DejaVu: Accelerating Resource Allocation in Virtualized Environments

Nedeljko Vasić, Dejan Novaković, Svetozar Miučin *, Dejan Kostić, and Ricardo Bianchini†

School of Computer and Communication Sciences
EPFL, Switzerland
{firstname.lastname}@epfl.ch

†Department of Computer Science
Rutgers University, USA
ricardob@cs.rutgers.edu

Abstract

Effective resource management of virtualized environments is a challenging task. State-of-the-art management systems either rely on analytical models or evaluate resource allocations by running actual experiments. However, both approaches incur a significant overhead once the workload changes. The former needs to re-calibrate and re-validate models, whereas the latter has to run a new set of experiments to select a new resource allocation. During the adaptation period, the system may run with an inefficient configuration.

In this paper, we propose DejaVu – a framework that (1) minimizes the resource management overhead by identifying a small set of workload classes for which it needs to evaluate resource allocation decisions, (2) quickly adapts to workload changes by classifying workloads using signatures and caching their preferred resource allocations at runtime, and (3) deals with interference by estimating an "interference index". We evaluate DejaVu by running representative network services on Amazon EC2. DejaVu achieves more than 10x speedup in adaptation time for each workload change relative to the state-of-the-art. By enabling quick adaptation, DejaVu saves up to 60% of the service provisioning cost. Finally, DejaVu is easily deployable as it does not require any extensive instrumentation or human intervention.

Categories and Subject Descriptors D.4.8 [*Operating Systems*]: Performance; K.6.4 [*Management of Computing and Information Systems*]: System Management

General Terms Design, Measurement, Performance

Keywords Resource management, Data center, Virtualization

1. Introduction

Cloud computing is rapidly growing in popularity and importance, as an increasing number of enterprises and individuals have been offloading their workloads to cloud service providers, such as Amazon, Microsoft, IBM, and Google. One of the main reasons for the proliferation of cloud services is virtualization technology. Virtualization (1) enables providers to easily package and identify each customer's application into one or more virtual machines (VMs); (2) allows providers to lower operating costs by multiplexing their physical machines (PMs) across many VMs; and (3) simplifies VM placement and migration across PMs.

However, effective management of virtualized resources is a challenging task for providers, as it often involves selecting the best resource allocation out of a large number of alternatives. Moreover, evaluating each such allocation requires assessing its potential performance, availability, and energy consumption implications. To make matters worse, the workload of certain applications varies over time, requiring the resource allocations to be re-evaluated and possibly changed dynamically. For example, the workload of network services may vary in terms of the request rate and the resource requirements of the request mix.

A service that is provisioned with an inadequate number of resources can be problematic in two ways. If the service is over-provisioned, the provider wastes money. If the service is under-provisioned, its performance may violate a service-level objective (SLO). As an illustration of the impact of such an SLO violation, Amazon reports that it loses 1% of sales for an increase of 100 ms in response latency [15]. Thus, it is very important that the service is adequately provisioned.

Given these problems, automated resource managers or the system administrators themselves must be able to evaluate many possible resource allocations quickly and accurately. Both analytical modeling and experimentation have been proposed for evaluating allocations in similar datacenter settings [10, 12, 14, 18, 30, 33, 36, 37, 40, 42, 43]. Unfortunately, these techniques may require substantial time. Although modeling enables a large number of allocations to be quickly evaluated, it also typically requires time-consuming (and often manual) re-calibration and re-validation whenever workloads change appreciably. In contrast, sandboxed experimentation can be more accurate than modeling, but requires executions that are long enough to produce representative results. For example, [42] suggests that each experiment may require minutes to execute. Finally, experimenting with resource allocations on-line, via simple heuristics and/or feedback control [2, 8, 21, 28, 38, 41], has the additional limitation that any tentative allocations are exposed to users.

This paper addresses this set of problems by proposing DejaVu, a system that simplifies and accelerates the management of virtualized resources in cloud computing services. The key idea behind DejaVu is to cache and reuse the results of previous resource allocation decisions. When the DejaVu framework detects that workload conditions have changed (perhaps because a VM or service is

* Work done during this author's internship at EPFL.

ASPLOS'12, March 3–7, 2012, London, England, UK.

not achieving its desired performance), it can lookup the DejaVu cache, each time using a VM identification and a *workload signature*. The signature is an automatically determined, pre-defined vector of metrics describing the workload characteristics, and the VM's current resource utilization. To enable the cache lookups, DejaVu automatically constructs a classifier that uses off-the-shelf machine learning techniques. The classifier operates on workload clusters that are determined after an initial learning phase. DejaVu clustering has a positive effect on reducing the overall resource management effort and overhead, because it reduces the number of invocations of the tuning process (one per cluster).

The resource manager can use the output of DejaVu to quickly reallocate resources. The manager only needs to resort to time-consuming modeling, sandboxed experimentation, or on-line experimentation when no previous workload exercises the affected VMs in the same way. When the manager does have to produce a new optimized resource allocation using one of these methods, it stores the allocation into the DejaVu cache for later use.

Like any other cache, DejaVu is most useful when its cached allocations can be repeatedly reused. Although DejaVu can be used successfully in a variety of environments, in this paper we focus on cloud computing providers that run collections of network services (these are also known as Web hosting providers). Previous works and our own experience suggest that DejaVu should achieve high "hit rates" in this environment. For example, it is well-known that the load intensity of network services follows a repeating daily pattern, with lower request rates on weekend days. In addition, these services use multiple VMs that implement the same functionality and experience roughly the same workload (e.g., all the application servers of a 3-tier network service).

Our approach to dealing with performance interference on the virtualized hosting platform recognizes the difficulty of pinpointing the cause of interference, and the inability of cloud users to change the hosting platform itself to eliminate interference. DejaVu uses a pragmatic approach in which it probes for interference and adjusts to it by provisioning the service with more resources.

The contributions of this paper are as follows:

1. We propose DejaVu, a framework for learning and reusing optimized VM resource allocations.
2. We describe a technique for automatically profiling, clustering, and classifying workloads. Clustering reduces the number of tuning instances and thus reduces the overall resource management overhead.
3. We evaluate DejaVu using realistic network services and real-world MSN messenger and HotMail traces. Our results show that DejaVu achieves more than 10x speedup in adaptation time for each workload change, relative to the state-of-the-art. Further, DejaVu saves between 35-45% and 55-60% of the provisioning cost when scaling up and scaling out, respectively, as compared to the approach that always overprovisions the service to ensure the SLO is met. The DejaVu-achieved savings translate to about $250,000 per year for 100 large EC2 instances.

We conclude that deploying DejaVu in the field would have two key benefits. First, it would enable cloud providers to meet their SLOs more efficiently as workloads change. It would also enable providers to lower their energy costs (e.g., by consolidating workloads on fewer machines, more machines can enter a low-power state [8, 22, 39]). Second, the more efficient adaptation to workload changes would enable users to purchase fewer resources from the provider. In addition, the lower provider costs would likely translate into savings for users as well.

2. Background and Motivation

In this section, we briefly describe the background for our work, and demonstrate the need for DejaVu.

2.1 Background

We assume that the user[1] of the virtualized environment deploys her service across a pool of virtualized servers. We use the term application to denote a standalone application or a service component running within a guest operating system in a virtual machine (VM). The service itself is mapped to a number of VMs. A typical example would be a 3-tier architecture which consists of a web server, an application server, and a database server component. All VMs reserved for a particular component can be hosted by a single physical server, or distributed across a number of them. The user and the provider agree on the Service Level Objective (SLO) for the deployed service.

While DejaVu is not restricted to any particular virtualized platform, we evaluate it using Amazon's Elastic Computing Cloud (EC2) platform. EC2 offers two mechanisms for dynamic resource provisioning, namely *horizontal* and *vertical* scaling. While horizontal scaling (scaling out) lets users quickly extend their capacities with new virtual instances, vertical scaling (scaling up) varies resources assigned to a single VM. EC2 provides many server instance types, from small to extra large, which differ in available computing units, memory and I/O performance. We evaluate DejaVu with both provisioning schemes in mind.

2.2 The Case for DejaVu

The key issue in resource provisioning is to come up with the sufficient, but not wasteful, set of virtualized resources (e.g., number of virtual CPU cores and memory size) that enable the application to meet its SLO. Resource provisioning is challenging due to: (1) workload dynamics, (2) the difficulty and overhead of deriving the resource allocation for each workload, and (3) the difficulty in enforcing the resource planning decisions due to interference. As a result, it is difficult to determine the resource allocation that will achieve the desired performance while minimizing the cost for both the cloud provider and the user.

The most important problem is that the search space of allocation parameters is very large and makes the optimal configuration hard-to-find. Moreover, the workload can change and render this computed setting sub-optimal. This in turn results in underperforming services or resource waste.

Once they detect changes in the workload, the existing approaches for dynamic resource allocation re-run time-consuming modeling and validation, sandboxed experimentation, or on-line experimentation to evaluate different resource allocations. Moreover, on-line experimentation approaches (including feedback control) adjust the resource allocation incrementally, which leads to long convergence times. The convergence problem becomes even worse when new servers are added to or removed from the service. Adding servers involves long boot and warm-up times, whereas removing servers may cause other servers to spend significant time rebalancing load (as in Casandra [1]).

The impact of the state-of-the-art online adaptation on performance is illustrated by our experiment using RUBiS [6] (an eBay clone) in which we change the workload volume every 10 minutes. Further, to approximate the diurnal variation of load in a datacenter, we vary the load according to a sine-wave. As shown in Figure 1, even if the workload follows a recurring pattern, the existing approaches are forced to repeatedly run the tuning process since

[1] We use the terms "user" and "tenant" interchangeably. We reserve the term "client" for the client of the deployed service itself.

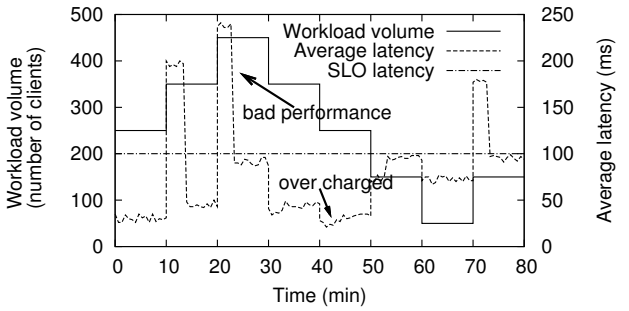


Figure 1. Every time the workload changes, the state-of-the-art approaches spend a considerable amount of time in performance retuning. During this time, the service can deliver insufficient performance due to a lack of resources. Alternatively, the service can be overprovisioned, and ultimately waste resources.

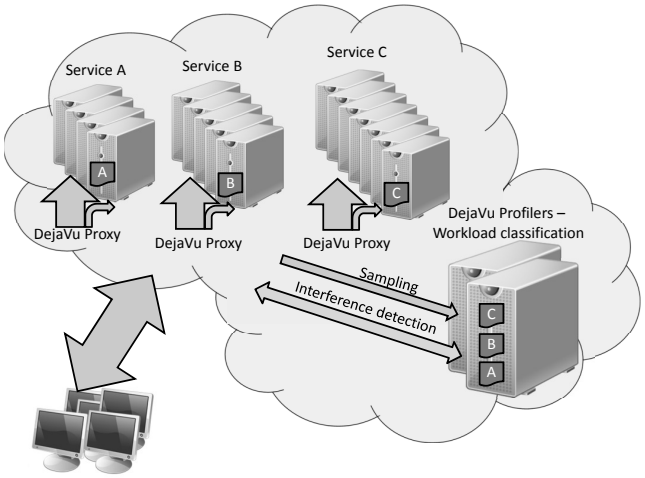


Figure 2. High-level overview of the way DejaVu's integrates with the service running in the cloud.

they cannot detect the similarity in the workload they are encountering. Unfortunately, this means that the hosted service is repeatedly running for long periods of time under a suboptimal resource allocation. Such magnitude of response latency increase (100 ms) has substantial impact on the revenue of the service [15]. Finally, computing the optimal resource allocation is an expensive task.

When faced with such long periods of unsatisfactory performance, the users might have to resort to overprovisioning, e.g., by using a large resource cap that can ensure satisfactory performance at foreseeable peaks in the demand. For the user, doing so incurs unnecessarily high deployment costs. For the provider, this causes high operating cost (e.g., due to excess energy for running and cooling the system). In summary, overprovisioning negates one of the primary reasons for the attractiveness of virtualized services for both users and providers.

Another problem with existing resource allocation approaches is that virtualization platforms do not provide ideal performance isolation, especially in the context of hardware caches and I/O. This implies that application performance may suffer due to the activities of the other virtual machines co-located on the same physical server. For instance, previous works and our own experience suggest that, due to the interference, even virtual instances of the same type might have very different performance over time.

3. Approach

In this section, we describe the DejaVu framework, starting with its high-level overview.

3.1 Overview

DejaVu operates alongside the services deployed in the virtualized environment of a cloud provider. Although other organizations are conceivable, we assume that the cloud provider itself deploys and operates DejaVu. Figure 2 highlights DejaVu's main components and the way they integrate with the cloud provider, while Figure 3 illustrates its operation. DejaVu accelerates the management of virtualized resources in datacenters by caching the results of past resource allocation decisions and quickly reusing them once it faces the same or a similar workload. For DejaVu to be effective in dealing with dynamic workloads, it first needs to learn about workloads and their associated resource allocations (e.g., the number and size of the required virtualized instances on EC2) during the learning phase (e.g., a week of service use).

To profile a workload, DejaVu deploys a proxy that duplicates the client requests directed to selected service VM instances to the DejaVu profiler. DejaVu then uses a dedicated profiling machine to compute a *workload signature* for each encountered workload. The workload signature itself is a set of automatically chosen low-level metrics. Further, DejaVu *clusters* the encountered workloads, and by doing so it reduces the resource management overhead, as well as the number of potentially time-consuming service reconfigurations to accommodate more or fewer virtual machines. The *Tuner* maps the clusters to the available virtualized resources, and populates the resource allocation repository.

After the initial learning phase, DejaVu profiles the workload periodically or on-demand (e.g., upon a violation of an SLO) using its proxy. It then uses each computed workload signature to automatically *classify* the encountered workload. If the classifier points to a previously seen workload (cache hit), DejaVu quickly reuses the previously computed resource allocation. In case of a different resource allocation, DejaVu instructs the service to reconfigure itself. In case of a failure to classify the workload (e.g., due to an unforeseen increase in service volume), DejaVu can either reinvoke the Tuner, or instruct the service to deploy its full capacity configuration. Compared to the state-of-the-art, DejaVu drastically reduces the time during which the service is running with inadequate resources. This translates to fewer and shorter SLO violations, as well as significant cost reduction for running the service itself.

To deal with interference from co-located workloads, DejaVu computes an *interference index* by contrasting the performance of the service on the DejaVu profiler with that in the production environment. It then stores this information in the resource allocation repository. Simply put, this information tells DejaVu how many more resources it needs to request to have a better probabilistic guarantee on the service performance. Using the historically collected interference information once again allows DejaVu to reduce the tuning overhead relative to the interference-oblivious case (state-of-the-art).

3.2 Workload Dispatching and Profiling

At a high level (Figure 2), DejaVu consists of two main components: *proxy* and *workload profiler*. To profile workloads under real-world conditions and traces, we introduce a proxy between the clients and the hosted services. The proxy forwards the client request to the system in production, but also duplicates and sends a certain fraction of the requests to the profiling environment for workload characterization.

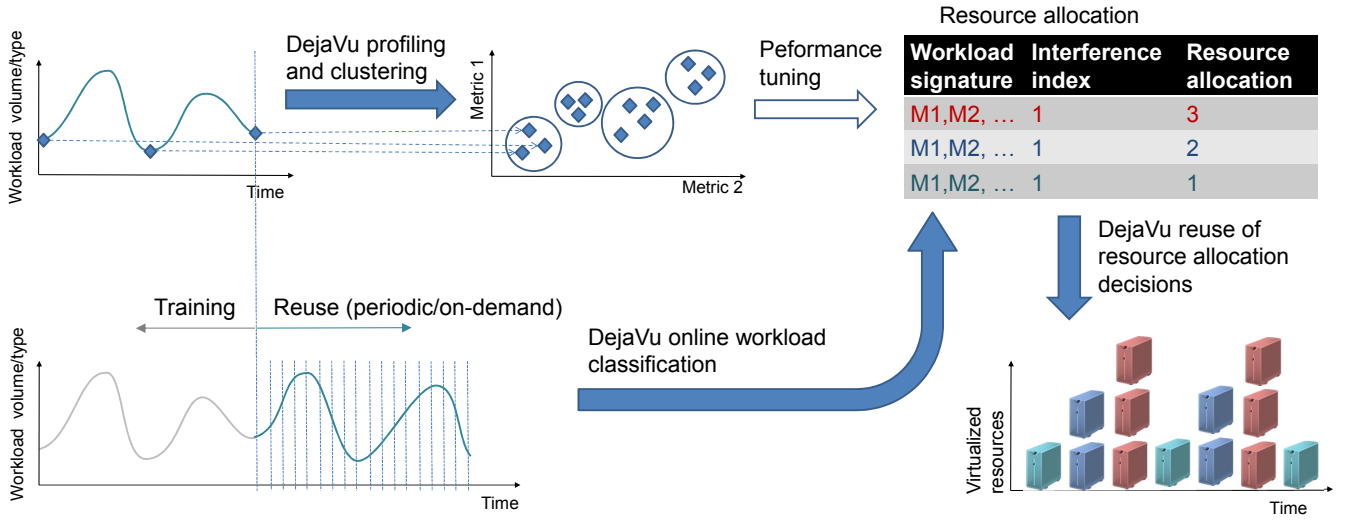


Figure 3. High-level overview of DejaVu's operation. It first profiles and clusters a dynamic workload during the learning phase. State-of-the-art performance tuning then maps the clusters to virtualized resources. Finally, DejaVu profiles workloads at runtime and reuses previous resource allocation decisions to allow the service to quickly adapt to workload changes.

3.2.1 DejaVu Proxy

The proxy needs to be careful when selecting the requests for profiling. In the case of Internet services, the sampling is done at the granularity of the client session to avoid issues with non-existent web cookies that might cause workload anomalies. Other types of applications may require more sophisticated approaches. For example, a distributed key-value storage system (e.g., Cassandra [1]) requires sampling where the dispatching needs to be aware of the node partitioning scheme, and duplicate only the requests with keys that belong to the particular server instance used for workload characterization.

Having a proxy between the service in production and the testing environment (in our case the profiling environment) has been addressed before [42]. The authors typically propose application protocol-aware proxies, ending up with numerous implementations that understand HTTP, mod-jk, jdbc, etc. In contrast, our workload characterization targets an arbitrary service; this in turn poses a need for a general proxy that can work with any service. Hence, we propose a novel proxy which which sits between the application and transport layers.

The proxy duplicates incoming network traffic (all the requests) of the server instance that DejaVu intends to profile, and forwards it to the clone. By doing so, DejaVu ensures that the clone VM serves the same requests as the profiled instance, resulting in the same or similar behavior. Finally, to make the profiling process transparent to the other nodes in the cluster, the clone's replies are dropped by the profiler. To avoid instrumentation of the service (e.g., changing the listening ports), we transparently redirect incoming traffic to the DejaVu proxy using iptables routines [19].

It is particularly hard to make the profiler behave just like the production instance in multi-tier services. For example, consider a three-tier architecture. In this architecture, it is common for the front-end web server to invoke the application server which then talks to the database server before replying back to the front-end. In this case, if DejaVu is instructed to profile only the application server (the middle tier), it is obvious that we need to deal with the absence of the database server.

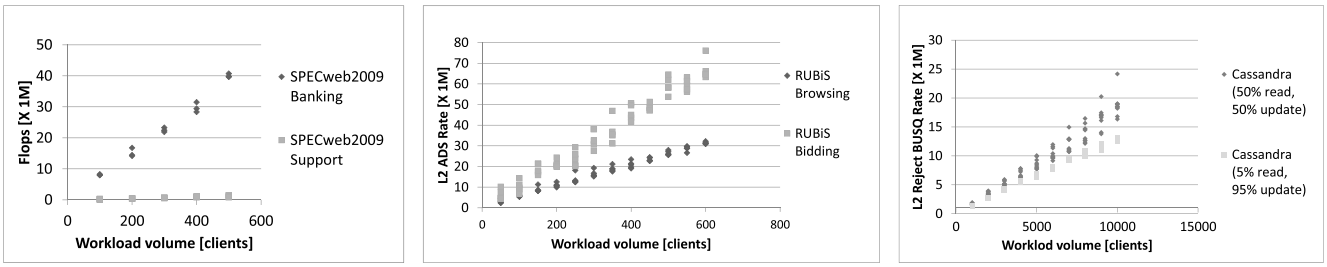
DejaVu addresses this challenge by having its proxy cache recent answers from the database such that they can be re-used by the profiler. Requests coming from the clone are also sent to the proxy. Upon receiving a request from the profiler, the proxy computes its hash and mimics the existence of the database by looking up the most recent answer for the given hash. Note that the proxy's lookup table exhibits good locality since both the production and the profiler deal with the same requests, only slightly shifted in time as one of these two might be running faster. This caching scheme does not necessarily produce the exact same behavior as in the production system because the proxy can: (1) miss some answers due to minor request permutations (i.e. the profiler and the production instance generate different timestamps), or (2) feed the profiler with obsolete data. However, the scheme still generates the load on the profiler that is similar to that of the production system (recall that DejaVu does not need a verbatim copy of the production system).

3.2.2 DejaVu Profiler

During the workload characterization process, DejaVu's profiler serves realistic requests (sent by the proxy) in the profiling environment, allowing us to collect all the metrics required by the characterization process, without interfering with the system in production. DejaVu relies on VM cloning to reconstruct a subset of the monitored service that can serve the sampled requests, but makes sure that VM clones have different network identities (recall that the clone is running in our private profiling environment). To minimize the cloning cost, DejaVu profiles only a subset of the service, typically a single server instance (e.g., one VM per tier) and assumes that services balance the load evenly across its server instances.

The services with little or no state are quickly brought to an equivalent operational state to that of the system in production. In contrast, replicating a database instance might be time-consuming, and it is important to consider the cloning of the disk storage [7, 20]. However, our goal is not to exactly capture the service's complex behavior and resulting performance, but only to label the current workload which gives us additional flexibility – our VM clone does not need to be tightly synchronized with the system in production. Instead, we envision periodic synchronization and resort to the current image, as long as DejaVu manages to identify the minimal set of resources that enable the service to meet its SLO.

To avoid the VM cloning and DejaVu overhead altogether, one can perform profiling on-line, without cloning the monitored VM.



(a) SPECweb2009 (b) RUBiS (c) Cassandra

Figure 4. Low-level metrics can serve as a signature which reliably identifies workloads that differ either in their type (i.e., read/write ratio) or intensity.

This again comes with two major obstacles. First, some metrics might be disturbed by co-located tenants during the sampling period; hence one would need to carefully identify the signature-forming metrics that are immune to interference. The second obstacle is that the cloud provider would need to make all low-level metrics available for our profiling.

3.3 Choosing the Workload Signature

For any metric-based workload recognition, it is crucial that the set of metrics chosen as the *workload signature* can uniquely identify all types of workload behaviors. Before going into the details of our workload signature selection process, we discuss how we collect the workload-describing metrics.

The DejaVu framework relies on its *Monitor* to periodically or on-demand (e.g., upon a violation of an SLO) collect the workload signatures. The design of the Monitor includes several key challenges:

- *Non-intrusive monitoring.* Given the diverse set of applications that might run in the cloud, DejaVu cannot rely on any prior service knowledge, semantics, implementation details, or highly-specific logs. Further, DejaVu assumes that it has to work well without having any control over the guest VMs or applications running inside them. This is a desirable constraint given that we target hosting environments like Amazon's EC2 that provides only a "bare-bones" virtual server.
- *Isolation.* Because the DejaVu profiler (possibly running on a single machine) might be in charge of characterizing multiple services, we need to make sure that the obtained signatures are not disturbed by other profiling processes running on the same profiler.
- *Small overhead.* Since DejaVu might be running all the time, its proxy must induce negligible overhead while duplicating client requests, to avoid an impact on application performance.

Using low-level metrics to capture the workload behavior is attractive as it allows us to uniquely identify different workloads without requiring knowledge about the deployed service. The virtualization platforms are already equipped with various monitoring tools that are useful to us. For instance, Xen's `xentop` command reports individual VM resource consumption (CPU, memory, and I/O). Further, modern processors usually have a set of special registers that allow monitoring of performance counters without affecting the code that is currently running. It has been shown that these *Hardware Performance Counters (HPC)* can be used for workload anomaly detection [17] and online request recognition [29]. In addition, the HPC statistics can conveniently be obtained without instrumenting the guest VM. We only read a hardware counter value before a VM is scheduled, and right after it is preempted. The difference between the two gives us the exact number of events for which the VM should be "charged". Tools such as `Xenoprof` already provide this functionality with passive sampling.

Leveraging low-level metrics brings another practical question: given the complexity of the hosted services, can we rely on these metrics as a reliable signature to distinguish different workloads? We assume that as long as a relevant counter value lies in a certain interval, the current workload belongs to the class associated with the interval.

To validate this assumption in practice, we run experiments with realistic applications. In particular, we run typical cloud benchmarks under different load volumes, with 5 trials for each volume. Figures 4(a) - 4(c) present the results with each point representing a different trial. In the most obvious example, Figure 4(a) clearly shows that the hardware metric (Flops rate in this case) can reliably differentiate the incoming workloads. Moreover, the results for each load volume are very close. Once we change either workload type (e.g., read/write ratio) or intensity, a large gap between counter values appear. Similar trends are seen in the other benchmarks as well, but with a bit more noise. Nevertheless, the remaining metrics that belong to the signature (we are plotting only a single counter for each benchmark) typically eliminate the impact of noise.

While we can choose an arbitrary number of xentop-reported metrics to serve as the workload signature, the number of HPC-based metrics is limited in practice – for instance, our profiling server, Intel Xeon X5472, has only four registers that allow monitoring of HPCs, with up to 60 different events that can be monitored. It is possible to monitor a large number of events using time-division multiplexing, but this causes a loss in accuracy [16]. Moreover, many of these events are not very useful for workload characterization, as they provide little or no value when comparing workloads. Finally, we can reduce the dimensionality of the ensuing classification problem and significantly speed up the process by selecting only a subset of relevant events.

The task at hand is a typical feature selection process, which evaluates the effect of selected features on classification accuracy. The problem has been investigated for years, resulting in a large number of machine learning techniques. As our focus is not on inventing new models, we simply apply various mature methods from the WEKA machine learning package [13] on our datasets obtained from profiling (Section 3.4). During this phase, we form the dataset by collecting all HPC and xentop-reported metric values.

Applying different techniques on our dataset, we note that the `CfsSubsetEval` technique, in collaboration with the `GreedyStepWise` search, results in high classification accuracy. The technique evaluates each attribute individually, but also observes the degree of redundancy among them internally to prevent undesirable overlap. As a result, we derive a set of N representative HPCs

and xentop-reported metrics, which serve as the workload signature (WS) in the form of an ordered N-tuple:

$$WS = \{m^1, m^2, ..., m^N\} \quad (1)$$

where m^i represents the metric i. We further analyze the feature selection process by manually inspecting the chosen counters. For instance, the HPC counters chosen to serve as the workload signature in case of the RUBiS workload are depicted in Table 1 (the xentop metrics are excluded from the table). Indeed, the signature metrics provide performance information related to CPU, cache, memory, and the bus queue.

Given that the selection process is data-driven, the metrics forming the workload signatures are *application-dependent*. We however do not view this as an issue since the *metric selection process is fully automated* and transparent to the user.

To ensure that our workload signature is robust to arbitrary sampling duration, we normalize the values with the sampling time. This is important as it allows us to generalize our signatures across workloads regardless of how long the sampling takes.

3.4 Identifying Workload Classes

Given that the majority of network services follow a repeating daily pattern, DejaVu should achieve high "cache hit rates". But still a challenge remains. To achieve high hit rates, DejaVu first needs to populate preferred resource allocations for representative workloads, i.e. those workloads that will most likely reoccur in the near future and result in a cache hit.

Note that there is a tradeoff between the overhead of adjusting resource allocations (tuning) and the achieved hit rates. One can achieve high hit rates by naively marking every workload as representative. However, this would cause DejaVu to perform costly tuning for too many workloads. On the other hand, omitting some important workloads could lead to unacceptable resource allocations during certain periods.

DejaVu addresses this tradeoff by automatically identifying a small set of workload classes for a service. First, it monitors the service for a certain period (e.g., a day or week) until the administrator decides that DejaVu has seen most, or ideally all, workloads. During this initial profiling phase, DejaVu collects the low-level metrics discussed in Section 3.3. Then, it analyzes the dataset to identify workload signatures, and represent each workload as a point in N-dimensional space (N is the number of metrics in the signature). Finally, DejaVu clusters workloads into classes.

DejaVu leverages a standard clustering technique, simple k means, to produce a set of workload classes for which the Tuner needs to obtain the resource allocations. The framework can automatically determine the number of classes, as we did in our experiments, but also allows the administrators to explicitly strike the appropriate tradeoff between the tuning overhead and hit rate. As an example, Figure 5 shows the representative workload classes that we obtain from a service after replaying the day-long Microsoft HotMail trace [35]. Each workload is projected onto the two-dimensional space for clarity. DejaVu collected a set of 24 workloads (an instance per hour), and it identified only four different workload classes for which it has to perform the tuning. For instance, a workload class holding a single workload (the top right corner) stands for the peak hour.

DejaVu assumes that the workload classes obtained in the profiling environment are also relevant for the production system. This does not mean that the counter values reported by the profiler need to be comparable to corresponding values seen by the service in production. This would be too strong of an assumption, as DejaVu would then have to make the profiling environment a verbatim copy of the hosting platform, which is most likely infeasible. Instead,

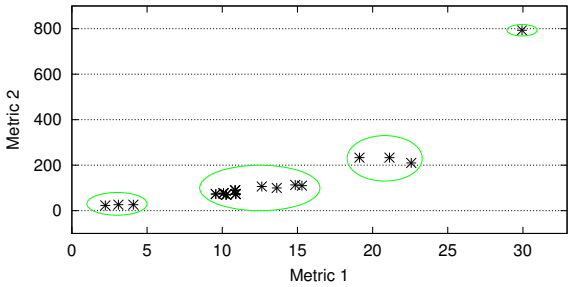


Figure 5. Identifying the representative workloads - DejaVu substantially reduces the tuning overhead by producing only 4 workload classes out of 24 initial workloads.

DejaVu only assumes that the relative ordering among workloads is preserved between the profiling and the production environment. For instance, if workload A is closer to workload B than to workload C in the profiling environment, the same also holds in the production environment. We have verified this assumption empirically using machines of different types in our lab.

After DejaVu identifies the workload classes, it triggers the tuning process for a single workload from each workload class. It typically chooses the instance that is closest to the cluster's centroid. The Tuner's job is to determine the sufficient, but not wasteful, set of virtualized resources (e.g., number and type of virtual instances) that ensure the application meets its SLO. The Tuner can use modeling or experiments for this task. Moreover, it can be manually driven or entirely automated. The choice of a tuning mechanism is orthogonal to our work. After the Tuner determines resource allocations for each workload class, DejaVu has a table populated with workload signatures along with their preferred resource allocations – *the workload signature repository* – which it can re-use at runtime.

Since our focus is not on the Tuner itself, we resort to a very simple technique – linear search – in our evaluation. In more detail, we replay a sequence of runs of the workload, each time with an increasing amount of virtual resources. We then choose the minimal set of resources that fulfill the target SLO. For instance, one can incrementally increase the CPU or memory allocation (by varying the VMM's scheduler caps) until the SLO is fulfilled. Since our experiments involve EC2, we can only vary the number of virtual instances or instance type. Note that we can accelerate the tuning process by using more sophisticated methods, as in [30]. We leave this avenue for our future work.

3.5 Quickly Adapting to Workload Changes

Since DejaVu's goal is to re-use resource allocation decisions at runtime, it needs a mechanism to decide to which cluster a newly encountered workload belongs – the equivalent of the cache lookup operation. DejaVu uses the previously identified clusters to label each workload with the cluster number to which it belongs, such that it can train a *classifier* to quickly recognize newly encountered workloads at runtime. The resulting classifier stands as the explicit description of the workload classes. We have experimented with numerous classifier implementations from the WEKA package and observe that both Bayesian models and decision trees work well for the network services we considered. We use the C4.5 decision tree in our evaluation, or more precisely its open source Java implementation – J48).

Upon a workload change, DejaVu promptly collects the relevant low-level metrics to form the workload signature of the new workload and queries the DejaVu repository to find the best match

Name	Description	Name	Description
busq_empty	Bus queue is empty	cpu_clk_unhalted	Clock cycles when not halted
l2_ads	Cycles the L2 address bus is in use	l2_reject_busq	Rejected L2 cache requests
l2_st	Number of L2 data stores	load_block	Events pertaining to loads
store_block	Events pertaining to stores	page_walks	Page table walk events

Table 1. The HPC metrics included in RUBiS's workload signature.

among the existing signatures. To do this, it uses the previously defined classification model and outputs the resource allocation of the cluster to which the incoming signature belongs. Given that the number of workload classes is typically small and the classification time practically negligible, DejaVu can adjust to workload changes on the order of a few or several seconds, as needed by the DejaVu profiler to collect the workload signatures.

Along with the preferred resource allocations, the repository also outputs the certainty level with which the repository assigned the new signature to the chosen cluster. If the repository repeatedly outputs low certainty levels, it most likely means that the workload has changed over time and that the current clustering is no longer relevant. DejaVu can then initiate the clustering and tuning process once again, allowing it to determine new workload classes, conduct the necessary experiments (or modeling activities), and update the Repository. Meanwhile, DejaVu configures the service with the maximum allowed capacity to ensure that the performance is not affected when experiencing non-classified workloads.

3.6 Addressing Interference

In the previous subsection, we described how to populate the repository with the smallest (also called *baseline*) resource allocation that meets the SLO at the time of tuning. The baseline allocation however, due to interference, may not guarantee sufficient performance at all times. DejaVu deals with this problem by estimating the *interference index* and using it, along with the workload signature, when determining the preferred resource allocation.

In more detail, after DejaVu deploys the baseline resource allocation for the current workload, it monitors the resulting performance (e.g., service latency). If it observes that the SLO is still being violated, DejaVu blames interference for the performance degradation. Workload changes are excluded from the potential reasons, because the workload class has just been identified in isolation. It then proceeds by computing the interference index as:

$$\textit{Interference index} = \frac{PerformanceLevel_{production}}{PerformanceLevel_{isolation}} \qquad (2)$$

The index contrasts the performance of the service in production after the baseline allocation is deployed with that obtained from the profiler. Note that DejaVu relies on each application to report a performance-level metric (e.g., response time, throughput). This metric already needs to be collected and reported when the performance is unsatisfactory. Others have argued for computing this metric [18].

Finally, DejaVu queries the repository for the preferred resource allocation for the current workload and the interference amount. If the repository does not contain the corresponding entry, DejaVu triggers the tuning process and sends the obtained results, along with the estimated index, to the repository for later use. After this is done, DejaVu will be able to quickly lookup the best resource allocation for this workload given the same amount of interference.

Interference may vary across the VM instances of a service, making it hard to select a single instance for profiling that will uniquely represent the interference across the entire service. Inspired by typical performance requirements (e.g., the $Xth - percentile$ of the response time should be lower than Y seconds), we envision a selection process that chooses an instance at which interference is higher than in $X\%$ of the probed instances. This conservative performance estimation would give us a probabilistic guarantee on the service performance.

For the time being, DejaVu quantifies the interference impact and reacts upon it to maintain the SLO. However, we plan to further investigate this aspect of DejaVu and ensure it provides finer information about the interference. Assuming that the cloud provider collects the low-level metrics from its VM instances, it might compare the metric values imposed by the same workload class over time to reveal which resource is primarily affected by the interference (e.g., cache, I/O).

3.7 Discussion

We now discuss a few interesting questions about DejaVu.

Who should run DejaVu: the cloud provider or a third party? Although we view this choice as orthogonal to our work, we believe that it is more practical for the cloud provider to run DejaVu. In fact, this is the setup we have assumed so far. This deployment scenario eliminates the privacy and network traffic concerns with shipping code (clones of the services' VMs) and client requests to a third party.

Nevertheless, it is conceivable that a third party could run DejaVu. In this case, users would likely have to explicitly contract with both the provider and the third party. Alternatively, the DejaVu proxy could be configured to selectively duplicate the incoming traffic such that private information (e.g., e-mails, user-specific data) is not dispatched to the profiler. However, having to share the service code with the third party would still be a problem.

Regardless of who runs DejaVu, a tenant needs to reveal certain information about their service. Specifically, the proxy needs to know the port numbers used by the service to communicate with the clients and, internally, among VMs. Finally, to completely automate the resource allocation process, DejaVu assumes that it can enforce a chosen resource allocation policy without necessitating user involvement. Amazon EC2, for instance, allows us to automatically adjust the number of running instances by using its APIs.

How does DejaVu deal with unforeseen workloads? DejaVu provides no worse performance than the existing approaches when it encounters a previously unknown workload (e.g., large and unseen workload volume [4]). In this case, DejaVu has to spend additional time to identify the resource allocation that achieves the desired performance at minimal cost (just like the existing systems). To try to avoid an SLO violation by the service, the current version of DejaVu responds to unforeseen workloads by deploying the maximum resource allocation (full capacity). If the workload occurs multiple times, DejaVu invokes the Tuner to compute the minimal set of required resources and then readjust the resource allocation.

What is the scope of DejaVu? Although DejaVu primarily targets "request-response" Internet services, we believe that our interference mechanism can be useful even for long-running batch workloads (e.g., MapReduce/Hadoop jobs). In this case, DejaVu would require the equivalent of an SLO. For example, for Hadoop map tasks, the SLO could be their user-provided expected running

times (possibly as a function of the input size). Upon an SLO violation, DejaVu would run a subset of tasks in isolation to determine the interference index. This computation would also expose cases in which interference is not significant and the user simply mis-estimated the expected running times. We leave the issue of applying DejaVu to other types of workloads for future work.

4. Evaluation

Our evaluation uses realistic traces and workloads to answer the following questions. First, can DejaVu produce significant savings while scaling network services horizontally (scaling out) and vertically (scaling up)? Second, how does DejaVu compare with: (1) a time-based controller (called Autopilot) which attempts to leverage the re-occurring (e.g., daily) patterns in the workload by repeating the resource allocations determined during the learning phase at appropriate times, and (2) an existing autoscaling platform, such as RightScale [24]? Third, is DejaVu capable of detecting and mitigating the effect of interference? Finally, can the profiling overhead, and to what extent, affect the performance of the production system? This section starts by describing our experimental setup.

Testbed. Our profiling environment consists of two servers: Intel SR1560 Series rack servers with Intel Xeon X5472 processors (eight cores at 3 GHz), 8 GB of DRAM, and 6 MB of L2 cache per every two cores. We use them to collect the low-level metrics while hosting the clone instances of Internet service components.

We evaluate the DejaVu framework by running widely-used benchmarks on Amazon's EC2 cloud platform. We ran all our experiments within an EC2 cluster of 20 virtual machines (both clients and servers were running on EC2). To demonstrate DejaVu's ability to scale out, we vary the number of active instances from 2 to 10 as the workload intensity changes, but resort only to EC2's large instance type. In contrast, we demonstrate its ability to scale up by varying the instance type from large to extra-large, while keeping the number of active instances constant.

To focus on DejaVu rather than on the idiosyncrasies of EC2, our scale out experiments assume that the VM instances to be added to a service have been pre-created and stopped. In our scale up experiments, we also pre-create VM instances of both types (large and extra large). Pre-created VMs are ready for instant use, except for a short warm-up time. In all cases, state management across VM instances, if needed, is the responsibility of the service itself, not DejaVu.

Internet services. We evaluate DejaVu for two representative types of Internet services: (1) a classic multi-tier web site with an SQL database back-end (SPECweb2009), and (2) a NoSQL database in the form of a key-value storage layer (Cassandra).

SPECweb2009 [32] is a benchmark designed to measure the performance of a web server serving both static and dynamic content. Further, this benchmark allows us to run 3 workloads: e-commerce, banking, and support. While the first two names speak for themselves, the last workload tests the performance level while downloading large files.

Cassandra [1] differs significantly from SPECweb2009. It is a distributed storage facility for maintaining large amounts of data spread out across many servers, while providing highly available service without a single point of failure. Cassandra is used by many real Internet services, such as Facebook and Twitter, whereas the clients to stress-test it are part of the Yahoo! Cloud Service Benchmark [11].

In Section 4.4, we also profile RUBiS [6], a three-tier e-commerce application (given its similarity to SPECweb2009, we do not demonstrate the rest of DejaVu's features on this benchmark). RUBiS consists of a front-end Apache web server, a Tomcat application server, and a MySQL database server. In short, RUBiS defines 26 client interactions (e.g., bidding, selling) whose frequencies are defined by RUBiS transition tables. Our setup has 1,000,000 registered clients and that many stored items in the database, as defined by the RUBiS default property file.

Given that these are widely-used benchmarks, client emulators are publicly available for all of them and we use them to generate client requests. Each emulator can change the workload type by varying its "browsing habits", and also collect numerous statistics, including the throughput and response time, which we use as the measure of performance. Finally, all clients run on EC2 instances to ensure that the clients do not experience network bottlenecks.

Workload traces. To emulate a highly dynamic workload of a real application, we use real load traces from HotMail (Windows Live Mail) and Windows Live Messenger from September, 2009 [35]. Figures 6(a) and 7(a) plot the normalized load from these traces. Both traces contain measurements at 1-hour increments during one week, aggregated over thousands of servers. We proportionally scale down the load such that the peak load from the traces corresponds to the maximum number of clients that we can successfully serve when operating at full capacity (10 virtual instances).

In all our experiments, we use the first day from our traces for initial tuning and identification of the workload classes, whereas the remaining 6 days are used to evaluate the performance/cost benefits when DejaVu is used.

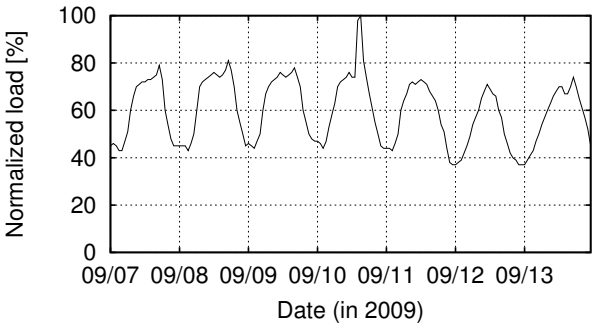
4.1 Case Study 1: Scaling Out

Our first set of experiments demonstrates DejaVu's ability to reduce the service provisioning cost by dynamically adjusting the number of running instances (scale out) as the workload intensity varies according to our live traces. We show DejaVu's benefits with Cassandra's update-heavy workload which has 95% of write requests and only 5% of read requests.

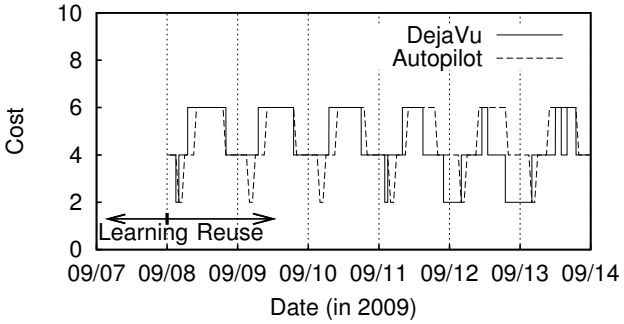
Figure 6(b) plots how DejaVu varies the number of active server instances as the workload intensity changes according to the Messenger traces. The initial tuning produces 4 different workload classes and ultimately 4 preferred resource allocations that are obtained using the Tuner. DejaVu collects the workload signature every hour (dictated by the granularity of the available traces) and classifies the workload to promptly re-use the preferred resource allocation. While the savings compared to the fixed maximum allocation are promising, about 55% over the 6-day period, we need to ensure that the desired performance level is maintained.

Figure 6(c) shows the response latency in this case. The SLO latency is set to 60 ms. Although this is masked by the monitoring granularity, we note that Cassandra takes a long time to stabilize (e.g., tens of minutes) after DejaVu adjusts the number of running instances. This delay is due to Cassandra's re-partitioning; a well-known problem that is the subject of ongoing optimization efforts [11]. Apart from Cassandra's internal issues, DejaVu keeps the latency below 60 ms, except for short periods when the latency is fairly high – about 100 ms. These latency peaks correspond to DejaVu's adaptation time, around 10 seconds, which is needed by the profiler to collect the workload signature and deploy a new preferred resource allocation. Note that this is 18x faster than the reported figures of about 3 minutes for adaptation to workload changes by state-of-the-art experimental tuning [42].

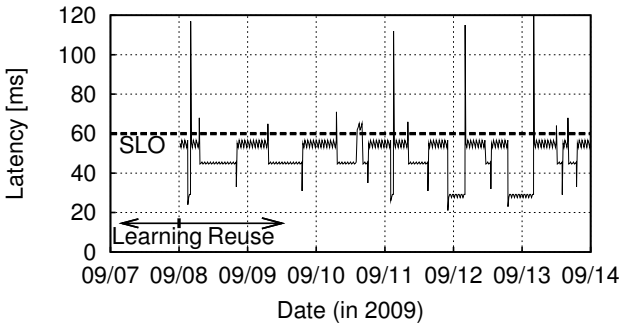
We now conduct a similar set of experiments, but drive the workload intensities using the HotMail trace. Figures 7(b) and 7(c) visualize the cost (in number of active instances) and latency over time, respectively. While the overall savings compared to the maximum allocation are again similar (60% over the 6-day period), there are few points to note. First, the initial profiling identified 3 workload classes for the HotMail traces, instead of



(a) Windows Live Messenger load trace.

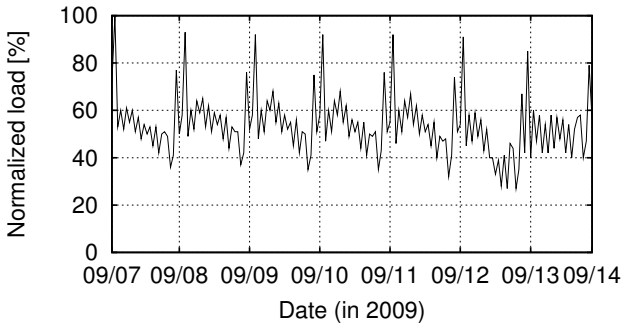


(b) Number of virtual instances used to accommodate the load.

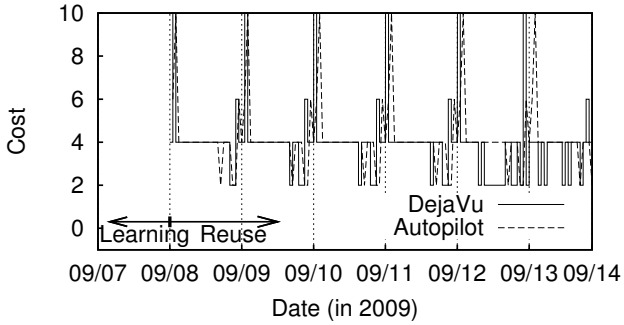


(c) Service latency as DejaVu adapts to workload changes. SLO = 60 ms.

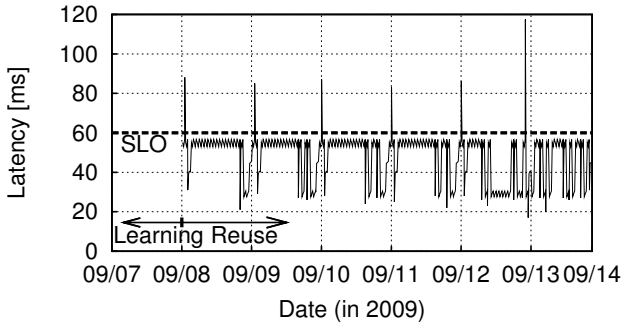
Figure 6. Scaling out Cassandra with the Messenger trace.



(a) HotMail load trace.



(b) Number of virtual instances used to accommodate the load.



(c) Service latency as DejaVu adapts to workload changes. SLO = 60 ms.

Figure 7. Scaling out Cassandra with the Hotmail trace.

4 for the Messenger traces. Second, during the 4th day, DejaVu could not classify one workload with the desired confidence, as it differs significantly from the previously defined workload classes. The reason is that the initial profiling had not encountered such a workload in the first day of the traces. To avoid performance penalties, DejaVu decided to use the full capacity to accommodate this workload. If this scenario were to re-occur, DejaVu would resort to repeating the clustering process.

Comparison with existing approaches. Next, we compare DejaVu's behavior with that of two existing approaches. Figure 6(b) depicts the resource allocation decisions taken by Autopilot. Specifically, Autopilot simply repeats the hourly resource allocations learned during the first day of the trace. The Autopilot approach leads to suboptimal resource allocations and the associated provisioning cost increases. Due to poor allocations, Autopilot violates the SLO at least 28% of the time, in both traces. These measurements illustrate the difficulty of using past workload information blindly.

We further compare DejaVu with an existing autoscaling platform called RightScale [24]. Because we are not RightScale customers, we reproduced their approach based on publicly available information. The RightScale algorithm reacts to workload changes by running an agreement protocol among the virtual instances. If the majority of VMs report utilization that is higher than the predefined threshold, the scale-up action is taken by increasing the number of instances (by two at a time, by default). In contrast, if the instances agree that the overall utilization is below the specified threshold, the scaling down is performed (decrease the number of instances by one, by default). To ensure that the comparison is fair, we run the Cassandra benchmark which is CPU and memory intensive, as assumed by the RightScale default configuration [25].

Figure 8 shows the average adaptation time for DejaVu and RightScale (assuming its default configuration) for the HotMail and Messenger traces. In case of RightScale, we experiment with 3 (the minimum used in [25]) and 15 minutes (the recommended value) for the "resize calm time" parameter – the minimum time between successive RightScale adjustments. DejaVu's reaction time is about

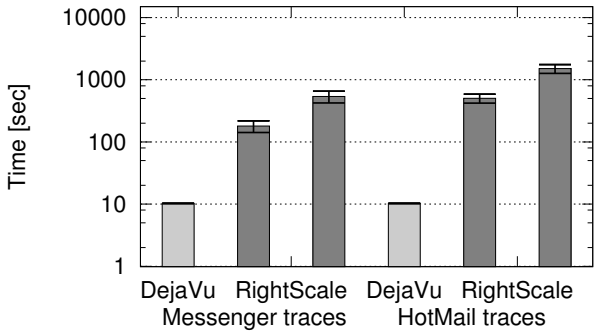


Figure 8. DejaVu and RightScale decision times (error bars show the standard error). RightScale decision times are shown for the "resize calm time" of 3 and 15 minutes in the middle and on the right for each trace, respectively.

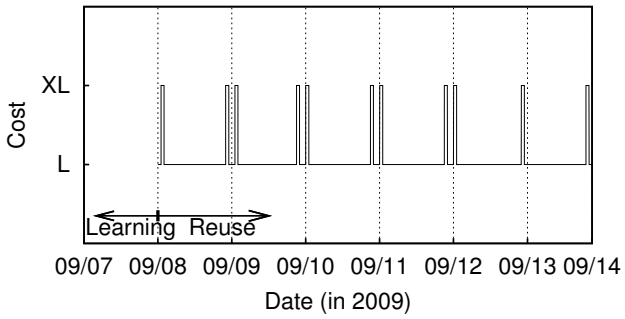
10 seconds in the case of a "cache hit". Note that this time can vary depending on the length of the workload signature (e.g., a larger number of HPCs may take longer to collect). When a single resize operation is sufficient for RightScale, we record an instantaneous adaptation time (zero seconds). However, multiple resize operations are often needed. As a result, RightScale's adaptation time is between one and two of orders of magnitude longer than DejaVu's (note the log scale on the Y axis). This is because DejaVu can automatically jump to the right configuration, rather than gradually increase or decrease the number of instances as RightScale does. Note that the resize calm time is different in nature from the VM boot up time and cannot be eliminated for RightScale; RightScale has to first observe the reconfigured service before it can take any other resizing action.

4.2 Case Study 2: Scaling Up

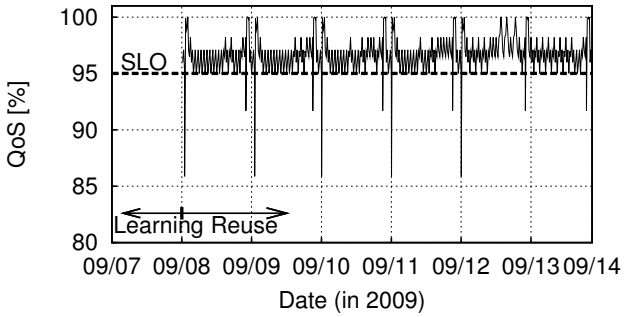
We next evaluate DejaVu's ability to reduce the service provisioning cost while varying the instance type (scaling up) from large to extra-large or vice versa, as dictated by the workload intensity. Toward this end, we monitor the SPECweb service with 5 virtual instances serving at the front-end, and the same number of them at the back-end layer. We use the support benchmark which is mostly I/O-intensive and read-only to contrast with the Cassandra experiments which are CPU-, memory-, and write-intensive. Similar to the previous experiments, DejaVu uses the first day for the initial profiling/clustering, while the remaining days are used to evaluate its benefits.

Figures 9(a) plots the provisioning cost, shown as the instance type used to accommodate the HotMail load over time. Note that the smaller instance was capable of accommodating the load most of the time. Only during the peak load (two hours per day in the worst case), DejaVu deploys the full capacity configuration to fulfill the SLO. In monetary terms, DejaVu produces savings of roughly 45%, relative to the scheme that has to overprovision at all times with the peak load in mind. Figure 9(b) demonstrates that the savings come with a negligible effect on the performance levels; the quality of service (QoS, measured as the data transfer throughput) is always above the target that is specified by the SPECweb2009 standard. The standard requires that at least 95% of the downloads meet a minimum 0.99Mbps rate in the support benchmark for a run to be considered compliant.

We perform a similar set of experiments with the Messenger trace. In this case, Figures 10(a) and 10(b) show the provisioning cost and performance levels, respectively. The savings in this case are about 35% over the 6-day period. Excluding a few seconds after



(a) Virtual instance types used to accommodate the load.



(b) Service latency as DejaVu adapts to workload changes. QoS = 95%.

Figure 9. Scaling up SPECweb with the Hotmail trace.

each workload change spent on profiling, QoS is as desired, above 95%.

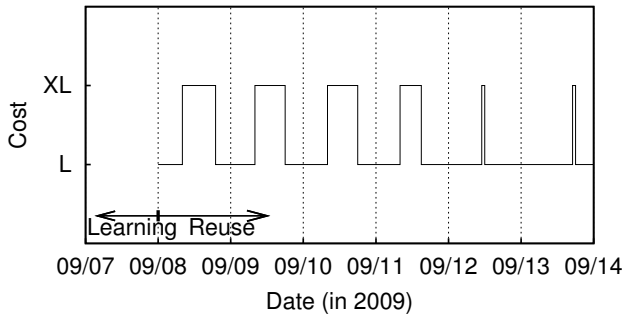
4.3 Case Study 3: Addressing Interference

Our next experiments demonstrate how DejaVu detects and mitigates the effects of interference. We mimic the existence of a co-located tenant for each virtual instance by injecting into each VM a microbenchmark which occupies a varying amount (either 10% or 20%) of the VM's CPU and memory over time. The microbenchmark iterates over its working set and performs multiplication while enforcing the set limit. These amounts of interference mimic the amount of performance degradation reported in similar settings [44].

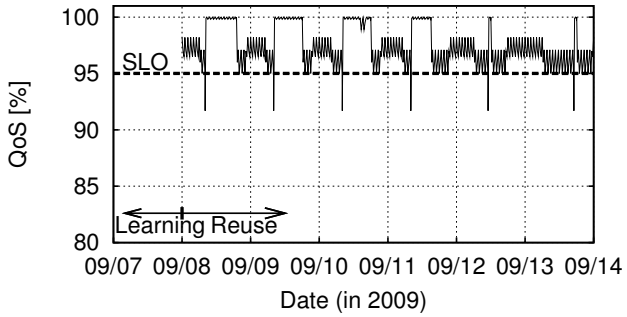
Figure 11(a) contrasts DejaVu with an alternative in which its interference detection is disabled. Without interference detection, one can see that the service exhibits unacceptable performance most of the time. Recall that the SLO is 60 ms. In contrast, DejaVu relies on its online feedback to quickly estimate the impact of interference and lookup the resource allocation that corresponds to the interference condition such that the SLO is met at all times. Figure 11(b) shows that DejaVu indeed provisions the service with more resources to compensate for interference.

4.4 Measuring DejaVu's Overhead

DejaVu requires only one or a few machines to host the profiling instances of the services that it manages. Its network overhead corresponds to the amount of traffic that it sends to the profiling environment. This overhead is roughly equal to $1/n$ of the incoming network traffic, where n is the number of service instances, assuming the worst case in which the DejaVu proxy is continuously duplicating network traffic and sending it to the DejaVu profiler. Given that the inbound traffic (client requests) is only a fraction of the outbound traffic (service responses) for typical services, the

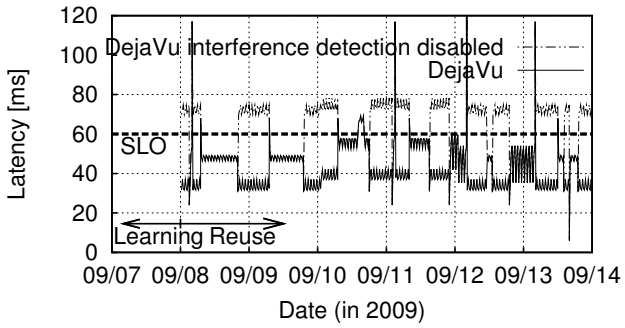


(a) Virtual instance types used to accommodate the load.

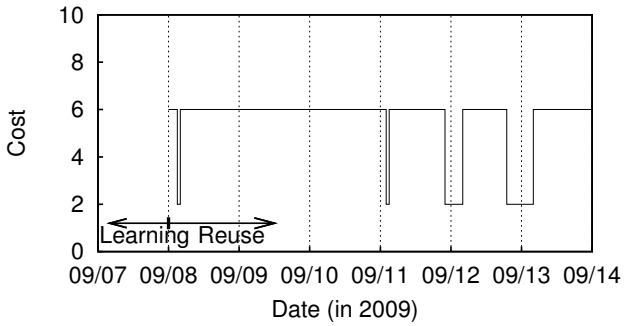


(b) Service latency as DejaVu adapts to workload changes. QoS = 95%.

Figure 10. Scaling up SPECweb with the Messenger trace.



(a) Service latency as DejaVu adapts to workload changes.



(b) Number of virtual instances used to accommodate the load.

Figure 11. Scaling out Cassandra with the Messenger trace under interference. The amount of interference varies, and is set to either 10% or 20%.

network overhead is likely to be negligible. For example, it would be 0.1% of the overall network traffic for a service that uses 100 instances, assuming a 1:10 inbound/outbound traffic ratio that is typically used for home broadband connections.

We now turn our attention to a more important question: To what extent does the DejaVu proxy affect the performance of the system in production, as it duplicates the traffic of a single service instance? To answer this question, we run a set of experiments with the RUBiS benchmark, while profiling its database server instance. We compare the service latency under a setup where the profiling is disabled against a setup with continuous profiling. To exercise different workload volumes, we vary the number of clients that are generating the requests from 100 to 500. Our measurements show that the presence of our proxy degrades response time by about 3 ms on average.

4.5 Summary

To summarize, our evaluation shows that DejaVu maps multiple workload levels to a few relevant clusters. It uses this information at runtime to quickly adapt to workload changes. The adaptation is short (about 10 seconds) and more than 10 times faster than the state-of-the-art. Having such quick adaptation times effectively enables online matching of resources to the offered load in pursuit of cost savings.

We demonstrate provisioning cost savings of 35-60% (compared to a fixed, maximum allocation) using realistic traces and two disparate and representative Internet services: a key-value store and a 3-tier web service. The savings are higher (50-60% vs. 35-45%) when scaling out (varying the number of machines) vs. scaling up (varying the performance of machines) because of the finer granularity of possible resource allocations. The scaling up case had only two choices of instances (large and extra-large) with a fixed number of instances vs. 1-10 identical instances when scaling out.

DejaVu successfully manages interference by recognizing the existence of interference and pragmatically using more resources to compensate for it.

The DejaVu-achieved savings translate to more than $250,000 and $2.5 Million per year for 100 and 1,000 instances, respectively (assuming $0.34/hour for a large instance on EC2 and $0.68/hour for extra large as of July 2011). We draw these service sizes from the available data: the Reddit aggregation web site reportedly uses about one hundred EC2 instances (218 virtual CPUs) [23], whereas the Animoto video creation site uses a few thousand EC2 instances [26].

In terms of overheads, we argue that the network traffic induced by DejaVu is negligible, while our final experiments demonstrate that DejaVu's impact on the performance of the system in production is also practically negligible.

5. Related Work

There has been a large body of recent work on various aspects of data center resource management.

Automated resource management in virtualized data centers. Industrial efforts such as Rightscale [24] use a load-based threshold to automatically trigger creation of a previously configured number of new virtual instances in a matter of minutes. This approach uses an additive-increase controller, and as such may take long to converge.

Applying modeling and machine learning to resource management in data centers. Urgaonkar *et al.* [37] propose a closed queuing network model along with Mean Value Analysis (MVA) algorithm for multi-tier applications. Watson *et al.* [40] follow a simi-

lar approach, and develop queuing-based performance models for enterprise applications, but with emphasis on the virtualized environment. Another example of explicitly using models to enhance coordinated provisioning of various computer resources was presented in [12]. Stewart *et al.* [33] significantly enhance the accuracy of models by explicitly modeling a non-stationary transaction mix; their main point is that the workload type (as in a different type of incoming requests to a service) is equally important as the workload volume itself. In general, these efforts work well for the workloads used during parameter calibration, but may require (manual) adjustment when the workload changes. Further, achieving higher accuracy requires highly skilled labor, along with a deep understanding of the application.

Running actual experiments instead of using models. Zheng *et al.* [42] advocated running actual experiments for resource management in a virtualized environment. Relative to this work, DejaVu quickly characterizes the workloads at runtime to avoid re-running experiments. In doing so, it dramatically reduces the amount of time the service is running with suboptimal parameters.

Performance counters and workload characterization. Recently, HPCs have been extensively used to characterize activities within the entire system for various purposes. For instance, Merkel *et al.* [17] leverage HPCs to predict task energy consumptions. Using these predictions, a scheduler can get the maximum performance out of a multiprocessor system, and still avoid overheating of system components. Sweeney *et al.* [34] demonstrate that HPCs are also useful in understanding the behavior of Java applications. Namely, their tool is effective in identifying performance anomalies and can help in pinpointing their cause. Finally, Shen *et al.* [29] argue that the HPCs might be used for on-the-fly prediction at the request granularity, thus enabling online system adaptation. Although related, our work is fundamentally different as we are aiming at workload classification, rather than fine-granularity recognition. For instance, our monitoring module could provide feedback such as "workload volume is medium, and the requests are mostly read requests".

Even earlier, sample-based profiling was used to identify different activities running within the system (e.g., Magpie [3] and Pinpoint [9]). For instance, Magpie uses clustering to classify requests and produce a workload model. Although such tools can be useful for post-execution decisions, they do not provide online identification and the ability to react during execution. This ability is crucial for our framework to quickly adapt to workload changes.

Automatic benchmarking. Developers devote considerable time to benchmarking to obtain insight into the performance, inter-layer interactions, and most important for our work, workload characterization. There have been a few works that automate this laborious task [30, 43]. Although this is orthogonal to our approach, our framework would greatly benefit from the existence of a tool that determines the most representative workloads to benchmark.

Automatic VM configuration and performance crisis detection. Soror *et al.* [31] address the problem of automatically configuring a database management system (DBMS) by adjusting the configurations of the VM in which they run. They use information about the anticipated workload and then compute the workload-specific configuration. However, their framework assumes help from the DBMS which describes a workload in the form of a set of SQL statements. In contrast, DejaVu does not require any information from the guest OS (VM), and very little information from the application running inside it.

Bodik *et al.* [5] propose a methodology for automatic classification and identification of performance crises, and in particular for detecting whether a given crisis has been seen before, so that a known solution may be immediately applied. As opposed to DejaVu, the focus of the work is mostly on (1) identifying performance anomalies due to bugs or unexpected behaviors, and (2) speeding up the stabilization. In contrast, DejaVu accelerates the management of virtualized resource allocations under workload changes.

Cost-aware elasticity. Sharma *et al.* [27] propose Kingfisher, a system that tries to minimize the cloud tenant's deployment cost, while being elastic to workload changes. Kingfisher takes into account the cost of each VM instance, the possibilities of scaling up and scaling out, as well as the transition time from one configuration to another. It then solves an integer linear program to derive the minimum cost configuration under each workload change. Kingfisher and DejaVu are orthogonal and can benefit from one another. Kingfisher assumes a perfect workload predictor, and it would benefit from storing its resource allocation decisions in the DejaVu cache (and avoid re-running the ILP solver every time a workload change dictates a configuration change). DejaVu could simply use Kingfisher as its Tuner.

6. Conclusion

The problem of resource allocation is challenging in the cloud, as the co-located workloads constantly evolve. The result is that system administrators find it difficult to properly manage the resources allocated to the different virtual machines, leading to suboptimal service performance or wasted resources for significant periods of the time.

In this paper we described the design and implementation of DejaVu, a framework that quickly and automatically reacts to workload changes by learning the preferred virtual resource allocations from past experience. DejaVu also detects performance interference across virtual machines and adjusts the resource allocation to counter it.

Though this work provides a solid foundation, there are multiple challenging directions that we want to pursue in the future. For example, we demonstrated that an application can significantly benefit from its own resource allocation experience. However, we believe that it can benefit from the experience of other cloud tenants as well, and we plan to further explore this potential.

Acknowledgments

This research is funded by the Swiss NSF (grant FNS 200021-130265). Dejan Novaković is also supported by the Swiss NSF (grant FNS 200021-125140). We thank the anonymous reviewers for their valuable feedback. We are grateful to Eno Thereska, Austin Donnelly, and Dushyanth Narayanan for providing us with their HotMail and Messenger traces.

References

[1] Apache Foundation. The Apache Cassandra Project. `http://cassandra.apache.org/`.

[2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *SIGMETRICS*, 2000.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[4] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Symposium on Cloud Computing*, 2010.

[5] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys*, 2010.

[6] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. *SIGPLAN Not.*, 2002.

[7] E. Cecchet, R. Singh, U. Sharma, and P. Shenoy. Dolly: virtualization-driven database provisioning for the cloud. In *VEE*, 2011.

[8] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat. Managing energy and server resources in hosting centers. In *SOSP*, 2001.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[10] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. In *SIGMETRICS*, 2005.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[12] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *USITS*, 2003.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[14] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and freon: temperature emulation and management for server systems. In *ASPLOS*, 2006.

[15] G. Linden. Make Data Useful. `https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt`.

[16] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counters. In *MASCOTS*, 2005.

[17] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *EuroSys*, 2006.

[18] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys*, 2010.

[19] Netfilter. netfilter/iptables. `http://www.netfilter.org/`.

[20] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and validating database system administration. In *USENIX*, 2006.

[21] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[22] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, 2001.

[23] Reddit. `http://www.reddit.com/r/IAmA/comments/a2zte/i_run_reddits_servers_and_do_a_bunch_of_other/`.

[24] RightScale. `http://www.rightscale.com/`.

[25] RightScale. `http://support.rightscale.com/12-Guides/Lifecycle_Management/03_-_Understanding_Key_Concepts/RightScale_Alert_System/Alerts_based_on_Voting_Tags/Understanding_the_Voting_Process`.

[26] RightScale. Animoto's facebook scale-up. `http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/`.

[27] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559 –570, june 2011.

[28] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. *SIGOPS Oper. Syst. Rev.*, 2002.

[29] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. *ASPLOS*, 2008.

[30] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting corners: workbench automation for server benchmarking. In *USENIX*, 2008.

[31] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.

[32] SPECweb2009. `http://www.spec.org/web2009/`.

[33] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys*, 2007.

[34] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In *VM*, 2004.

[35] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *EuroSys*, 2011.

[36] R. Thonangi, V. Thummala, and S. Babu. Finding good configurations in high-dimensional spaces: Doing more with less. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecomunication Systems*, 2008.

[37] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS*, 2005.

[38] B. Urgaonkar, P. J. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, 2002.

[39] N. Vasić, M. Barisits, V. Salzgeber, and D. Kostić. Making Cluster Applications Energy-Aware. In *ACDC*, 2009.

[40] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. Probabilistic performance modeling of virtualized resource allocation. In *ICAC*, 2010.

[41] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[42] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: Experiment-based management of virtualized data centers. In *USENIX*, 2009.

[43] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *EuroSys*, 2007.

[44] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.