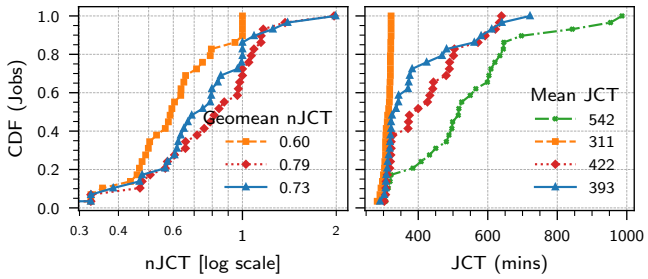
We delegate a subset of these metrics to Appendix B.5.

4.2 Scheduler Evaluation

We evaluate SLACKSCHED on the production clusters from our dataset under the same workload trace. Figure 9 shows the geomean nJCT relative to *CapacityScheduler*. Mean reduction from SLACKSCHED ranges from 0 to 24% (i.e., geomean nJCT from 1 to 0.76). Improvements from SLACKSCHED are close (within 13%) to the *Oracle*.



(a) High-activity cluster (C2, frequent resource changes) with shorter tasks.



(b) Low-activity cluster (C8, infrequent resource changes) with longer tasks.

Figure 10: SLACKSCHED's mean reduction in JCT is 20–27%. Waiting for more stable nodes provides a further 0–7.5% reduction. SLACKSCHED reduces mean JCT by 25–27% and p90 JCT by 20–44%. We use Seismic A workload (§2.2) for this experiment which has uniformly distributed task runtimes.

The frequency of resource changes relative to the task durations govern how much overhead HVMs impose and in turn govern how much benefit we can get from SLACKSCHED. We observe that benefits are greater in clusters that witness more activity from regular VMs (i.e., more HVM resource changes). Specifically, for the last two clusters (C7 and C8) which have the least amount of activity from regular VMs (§2.1), HVMs cause little slowdown for the particular workload.

We zoom into the JCT and nJCT distributions for a high-activity cluster (C2). We also study a low-activity cluster (C8) with longer tasks. These represent cases when HVMs impose non-trivial overhead. Figure 10 shows that SLACKSCHED's mean reduction in JCT is 20–27%. 40% of the jobs see more than 30% reduction in their completion times. 0–15% jobs have nJCT > 1, implying their JCT increases. Since our method is probabilistic, SLACKSCHED can make poor decisions for some individual tasks compared to a random matching decision (taken by *CapacityScheduler*). However, *on average*, SLACKSCHED's matching decisions are better than random. SLACKSCHED reduces mean JCT by 25–27% and p90 JCT by 20–44%. In §4.2.1, we vary various aspects of the workload.

Impact of "waiting" for stable nodes. Figure 10 also quantifies the impact of waiting for stable, but currently busy nodes (§3.1.2). As shown, SLACKSCHED provides an additional

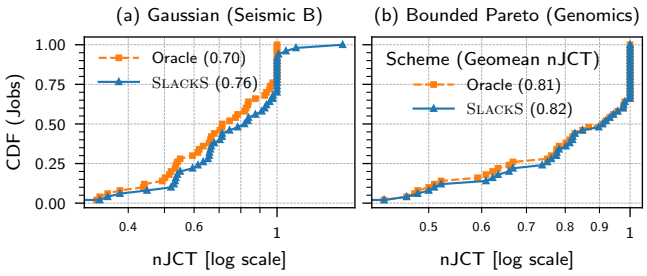


Figure 11: nJCT for workloads with different task runtime distributions. SLACKSCHED consistently improves JCT.

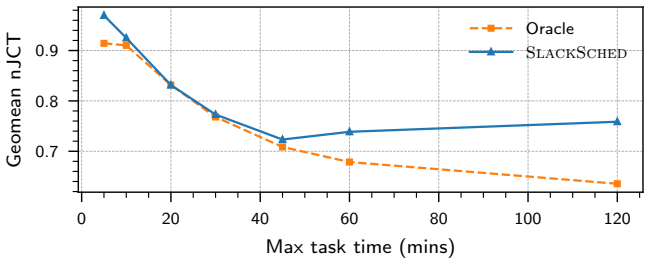


Figure 12: nJCT with varying task runtimes. SLACKSCHED is effective for a wide range of task runtimes.

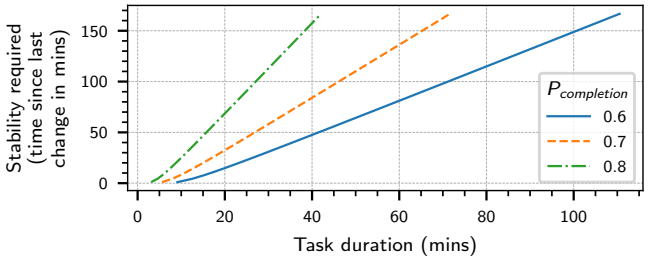


Figure 13: HVM stability required for different task durations at different confidence levels. To obtain 70% confidence in completion without preemption for a task with duration d, we roughly need a node to have been stable for $2.6 \times d$ time. We compute "stability required" using inverse CDF of inter-change-time distribution.

0–6% mean reduction in JCT compared to when this feature is disabled (SLACKSCHEDNODELAY). Specifically, for the high-activity cluster (C2), SLACKSCHEDNODELAY is already close to *Oracle* and there is little scope of further improvement from waiting.

Resource waste reduction. We compare the fraction of resource waste (i.e., work wasted due to preemptions divided by total work done). We find that *CapacityScheduler*, SLACKSCHED, and *Oracle* waste 20–40%, 3–20%, and 0% work respectively, i.e., SLACKSCHED reduces resource waste by $\approx$ 20% compared to *CapacityScheduler*. This saving comes from the better task-to-HVM matching that avoids preemptions and reduces wasted work or resources.

4.2.1 Improvement Conditions and Robustness

SLACKSCHED's gains depend on the range and distribution of task runtimes relative to the inter-change times of HVMs. We find that SLACKSCHED provides benefits in JCT when:

- There is spatial variation in stability of HVMs and spatial variation in runtime of concurrently running tasks. Otherwise, different mappings from tasks to nodes are equivalent.
- The range of HVM inter-change times is similar to the range of task runtimes. If tasks are too short, there is not much room for improvement as there are few preemptions. If tasks are too long, preemptions cannot be avoided.

Both conditions hold for a large set of workloads and HVM resource profiles (regions/clusters) (§2). We now describe our robustness experiments that led to these findings.

Task runtime distribution. We change the task runtime distribution in accordance with different workloads (§2.2) under the same HVM resource variations (high activity cluster, C2). This is shown in Figures 10a (Uniform), 11a (Gaussian), and Figure 11b (Bounded Pareto). Workloads with higher variance (Uniform and Pareto) see more improvements from SLACKSCHED since they benefit more from matching of tasks to resource variability.

Task duration. We analyzed the impact of varying the range of task runtimes. We use uniformly distributed task runtimes between 1 and X minutes (max task time) and vary X. Figure 12 shows that the improvements in both SLACKSCHED and *Oracle* diminish with short tasks (< 20 min) as they are rarely preempted (no scope for improvement).

SLACKSCHED's improvements also diminish when tasks become too long as it becomes harder to find stable nodes. This is because SLACKSCHED relies on past history to predict the future, e.g., to schedule a task of duration d with 70% confidence, it needs a node which has remained stable for roughly $2.6d$ time (Figure 13). On the other hand, *Oracle* can identify future stable nodes even if they have been unstable in the past, allowing it to find matching nodes for long tasks.

Cluster load. We increase the load by reducing the mean inter-arrival time of jobs while keeping the same set of HVMs, task runtime distributions, and task resource requirements. Figure 14 shows that when load becomes very high ($\approx$ 100% at 3 mins mean inter-arrival, as shown in the first data point), the benefits of SLACKSCHED diminish, since the relative number of stable nodes decrease and SLACKSCHED has a harder time finding nodes for longer tasks (same reason as shown before with Figure 13). *Oracle* still provides improvement since it has full future knowledge and can still find stable nodes.

Task runtime misestimates. SLACKSCHED uses estimates of task runtimes to compute completion probabilities and expected completion times. We evaluate SLACKSCHED's sensitivity to misestimates by injecting errors into the runtimes reported while still running tasks with their original runtimes. To inject errors, for each task, we deviate its duration estimate by a number sampled uniformly between 0 and some max percentage error. Negative error implies underestimates and positive error implies overestimates. Figure 15 shows there is little impact of misestimates on SLACKSCHED. Completion times inflate when runtimes are underestimated. This is be-

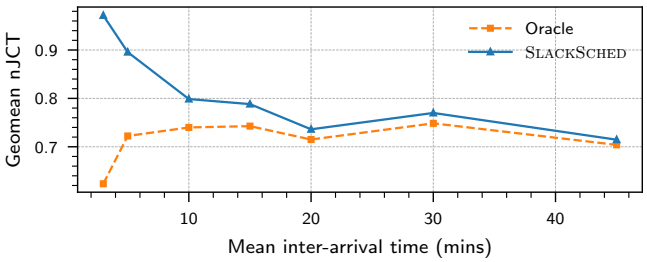


Figure 14: nJCT with varying job inter-arrival rates. SLACKSCHED improves JCT at different cluster loads.

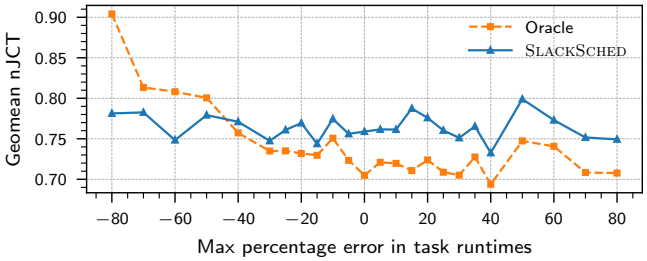


Figure 15: nJCT with varing errors in task runtimes estimates. SLACKSCHED improves JCT even under modest errors.

cause runtime underestimation results in overestimating the completion probabilities and matching tasks to nodes which may not remain stable through the lifetime of the task. We see more impact on *Oracle*, as it makes many close calls which are rendered incorrect due to inaccurate duration estimates.

4.3 Acquirer Evaluation

We study benefits from SLACKSCHED-Acquirer considering settings where there is room for improvement in job completion time between *Oracle* and SLACKSCHED-Scheduler.

Methodology. Different experiments differ in their scheduling and acquisition logic. In all cases, HVMs are continuously (minimum size chosen uniformly randomly) requested to maintain ≈ 64 HVMs worth of resources. For these experiments, we do not use the scaling policy (§3.2) and consider a constant job arrival rate. Our baseline uses the *CapacityScheduler* with no intentional HVM deallocations.

These experiments require a simulated HVM allocator and we test our system with three policies to decide which server to place an HVM on: (1) Balancing, picks the server with the most amount of unallocated resources, (2) Packing, picks the server with the least amount of unallocated resources, (3) Random, picks a random server. All policies only consider servers which have enough resources to support the minimum size of the HVM at the time of allocation.

Results. Figure 16 shows qualitatively similar improvements across different HVM allocation policies solely from the SLACKSCHED-Scheduler (16–24% mean reduction). We also obtain 8–23% mean reduction solely from the SLACKSCHED-Acquirer. The Scheduler and the Acquirer complement each other to provide a mean reduction of 27–32%.

Improvement conditions. In addition to the improvement

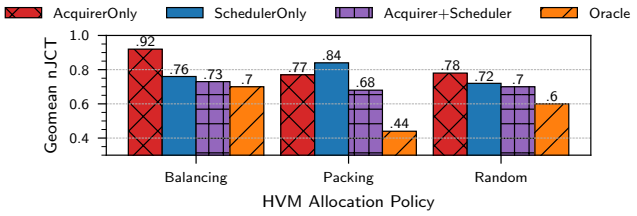


Figure 16: nJCT relative to baseline. SLACKSCHED's scheduling and resource acquisition complement each other to reduce JCT.

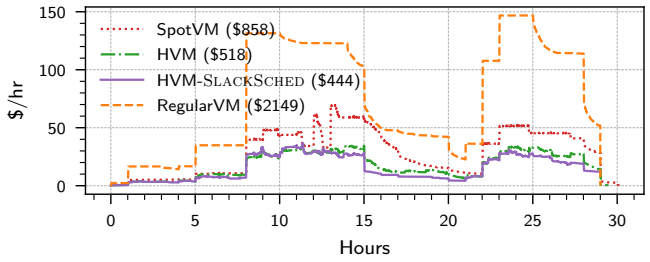


Figure 17: Cluster size over time in terms of cost ($) per hour. Acquisition logic adapts to time-varying job arrival rate. HVMs are cheaper than Spot and Regular VMs due to the discounted harvested cores. Legend lists the total cost ($) in parentheses.

conditions in §4.2.1, resource acquisition is useful when there is room for improving the stability of HVMs in the cluster. This happens if (1) there are not enough stable HVMs in the cluster and (2) there are better HVMs that can be returned by the allocation policy. For instance, when most HVMs are allocated on *buffer servers*, there is little scope to improve the HVM mix. Providers typically reserve buffer servers to satisfy sudden demand for regular VMs. In the absence of sudden demand changes for regular VMs, HVMs on buffer servers witness few resource variations and cause little to no job slowdown. In our implementation, we disallow the Balancing policy from placing HVMs on buffer servers (≈ 5% of all servers). This is because, in practice, multiple users will request HVMs from the cloud provider and a single user will only get a small portion of HVMs allocated on buffer servers. In general, we expect individual users to see HVMs that behave closer to those allocated by the Random policy.

4.4 Scaling and Cost Comparison

We study how SLACKSCHED adapts to a workload that varies over time. For such a workload, we compare the performance (JCT) and cost of maintaining homogeneous pools of HVMs, Spot VMs, and regular (on-demand) VMs.

Methodology. To generate the time-varying workload, we vary the mean job arrival rate between 1× and 4× the base rate every 6 hours. For the environment, the Acquirer maintains a homogeneous cluster of Spot VMs, HVMs or regular VMs in separate runs. Runs without SLACKSCHED (i.e., SpotVM, HVM, RegularVM in Figures 17 and 18) use *CapacityScheduler*. For these cases, the Acquirer does not intentionally deallocate servers and only uses the scaling policy to

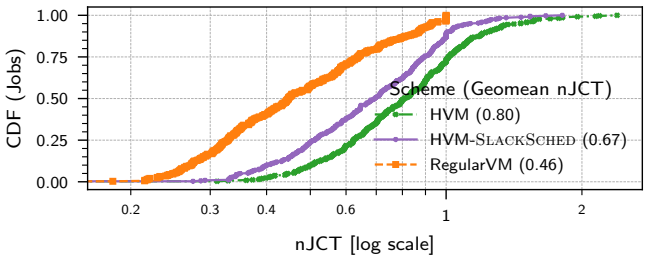


Figure 18: nJCT relative to Spot VM. SLACKSCHED improves JCT over out-of-the-box HVMs under a time-varying job arrival rate. Vanilla HVMs yield better JCT than Spot VMs.

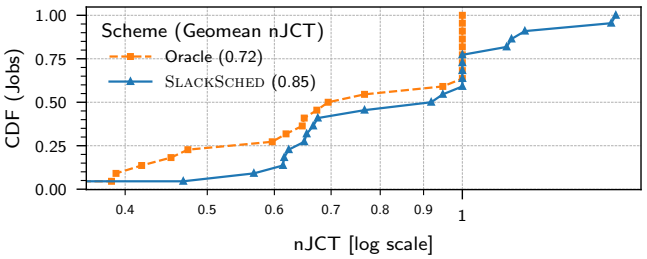


Figure 19: SLACKSCHED improves completion times even under power variations due to renewable energy sources.

tune the cluster size in response to changing job arrival rate, randomly choosing which VMs to deallocate.

We use the Random allocation policy (from §4.3) to determine the placement of HVMs and Spot VMs. Spot VMs are fully evicted if resources are needed for competing regular VMs allocations (i.e., they do not shrink like HVMs). We obtain the cost of regular and Spot VMs from [74]. For HVMs, we follow the same pricing scheme as [3, 87] and charge their minimum size at the same price as Spot VMs of the same size and charge additional harvested cores at a 50% discount.[7]

Scaling results. Figure 17 shows that the SLACKSCHED-Acquirer scales the cluster in and out in response to the changing job arrival rate for all VM types. It maintains a relatively constant core utilization over time (not shown). Across schemes, the Acquirer maintains similar core utilization and amount of resources. Different schemes have different number of preemptions and resource waste.

Cost results. Figure 17 also shows that HVMs are 40% (1.67×) and 75% (4×) cheaper than Spot and regular VMs respectively. The ≈ 14.2% difference in cost with and without SLACKSCHED ($444 vs. $518) mainly comes from (1) less resource waste, and (2) intentional deallocations that move utilization closer to the upper bound (80% threshold §3.2).

JCT results. Figure 18 shows that HVMs without SLACKSCHED already provide mean reduction of 20% compared to Spot VMs as HVMs shrink instead of getting evicted. The Scheduler+Acquirer in SLACKSCHED provide a mean reduction of 33% over Spot VMs even under a time-varying arrival rate and without incurring extra cost.

4.5 Case Study: Renewable Energy Sources

To further test the generality of our approach, we consider a future source of resource variability: renewable energy sources (e.g., wind and solar power). Power generated from renewable energy sources typically fluctuates over time as these sources are driven by weather conditions that are in turn variable over time. We study the case of HVMs that vary in resources due to a changing power supply in addition to on-demand VM arrivals/departures. In these scenarios, leveraging an unreliable resource supply is a necessity rather than an optimization.

Methodology. We use the same simulation environment and workloads as before. For generating HVM traces under renewable energy variations, we use wind power generation traces from the ELIA dataset [66]. We scale the power trace such that the cluster is fully powered at the max power in the trace. For simplicity, we also assume that cores powered on a server are proportional to the power supplied to the server. We replay the on-demand VM allocation traces and the power generation traces. Whenever power drops, we reduce the power supply to servers that have unallocated resources, effectively reducing the size of the HVMs. When no such servers exist, we evict on-demand VMs (assuming they are migrated out). When power rises, we increase the power supply to servers that have a lower supply than the max power, effectively increasing the size of the HVMs. Additionally, we relaunch previously-evicted on-demand VMs to maintain the cluster load (assuming they are migrated in).

Results. We observe that the inter-change time and shrink distributions are still relatively stationary when taken for a day. However there are more variations at smaller time scales (e.g. when taken for a window of 6 hours). Since SLACKSCHED maintains a large enough snapshot (i.e., 1 day) of the HVM distributions, it still prevents preemptions and improves completion times. Figure 19 shows that SLACKSCHED provides a mean reduction in JCT of 15%. Similar to Figure 10, ≈ 25% jobs degrade (nJCT > 1) relative to their execution with *CapacityScheduler*.

5 Related Work

Related work not covered in §3.1 can be classified into:

Checkpointing, migration, and replication. Work like [71, 80] use a combination of these techniques to mitigate the impact of preemptions caused by Spot VM evictions. These are prohibitive for long uninterruptible workloads. In Appendix B.1, we empirically show that SLACKSCHED outperforms these techniques for our target workloads by profiling their checkpointing overheads.

[7] We are only interested in price variations across space but not across time. Temporal variations in price would affect all HVMs (at least within the same region) symmetrically and would not significantly affect our acquisition decisions. Thus, we pick costs from [74] at a single point in time. In reality, Spot VM prices, and in turn HVM prices, are volatile over time and can range between 48% and 88% of the regular VM price.

Multiple markets. [20, 43, 70, 71, 80, 84] mitigate the impact of VM evictions by picking Spot VMs from different VM types/sizes and regions (i.e., markets). Different markets can have different prices and eviction rates. These efforts estimate Spot VM eviction likelihoods based on spot prices and bids. However, pricing-based prediction techniques may not necessarily predict resource changes (e.g., HVMs shrinking) which cause workload preemptions. Further, SLACKSCHED yields improvements even for a cluster of HVMs taken from a single market (i.e., the same minimum size and region) as shown in Figure 10b. For such setting, we expect multi-market techniques will perform similar to the *CapacityScheduler* as they will not distinguish between VMs taken from a single market.

[44, 70, 80, 84] use an ensemble of on-demand and Spot VMs. SLACKSCHED provides improvements without requiring on-demand VMs that cost 2–10× more than HVMs.

Bidding, pricing, and admission control. [57, 85] use bidding strategies to control VM eviction likelihoods. These may not control HVM resource variations which can preempt workloads. Further, pricing-based techniques only work when evictions and pricing are related. These methods do not generalize to flat pricing models [51, 74]. Many of these efforts are also scoped to different workloads (e.g., machine learning training [44], database queries [84]) and do not generalize to long uninterruptible workloads.

In a setting where jobs have different levels of importance, admission control [65] techniques may be useful. This is orthogonal to SLACKSCHED.

Scheduling and task placement. Most prior work assumes fixed resources over time [33, 37–40, 52, 64, 79, 86]. They leverage multi-dimensional packing, locality, fairness, and workload properties (e.g., dependencies). These ideas are orthogonal to our work and SLACKSCHED can leverage them to further improve scheduling objectives including efficiency, fairness, and completion times. However, HVMs incur large resource variations which are not addressed by this prior work.

Other proposals [3, 56, 87] adapt cluster scheduling frameworks to address resource variability. However, they are scoped to specific workloads that are less challenging than long-running uninterruptible workloads. [56] only considers elastic query processing workloads, [87] considers serverless functions with short tasks, and [3] just reacts to resource variations rather than avoiding preemptions. The closest to our work is SciSpot [51] that schedules tasks using a time-to-eviction distribution for Spot VMs. It does not consider waiting for VMs to free up resources. Waiting (delayed scheduling) provides better performance (§4.2). SciSpot also does not provide any empirical evaluation and only estimates potential for improvement using theoretical analysis. It does not consider resource acquisition and only works with bathtub-shaped time-to-eviction distributions.

Addressing underutilization. Prior work has also looked at underutilization in cloud environments [78, 88]. These try to co-locate latency critical services and batch workloads to reduce resource fragmentation and improve cluster utilization. These techniques often also leverage the fact that jobs do not use their peak resources all the time and thus oversubscribe resources. However, such oversubscription is typically only done for first party workloads and not customer facing services [3, 42, 72, 88]. Hence, providers still deal with unallocated resources [3, 88].

Serverless computing or FaaS (Function-as-a-Service) give up on the VM abstraction and allow providers more flexibility to dynamically spread computation and reduce resource fragmentation. HVMs try to preserve the VM abstraction allowing use of harvested capacity for workloads that are not suited for serverless computing, e.g., workloads that maintain state, need the abstraction of a machine, shared libraries, or operating system, or require significant software engineering effort for porting to FaaS abstractions.

While HVMs expose unallocated resources, SmartHarvest [82] also opportunistically exposes allocated but unused resources. It uses machine-learning techniques to decide when and how many resources can be harvested without harm. SmartHarvest resource variations are fine-grained (millisecond level) and would require additional scheduling solutions.

6 Conclusion

Cloud providers have started using new mechanisms like Spot VMs and Harvest VMs (HVMs) to monetize their unallocated resources. After a characterization of HVMs and workloads, we identified that prior work falls short at running long, uninterruptible workloads in such variable environments. To enable these workloads, we propose SLACKSCHED, which leverages distribution-based predictions to maintain a stable pool of HVMs and intelligently match tasks to their ideal resources. SLACKSCHED successfully enables running workloads on HVMs as if they were run on regular VMs.

We experimentally demonstrated that our proposal reduces resource waste by 20% and improves mean and tail (90^{th} percentile) completion time by 27% and 44% respectively, at 75% lower cost than regular VMs. We also show that our system generalizes to cases where resource variations are caused by a variable power supply. We plan to contribute our code to the Apache YARN project [5].

Acknowledgments

We would like to thank anonymous reviewers, our shepherd John Wilkes, and Srikanth Kandula for feedback that helped us improve this work. We would also like to thank Peeyush Kumar for useful discussions, Philipp Witte and Roberto Lleras for help with workloads, and Shivkumar Kalyanaraman and Srinivasan Iyengar for help with the renewable energy case study.

References

[1] *[YARN-1187] Add discrete event-based simulation to yarn scheduler simulator - ASF JIRA*. [Online; accessed 17. Mar. 2021]. 2021. URL: https://issues.apache.org/jira/browse/YARN-1187.

[2] *Amazon EC2 Spot – Save up-to 90% on On-Demand Prices*. [Online; accessed 12. Sep. 2021]. URL: https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc.

[3] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *OSDI*. 2020.

[4] *Apache Hadoop 3.3.1 – YARN Node Labels*. [Online; accessed 13. Sep. 2021]. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/NodeLabel.html.

[5] Apache Software Foundation. *Apache Hadoop YARN*.

[6] Apache Software Foundation. *Hadoop*. Version 0.20.2. Feb. 19, 2010.

[7] Apache Software Foundation. *Hadoop: Capacity Scheduler*.

[8] *AWS Auto Scaling*. [Online; accessed 17. Apr. 2022]. Apr. 2022. URL: https://aws.amazon.com/autoscaling.

[9] *AWS Batch — Easy and Efficient Batch Computing Capabilities - AWS*. [Online; accessed 13. Sep. 2021]. URL: https://aws.amazon.com/batch.

[10] *Azure Autoscale | Microsoft Azure*. [Online; accessed 17. Apr. 2022]. Apr. 2022. URL: https://azure.microsoft.com/en-us/features/autoscale.

[11] *Azure Kubernetes Service (AKS) | Microsoft Azure*. [Online; accessed 13. Sep. 2021]. URL: https://azure.microsoft.com/en-us/services/kubernetes-service/#overview.

[12] *Azure Sustainability—Sustainable Technologies | Microsoft Azure*. [Online; accessed 15. Sep. 2021]. URL: https://azure.microsoft.com/en-us/global-infrastructure/sustainability/#overview.

[13] Noman Bashir, Tian Guo, Mohammad Hajiesmaili, David Irwin, Prashant Shenoy, Ramesh Sitaraman, Abel Souza, and Adam Wierman. "Enabling Sustainable Clouds: The Case for Virtualizing the Energy System". In: *SoCC*. 2021.

[14] *Batch - Compute job scheduling service | Microsoft Azure*. [Online; accessed 13. Sep. 2021]. URL: https://azure.microsoft.com/en-us/services/batch/#overview.

[15] Edip Baysal, Dan D. Kosloff, and John W. C. Sherwood. "Reverse Time Migration". In: *GEOPHYSICS* 48.11 (1983), pp. 1514–1524. eprint: https://doi.org/10.1190/1.1441434.

[16] Broad Institute. *cromwell*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/broadinstitute/cromwell.

[17] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. "Syntactic Clustering of the Web". In: *Computer networks and ISDN systems* 29.8-13 (1997), pp. 1157–1166.

[18] *Burstable performance instances*. [Online; accessed 12. Sep. 2021]. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[19] A.W. Camargo, J. Ribeiro, N. Okita, C. Benedicto, T.A. Coimbra, J.H. Faccipieri, and M. Tygel. "Fault-Tolerant Wave Propagation Assisted By Independent Checkpointing Strategy". In: 2020.1 (2020), pp. 1–5.

[20] Marcus Carvalho, Walfredo Cirne, Franciso Brasileiro, and John Wilkes. "Long-term SLOs for reclaimed cloud computing resources". In: *SoCC*. 2014.

[21] *Chameleon - Texas Advanced Computing Center*. [Online; accessed 14. Sep. 2021]. URL: https://www.tacc.utexas.edu/systems/chameleon.

[22] ChevronETC. *Examples*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/ChevronETC/Examples.

[23] Andrew A. Chien, Richard Wolski, and Fan Yang. "The Zero-Carbon Cloud: High-Value, Dispatchable Demand for Renewable Power Generators". In: *The Electricity Journal* 28.8 (2015), pp. 110–118.

[24] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. "Unearthing inter-job dependencies for better cluster scheduling". In: *OSDI*. 2020.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *SOSP*. 2017.

[26] *CRIU*. [Online; accessed 23. Mar. 2022]. Feb. 2022. URL: https://criu.org/Main_Page.

[27] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. "Reservation-Based Scheduling: If You're Late Don't Blame Us!" In: *SoCC*. 2014.

[28] J.T. Daly. "A higher order estimate of the optimum checkpoint interval for restart dumps". In: *Future Generation Computer Systems* 22.3 (2006), pp. 303–312.

[29] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. San Francisco, CA: USENIX Association, Dec. 2004.

[30] Yanlei Diao, Abhishek Roy, and Toby Bloom. "Building Highly-Optimized, Low-Latency Pipelines for Genomic Data Analysis." In: *CIDR*. 2015.

[31] *docker checkpoint*. [Online; accessed 23. Mar. 2022]. Mar. 2022. URL: https://docs.docker.com/engine/reference/commandline/checkpoint.

[32] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. "Memory-Harvesting VMs in Cloud Platforms". In: *ASPLOS*. 2022.

[33] M. R. Garey, D. S. Johnson, and Ravi Sethi. "The Complexity of Flowshop and Jobshop Scheduling". In: *Mathematics of Operations Research* 1.2 (May 1976), pp. 117–129.

[34] *GATK*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://gatk.broadinstitute.org/hc/en-us.

[35] *Genomic data processing reference architecture*. [Online; accessed 13. Sep. 2021]. URL: https://cloud.google.com/architecture/genomic-data-processing-reference-architecture.

[36] *Get started with Batch | Google Cloud*. [Online; accessed 13. Sep. 2022]. URL: https://cloud.google.com/batch/docs/get-started.

[37] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *NSDI*. 2011.

[38] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. "Multi-resource Packing for Cluster Schedulers". In: *SIGCOMM*. 2014.

[39] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. "Altruistic Scheduling in Multi-Resource Clusters". In: *OSDI*. 2016.

[40] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. "GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters". In: *OSDI*. 2016.

[41] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, David Dion, Esaias E Greeff, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. "Protean: VM Allocation Service at Scale". In: *OSDI*. 2020.

[42] James Hamilton. *AWS Innovation at Scale*. Nov. 2014. URL: www.youtube.com/watch?v=JIQETrFC_SQ.

[43] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. "Tributary: spot-dancing for elastic services with latency SLOs". In: *USENIX ATC*. 2018.

[44] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. "Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets". In: *EuroSys*. 2017.

[45] *Helping the financial services industry transform with Google Cloud | Google Cloud Blog*. [Online; accessed 13. Sep. 2021]. URL: https://cloud.google.com/blog/topics/financial-services/helping-the-financial-services-industry-transform.

[46] *High Burst CPU Compute for Monte Carlo Simulations on AWS | Amazon Web Services*. [Online; accessed 13. Sep. 2021]. URL: https://aws.amazon.com/blogs/hpc/high-burst-cpu-compute-for-monte-carlo-simulations-on-aws.

[47] *High-Performance Computing for Financial Services | Microsoft Azure*. [Online; accessed 13. Sep. 2021]. URL: https://azure.microsoft.com/en-us/solutions/high-performance-computing/financial-services/#features.

[48] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *NSDI*. 2011.

[49] *Introducing B-Series, our new burstable VM size*. [Online; accessed 12. Sep. 2021]. URL: https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size.

[50] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Jana Kulkarni, and Sriram Rao. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *OSDI*. 2016.

[51] Jcs Kadupitiya, Vikram Jadhao, and Prateek Sharma. "SciSpot: Scientific Computing On Temporally Constrained Cloud Preemptible VMs". In: *IEEE Transactions on Parallel and Distributed Systems* (2022), pp. 1–1.

[52] Ian A. Kash, Greg O'Shea, and Stavros Volos. "DC-DRF: Adaptive Multi-Resource Sharing at Public Cloud Scale". In: *SoCC*. 2018.

[53] *Kubernetes*. [Online; accessed 13. Sep. 2021]. URL: https://kubernetes.io/docs/reference.

[54] *Kubernetes - Google Kubernetes Engine (GKE) | Google Cloud*. [Online; accessed 13. Sep. 2022]. URL: https://cloud.google.com/kubernetes-engine.

[55] *Kubernetes on AWS | AWS*. [Online; accessed 13. Sep. 2021]. URL: https://aws.amazon.com/kubernetes.

[56] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. "Dynamic Query Re-Planning using QOOP". In: *OSDI*. 2018.

[57] Sharmistha Mandal, Sunirmal Khatua, and Rajib K. Das. "Bid Selection for Deadline Constrained Jobs over Spot VMs in Computational Cloud". In: *Distributed Computing and Internet Technology*. Cham, Switzerland: Springer, Nov. 2016, pp. 118–128.

[58] Microsoft. *AzureClusterlessHPC.jl*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/microsoft/AzureClusterlessHPC.jl.

[59] Microsoft. *CromwellOnAzure*. [Online; accessed 30. Aug. 2021]. Aug. 2021. URL: https://github.com/microsoft/CromwellOnAzure.

[60] *National Energy Research Scientific Computing Center*. [Online; accessed 14. Sep. 2021]. URL: https://www.nersc.gov.

[61] Frank Austin Nothaft, Matt Massie, Timothy Danford, Zhao Zhang, Uri Laserson, Carl Yeksigian, Jey Kottalam, Arun Ahuja, Jeff Hammerbacher, Michael Linderman, Michael J. Franklin, Anthony D. Joseph, and David A. Patterson. "Rethinking Data-Intensive Science Using Scalable Analytics Systems". In: *SIGMOD*. 2015.

[62] Keiron O'Shea and Ryan Nash. "An Introduction To Convolutional Neural Networks". In: *arXiv preprint arXiv:1511.08458* (2015).

[63] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. "3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty". In: *EuroSys*. 2018.

[64] David C. Parkes, Ariel D. Procaccia, and Nisarg Shah. "Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities". In: *ACM Transactions on Economics Computation* 3.1 (Mar. 2015).

[65] Florentina I. Popovici and John Wilkes. "Profitable Services in an Uncertain World". In: *SC*. 2005.

[66] *Power generation*. [Online; accessed 12. Sep. 2021]. URL: https://www.elia.be/en/grid-data/power-generation.

[67] *Request real-time and forecasted weather data using Azure Maps Weather services*. [Online; accessed 13. Sep. 2021]. URL: https://docs.microsoft.com/en-us/azure/azure-maps/how-to-request-weather-data.

[68] *Running Geospacial Workloads On AWS*. [Online; accessed 14. Sep. 2021]. URL: https://anz-resources.awscloud.com/aws-summit-sydney-2019-analyse/room5-day1-1745-runninggeospacialworkloadsonaws-v3-3.

[69] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. "Soap: One Clean Analysis of All Age-Based Scheduling Policies". In: *SIGMETRICS*. 2018.

[70] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. "SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market". In: *EuroSys*. 2015.

[71] Supreeth Shastri and David Irwin. "HotSpot: Automated Server Hopping in Cloud Spot Markets". In: *SoCC*. 2017.

[72] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *SIGCOMM*. 2015.

[73] *Slurm Workload Manager - Documentation*. [Online; accessed 13. Sep. 2021]. URL: https://slurm.schedmd.com/documentation.html.

[74] *Spot Virtual Machines – Spot Pricing and Features | Microsoft Azure*. [Online; accessed 12. Sep. 2021]. URL: https://azure.microsoft.com/en-us/services/virtual-machines/spot/#overview.

[75] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *INTERSPEECH*. 2012.

[76] *Sustainability | Google Cloud*. [Online; accessed 15. Sep. 2021]. URL: https://cloud.google.com/sustainability.

[77] *Sustainability in the Cloud*. [Online; accessed 15. Sep. 2021]. URL: https://sustainability.aboutamazon.com/environment/the-cloud?energyType=true.

[78] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The next Generation". In: *EuroSys*. 2020.

[79] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters". In: *EuroSys*. 2016.

[80] Prateeksha Varshney and Yogesh Simmhan. "Autobot: Resilient and Cost-Effective Scheduling of a Bag of Tasks on Spot Vms". In: *TPDS* 30.7 (2019).

[81] Jean Virieux and Stéphane Operto. "An Overview of Full-Waveform Inversion in Exploration Geophysics". In: *GEOPHYSICS* 74.6 (2009), WCC1–WCC26. eprint: https://doi.org/10.1190/1.3238367.

[82] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. "SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud". In: *EuroSys*. 2021.

[83] Adam Wierman and Misja Nuyens. "Scheduling Despite Inexact Job-Size Information". In: *SIGMETRICS*. 2008.

[84] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. "Blending On-Demand and Spot Instances to Lower Costs for in-Memory Storage". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. San Francisco, CA, USA: IEEE Press, 2016, pp. 1–9.

[85] Murtaza Zafer, Yang Song, and Kang-Won Lee. "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs". In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 75–82.

[86] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *EuroSys*. 2010.

[87] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Sameh Elnikety Rodrigo Fonseca, Christina Delimitrou, and Ricardo Bianchini. "Faster and Cheaper Serverless Computing on Harvested Resources". In: *SOSP*. 2021.

[88] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. "History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters". In: *OSDI*. 2016.

[89] Hua-Wei Zhou, Hao Hu, Zhihui Zou, Yukai Wo, and Oong Youn. "Reverse Time Migration: a Prospect of Seismic Imaging Methodology". In: *Earth-Science Reviews* 179 (2018), pp. 207–227.

A Pseudocode

For completeness, we list the pseudocode for the algorithms used by the SLACKSCHED Scheduler and SLACKSCHED Acquirer. Algorithm 1 is used by the scheduler to match tasks to nodes, and Algorithm 2 is used by the acquirer to select nodes to deallocate.

Note, in our implementation we ensure that the allocations and deallocations by the Acquirer do not nullify those done by the scaling policy by considering the net change in number of nodes that needs to be there. We also let the scaling policy scale out the cluster after intentional deallocations instead of Acquirer directly requesting VMs after decommissioning VMs (i.e., lines 6, 7, 8 of Algorithm 2 are coupled in our implementation). The implementation can be done in a way that avoids this coupling. Decoupling would actually be needed for cases when scaling and acquisition are triggered independently.

SLACKSCHED for heartbeat-based scheduling. In some cluster managers (e.g., Apache YARN [5]) scheduling is triggered on heartbeats [7]. In these cases, the node to schedule on is implicitly decided based on the node which sent the heartbeat. However, SLACKSCHED needs to explicitly decide which node a task should be mapped to. Thus, we adapt the scheduling logic so that the scheduler can wait for heartbeats from other nodes rather than necessarily assigning tasks to the random node that sent a heartbeat. Specifically, for the node that sent the heartbeat, if the task meets a threshold completion probability, we directly schedule the task on that node, otherwise, we look at the expected completion times offered by all the other nodes and wait for more heartbeats if there are better nodes. The pseudocode for this logic is listed in Algorithm 3. This adds negligible overhead to the time complexity of the scheduler.

Algorithm 1: SLACKSCHED Scheduler

```
1  Function MatchMaker(Task t, Nodes N)
2      W ← []
3      d ← duration(t)
4      for n ∈ N do
5          x ← lastChangeDirection(n)
6          a ← nextAvailTime(n,t)
7          e ← timeSinceLastChange(n)
8          // Using formulation in §3.1
9          c ← expectedCompletionTime(x,a,e,d)
10         W.append((c,n))
11     end
12     (c*,n*) ← min(W)
13     if feasible(t, n*) then
14         schedule(t, n*)
15     end
16     // Otherwise re-visit decision
17     // at the next scheduling event
18 end
```

Algorithm 2: SLACKSCHED Acquirer

```
1  deallocateFraction = 0.1
2  Function OnEpochEnd(Nodes N)
3      N.sort()
4      // increasing order of timeSinceLastChange
5      toDeallocate ← N[: N.size() * deallocateFraction]
6      requestNVMs(toDeallocate.size())
7      gracefullyDecommission(toDeallocate)
8      // trigger scaling
9  end
10
```

Algorithm 3: SLACKSCHED Scheduler

```
1  Function ScheduleOnNodeUpdate(Tasks T, Machine m)
2      w ← []
3      x ← lastChangeDirection(m)
4      e ← timeSinceLastChange(m)
5      for t ∈ T do
6          if ¬ feasible(t, m) then
7              continue
8          end
9          d ← duration(t)
10         p ← completionProbability(x,e,d)
11         if p > threshold then
12             w.append((p,t))
13         end
14     end
15     if w is not empty then
16         (p*,t*) ← max(w)
17         schedule(t*, m)
18     end
19     else
20         N ← getAllNodes()
21         t ← shortestTask(T)
22         MatchMaker (t, N) // (Algorithm 1)
23     end
24 end
```

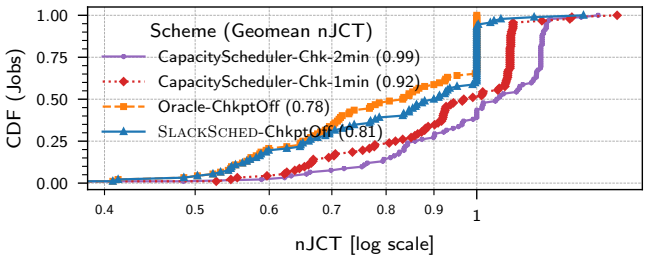


Figure 20: nJCT relative to *CapacityScheduler* without checkpointing. SLACKSCHED outperforms the checkpoint-migrate technique. We set the time-per-checkpoint as 1 min and 2 min in the two schemes with checkpointing.

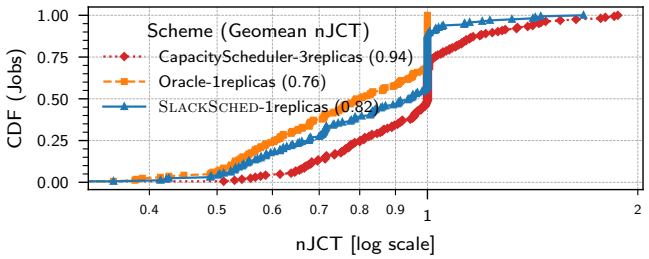


Figure 21: nJCT relative to *CapacityScheduler* without replication. Replication only slightly improves job completion time.

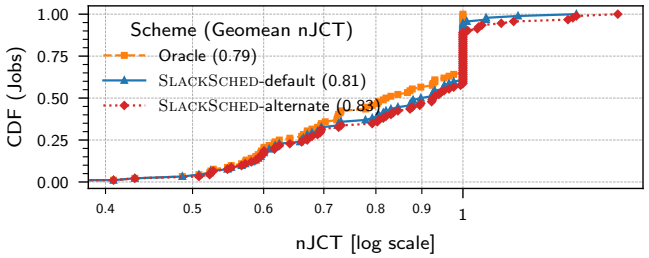


Figure 22: nJCT relative to *CapacityScheduler*. There may be multiple ways to define affinity between a task and an HVM and/or compute completion probability.

B Additional Evaluation

B.1 Comparison to prior work for Spot VMs

Prior work [28, 29, 71, 80] employs checkpointing, migration, and replication techniques to mitigate the impact of preemptions. We empirically show that these techniques are insufficient to curb the overheads imposed by HVMs on our target workloads.

Checkpointing and migration. A common approach to handle Spot VM preemptions is to periodically checkpoint tasks [28] and when they get evicted, restore them in other servers. Universal checkpointing techniques like the one docker uses based on CRIU [26, 31] only handle effects inside the container and ignores any external ones (e.g., external storage for intermediate outputs). The workloads we study do not perform checkpointing. On preemptions, tasks are simply restarted inside a new container with reinitialized external storage for intermediate results and outputs. Implementing efficient and complete checkpointing would require application specific insights and expensive engineering.

For the sake of evaluation, we measure overhead for checkpointing memory and processor state using the docker mechanism [31][8]. This checkpointing delays tasks by $\approx$ 1–2 minutes per checkpoint even for containers with less than 1 GB of memory[9]. This relatively large delay is because our applications rely on stateful shared libraries, open multiple TCP connections, and open a number of files in memory. These findings are consistent with [19].

To study the end-to-end impact of checkpointing, we extend our simulation framework to stall tasks being checkpointed. We delay each task by X=1,2 minute(s) per checkpoint (independent of container memory size) and run checkpoints every 10 minutes. On preemptions, tasks are restarted from the latest checkpoint and assume no overhead to migrate checkpoints. Figure 20 shows the normalized job completion time. We find that even with such optimistic checkpoint-migrate overheads, SLACKSCHED outperforms checkpoint-migrate techniques.

Replication. Another alternative to handle preemptions is to launch multiple tasks replicas and once one completes, kill the rest. This technique is common in MapReduce systems [29]. In our implementation, we launch 3 replicas for each task and kill other replicas as soon as any replica completes. Figure 21 shows that while replication reduces JCT for the *CapacityScheduler*, SLACKSCHED outperforms both *CapacityScheduler* with and without replication. In addition, replicas lead to a higher number of total preemptions and a much higher load.

B.2 Alternate method for computing the completion probability

There are other ways of computing completion probability beyond the method presented in §3.1.1. One notable way is to construct the distribution of time between consecutive resource change and resource shrink events[10] and then compute completion probability as:

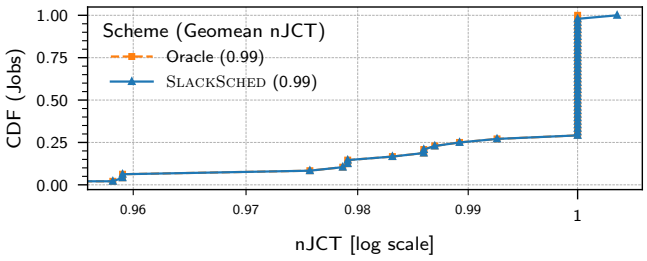
$$\begin{aligned} & P_c(e,d) \\ &= P\begin{bmatrix}\text{shrink does not occur}\\ \text{during task lifetime}\end{bmatrix} \\ &= P\begin{bmatrix}\text{shrink occurs}\\ \text{after task lifetime}\end{bmatrix} \\ &= P(Y > e+d | Y > e) \end{aligned} \tag{3}$$

where Y is the distribution of time between resource change and resource shrink events, e is the time elapsed since the

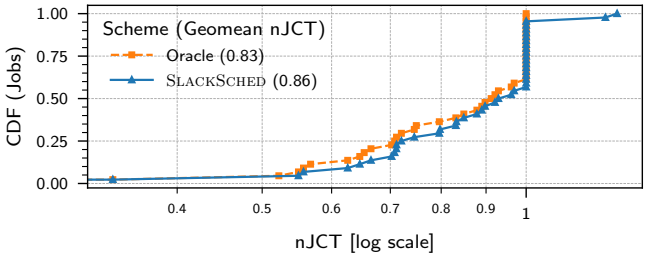
[8]These checkpoints are not restorable, a complete checkpoint would include local files and external storage/effects.

[9]The checkpointing latency (i.e., time when the checkpoint is available for restoration) is typically larger than the delay added to the task.

[10]Note, the inter-change time distribution also captures the time between two consecutive resource growth events.

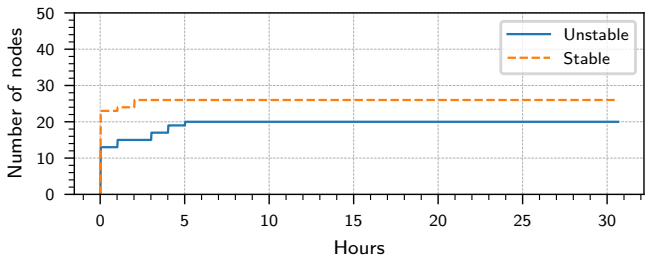


(a) Interruptible jobs.

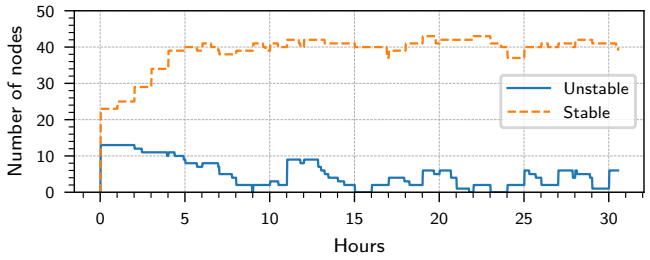


(b) Uninterruptible jobs.

Figure 23: SlackSched improves JCT in a cluster serving a mix of interruptible and uninterruptible workloads.



(a) Without SlackSched-Acquirer



(b) With SlackSched-Acquirer

Figure 24: SlackSched-Acquirer converges to a set of relatively more stable HVMs.

last resource change event, and d is the (remaining) duration of the task being considered for scheduling. We observe that using this alternate method provides similar benefits as SlackSched as shown in Figure 22. The default method of SlackSched provides mean reduction of 19% compared to *CapacityScheduler*, while the alternate method provides mean reduction of 17% compared to *CapacityScheduler*. The main difference is that SlackSched also conditions on whether the previous resource change event was growth or shrink to compute completion probability, which gives slightly better completion probability estimates.

B.3 Mixing interruptible and uninterruptible

We study how SlackSched performs in a cluster serving a mix of interruptible and uninterruptible jobs. Interruptible jobs are those that have negligible checkpointing and migration overhead.

Methodology. We simulate interruptible workloads by running checkpoints every minute and add zero overhead for checkpointing and migration. On preemption, the tasks restart from the latest checkpoint. No checkpointing or migration is done for uninterruptible jobs. The cluster serves a 50-50 mix of uninterruptible and interruptible jobs that arrive according to a Poisson process. We use the same distribution to generate task runtime and resource requirements for uninterruptible and interruptible jobs. Recall, these distributions were based on genomics and seismic workloads, which are hard to checkpoint and migrate. As such the interruptible jobs are synthetic — they do not necessarily resemble a real workload — unlike the uninterruptible jobs.

Results. Figure 23 shows the distribution of nJCT relative to *CapacityScheduler* separately for the uninterruptible and interruptible jobs. SlackSched shows qualitatively similar improvement as before for uninterruptible workloads. For interruptible workloads, there is no significant difference in completion times across schemes.

B.4 Acquirer with known ground truth

To verify the operation of SlackSched-Acquirer and to study its convergence properties, we study its operation on HVMs with known ground truth stability. To establish the ground truth, we generate a synthetic regular VM arrival and placement trace such that HVMs on half the servers are unstable (i.e., witness frequent regular VM arrivals and departures); while HVMs on other half are stable (i.e., witness infrequent regular VM arrivals and departures). We use the scaling policy that maintains a fixed resource budget and use the random policy for HVM allocation. We measure and compare the number of stable and unstable HVMs with and without using SlackSched-Acquirer. Figure 24 shows that the Acquirer converges to a mix with a larger portion of stable HVMs.

B.5 Absolute JCT metrics

Due to space constraints we show normalized JCT for most experiments in the main text. Here we show graphs for absolute JCT. In all our experiments, we observe that mean reduction in JCT follows similar trends as reduction in mean JCT. We compute reduction in mean and p90 JCT relative to *CapacityScheduler* unless mentioned otherwise.

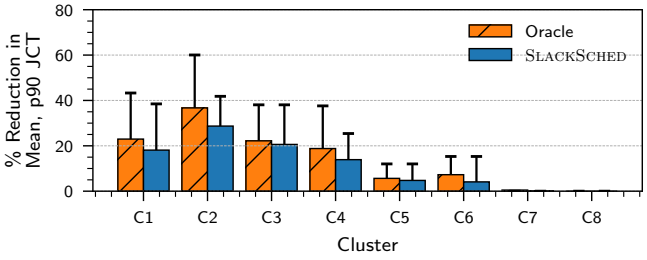


Figure 25: All clusters. cf. Figure 9. The bars correspond to mean JCT and whiskers correspond to p90 JCT.

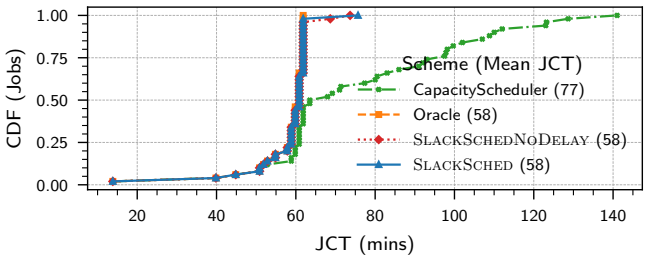


Figure 26: Stable cluster. cf. Figure 10b.

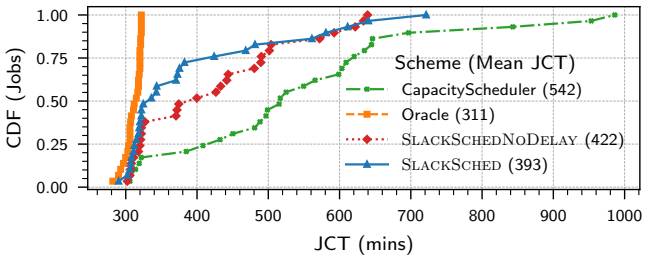


Figure 27: Volatile cluster. cf. Figure 10a.

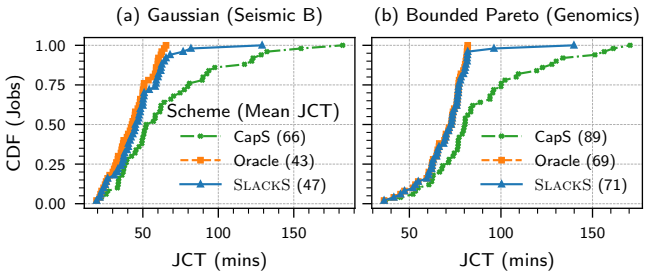


Figure 28: Different task runtime distributions. cf. Figure 11.

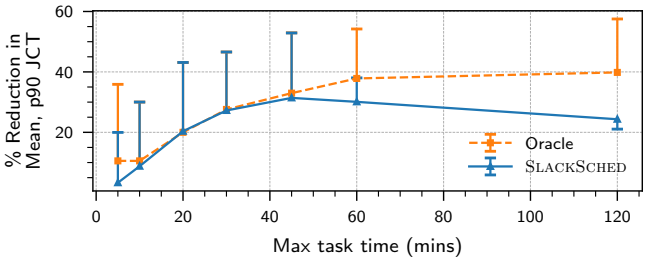


Figure 29: Varying max task time. cf. Figure 12. The lines correspond to mean JCT and whiskers correspond to p90 JCT.

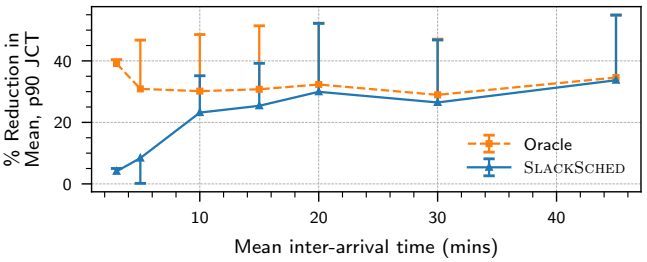


Figure 30: Varying load (job arrival rate). cf. Figure 14. The lines correspond to mean JCT and whiskers correspond to p90 JCT.

Figure 31: Robustness to errors in task runtime estimates. cf. Figure 15.

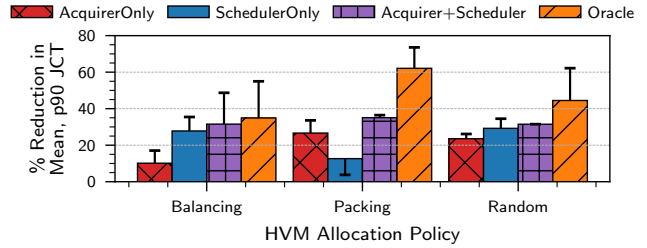


Figure 32: Resource acquisition evaluation. cf. Figure 16.

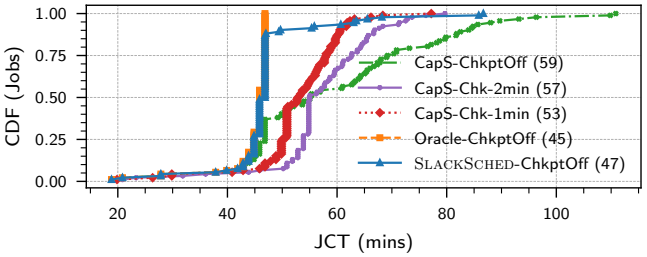


Figure 33: Comparison with checkpointing and migration. cf. Figure 20. Time in legend shows time-per-checkpoint for *CapacityScheduler*.

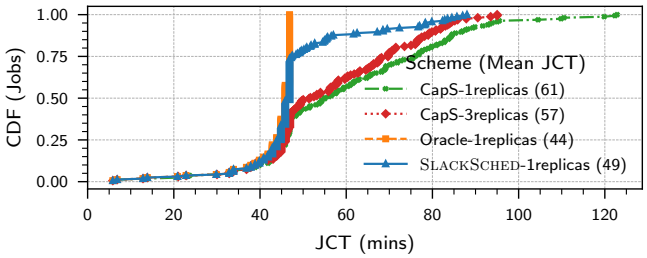


Figure 34: Comparison with replication. cf. Figure 21.

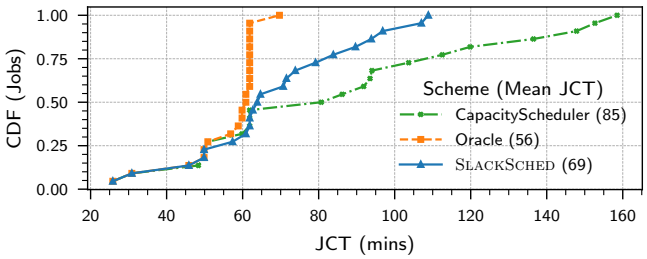


Figure 35: Renewable energy setting. cf. Figure 19.