Selecting the *Best* VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach

Neeraja J. Yadwadkar[1], Bharath Hariharan[2], Joseph E. Gonzalez[1], Burton Smith[3], and Randy Katz[1]
[1]*University of California, Berkeley* [2]*Facebook AI Research* [3]*Microsoft Research*

Submission Type: Research

Abstract

Users of cloud services are presented with a bewildering choice of VM types and the choice of VM can have significant implications on performance and cost. In this paper we address the fundamental problem of *accurately* and *economically* choosing the *best VM* for a given *workload* and *user goals*. To address the problem of optimal VM selection, we present PARIS, a data-driven system that uses a novel *hybrid* offline and online data collection and modeling framework to provide *accurate* performance estimates with *minimal* data collection. PARIS is able to predict workload performance for different user-specified metrics, and resulting costs for a wide range of VM types and workloads across multiple cloud providers. When compared to a sophisticated baseline linear interpolation model using measured workload performance on two VM types, PARIS produces significantly better estimates of performance. For instance, it reduces runtime prediction error by a factor of 4 for some workloads on both AWS and Azure. The increased accuracy translates into a 45% reduction in user cost while maintaining performance.

1 Introduction

As companies of all sizes migrate to cloud environments, increasingly diverse workloads are being run in the Cloud — each with different performance requirements and cost trade-offs [54]. Recognizing this diversity, cloud providers offer a wide range of Virtual Machine (VM) types. For instance, at the time of writing, Amazon [2], Google [7], and Azure [38] offered a combined total of over 100 instance types with varying system and network configurations.

In this paper we address the fundamental problem of *accurately* and *economically* choosing the *best VM* for a given *workload* and *user goals*. This choice is critical because of its impact on performance metrics such as runtime, latency, throughput, cost, and availability. Yet determining or even defining the *"best"* VM depends heavily on the users' goals which may involve diverse, application-specific performance metrics, and span tradeoffs between price and performance objectives.

For example, Figure 1 plots the runtimes and resulting costs of running a video encoding task on several AWS VM types. A typical user wanting to deploy a workload might choose the cheapest VM type (`m1.large`) and paradoxically end up not just with poor performance but also high total costs. Alternatively, overprovisioning by picking the most expensive VM type (`m2.4xlarge`) might only offer marginally better runtimes than much cheaper alternatives like `c3.2xlarge`. Thus, to choose the right VM for her performance goals and budget, the user needs accurate performance estimates.

Recent attempts to help users select VM types have either focused on optimization techniques to efficiently search for the best performing VM type [12], or extensive experimental evaluation to model the performance cost trade-off [64]. Simply optimizing for the best VM type for a particular goal (as in CherryPick [12]) assumes that this goal is *fixed*; however, different users might prefer different points along the performance-cost trade-off curve. For example, a user might be willing to tolerate mild reductions in performance for substantial cost savings. In such cases, the user might want to know precisely how switching to another VM type affects performance and cost.

The alternative, directly modeling the performance-cost trade-off, can be challenging. The published VM characteristics (e.g., memory and virtual cores) have hard-to-predict performance implications for any given workload [67, 35, 24]. Furthermore, the performance often depends on workload characteristics that are difficult to specify [27, 15, 35]. Finally, variability in the choice of host hardware and resource contention [54] can result in performance variability [50] that is not captured in the published VM configurations. Recent data driven approaches like Ernest [64] overcome these limitations through extensive performance measurement and modeling. However these techniques introduce an $O(n^2)$ data collection process as each workload is evaluated on each VM type.

The movement towards server-less compute frameworks such as AWS Lambda [4], Azure Functions [5], or Google

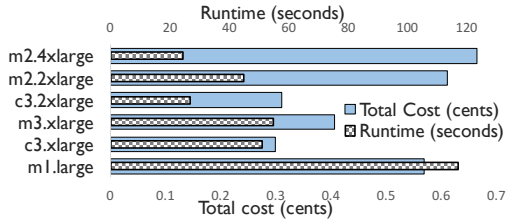
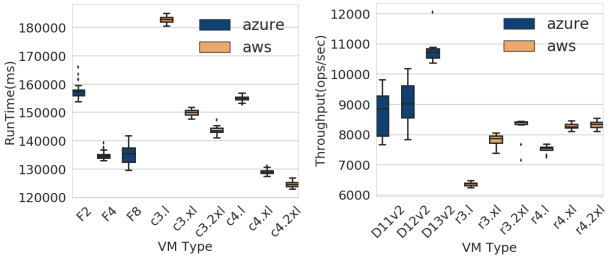


Figure 1: Execution time and total cost of a video encoding task on AWS, across various VM types.



(a) Building Apache Giraph (b) YCSB-benchmarks Workload A

Figure 2: **(a)** Runtime for building Apache Giraph (lower the better) and **(b)** Throughput for a 50/50 R/W serving workload on the Redis in-memory datastore using YCSB (higher the better) across different VM types offered by AWS and Azure.

Cloud Functions [6] may appear to eliminate the challenges of VM selection, but in fact simply shift the challenges to the cloud provider. While cloud providers may have detailed information about their resources, they have limited visibility into the requirements of each workload.

In this paper we present PARIS, a **P**erformance-**A**ware **R**esource **I**nference **S**ystem, which estimates the performance-cost trade-off for all VM types, allowing users to balance performance gains with cost reductions. PARIS is applicable to a broad range of workloads and performance metrics and works across cloud providers. PARIS introduces a novel *hybrid* offline and online data collection and modeling framework which provides *accurate* performance estimates with *minimal* data collection, eliminating the $O(n^2)$ data collection complexity.

The key insight in PARIS is to decouple VM performance characterization from the characterization of workload-specific resource requirements. By leveraging a shared profiling framework and established machine learning techniques PARIS is able to combine these separate stages to achieve accurate performance predictions for all combinations of workload and VM type.

In the offline stage, PARIS runs a broad set of benchmarks with diverse resource requirements and collects extensive profiling information for each VM type. Intuitively, the diversity of the resource requirements in the benchmarks ensures that we observe how each VM type responds to demands on its resources. Because these benchmarks are *independent* of the query workloads, the benchmarks only need to be run once for each new VM type.

In the online stage, PARIS characterizes each new query workload by executing a user-specified task that is representative of her workload on a pair of *reference VMs* and collecting the *same* profiling statistics as in the offline stage. These profiling statistics form a *fingerprint* characterizing the workload in the same *dimensions* as the offline benchmarking process. PARIS then combines this fingerprint with the offline VM benchmarking data to build an accurate model of the workload performance characteristics across *all* VM types spanning multiple cloud providers.

We demonstrate that PARIS is sufficiently general to accurately predict a range of performance metrics and their variability for widely deployed batch processing and serving-style workloads across VMs from multiple public cloud providers. For instance, it reduces the prediction error for the runtime performance metric by a factor of 4 for some workloads on both AWS and Azure. The increased accuracy translates into a 45% reduction in user cost while maintaining performance (runtime).

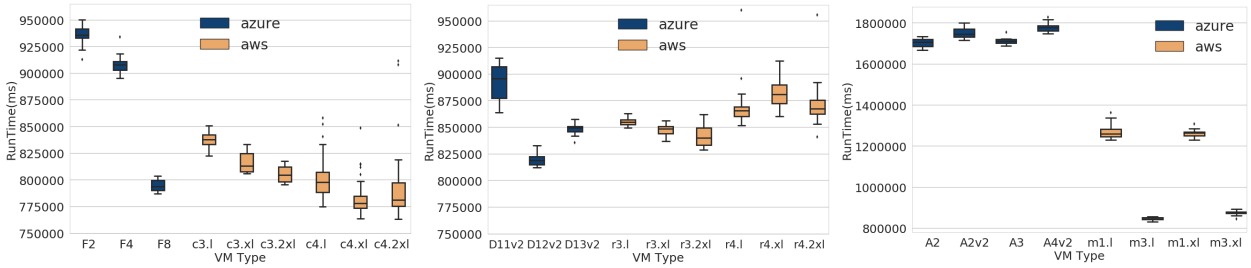
The key contributions of this paper are:

- an experimental characterization of performance trade-off of various VM types for realistic workloads across Amazon AWS and Microsoft Azure (Sec. 2).
- a novel hybrid offline (Sec. 4) and online (Sec. 5) data collection and modeling framework which eliminates the $O(n^2)$ data collection overhead while providing accurate predictions across cloud providers.
- a detailed experimental evaluation demonstrating that PARIS accurately estimates multiple performance metrics and their variabilities (P90 values), for several real-world workloads across two major public cloud providers, thereby reducing user cost by up to 45% relative to strong baseline techniques (Sec. 6.3).

2 Motivation

To illustrate the challenges involved in selecting VM types, we evaluated three different workloads on a range of VM types spanning two cloud providers: Amazon AWS and Microsoft Azure. Below, we present the complex and often counterintuitive trade-offs between performance and cost.

As an example of a software-build system, we studied the compilation of Apache Giraph (see Figure 2a) on a range of compute-optimized instances. As an example serving application, we ran a YCSB query processing benchmark on the Redis in-memory data-store (Figure 2b) on a range of memory-optimized instances. Finally, as an example of a more complex task that utilizes multiple resources, we experimented with a compression workload that downloads, decompresses, and then re-compresses a



(a) Compute-optimized VMs (b) Memory-optimized VMs (c) General-purpose VMs

Figure 3: Runtimes for a compression workload across VM types from different families offered by AWS and Azure. In plot (c), note the different scale on y-axis.

remote file (Figure 3). This task emulates many standard cloud-hosted applications, such as video transcoding, that utilize network, compute, disk, and memory at different stages in the computation. We ran the compression workload on both specialized and general-purpose cloud VMs.

Bigger is not always better: Often users choose to defensively provision the most expensive or the "largest" VM type under the assumption that larger or more expensive instances provide improved performance. This is not always true: for building Giraph, the Azure `F8` VM type performs *worse* than the `F4` VM type in spite of being larger. Similarly, for the YCSB serving benchmark, the throughput doesn't improve much when going from `r4.xlarge` to the more expensive `r4.2xlarge`, making `r4.xlarge` a more cost-efficient choice. This suggests that provisioning more resources than the workload needs might be unnecessary for good performance.

Similar configurations but different performance: For the YCSB workload (Figure 2b), the AWS `R4` family performs worse than Azure `Dv2` in spite of having a very similar configuration. By contrast, the `R3` and `R4` families perform similarly despite the latter using a newer generation processor. These observations indicate other factors at play: differences in the execution environment, and hardware or software differences that are not reflected in the configuration. Thus, VM type configuration alone does not predict performance.

Optimizing for mean performance may not optimize for the tail: For the YCSB workload, Azure VMs provide improved throughput while AWS VMs provide more consistent performance. A developer of a cloud-hosted service might prefer a guaranteed throughput to improved but less predictable throughput. For the compression workload (Figure 3), some of the Azure VMs showed reduced variability, even when they lead to a longer expected runtime. Thus, the best VM type may differ depending on whether we are interested in the mean or the tail.

Workload resource requirements are opaque: For workloads that use different resources at different points during their execution, it can be hard to figure out which resources are the most crucial for performance [51]. This is especially challenging for hosted compute services such as AWS-Lambda where the workload is treated as a black-box function. For the compression workload (Figure 3a), memory- and compute-optimized VM types offered lower runtimes compared to general purpose VM types, indicating that memory or compute, or both, might be the bottleneck. Yet, counterintuitively, going from `r4.l` to `r4.xl`, or `c4.xl` to `c4.2xl` actually *hurts* performance. This might be because of the underlying execution environment, issues of performance isolation, or the non-linear dependence of performance on resource availability, none of which is captured in the resource configuration alone.

Monitoring resources consumed while a task is running might help identify resources utilized for that run, but will not tell us how performance is impacted in constrained settings or on different hardware / software. Profiling the workload on each VM across all cloud providers will be informative but prohibitively expensive. We need a much cheaper solution that can nevertheless predict the performance of arbitrary workloads on all VM types accurately.

3 System Overview

PARIS enables cloud users to make better VM type choices by providing performance and cost estimates on different VM types tailored to their workload.

PARIS runs as a light weight service that presents a simple API to the cloud user. The cloud user (or simply "user") interacts with PARIS by providing a representative task of her workload, the desired performance metric, and a set of candidate VM types. PARIS then calculates the predicted *performance* and *cost* for all of the provided candidate VM types. The user can then use this information to choose the best VM type for any performance and cost goals. For the user, the interaction looks like this:

```
# Get performance and cost est. for targetVMs
perfCostMap = predictPerfCost(userWorkloadDocker,
     candidateVMs, perfMetric)
# Choose VM with min cost subj. to a perf. req.
chosenVMType = minCost(perfCostMap, perfReq)
```

To make accurate performance prediction, PARIS needs to model two things: a) the resource requirements of the workload, and b) the impact of different VM types on workloads with similar resource requirements. However, exhaustively profiling the user's workload on all VM types is prohibitively expensive. To avoid the cost overhead, PARIS divides the modeling task into two phases (Figure 4): a *one-time, offline, extensive* VM type benchmarking phase (Section 4) and an *online, inexpensive* workload profiling phase (Section 5). We provide a high-level overview of each phase below and then elaborate on each phase in the subsequent sections.

In the *offline* VM-benchmarking phase, PARIS uses a *Profiler* to run a suite of benchmarks for each VM type and collect detailed system performance metrics. The benchmark suite is chosen to span a range of realistic *workload patterns* with a variety of resource requirements. This benchmarking can be run by the cloud providers or published[1] by a third party. As new VM types or physical hardware is introduced, the benchmark only needs to be rerun on the new VM types. The offline phase has a fixed one-time cost and removes the extensive profiling and data collection from the critical path of predicting performance characteristics of new user workloads.

In the *online* phase, end users interact with PARIS by providing an example or representative task of their workload. PARIS first characterizes the resource usage patterns of the workload by invoking a *Fingerprint-Generator*. The Fingerprint-Generator runs the representative task on a *small* (typically, 2) set of *reference VM types* and collects runtime measurements. We choose reference VM types that are farthest apart in terms of their configurations, to capture workload performance in both resource-abundant and resource-constrained settings. These measurements capture the resource usage patterns of the task and form the workload *fingerprint*. While the fingerprinting process incurs additional cost, this cost is small and independent of the number of candidate VM types.

PARIS then combines the fingerprint with the offline benchmarking data to construct a machine learning model that accurately estimates the desired performance metrics as well as the 90^{th} percentile values for corresponding performance metrics for the user workload[2]. Finally, PARIS assembles these estimates into a performance-cost trade-off map across all VM types.

4 Offline VM-benchmarking phase

In the offline benchmarking phase, the profiler uses a set of benchmark workloads to characterize VM types. These

[1]We plan to publish our benchmark data across VM types.

[2]PARIS can predict higher percentiles too, but these require more samples during fingerprinting, raising user costs.

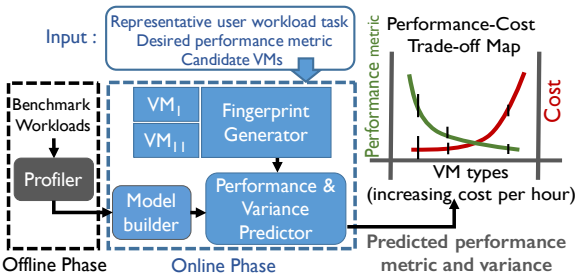


Figure 4: Architecture of PARIS (Sec. 3).

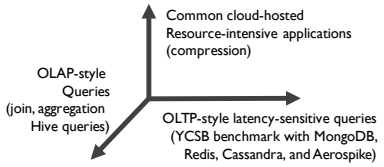


Figure 5: Benchmark workloads chosen from a diverse set of cloud use-cases [3].

benchmark workloads are chosen to be diverse in terms of their type, the performance metrics they use, and their resource requirements (Figure 5). This allows PARIS to characterize how the different VM types respond to different patterns of resource usage. The set of benchmark workloads is not exhaustive but rather intended to span the space of requirements workload requirements. Below we describe the benchmark workloads in more detail.

We evaluated each VM type on a range of realistic benchmarks. To represent OLAP-style analytical queries, we included the join and aggregation queries of Hive [61]. These model complex analytical queries over structured relational tables and exercise CPU, disk (read), and network. As a representation of latency-sensitive serving workloads in the cloud, we added YCSB core benchmark workloads [25] with Aerospike [1], MongoDB [23], Redis [20], and Cassandra [37] datastores. Finally, as an example of a multi-stage workload, we constructed a benchmark that simulates a hosted compression service, using the squash compression benchmark [9]. This benchmark downloads a compressed file over the network and then decompresses and re-compresses the file thereby exercising compute, memory and disk resources.

The Profiler: The profiler records the performance of each benchmark task for a range of metrics. To accurately estimate performance variability and $p90$ values, each task is run 10 times on each VM type and the empirical 90^{th} percentile performance is computed over all 10 trials (see Section 6.2 and Table 2 for details).

During each run, the profiler also records aggregated measurements that represent the task's resource usage and

performance statistics. This leverages instrumentation mechanisms that are in place in most of today's infrastructure [54]. Concretely, we used Ganglia [42] to instrument the VMs to capture performance and resource counters at a regular 15 second intervals, and record the average (or sum, depending on the counter) of these counters over the task's run. We collected about 20 resource utilization counters. These counters span following broad categories:

(a) **CPU utilization:** CPU idle, system, and user time.
(b) **Network utilization:** Bytes sent and received.
(c) **Disk utilization:** Ratio of free to total disk space.
(d) **Memory utilization:** Available virtual, physical, and shared memory, and the cache and buffer space.
(e) **System-level features:** Number waiting, running, terminated, and blocked threads and the host load in the last 1, 5, and 15 minutes.

5 Online performance prediction

PARIS interacts with users in the online phase. The user provides PARIS with three things: an example or representative task from her workload, the performance metric she cares about, and a set of target or candidate VM types for which she needs performance and cost estimates.

PARIS first invokes the *Fingerprint-Generator*, which runs the user-specified task on the pre-defined set of reference VM types[3], and in the process uses the profiler described above to collect resource usage and performance statistics. Because we want to predict the 90^{th} percentile performance, we run the task 10 times on each reference VM type and record the 90^{th} percentile performance on these reference VMs. The resource usage measurements, and the mean and 90^{th} percentile performance on the two reference VM types, are put together into a vector F called the *workload fingerprint*. Intuitively, because the fingerprint records resource usage information and not just performance, this fingerprint can help us understand the resource requirements of the task. This can help us predict the workload's performance on other VM types.

The fingerprint tells us the resources used by the task, and the VM type configuration tells us the available resources. For a single task in isolated environments, if the relationship between its performance and the available resources is known, then this information is enough to predict performance. For example, if, when run on a large machine, the profile indicates that the task used 2 GB of memory, and it performs poorly on a reference VM type with 1 GB of memory, then it might perform poorly on other VM types with less than 2 GB of memory. Otherwise, if the task is performing a lot of disk I/O and spends a lot of time blocked on I/O-related system calls, then I/O might be the bottleneck. This kind of reasoning can be represented as a *decision tree* comprising of a sequence of if-then-else statements (Figure 6). Given the workload fingerprint and the target VM configuration, we follow the appropriate path down the tree, finally leading to a performance prediction. Note that a decision tree can make fairly complex, non-linear decisions.

[3]The reference VM types can also be chosen by the user

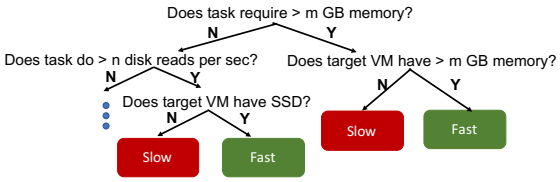


Figure 6: A possible decision tree for predicting performance from the task fingerprint and VM type configuration.

Manually specifying a decision tree for each workload would be prohibitively challenging. We therefore leverage the data collected from the extensive *offline* benchmarking phase in conjunction with established *random forest* algorithms to automatically train a *collection* of decision trees for each workload. Random forests extend the reasoning behind decision trees to a collection of trees to provide more robust predictions [18].

5.1 Training the Random Forest Model

To accurately predict the average and tail workload performance using the offline benchmark data we train a random forest model which approximates the function:

$$g(\texttt{fingerprint, target\_vm}) \rightarrow (\texttt{perf, p90})$$

To learn this function we transform the offline benchmarking dataset into a training dataset where each benchmark has a corresponding fingerprint and known mean and tail performance for all target VM types.

The fingerprint for each benchmark task is easily obtained by putting together the resource utilization counters collected while running the task on the reference VMs. Because we profile each benchmark on each VM type in the offline phase, these resource utilization counters are available irrespective of the choice of reference VM types. The target VM in our model is characterized by the VM configuration consisting of the number of cores (Azure) or vcpus (AWS), amount of memory, disk size, and network performance and bandwidth. Similarly, the offline benchmarking phase collected both mean and tail latencies for each benchmark which we use as the targets when training our model. We feed this training dataset to an off-the-shelf random forest training algorithm [52]. In our experiments, training a random forest predictor took less than 2 seconds in most cases. As an implementation detail, instead of predicting absolute performance, we predict the performance

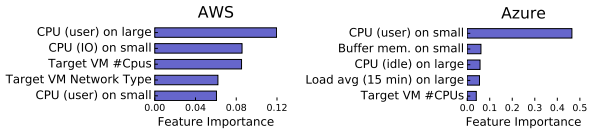


Figure 7: Importance of the various features for AWS (left) and Azure (right). The random forests were trained to predict runtime using a compression benchmark workload suite (See Section 6.2). Reference VMs used: `c3.large` and `r3.2xlarge` for AWS and `F2` and `D13v2` for Azure.

scaling relative to the first reference VM type. We found that this led to a simpler learning problem.

Invoking the performance predictors: Once the model builder has trained random forests for the performance metric of interest, for each candidate VM type j, we feed the user task fingerprint F and the VM configuration c^j as inputs to the two random forests. The random forests output the mean and 90^{th} percentile performance relative to the first reference VM. We get absolute performance by multiplying these predictions with the corresponding mean and 90^{th} percentile performance on the first VM type.

Performance-Cost Map: Finally, PARIS uses the performance predictions to also estimate the cost for each VM type. For this we assume that the cost is a function of the performance metric and the published cost per hour of the VM, that is either known (for standard performance metrics such as throughput or latency) or specified by the user as an additional argument in the call to PARIS. For example, for a serving-style workload where performance is measured by latency, then the total cost per request would be the latency times the published cost per hour.

PARIS' estimated performance-cost trade-off enables users to implement a high-level policy to pick a VM type for a given workload. For example, a policy could be to choose a VM type for a workload that has: (a) an estimated cost below a certain constraint C and (b) the best performance in the worst case. We specify the worst case performance with a high percentile execution time, such as 90^{th} percentile. An alternative policy might pick an "optimal" VM type that achieves the least cost and the highest predictable worst-case performance.

5.2 Interpreting the Learned Models

Figure 7 illustrates the top 5 features that the random forest considers important, for runtime prediction on AWS and Azure. Here feature importance is based on the intuition that the decision tree will make early splits based on the most informative features, and then gradually refine its prediction using less informative features. Thus important features are those which frequently appear near the top of the decision tree. We find that various measures of CPU usage and the number of CPUs in the target VM figure prominently, for both AWS and Azure. This makes sense, since in general the more CPUs, the more the compute available to the task. However, measures of memory usage and disk utilization are also important. Note that the actual features that are used to estimate performance will depend on the path taken down the tree, which in turn will be different for different workloads.

6 Evaluation

In this section we answer following questions:

1. **Prediction accuracy (Section 6.3):** How accurately does PARIS predict the mean and 90^{th} percentile values for different performance metrics?
2. **Robustness (Section 6.4):** Is PARIS robust to changes in (a) the number and choice of VM types (6.4.1, 6.4.2), (b) the benchmark workloads used in the offline profiling phase (6.4.5), and (c) the choice of modeling technique (regressor) (6.4.3, and 6.4.4)?
3. **Usefulness (Sections 6.5, 6.6):** (a) Can we convert PARIS' performance estimates into actionable information (6.5) that reduces cost (6.6)?

6.1 Baselines

No off-the-shelf approach exists for predicting the performance of arbitrary workloads on all VM types in the cloud. Often users defensively provision the most expensive VM type, but this can lead to excessive costs without performance gains (Sec. 2). Alternatively, exhaustively profiling the workload on every available VM type provides accurate performance estimates, but is prohibitively expensive.

Instead, we chose two baselines that are similar to PARIS in terms of user cost, use the published VM configurations intelligently, and correspond to what users might do given the available information and a tight budget:

(a) Baseline$_1$: To reduce the cost, a user might profile her workload on the "smallest" and "largest" VM types according to some resource, and then take *average* performance to be an estimate on an intermediate VM type. Concretely, suppose VM type 1 obtains performance, (for instance, runtime), p_1, and VM type 2 achieves performance p_2. Then for a target VM type, one might simply predict the performance to be $p_{target} = \frac{p_1+p_2}{2}$.

(b) Baseline$_2$: Instead of simply averaging the performance, Baseline$_2$ *interpolates* performance based on published configurations. Concretely, suppose VM type 1 has memory m_1 and gets performance p_1, and VM type 2 has memory m_2 and gets performance p_2. Then for a VM type offering memory m, one might simply predict the performance to be $p_{memory}(m) = p_1 + \frac{p_2-p_1}{m_2-m_1}(m-m_1)$. Since the user may not know which resource is important,

Workload	Operations	Example Application
D	Read latest: 95/5 reads/inserts	Status updates
B	Read mostly: 95/5 reads/writes	Photo tagging
A	Update heavy: 50/50 reads/writes	Recording user-actions

Table 1: Serving benchmark workloads we used from YCSB. We did not use the Read-Only Workload C, as our benchmark set covers read-mostly and read-latest workloads.

Workload	Number of tasks	Time (hours)
Cloud hosted compression (Benchmark set)	740	112
Cloud hosted video encoding (Query set)	12983	433
Serving-style YCSB workloads D,B, A (Benchmark set)	1830	2
Serving-style new YCSB workloads (Query set)	62494	436

Table 2: Details of the workloads used and Dataset collected for PARIS' offline and online phases.

she might do such linear interpolation for each resource and average the predictions together.

6.2 Experimental Set-up

We evaluated PARIS on AWS and Azure, using two widely recognized types of cloud workloads [14]: (a) Applications such as video encoding, and compression, and (b) Serving-style latency and throughput sensitive OLTP workloads.

Common cloud-hosted applications: Video encoding and compression are common use-cases of the cloud. We used the squash compression library [9], an abstraction layer for different compression algorithms that also has a large set of datasets. For a video encoding workload, we used libav [47], a set of open source audio and video processing tools. We set both of these applications in the cloud, to first download the relevant input data and then process it. The video encoding application first downloads a video using the specified URL, then converts it to a specified format using various frame-rates, codecs, and bit-rates. The compression workload downloads a compressed file, decompresses it, and re-compresses it using different compression algorithms. These workloads have different resource usage patterns. To show that PARIS can generalize well across workloads, we chose the compression application for the offline benchmarking and tested the models using the video encoding application (Table 2).

Serving-style workloads: We used four common cloud serving datastores: Aerospike, MongoDB, Redis, and Cassandra. These systems provide read and write access to the data, for tasks like serving a web page or querying a database. For querying these systems, we used multiple workloads from the YCSB framework [25]. We used the core workloads [11], which represent different mixes of read/write operations, request distributions, and datasizes. Table 1 shows the benchmark serving workloads we used in the offline phase of PARIS. For testing PARIS' models, we implemented new realistic serving workloads by varying the read/write/scan/insert proportions and request distribution, for a larger number of operations than the benchmark workloads [10].

Dataset details: Table 2 shows the number of tasks executed in the offline phase and the corresponding amount of time spent. Also shown are the workloads and the number of query tasks used for online evaluation.

Metrics for evaluating model-predictions: We use the same error metrics for our predictions of different performance metrics. We measured actual performance recorded by running a task on the different VM types, and computed the percentage RMSE (Root Mean Squared Error), relative to the actual performance:

$$\%Relative\ RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{p^i - a^i}{a^i}\right)^2} * 100$$

where N is the number of query tasks, and p^i and a^i are the predicted and actual performance of the task respectively, in terms of the user-specified metric. We want the % Relative RMSE to be as low as possible.

RMSE is a standard metric in regression, but is scale-dependent: an RMSE of 10 ms in runtime prediction is very bad if the true runtime is 1 ms, but is acceptable if the true runtime is 1000 ms. Expressing the error as a percentage of the actual value mitigates this issue.

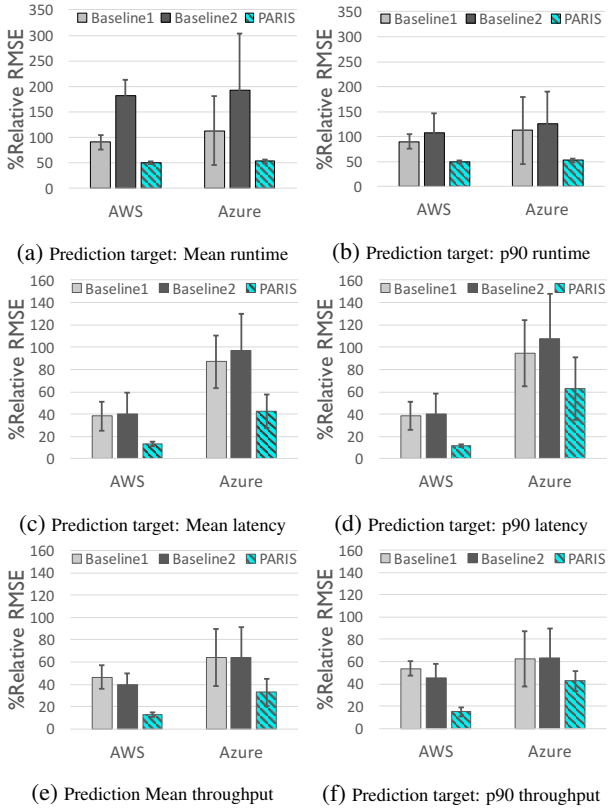
6.3 Prediction accuracy of PARIS

We first evaluate PARIS' prediction accuracy by comparing PARIS' predictions to the actual performance obtained by exhaustively running the same user-provided task on all VM types. We evaluated PARIS on both AWS and Azure for (a) Video encoding tasks using *runtime* as the target performance metric, and (b) serving-type OLTP workloads using *latency* and *throughput* as the performance metrics.

Overall Prediction Error: Figure 8 compares PARIS' predictions to those from Baseline$_1$ and Baseline$_2$ for the mean and 90^{th} percentile runtime, latency and throughput. Results are averaged across different choices of reference VMs, with standard deviations shown as error bars.

PARIS reduces errors by a factor of 2 compared to Baseline$_2$, and by a factor of 4 compared to baseline$_2$. Note that the cost of all three approaches is the same, corresponding to running the user task on a few reference VMs. This large reduction is because the nonlinear effects of resource availability on performance (such as hitting a memory wall) cannot be captured by linear interpolation (baseline$_2$) or averaging (baseline$_1$).

To better understand why Baseline$_2$ gets such a high error for some VM types, we looked at how predictions by Baseline$_2$ varied with the different resources of the target VMs (num CPUs, memory, disk). In one case, when using `m3.large` and `c4.2xlarge` as our reference VMs, we



(a) Prediction target: Mean runtime (b) Prediction target: p90 runtime

(c) Prediction target: Mean latency (d) Prediction target: p90 latency

(e) Prediction Mean throughput (f) Prediction target: p90 throughput

Figure 8: Prediction Error for Runtime, Latency, and Throughput (expected and p90) for AWS and Azure. a, b: Runtime prediction for video encoding workload tasks, c-f: Latency and throughput prediction for Serving-style latency and throughput sensitive OLTP workloads. The error bars show the standard deviation across different combinations of reference VMs used.

observed that surprisingly, Baseline$_2$ predicted *higher* runtimes for VM types with higher disk capacity. Why did the baseline latch on to this incorrect correlation? In this example, the larger reference VM we used, `c4.2xlarge`, offered lower runtimes than the smaller reference VM used, `m3.large`; however, the smaller reference VM had larger disk (32GB) than the larger reference VM.

This reveals a critical weakness of the baseline: from only the performance on two reference VMs and the published configurations, the baseline *cannot* know which resource is important for workload performance. PARIS, on the other hand, looks at the usage counters and might figure out that disk is not the bottleneck for this workload.

We also note that prediction errors are in general larger for Azure for latency and throughput prediction on the OLTP workloads. We surmise that this is probably due to the higher variability of performance on Azure instances for these workloads, which we pointed out in Section 2.

Prediction Error per VM-type: Figure 9 shows how the prediction error breaks down over different target VM.

Figure 9: Errors per target VM type for predicting mean latency, on AWS (top) and Azure (bottom). Reference VMs used on AWS: `m3.large` and `c4.2xlarge`, and on Azure: `A2` and `F8`. Error bars show the standard deviation in % Relative RMSE across the set of over 62K latency-sensitive YCSB queries run on Aerospike, Cassandra, MongoDb, and Redis data-stores.

This is a representative result, with mean latency as the performance metric. Reference VM types are `m3.large` and `c4.2xlarge` on AWS, and `A2` and `F8` on Azure. PARIS' errors are *consistently low* across VM types and much lower than both baselines.

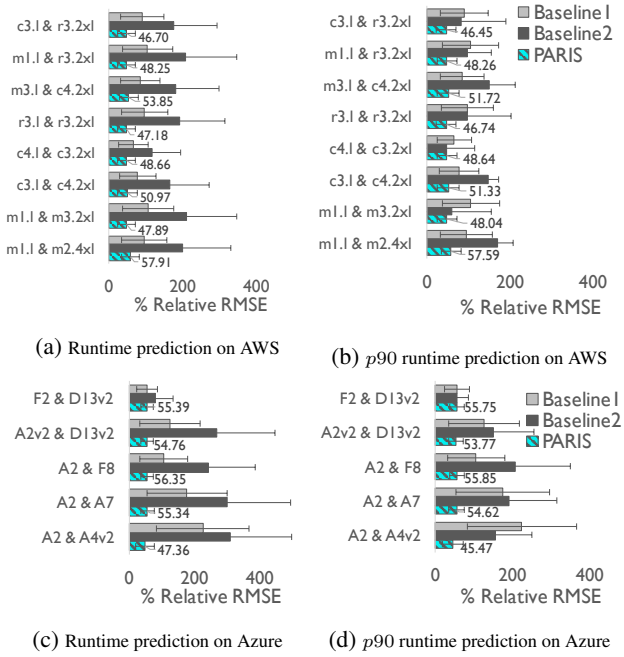
As before, Baseline$_2$ discovers spurious correlations and thus often performs worse than Baseline$_1$. Further, *both* baselines perform significantly worse than PARIS for most VM types, perhaps because they lack access to (i) PARIS' offline VM benchmarking data, and (ii) PARIS' resource-utilization statistics, collected in the offline phase as well as when fingerprinting in the online phase.

6.4 Robustness

6.4.1 Sensitivity to the choice of reference VM types

We experimented with several choices for the 2 reference VM types. We picked pairs of VM types that were the farthest apart in terms of a particular resource (number of cores, amount of memory, disk or storage bandwidth). We also experimented with randomly chosen reference VMs. In general, we found PARIS' predictors to be robust to the choice of reference VM types.

As a representative result, Figure 10 compares PARIS' mean and $p90$ runtime predictions to the baselines for several reference VM choices using the video encoding workload. PARIS is both more accurate and more consistent across different reference VM choices. Thus, *PARIS maintains accuracy irrespective of the choice of reference VM types*. The profiling information used by PARIS is



(a) Runtime prediction on AWS

(b) $p90$ runtime prediction on AWS

(c) Runtime prediction on Azure

(d) $p90$ runtime prediction on Azure

Figure 10: Sensitivity of PARIS to the choice of Reference VMs. Errors in predicting mean runtime and 90^{th} percentile runtime for video encoding tasks using different reference VM types on AWS (a and b) and Azure (c and d) (Sec. 6.4.1).

consistent and reliable, even when the performance on the two reference VM types is not informative. Further, separate predictors for each selection of reference VM types allow PARIS to learn how performance on the reference VM types extrapolates to other VM types. In contrast, with only the runtime on the two VMs and their published configurations to rely on, the baseline latches on to spurious correlations, and is thus inaccurate.

6.4.2 Sensitivity to number of reference VMs

We experimented with increasing the number of reference VMs from 2 (`m3.large` and `c4.2xlarge`) to 3 (`m3.large`, `c4.2xlarge` and `c3.large`) and 4 (`m3.large`, `c4.2xlarge`, `c3.large`, and `c3.xlarge`). We found that latency and throughput prediction error decreased slightly or remained the same as the number of reference VM types increased. % Relative RMSE for latency prediction remained around 9%, while for throughput prediction, it decreased from 11.21% with 2 reference VMs to 10.38% with 3 and 10.27% with 4.

Similarly on Azure, the latency prediction error dropped slightly from 22.89% with 2 reference VMs (`A2` and `A7`) to 21.85% with an additional reference VM (`D13v2`) and to 19.69% with a 4^{th} additional reference VM (`F2`). Throughput prediction error similarly decreased from 24.69% (2 reference VMs) to 18.56% and 18.21% respectively.

This indicates that PARIS is quite robust to the number of reference VM types and is able to make accurate predictions *with only 2 reference VM types*. This is because the profiling information used by PARIS is very informative.

6.4.3 Importance of the choice of regressor

Besides random forests, we also experimented with linear regression and decision trees for throughput and latency prediction on AWS (Figure 11). Similar patterns emerged using Azure VMs (not shown). Linear regression performs the worst as it isn't able to capture non-linear relationships between resource utilization and performance, but owing to the availability of resource usage counters still performs better than Baseline$_2$. Regression trees and forests perform equally better, but the forest provides better accuracy by combining complementary trees.

6.4.4 Sensitivity to random forest hyperparameters

Figure 12 shows the percentage relative RMSE of PARIS' latency and throughput predictors for different values of the two most important hyperparameters used by the random forest algorithm: (i) Number of features used per tree (NF), and (ii) Maximum depth of the trees (MD). The predictors for latency and throughput achieve comparable accuracies across the different values of NF and MD. This suggests that *the predictors are robust to hyperparameter choices*.

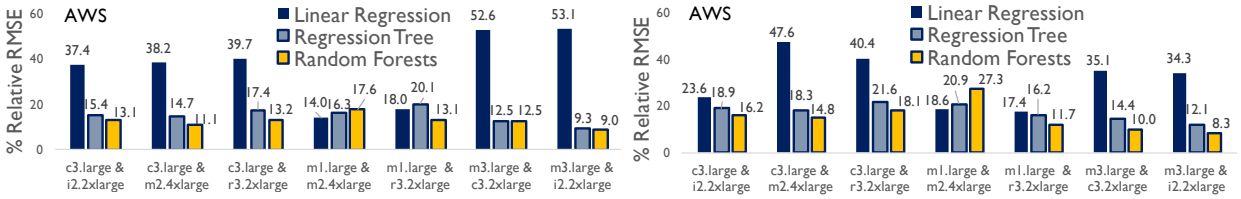
6.4.5 Sensitivity to benchmark workloads

Figure 13 shows the percentage relative RMSE of PARIS' latency and throughput predictors when one of the benchmark workloads is removed from the training set at a time. This figure shows the error averaged over different combinations of reference VM types and the error bars indicate the standard deviation. The predictors achieve comparable accuracy on removal of a benchmark workload. We observed a similar trend using the data on Azure for runtime, latency and throughput predictors of PARIS. This shows that *the predictors are robust to different choices of the benchmark workloads*.

6.5 From Estimated Performance to Action

PARIS presents its performance predictions as a *performance-cost trade-off map* that maps each VM type to the corresponding performance-cost trade-off, for a given user workload. We first qualitatively explain why we expect this map to be useful and then quantitatively show cost-savings in the next section.

Why common VM selection strategies fail: Without good estimates of performance or cost, users wanting to deploy workloads on the cloud might:



(a) Latency prediction

(b) Throughput prediction

Figure 11: Prediction errors for different regressors using different choices of reference VM types on AWS (Sec. 6.4.3)

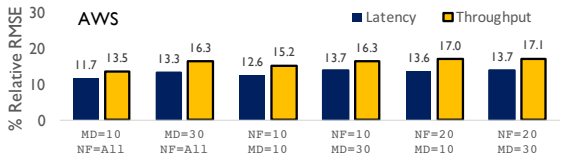


Figure 12: Error of PARIS' latency and throughput predictors for different random forest hyperparameters, for test YCSB workloads on Aerospike, MongoDB, Redis, and Cassandra datastores, on AWS. Reference VMs: c3.large & i2.2xlarge. (Sec. 6.4.4)

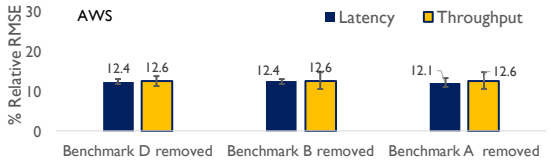


Figure 13: Error of PARIS' latency and throughput predictors by removing one benchmark workload at a time, averaged across reference VM types combinations on AWS (Sec. 6.4.5).

(a) Try to minimize cost by choosing the cheapest VM.
(b) Defensively choose a large enough VM, assuming 'the higher the cost, the better the performance', or
(c) Pick the largest VM cheaper than a cost constraint.

Figure 1 shows the actual performance and cost for a video encoding task on each VM type. Note that this information is unavailable to users unless they are willing to incur heavy profiling-costs. We can see that strategy a) would choose `m1.large`, and lead to higher costs and higher and less predictable runtimes, possibly violating SLOs: a bad decision. Strategy b) would select `m2.4xlarge` and keep runtime low and predictable but incur higher costs than an alternative such as `c2.2xlarge`, which also provides similar runtime. Strategy c), while reasonable, might still lead to sub-optimal choices like `m3.xlarge`, which offers worse performance than `c3.2xlarge` for higher cost. Choosing a VM from over a 100 types across multiple cloud providers is even harder.

How does PARIS help? PARIS generates a performance-cost trade-off map with predictions of mean and p90 values of performance according to the user-specified performance metric and tailored to a user-specified task that represents her workload. Figure 14 shows such a trade-off map with predicted latencies (top) and corresponding *task completion costs* for a representative task consisting of a set of 225K YCSB queries on a Redis data-store that with 225K records. The p90 values are shown as error bars. The X-axis has different AWS and Azure VM types in an increasing order of their cost-per-hour. The reference VMs were `A2` and `F8` for Azure and `c4.2xlarge` and `m3.large` for AWS.

The user can use this map to choose the best VM for any performance and cost goals, then run their entire workload on the chosen VM. The last plot in Figure 14 shows the true latencies observed when *all* query tasks from the user workload are run on each VM. PARIS' predictions match these true latencies well. As before, the latencies do not directly correlate with the published cost-per-hour of the VMs; `F2`, for instance, achieves lower latencies than `A4v2`. PARIS predicts these counterintuitive facts correctly.

6.6 Quantifying cost savings

PARIS offers users considerable flexibility in choosing their own performance and cost goals. The precise gains a user gets from PARIS will depend on these goals. Nevertheless, below we consider two example policies that the user might follow, and quantify cost savings for each.

6.6.1 Reduced user costs through better decisions

We performed this experiment on AWS, using YCSB-based serving workloads on Aerospike, MongoDB, Redis, and Cassandra data stores. We generated two performance-cost trade-off maps: one using predictions from PARIS and the other using baseline predictors. For each map, we chose a VM type for this workload using the policies described below, executed the workload on this VM type, and compared costs. We considered two example policies:

Policy I: Policy I picks the VM type with the least estimated cost provided the predicted runtime is less than a user-specified threshold, which is expressed as a fraction β of the mean predicted runtime across VM types.

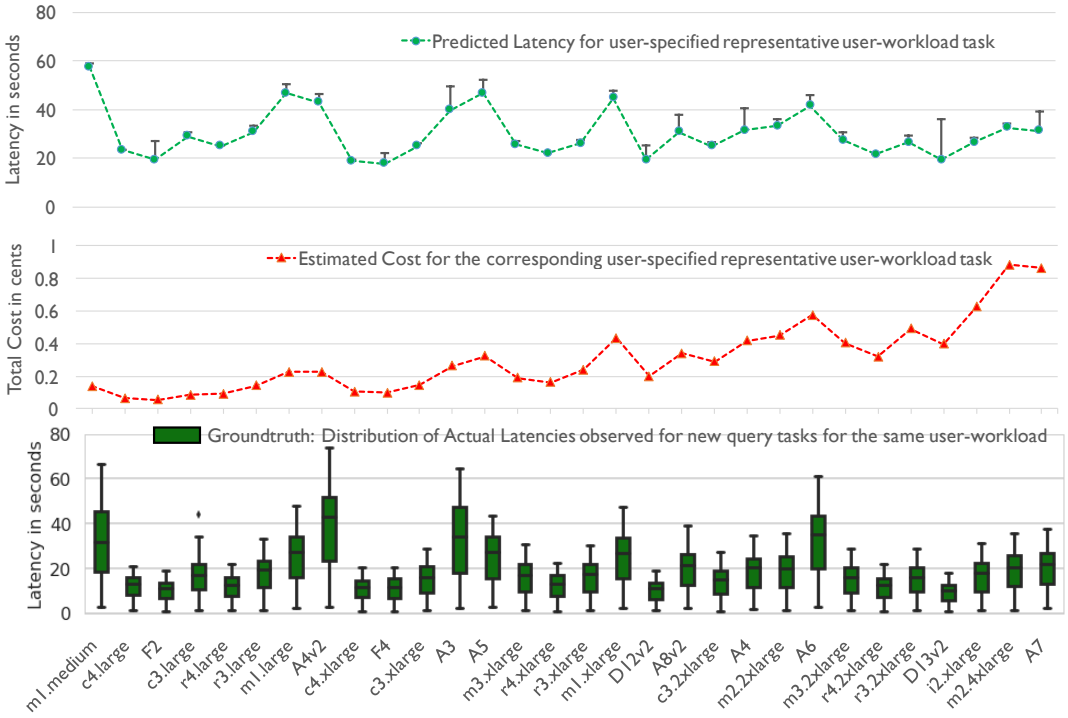


Figure 14: Performance-cost trade-off Map generated by PARIS using user-specified representative task that consisted of a set of 225K YCSB queries with a mix of 10/90 Reads/Writes, on a Redis data-store with 225K records. X-axis: AWS and Azure VM types ordered by increasing cost per hour. Reference VMs: `A2`, `F8` for Azure and `c4.2xlarge`, `m3.large` for AWS. **Top:** Predicted mean and p90 latencies (shown by whiskers). **Middle:** Estimated cost in cents for the representative task. **Bottom:** Distribution of actual observed latencies across different AWS and Azure VM types, for a set of 2.5K query user-tasks on Redis. (Sec. 6.5).

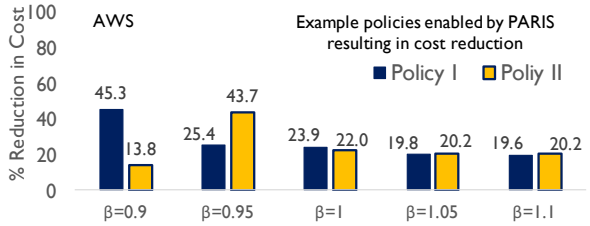


Figure 15: Percentage reduction in user costs enabled by PARIS' predictors over the baseline predictors on AWS for a number of policies. Policy I chooses the VM type with least predicted cost provided mean runtime $\leq \beta$ times the mean across VMs. Policy II is similar but thresholds $p90$ values instead. (Sec. 6.6.1).

Policy II: Instead of predicted runtime, Policy II uses predicted $p90$ to choose a VM type based on the same criterion. This policy optimizes for worst case performance.

We varied β in $[0.9, 1.1]$. As shown in Figure 15, the user can reduce costs by upto 45% by using performance and cost estimates from PARIS instead of the baseline.

6.6.2 Cost overheads of PARIS

PARIS does incur some limited overhead to produce the performance estimates. Part of this overhead is the *one-time* cost of offline benchmarking of VM types (see Table 2), which is amortized across all user workloads.The rest of the overhead is the cost of running a user-specified task on the reference VMs. As shown in Section 6.4.2, two reference VMs are enough for accurate predictions.

To quantify the cost overheads of PARIS empirically, we computed the cost of the offline VM benchmarking phase and the cost for fingerprinting each user-specified representative task in the online performance prediction phase. We compared this cost to the cost incurred by an alternative that exhaustively runs the task on each VM type to choose the right VM type. This alternative strategy is what would be followed by systems like Ernest [64] (Ernest also performs additional profiling to determine the number of VMs; this is not included in our comparison). Figure 16 shows this comparison for the mean and p90 latency prediction task using core YCSB queries A and B as train and a set of 50 newly implemented workloads as the user-specified representative tasks. For this experiment, we used the cost of the VMs per unit time published by the cloud providers. We note that PARIS has a non-zero

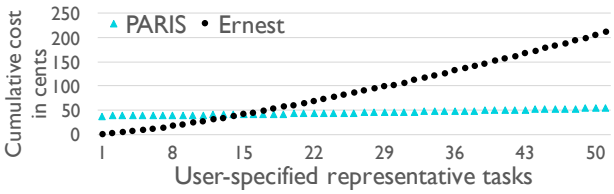


Figure 16: Cost overheads of PARIS compared to brute-force profiling on all VM types (e.g., in Ernest [64]).

initial cost due to the offline VM benchmarking phase, but once this phase is over, the additional cost of fingerprinting each new user-specified task is much lower than the cost of Ernest's exhaustive search. Ernest's cumulative cost grows at a much higher rate than PARIS' and overtakes the latter after about 15 tasks. PARIS is therefore lightweight.

7 Limitations and Next Steps

PARIS assumes the availability of a representative task from a user workload. Including task-specific features, such as input size, can enable generalization across tasks. While our current version requires separate fingerprints for each cloud provider, our modeling framework can be extended to include multiple providers. PARIS is not aimed at estimating scaling behavior, but can be combined with approaches such as Ernest [64] that tackle that problem. PARIS can also be extended to work with customizable VM sizes in the cloud, for instance, custom images by Google Cloud Engine [7].

8 Related Work

Both classical batch systems [28, 29, 13] and modern cluster management systems such as Amazon EC2 [2], Eucalyptus [48], Condor [53], Hadoop [69, 8, 70], Quincy [34], and Mesos [33, 31] need resource requirements from the user. By contrast, PARIS *does not need* knowledge of resource requirements and complements these systems.

Performance prediction based on system modeling: There is prior work on predicting performance based on system properties and workload patterns [21, 45, 49, 16].

Pseudoapp [58] creates a pseudo-application with the same set of distributed components and executes the same sequence of system calls as those of the real application. This assumes complete knowledge of what the real application is doing, which is often unavailable. Ernest [64] predicts the runtime of distributed analytics jobs as a function of cluster size. However, Ernest cannot infer the performance of new workloads on a VM type without first running the workload on that VM type. Quasar [27] tries to predict the performance impact of various resource allocation decisions on workload performance by extrapolating performance from a few profiling runs. This cannot capture the kind of detailed resource utilization information that is present in the workload fingerprints used by PARIS.

Interference Prediction: Interference is a major hindrance in accurate performance estimation. There is work on placing applications on particular resources to reduce interference, either by co-scheduling applications with disjoint resource requirements [59, 19, 56, 71, 73, 43, 44], or by trial and error[60, 41, 72]. However, users requesting VM types in cloud services like Amazon EC2 cannot usually control what applications get co-scheduled.

Prior work has used performance models to predict interference among applications [32, 66, 65, 36, 62, 57, 26, 27]. Some approaches rely on dynamically monitored hardware-level features, such as CPI (Cycles Per Instruction) or CMR (Cache Miss Rate) for interference prediction; however they aim to consolidate VMs on underlying physical machines [22, 39, 40]. Compared to these hardware-level counters, the 40 VM-level resource usage counters used by PARIS are both more informative and more easily available in public cloud environments.

Adaptive control systems: Instead of, or in addition to, predicting performance, some systems adaptively allocate resources based on feedback. For example, Rightscale [55] for EC2 creates additional VM instances when the load of an application crosses a threshold. Yarn [63] determines resource needs based on requests from the application. Other systems have explicit models to better inform the control system, e.g., [17, 30, 46].

Wrangler [68] identifies overloaded nodes in map-reduce clusters and delays scheduling jobs on them. Quasar [27] dynamically updates estimates of the sensitivity of an application's performance to heterogeneity, interference, scale-up and scale-out of resources. Unlike these systems, PARIS does not control online scheduling decisions, but can be used to inform the resource management system of the requirements for the application.

9 Conclusion

In this paper we presented PARIS, a system that allows users to choose the right VM type for their goals through accurate and economical performance estimation. PARIS decouples the characterization of VM types from the characterization of workloads, thus eliminating the $O(n^2)$ cost of performance estimation while delivering accurate performance predictions across VM types. We showed empirically that PARIS accurately predicts mean and tail performance for many realistic workloads and performance metrics across multiple clouds, and results in more cost effective decisions while meeting performance goals.

References

[1] Aerospike Datastore. https://www.aerospike.com.

[2] Amazon ec2. https://aws.amazon.com/ec2/.

[3] Aws customer success. https://aws.amazon.com/solutions/case-studies/.

[4] Aws lambda. https://aws.amazon.com/lambda/.

[5] Azure functions. https://azure.microsoft.com/en-us/services/functions/.

[6] Google cloud functions. https://cloud.google.com/functions/.

[7] Google cloud platform. https://cloud.google.com/compute/.

[8] Hadoop's Capacity Scheduler. http://hadoop.apache.org/core/docs/current/capacity_scheduler.html.

[9] Squash compression benchmark. https://quixdb.github.io/squash-benchmark/.

[10] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/wiki/Implementing-New-Workloads.

[11] Yahoo! Cloud Serving Benchmark . https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[12] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[13] G. Alverson, S. Kahan, R. Korry, C. Mccann, and B. Smith. Scheduling on the Tera MTA. In *In Job Scheduling Strategies for Parallel Processing*, pages 19–44. Springer-Verlag, 1995.

[14] Amazon.com. Amazon Web Services: Case Studies. https://aws.amazon.com/solutions/case-studies/.

[15] S. Bird. *Optimizing Resource Allocations for Dynamic Interactive Applications*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[16] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[17] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 1–6, New York, NY, USA, 2009. ACM.

[18] L. Breiman. Random forests. *Mach. Learn.*, Oct. 2001.

[19] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *ECRTS*, pages 194–204, 2009.

[20] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[21] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, 2004.

[22] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, W. Knottenbelt, and F. Franciosi. Cloudscope: Diagnosing performance interference for resource management in multi-tenant clouds. In *23rd IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, Atlanta, GA, USA, 10/2015 2015.

[23] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[24] M. Conley, A. Vahdat, and G. Porter. Achieving cost-efficient, data-intensive computing in the cloud. SoCC '15, 2015.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[26] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*, March 2013.

[27] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. ASPLOS '14, 2014.

[28] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems, 1997.

[29] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - a status report. In *JSSPP*, pages 1–16, 2004.

[30] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[31] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. NSDI'11, 2011.

[32] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. SOCC '11, pages 22:1–22:14, 2011.

[33] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[34] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, 2009. ACM.

[35] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. SoCC '12, 2012.

[36] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *ISPASS*, 2007.

[37] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), Apr. 2010.

[38] H. Li. *Introducing Windows Azure*. Apress, Berkely, CA, USA, 2009.

[39] A. K. Maji, S. Mitra, and S. Bagchi. Ice: An integrated configuration engine for interference mitigation in cloud services. In *ICAC*, 2015.

[40] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, 2014.

[41] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. MICRO-44, pages 248–259, 2011.

[42] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:2004, 2003.

[43] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. EuroSys '06, pages 403–414, 2006.

[44] A. Merkel and F. Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In *Proc. Eurosys '08*, pages 1–12, New York, NY, USA, 2008.

[45] K. Morton, M. Balazinska, and D. Grossman. Paratimer: A progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518, 2010.

[46] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, 2010.

[47] J. Newmarch. *FFmpeg/Libav*, pages 227–234. Apress, Berkeley, CA, 2017.

[48] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. CCGRID '09, pages 124–131, 2009.

[49] R. U. D. of Computer Science, V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. *A Static Performance Estimator to Guide Data Partitioning Decisions*. Number 136. Rice University, Department of Computer Science, 1990.

[50] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. HotCloud'12, 2012.

[51] K. Ousterhout, C. Canel, M. Wolffe, S. Ratnasamy, and S. Shenker. Performance clarity as a first-class design principle. In *16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, 2017.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[53] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2), Apr. 1999.

[54] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. SoCC '12.

[55] I. Rightscale. Amazon EC2: Rightscale. `http://aws.amazon.com/solution-providers/isv/rightscale`.

[56] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. *SIGOPS Oper. Syst. Rev.*, 42(2):189–200, 2008.

[57] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. EuroSys '07, pages 31–44, 2007.

[58] B. Tak, C. Tang, H. Huang, and L. Wang. Pseudoapp: Performance prediction for application migration to cloud. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013*, pages 303–310, 2013.

[59] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07*, 2007.

[60] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. ISCA '11, pages 283–294, 2011.

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

[62] N. Vasic, D. M. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *ASPLOS*, pages 423–436, 2012.

[63] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. SOCC '13.

[64] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, Mar. 2016. USENIX Association.

[65] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. ICS '08, pages 175–184, 2008.

[66] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev.*, 44(4):19–29, Dec. 2010.

[67] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. NSDI'12, 2012.

[68] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. SoCC '14, 2014.

[69] M. Zaharia. The Hadoop Fair Scheduler. `http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt`.

[70] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[71] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *HOTOS'07*, pages 1–6, 2007.

[72] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: Experiment-based management of virtualized data centers. USENIX'09, pages 18–18, 2009.

[73] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. ASPLOS XV, pages 129–142, 2010.