DIAL: Reducing Tail Latencies for Cloud Applications via Dynamic Interference-aware Load Balancing

Seyyed Ahmad Javadi, Anshul Gandhi; Stony Brook University

***Abstract*—Many online application services are now provided by cloud-deployed VM clusters. Although economical, VMs in the cloud are prone to interference due to contention for physical resources among colocated users. Worse, this interference is dynamic and unpredictable. Current provider-centric solutions are application-oblivious and are thus not always aware of the user's SLO requirements or application bottlenecks. Further, such solutions rely on VM scheduling and migration, approaches that are not agile enough to mitigate volatile interference.**

This paper presents DIAL, an interference-aware load balancer that can be employed by cloud users without requiring any assistance from the provider. DIAL addresses time-varying interference by dynamically shifting load away from compromised VMs without violating the application's tail latency SLOs. The key idea behind DIAL is to infer the demand for contended resources on the physical hosts, which is otherwise hidden from users. Estimates of the colocated load are then used to drive the load distribution for the application VMs. Our experimental results on OpenStack and AWS clouds show that DIAL can reduce application tail latencies by as much as 70% and 48% compared to interference-oblivious and existing interference-aware load balancers, respectively.

I. Introduction

Cloud computing users can access economical and virtually unlimited computing resources, allowing them to deploy their applications in the cloud, and let the applications grow elastically with them. Many online services and applications, such as Netflix and Expedia, are now provided by cloud-deployed Virtual Machines (VMs). Users typically deploy multiple VMs on the cloud to host their application; load balancers are then employed to facilitate scaling by distributing incoming load among the VMs.

Despite the benefits of cloud computing, user applications deployed in such environments can experience undesirable performance effects, the most severe of which is *interference*. Performance interference is caused by contention for physical resources, such as CPU, network, or last-level cache, among colocated VM users. While certain resources, such as CPU, can be partitioned among colocated VMs, other resources, such as processor caches, are notoriously hard to partition [1]; regardless, static partitioning of resources can adversely impact resource utilization in cloud data centers. Worse, resource contention among independent colocated VMs is often *dynamic and unpredictable* [2]. Dynamic interference in public and private cloud environments can degrade application performance by as much as 27× [3, 4, 5], despite provider efforts to isolate performance [6, 7, 8].

Existing provider-centric solutions are not always aware of the cloud user's Service Level Objective (SLO) requirements or the user application's bottleneck resources; this is especially true in public clouds where providers have *limited visibility* into user deployments [6]. Provider-centric solutions focus on co-scheduling VMs that do not contend on the same resource(s) [5, 7, 9, 10, 11]. However, since interference is dynamic and can emerge unpredictably, statically co-scheduling VMs will not suffice. VM migration can help in this case, but interference is volatile and short-lived, often lasting for only a couple minutes [2]; by contrast, migration can take several minutes [12] and can incur overheads [13], especially for stateful applications [14]. Relying only on provider-centric solutions for interference leaves application performance, which is often the most important criteria for users, at the mercy of providers.

We present DIAL, a dynamic solution for mitigating interference in load-balanced cloud deployments. Specifically, we consider a cloud-deployed web application hosted on multiple VMs behind a load balancer, and experiencing unpredictable interference from colocated VMs (owned by other cloud tenants). DIAL is *user-centric*, meaning that it can be employed directly by cloud users without requiring any support from the cloud provider. Further, since DIAL is user-centric, it is aware of the user application and its SLOs, and can accordingly react to the onset or termination of interference by quickly adapting the load distribution among application VMs.

The main challenge for user-centric approaches is the lack of visibility into colocated applications. Our key insight to addressing this problem is to *infer* contention by monitoring the application performance from within the user deployment and estimating the colocated load that can induce the observed level of performance degradation. Importantly, we do so without requiring any assistance from colocated users or the hypervisor. We find that, in addition to estimating the colocated load, it is also important to determine the resource that is under contention, as this dictates the impact of interference on performance.

To address the dynamic nature of interference, DIAL adapts the load distribution of incoming requests among user VMs. We introduce a model for interference, based in queueing theory [15, 16], to understand the impact on performance of contention at the shared physical resources. We then optimize the time-varying load distribution among the VMs to minimize tail latency, thus guiding our design of DIAL.

We implement DIAL on HAProxy [17], and evaluate DIAL's benefits using popular web applications with varying workload under CPU, network, disk, and cache interference on OpenStack and AWS clouds. Our experimental results show that DIAL reduces 90%ile response times by as much as 70% compared to interference-oblivious load balancers. Further, compared to existing interference-aware solutions, including those that rely on the hypervisor to detect interference, DIAL reduces tail response times by as much as 48%.

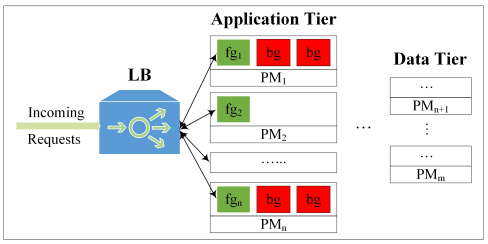


Fig. 1. Illustration of a load-balanced cloud application deployed on multiple foreground (fg) VMs experiencing interference from background (bg) VMs.

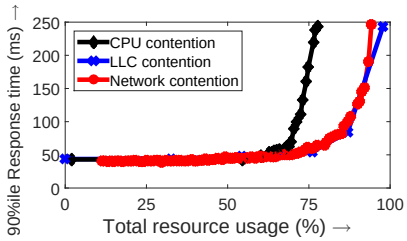


Fig. 2. Performance of an OpenStack-deployed Apache web server under interference from colocated VMs running microbenchmarks.

II. DIAL System Design

DIAL is a user-centric interference mitigation solution designed for clouds that directly empowers the users. DIAL can complement provider-centric solutions, especially when provider efforts to mitigate interference are not enough to avoid specific SLO violations for user applications. We first provide an overview of DIAL, and then discuss the design of its two key contributions: (i) estimating co-located load, and (ii) optimizing the interference-aware load distribution.

A. Overview and Scope

We consider user cloud deployments consisting of multiple VMs to scalably handle incoming traffic; *any* of the VMs could be under interference at *any* time. Multi-VM applications often employ an internal scheduler or load balancer. In this paper we focus on load-balanced *web* applications.

Figure 1 illustrates a typical multi-tier web application, similar to the ones we employ in our experiments (see Section III-B), composed of multiple tiers including application and data tiers. Our focus in this paper is on the application tier which is behind the load balancer. The web application is hosted on multiple foreground (fg) VMs, each of which is hosted on a physical machine (PM); we highlight the application tier fg VMs in Figure 1. The incoming requests for the user application are load balanced among fg VMs via a load balancer (LB). Each PM may also host background (bg) VMs that do not belong to the fg user, as shown in Figure 1. The fg and bg VMs on a PM can contend for shared physical resources, such as CPU, network bandwidth (NET), disk I/O bandwidth (DISK), and last-level-cache (LLC), resulting in interference. Note that the fg user does not have visibility into the bg VMs; in fact, the fg user is unaware of bg VMs.

We assert that the LB is ideally suited to mitigate the effects of volatile interference. Our solution, DIAL, is a user-centric Dynamic Interference-Aware Load balancer. The design of DIAL addresses two key questions:

(i) How can users estimate the interference that their VMs are experiencing without any assistance from the provider, hypervisor, or colocated users? (Section II-B)

(ii) Given this information, how should users dynamically distribute load among their VMs to minimize tail latencies in the presence of interference? (Section II-C)

B. User-Centric Estimation of Interference

To effectively mitigate interference, DIAL must first estimate the amount of interference that each user VM is experiencing. To this end, we define *amount of interference: the fraction of available physical resources that are in use by colocated background VMs.* In the context of Figure 1, the amount of interference is the fraction of physical resources on a PM that are in use by all the colocated bg VMs, and are thus unavailable to the fg VM on that PM. As we show below, estimating the amount of interference is not an easy task as it requires analysis, classification, and modeling of interference.

1) Impact of interference on tail latencies

Interference is known to impact application response times [3, 4, 5]. DIAL leverages this fact to estimate the amount of interference that an fg VM is experiencing because of resource contention created by colocated bg VMs. Specifically, DIAL aims to infer the amount of interference, or resource contention, that the bg VMs must be creating to effect the observed rise in fg response times.

Figure 2 shows the impact of different types of resource contention on the 90%ile response time, T_{90}, of an OpenStack cloud-deployed Apache web server VM hosting files and driven by the httperf load generator. We create contention for this fg VM by running various microbenchmarks in colocated bg VMs. The x-axis denotes the percentage of total resource usage, which is the sum of resource usage by the fg VM and all colocated bg VMs, normalized by peak resource capacity or bandwidth. For example, if the total network bandwidth usage is 80MB/s, and the peak network bandwidth is about 115MB/s, then the resource utilization is $80/115 \approx 0.7$. We defer the details of our experimental setup and resource usage calculation to Section III, where we also discuss our other, more realistic, web applications.

We make two observations from this figure:
(i) *response time increases considerably under interference*,
(ii) *the relationship between total resource usage and response time depends on the exact resource under contention*.

Detecting interference: The first observation can be used by DIAL to *detect* when the fg application VM is under interference. Specifically, from Figure 2, we see that application response times, or T_{90}, are initially *low and stable* (left of the graph). However, once the total resource usage increases (right of the graph), because of the increased resource demand from bg VMs, the fg response times *rise sharply*. We also observe the same behavior in other environments, such as AWS clouds. Based on this observation, DIAL signals interference when T_{90} goes beyond the 95% *confidence intervals* (around the mean) observed during no or low interference. Validation experiments for realistic web applications in Section IV show that our detection approach provides a low false positive rate ($\sim$ 6%).

Need for identifying the source of interference: The second observation suggests that using tail response times to estimate interference will require knowledge of the specific resource that is under contention.

2) Classifying interference using Decision Trees

Our next task is to classify the source of interference, which is defined as the *dominant resource under contention*. Note that it is possible for several resources to be simultaneously under contention; however, we only consider dominant resource contention. We defer the analysis of multiple resources under contention to future work. Our key idea in classification is to observe the impact of interference on easily observable user metrics such as CPU utilization, I/O wait time, etc., which can be obtained from within the VM via the `/proc` subsystem. In general, one can monitor all available metrics and use feature selection algorithms, such as LASSO or ridge regression, to derive the list of useful features, which can then be used for classification.

DIAL uses decision trees to classify contention. The decision tree classifier is trained by running controlled interference experiments using microbenchmarks and monitoring the metrics in each case. After training, the decision tree can classify the source of interference, even for unseen workloads, based on the observed metric values (Section IV-B).

Distinguishing interference from workload variations: To distinguish interference from workload variations, DIAL normalizes the observed metric values with *predicted* values based on monitored workload intensity. Prior work has shown that linear models can accurately predict CPU usage based on workload intensities [18]. We thus use linear regression to predict the metric values as a linear function of the number of requests seen in the past monitoring interval; since web applications often serve different types/classes of requests, such as browse, store, etc., we consider the number of requests of *each* type in our prediction.

The intuition behind this approach is that, in the absence of interference, the normalized values will be close to 1 under workload variations. The decision tree can thus use the deviation of the observed metrics from the normalized metrics to distinguish workload changes from interference. Experimental results in Section IV highlight the efficacy of our classifier under various contentions (91–93% accuracy).

3) Queueing-based model for interference

The final step is to use the classification information to estimate the amount of interference, which is the *fraction of resources that are in use by colocated bg VMs*. Once we have these estimates, DIAL can redistribute incoming load accordingly to mitigate the impact of interference (Section II-C).

From Figure 2, we see that tail response times increase non-linearly with the total usage of the resource under contention. Recall that the total resource usage is the sum of resource usage of the fg VM (can be monitored by the fg user) and all colocated bg VMs (cannot be monitored by the fg user). Our key idea is to model this non-linear relationship for each resource; this allows inferring the resource usage of the colocated bg VMs based on observed fg tail latencies, which in turn gives us the amount of interference.

Modeling interference: We employ queueing theory to model the non-linear relationship between resource usage and tail latencies. Queueing models suggest that the tail response time for an application is inversely proportional to the unused capacity of the VM [15]. Mathematically, $T_{90} \sim 1/(1-\rho_{fg})^{\alpha}$, for some parameter α, where ρ_{fg} is the resource load of the fg application (such as CPU utilization or I/O bandwidth utilization), normalized to peak resource usage; that is, $0 \leq \rho_{fg} \leq 1$. Prior work [19] has shown that $\alpha = 2$ works well for practical settings given the high variability in real workloads. However, such models do *not* account for interference.

Under interference, the fg application experiences congested resources due to colocated bg VMs. As a result, the application experiences higher load than it would in the absence of interference. We model this effect by adding the resource usage of colocated bg VMs to that of the fg VM, resulting in fg response times being inversely proportional to $(1-(\rho_{fg}+\rho_{bg}))$. The sum of loads exerted by the fg and bg VMs, $(\rho_{fg}+\rho_{bg})$, represents the normalized total utilization of the resource. We thus approximate 90%ile response time as:

$$T_{90} = c_0 + c_1/(1-\rho_{fg}-\rho_{bg}) + c_2/(1-\rho_{fg}-\rho_{bg})^2, \quad (1)$$

where $\vec{c}$ is the coefficient vector that depends on the *specific resource under contention*. The polynomial function in Eq. (1) is inspired by prior work on queueing systems [20, 21] to interpolate between low load and high load regimes.

To determine the coefficients, we train the model in Eq. (1) by creating different levels of resource usage and monitoring the T_{90} of fg VMs (see Section II-D2). We then use multiple linear regression over this training data to derive the resource-specific coefficients. While Eq. (1) is inspired by queueing models, we find that it is able to accurately track the relationship between tail response times and resource usage even for realistic web applications as we show in Section IV.

Applying the model to estimate interference: Eq. (1) can be easily employed to estimate the amount of interference. After detection and classification, the fg user can estimate ρ_{bg} by monitoring T_{90} and ρ_{fg}, and solving Eq. (1) for ρ_{bg}.

C. Interference-Aware Load Balancing

Interference-aware load balancing is the key component of DIAL. When there is no interference, balancing the load equally among all VMs works well to provide low response times. However, if one of the VMs is facing interference (can be estimated via the above-described interference modeling), then its share of the load must be adjusted accordingly.

One might think that reducing the share of load in proportion to the available capacity at the compromised VM, $(1-\rho_{bg})$, should work well. Unfortunately, this approach can be far from optimal, as we show via experiments in Section IV.

1) Minimizing tail response times

To minimize application tail response times under interference, we again employ queueing theory. Consider a cluster of n VMs, with VM i facing interference of $\rho_{bg,i}$. Let the fraction

of total incoming load that is directed to VM i be p_i; we refer to p_i as the weight assigned by the LB to VM i. If the total arrival rate for the application is a, the arrival rate for VM i is $a \cdot p_i$. Our goal is to determine the p_is that minimize the 90%ile response time, T_{90}. To obtain a simple closed-form expression for the theoretically optimal p_is, we model each VM as an M/M/1 system [15]. Under this assumption, T_{90} for a cluster of n VMs can be approximated as:

$$T_{90} \approx \sum_{i=1}^{n} p_i \cdot \ln 10/(r_i - a \cdot p_i), \quad (2)$$

where r_i represents the throughput of VM i (with contention). Since interference reduces the throughput of the compromised VM, we set $r_i = r \cdot (1 - \rho_{bg,i})$, where r is the peak throughput of an application VM. For example, if the peak throughput of our Apache server is $r = 1000$ req/sec, and it is experiencing an estimated interference of $\rho_{bg} = 0.6$, then we set $r = 1000 \times 0.4 = 400$ req/sec.

Note the $\ln 10$ term in the numerator. This comes from the term $-\ln(1 - 0.9)$, where 0.9 is due to the 90%ile response times. If we instead focus on, say, 95%ile response times, the only change in Eq. (2) will be $\ln 20$ instead of $\ln 10$. Interestingly, the optimization for p_is discussed below does not depend on this constant value, and thus *our results also apply, as-is, to other percentiles of response times*.

Given a (which can be monitored) and r_i (derived as discussed above using interference estimation from Section II-B), T_{90} can be expressed as a function of p_i via Eq. (2). We can now derive the theoretically optimal weights, p_is, that minimize T_{90} in Eq. (2) using calculus. Due to lack of space, we omit the proof and present the final result here:

$$p_i^* = \left(r_i \sum_{j=1}^{n} \sqrt{r_j} - \sqrt{r_i} \sum_{j=1}^{n} r_j + a\sqrt{r_i} \right) / \left(a \sum_{j=1}^{n} \sqrt{r_j} \right) \quad (3)$$

Note that p_i^* depends on the estimates of r_i, thus necessitating the interference estimation of Section II-B. Also note that p_i^* depends on the total arrival rate, a. This is to be expected since, for example, if the arrival rate is very low, we can send all requests to the VM with the highest throughput to minimize response times; however, if the arrival rate is very high, then a single VM cannot handle all requests, and we have to distribute the load. Importantly, both r_i and a can change unpredictably at any time (r_i due to interference and a due to variable customer traffic), motivating the need for a *dynamic* solution instead of existing static solutions.

D. The DIAL Controller

The DIAL controller implements dynamic interference-aware load balancing as follows:

0) Monitoring: DIAL monitors the fg application's T_{90}, arrival rate, a, fg load, $\rho_{fg,i}$, and classification metrics (e.g., connection time), averaged every interval, for all fg VMs.
1) Detection: DIAL signals interference if T_{90} exceeds the 95% confidence bounds for two successive monitoring intervals.
2) Classification: DIAL next employs the decision tree to identify the dominant resource under contention, if any.
3) Estimation: DIAL then uses the T_{90} and $\rho_{fg,i}$ values in Eq. (1), with the dominant resource-specific coefficients, to estimate the interference, $\rho_{bg,i}$. DIAL then adjusts the interference-aware throughput for fg VM i by $(1 - \rho_{bg,i})$.
4) Interference-aware load balancing: Given these estimates, and the monitored a value, DIAL derives the LB weights, $\vec{p^*}$, via Eq. (3), and inputs them to the LB.

We continue monitoring the VMs' performance to detect further changes in interference and to detect the end of interference. To this end, we ensure that a small number of requests are sent to the affected VMs so we can monitor the progress of interference; we can also use probes for this purpose, as suggested by recent work [22]. When T_{90} returns to normal (for successive intervals), we reset the LB weights.

The monitoring interval length employed by the controller depends on the stability of the fg application and the noise in the system. We use a length of 10s based on the sensitivity analysis for our specific setup as discussed in Section IV-A.

1) DIAL implementation

We implement the above DIAL controller logic using: (i) a C program to execute the detection, classification, and estimation tasks, and (ii) a set of bash scripts to monitor metrics from the `/proc` subsystem and the LB logs, and to communicate with the LB to reconfigure the weights. The overhead of the DIAL controller is negligible in practice since the decision tree building, response time modeling, and LB weights optimization are performed offline, and only need to be leveraged during run time using the monitored metrics. Our evaluation results show that the average increase in CPU utilization of the LB VM under DIAL is about 2%. Of course, if the CPU usage at the LB VM is a concern, we can implement DIAL on a separate VM. In our experiments, we use the HAProxy LB [17]; however, DIAL can also be integrated with the nginx and Apache LBs.

2) Training the DIAL controller

DIAL requires some model training to build the decision tree (Section II-B2) and derive the coefficients of the estimation model (Eq. (1)). The above training tasks can be performed offline on a dedicated server in a cloud environment by controlling the bg VMs to run microbenchmarks at different intensities while monitoring relevant metrics. In a private cloud environment, such as OpenStack, we can set aside a dedicated host using Availability Zones and Host Aggregates. In some public cloud environments, such as Amazon, dedicated hosts can be rented on a pay-as-you-go basis. We use these options for training the DIAL controller using a simple set of microbenchmarks and then highlight the performance improvements under a different set of realistic workloads that were not used for training (see Section IV-B).

III. Experimental Setup and Methodology

We now detail our cloud environments, fg applications (that face contention), and bg workloads (that create contention).

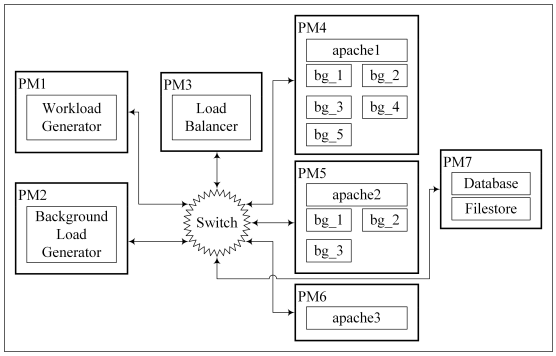


Fig. 3. Illustration of our OpenStack cloud setup.

A. Cloud environments

We set up two cloud environments for our evaluation, an OpenStack based private cloud environment and an AWS-based public cloud environment. Unless specified otherwise, we use the OpenStack environment.

OpenStack-based private cloud: Figure 3 depicts our experimental setup. We use an OpenStack Icehouse-based private cloud with several dedicated Dell C6100 physical machines, referred to as *PMs* (each with two 6-core CPUs and 48 GB memory). All PMs are connected to a network switch via a 1Gb Ethernet cable. Our experiments reveal that the maximum achievable network bandwidth is about 115 MB/sec (we flood the network using a simple load generator, httperf, and measure the peak observed bandwidth under various request rates and request sizes). Likewise, we find that the maximum achievable memory and (sequential) hard disk drive I/O bandwidths are about 11GB/sec (using RAMspeed) and about 50MB/sec (using sysbench), respectively.

AWS-based public cloud: We rent 10 c4.large instances, each with 2 vCPUs and 3.75GB of memory, in AWS EC2's US East (N. Virginia) region. We also rent a c4 dedicated server (PM) for hosting one of the instances colocated with bg VMs.

B. Foreground (fg) applications

We employ two multi-tier web applications for our fg evaluation, CloudSuite [23] and WikiBench [24]. Unless specified otherwise, we use CloudSuite as our fg application.

CloudSuite: The CloudSuite 2.0 Web Serving benchmark is a multi-tier, multi-request class, PHP-MySQL based social networking application. The benchmark uses several request classes such as HomePage, TagSearch, EventDetail, etc.

Our CloudSuite setup consists of: (i) Faban workload generator for creating realistic session-based web requests; we set the number of users to 1000 for OpenStack and 5000 for AWS; the think time is 5s (default). (ii) HAProxy LB distributes incoming http requests (from Faban) among the back-end application VMs. We use the default Round Robin policy, similar to the LBs under Google Cloud Platform [25], unless stated otherwise (as in Section IV-F where we compare with other policies). (iii) Application VMs installed with Apache, PHP, Memcached, and an NFS-Client. We employ 3 application VMs in OpenStack and 10 in AWS. (iv) A MySQL server and an NFS server, hosting the file store, are installed on separate, large VMs (to avoid being the bottleneck).

WikiBench: WikiBench is a Web hosting benchmark that mimics wikipedia.org. Our WikiBench setup consists of: (i) wikijector load generator to replay real traffic from past traces of requests to Wikipedia, (ii) HAProxy LB, (iii) Three servers running the MediaWiki application (the same application that hosts wikipedia.org), and (iv) a MySQL database to store data from the Wikipedia database dump.

C. Background (bg) workloads

In our experiments, we emulate interference by employing several bg workloads to create contention for the fg application. This approach of creating interference, which is similar to other prior works such as ICE [1], Paragon [11], and DeepDive [6], allows us to reproduce the same interference pattern to fairly evaluate performance with and without DIAL. The bg workloads are hosted on VMs colocated with the fg application layer VMs. Each fg VM under interference is hosted separately from other fg VMs, and is colocated with bg VMs. We first employ *microbenchmarks* to stress individual resources for analyzing fg interference. We then employ *test workloads* to evaluate DIAL for fg applications under realistic cloud workloads.

Microbenchmarks: We employ: (i) stress-ng tool on bg VMs to create controlled CPU contention; (ii) httperf load generator (on a separate VM and PM) to retrieve hosted files from the colocated bg VMs at different, controllable request rates to create NET contention; (iii) dcopy benchmark on bg VMs to create LLC contention; and (iv) stress on bg VMs to create DISK contention.

Test workloads: We employ: (i) SPEC CPU to create CPU contention, (ii) Memcache server (driven by mutilate client) to create NET contention, (iii) STREAM to create LLC contention, and (iv) Hadoop running TeraSort with a large data set to create DISK contention.

D. Resource usage monitoring

- NET: We use the dstat Linux tool to monitor the used NET bandwidth for bg and fg VMs. We then normalize their sum by the peak network bandwidth to get NET resource usage.
- CPU: We consider fair-sharing of the possibly over-committed PM cores among VMs to compute CPU usage. If a PM has n cores available and all VMs together require m cores, then the CPU usage of each VM is normalized by $max\{m, n\}$. For example, a PM may have 12 cores ($n = 12$); if we launch 4 VMs with 4 vCPUs each on this PM, since oversubscription is allowed, then the total request is 16 ($m = 16$). If one of the VMs has a CPU usage of x% out of 400% (or y% out of 100%), then we estimate its CPU usage as $\frac{x}{16}$ (or $\frac{4 \cdot y}{16}$). Thus, if all 4 VMs are at 400% each (or 100% per vCPU), then total usage is 1 or 100%.
- LLC: Since memory bandwidth for a VM cannot be easily monitored, we employ the RAMspeed benchmark to measure the available memory bandwidth. We obtain this bandwidth for each experiment and then estimate the LLC usage by computing the difference between peak bandwidth and

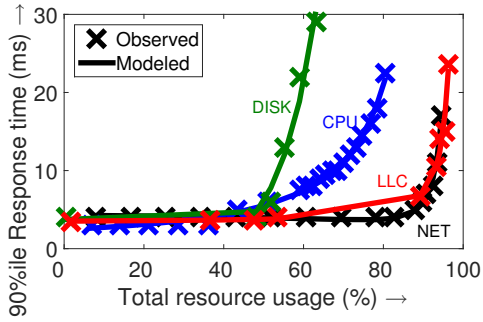


Fig. 4. Observed and modeled response times for CloudSuite under various resource contentions via microbenchmarks. Average modeling error is 6.1%.

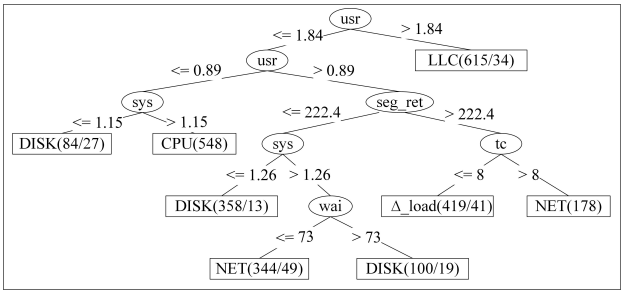


Fig. 5. Illustration of our trained decision tree created using WEKA. Leaves represent the contention classification. Numbers in the leaf represent the total classification instances (left) and the number of misclassified ones (right).

experiment bandwidth. Finally, we normalize this difference by peak bandwidth to estimate LLC usage.

- DISK: Disk usage typically depends on the access pattern (sequential vs. random). We thus use the same approach as for LLC, but with sysbench instead of RAMspeed, for estimating DISK usage.

IV. Evaluation

We first present results for classification and estimation of test workloads. We then present results for performance improvement (reduction in T_{90}) under DIAL for OpenStack and AWS setups for CloudSuite and WikiBench. Unless mentioned otherwise, we compare performance under DIAL with performance without DIAL, referred to as *baseline*. In Section IV-F we compare DIAL against existing interference-aware techniques that are popularly employed.

A. Evaluating detection, classification, and estimation

Detection: The crosses in Figure 4 show the impact of different resource contentions, created by microbenchmarks, on CloudSuite's HomePage request class response time under the OpenStack setup. Every data point (cross) in Figure 4 is obtained by averaging the 90%ile of response times in every monitoring interval over three different experiments, each of which takes 300s. To detect contention, we use the 95% confidence intervals around the mean (see Section II-B1) to obtain the following detection rule for both the OpenStack and AWS setups: $T_{90} > 5ms$, for HomePage; similar rules can be derived for other request classes. We run several experiments using the bg test workloads and find that our detection rule results in a low false positive rate of 5.7%.

Prior work has employed similar techniques to detect and analyze interference using hardware performance counters such as CPI [26], MIPS [11], cache miss rate [1, 2], etc.; such values are visible to the hypervisor, but are difficult and often infeasible to obtain from within the VM. We tried accessing such counters through VMs hosted by AWS EC2, Google Cloud Platform, and our OpenStack environment, but the values were either not supported or were incorrectly reported as all zeros; similar observations were made for AWS EC2 VMs in prior work [2]. By logging application response times to analyze interference, DIAL can be employed directly by cloud users *without* requiring access to such counters.

Classification: We monitor the user space CPU utilization, usr, the kernel space CPU utilization, sys, the I/O wait time, wai, the rate of segments retransmitted, seg_ret, and the 90%ile time taken to establish a connection to the application VM, T_c (via HAProxy logs). We normalize usr and sys using predicted values to distinguish from workload variations, as discussed in Section II-B2. The usr and sys metrics can help detect CPU and LLC contention as the processor might have to do more work under these contentions. wai could potentially help classify DISK contention. Finally, seg and T_c could help classify NET contention because of the reduced available network bandwidth.

Our decision tree for CloudSuite, trained using microbenchmarks, is shown in Figure 5. The decision tree is generated using WEKA [27]; in particular, WEKA determines the nodes and cutoff values using the J48 algorithm. The tree structure may be different for different applications. However, we expect the high level rules to be the same. For example, we expect that LLC interference will lead to an increase in CPU usage.

Our 10-fold cross-validation error is *7.8%*. Our classifier shows that high (normalized to predicted contention) usr signals LLC contention, possibly because more work has to be done to service the LLC misses. A high T_c signals NET contention, which seems intuitive. A moderate drop in usr and moderate rise in sys signals CPU contention; we believe this is because throughput decreases under contention, resulting in lower usr, and thus exhibiting a relative rise in sys. A high wai suggests DISK contention. Finally, a moderate rise in seg_ret and T_c signals workload variations (denoted as Δ_load in Figure 5).

We also evaluated our classifier using test workloads that were *not* seen during classifier training. Here, we run 50 total experiments using 10 experiments each for Memcache (NET contention), SPEC (CPU contention), Hadoop (DISK contention) and STREAM (LLC contention), in addition to 10 experiments under varying CloudSuite application load. Table I summarizes our results. The decision tree successfully classifies 44 of the 50 test instances, including change in workload. The "misclassifications" for DISK contention (as LLC) under Hadoop are because of the numerous memory accesses made by the colocated Slave VMs; we believe that Hadoop interference cannot always be classified as a single resource because of its complex and dynamic resource needs.

Class /	workload	NET	CPU	DISK	LLC
Accuracy	70%	100%	90%	80%	100%

TABLE I
EVALUATING OUR INTERFERENCE CLASSIFICATION USING TEST WORKLOADS. OVERALL ACCURACY=88%.

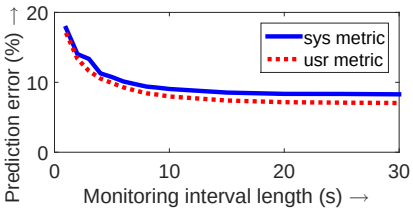


Fig. 6. Smaller monitoring interval lengths lead to higher inaccuracy.

Estimation: The solid lines in Figure 4 show our modeling results for CloudSuite interference estimation (see Section II-B3) under different resource contentions via training. Our average modeling error across all contentions is *6.1%*. If we instead use $\alpha = 1$ in Eq. (1) to model T_{90} simply as $c_0 + c_1/(1-\rho_{fg}-\rho_{bg})$, the modeling error increases to about 15%. However, when we increase the value of α beyond 2, we find only modest improvements in accuracy.

Effect of monitoring interval length: We use a metrics monitoring interval length of 10s for the above evaluation. Experimentally, we find that shorter interval lengths can lead to inaccurate classification and estimation due to system noise and load fluctuations. For example, Figure 6 shows the prediction error for CPU metrics used in our decision tree classifier (see Section II-B2). We see that an interval of 1s or 5s can lead to high inaccuracy. On the other hand, intervals larger than 10s do not significantly improve accuracy. Likewise, we find a significant increase in the number of false positives for interference detection at smaller interval lengths, leading to cases where DIAL overlooks interference. For these reasons, we choose an interval length of 10s; prior work has also reported such reaction times to avoid rash decisions [2, 28, 29].

B. Evaluating DIAL under real workloads

Figure 7 illustrates our experimental results for CloudSuite under OpenStack for various contentions created using test workloads in bg VMs. The y-axis shows the tail response time for CloudSuite across all request classes. Here, we create NET, DISK, and LLC contention for apache1 VM using Memcache, Hadoop (running TeraSort), and STREAM, respectively. We create CPU contention for apache2 using SPEC.

We see that DIAL significantly reduces tail response times, when compared to the baseline, under all contentions; the reduction ranges from 16% under DISK contention to 59% under LLC contention. The relatively low improvement under DISK contention is because, during its execution, Hadoop does not always utilize disk I/O bandwidth; further, not all CloudSuite request classes require (or contend for) disk access.

Without DIAL, the tail response time can be as high as 20-30ms; with DIAL, the tail response time is almost always around 4-5ms. Note that DIAL requires some time (at least two successive intervals of high response time) for interference detection during which response time continues to be high, as seen at the start of each contention.

Figures 8 and 9 show our classification metrics for apache1 and apache2, respectively; we only show T_c, wai, sys, and usr (and not seg_ret) for ease of presentation. Note that the y-axis range in Figure 9 is intentionally smaller to focus on the rise in the sys metric. For apache1, we see that under NET contention, T_c is high while the other metrics are unaffected. For DISK and LLC contentions, sys is high, especially for LLC; further, usr is also high under LLC contention. Finally, the wai metric, though noisy, is higher under DISK contention. By contrast, these metrics are unaffected for the corresponding time periods under apache2.

Likewise, for apache2, for CPU contention, sys is moderately high but not as high as that under DISK and LLC contention under apache1. Again, the metrics are unaffected for the CPU contention period under apache1. This shows that the relevant metrics on the compromised VM change under contention, but are *unaffected for uncompromised VMs*. Further, the change in metric values under the contention periods are in agreement with the rules of the decision tree classifier in Figure 5, even though the classifier was trained on microbenchmarks and not on these test workloads. This highlights the efficacy of our classifier.

Note that it is possible for several resources to be *simultaneously under contention*, as in the case of Memcache (NET, LLC, and DISK); however, it is typically the dominant resource that has most impact on performance. In the case of Memcache, the server is hosted on a bg VM and is driven by mutilate clients (running on different hosts) issuing a high request rate for a small set of key-value pairs, resulting in NET being the dominant resource. DIAL correctly classifies this Memcache bg VM as creating NET contention. For Hadoop, there is significant demand for disk and memory bandwidth; however, our classifier suggests DISK contention.

Finally, this experiment also shows that DIAL can handle *time-varying contentions* exhibited by test workloads (as evidenced by the time-varying metrics during contention in Figures 8 and 9), which are common for real workloads.

C. Evaluating DIAL under multiple contentions

DIAL is also capable of dynamically responding to multiple compromised VMs. This is because our model allows for arbitrary levels of interference on different VMs simultaneously. The optimization in Section II-C1 provides estimates for LB weights, via Eq. (3), for all VMs. Note that this is different from the case of multiple resource contentions on the *same* VM, which is beyond the scope of this paper.

Figure 10 shows our experimental results for CloudSuite where initially apache2 VM is under CPU contention, but then, after about 5 mins, apache1 (on a different host) also starts experiencing NET contention, resulting in very high interference for the application. After an additional 5 mins, both contentions are terminated. We see that DIAL substantially reduces T_{90} under interference. This example highlights the dynamic nature of DIAL. Compared to existing techniques that employ (static) VM placement to mitigate interference, DIAL is able to adapt to *variations in interference* by constantly

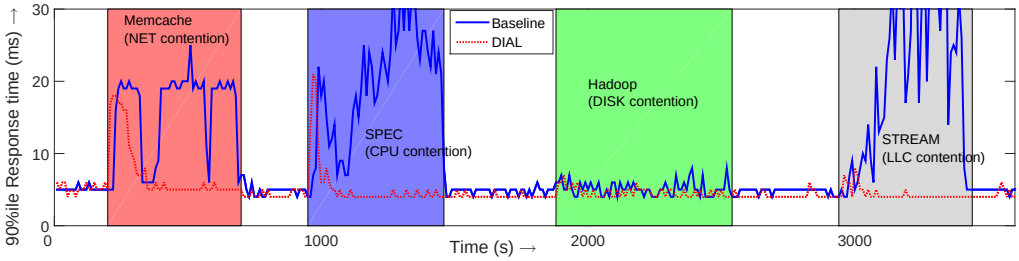


Fig. 7. Performance comparison between DIAL and baseline for test background workloads. The red, blue, green, and gray regions represent NET, CPU, DISK, and LLC contention, respectively. DIAL reduces 90%ile response time during these contentions by 39.1%, 56.3%, 16.2%, and 59.2%.

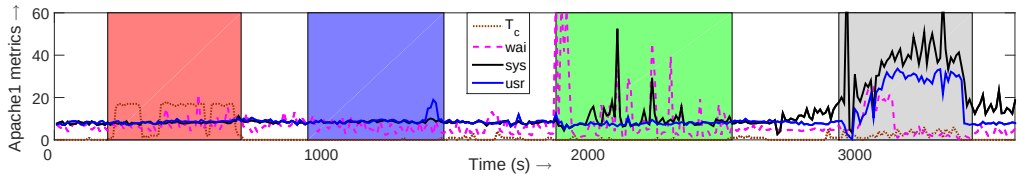


Fig. 8. T_c, wai, sys, and usr metrics for apache1 application layer VM for the experiments in Figure 7. apache1 VM experiences NET, DISK, and LLC contention, and shows an increase in relevant metrics under those contentions.

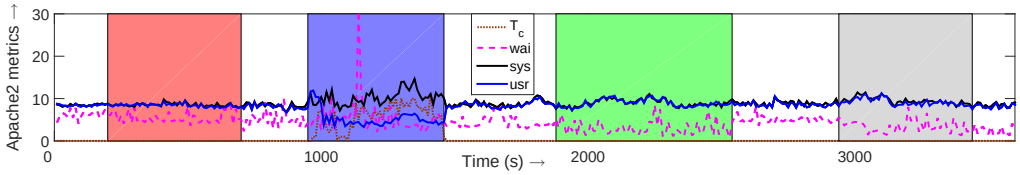


Fig. 9. T_c, wai, sys, and usr metrics for apache2 application layer VM for the experiments in Figure 7. apache2 VM experiences only CPU contention, and consequently shows an increase in relevant metrics under CPU contention.

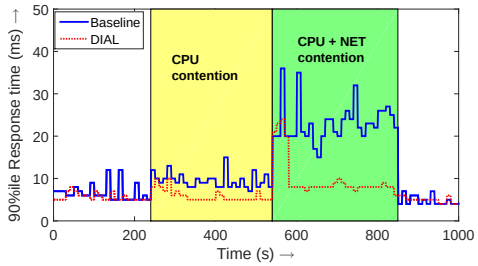


Fig. 10. DIAL reduces the response time of all request classes by 37% under CPU contention and by 56% under the combined CPU + NET contention.

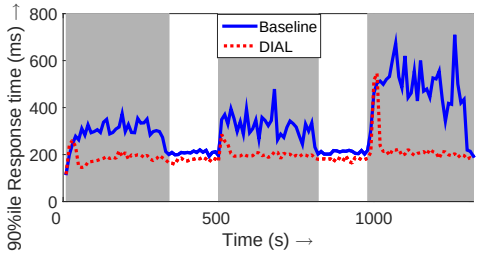


Fig. 11. Performance under LLC contention for fg WikiBench. DIAL reduces response times by around 23.6% during contention (gray regions).

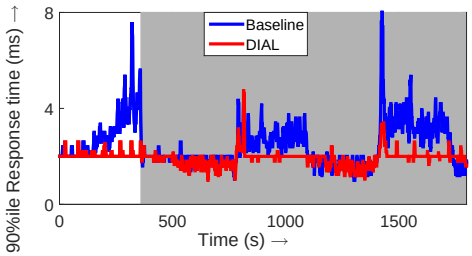


Fig. 12. Performance under LLC contention for AWS setup. DIAL reduces response times by around 22.3%.

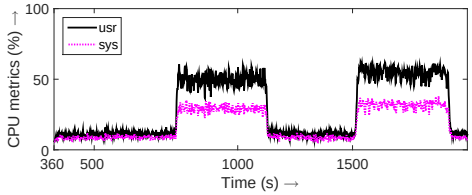


Fig. 13. usr and sys metrics for the gray region in Figure 12. These metrics clearly increase during contention.

updating its estimates and re-distributing load accordingly. For the above experiment, for CPU contention, the DIAL weights are $\{0.45, 0.1, 0.45\}$ (apache2 under contention), and for combined CPU and NET contention, the weights are $\{0, 0.27, 0.73\}$ (apache1 under severe NET contention). Note that if several VMs are under severe contention, we may have to scale out to maintain acceptable response times.

D. Evaluating DIAL for the WikiBench fg application

Figure 11 shows our results for WikiBench under LLC contention created by the dcopy microbenchmark. Here, we have two application VMs and one of them is under contention. The figure shows the response time for baseline and DIAL for all request classes. We create three different contention levels for this experiment, shown in gray. DIAL reduces response time by about 23% when compared to the baseline. We also measure the usr and sys metrics for classification and find that both increase considerably, by about 62% and 41%, respectively, under interference; this is in agreement with our decision tree classifier.

E. Evaluating DIAL in the AWS setup

Figure 12 shows our results for CloudSuite under LLC contention created by the dcopy microbenchmark in the AWS

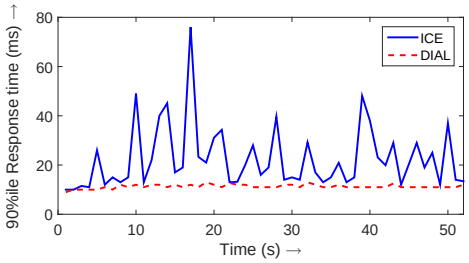


Fig. 14. Comparison of DIAL with ICE under CPU contention. DIAL reduces T_{90} for EventDetail by about 48%; reduction for HomePage and TagSearch classes is about 25%.

setup. Here, we have 10 application VMs and only one of them is under contention. The figure shows the response time for baseline and DIAL for all request classes served by all VMs in the AWS setup. We create several different contention levels for this experiment. We see that DIAL reduces response time by about 22% when compared to the baseline. This shows that *even one* compromised VM (out of 10) can have a considerable impact on the overall response time.

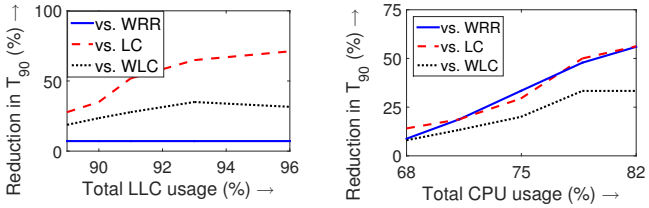
Figure 13 shows the usr and sys metrics for the shaded region in Figure 12 to assess classification. Clearly, both the usr and sys metrics increase considerably during contention when compared to the low, flat lines during no contention. Further, the regions of contention can be easily discerned from the figure, resulting in good detection accuracy.

F. Comparison with existing user-centric techniques

We now present experimental results comparing DIAL with existing user-centric interference mitigation strategies. We do not evaluate cluster management strategies, such as CloudScope [30], and Tarcil [13], since these strategies cannot be implemented by the cloud user who does not have a global view of the infrastructure. Further, users cannot control VM placement or migration.

Utilization-based strategies. Figure 14 shows our experimental results for high CPU contention under DIAL and under ICE [1]. Similar to DIAL, ICE is an interference-aware load balancing strategy that adjusts the traffic directed towards compromised VMs. However, instead of using LB weights, ICE ensures that the CPU utilization for the compromised VMs stays below a certain threshold. The authors do not mention this threshold value in the paper, and so we experimentally determine the best threshold value across experiments. Unfortunately, we find that the optimal CPU utilization threshold varies with the amount and type of interference. For example, we find that 15% CPU utilization works well for moderate CPU interference under ICE, but does not work well for high CPU interference, as illustrated in Figure 14; under DIAL, with theoretically optimal weights, the response time is significantly lower, and the observed CPU utilization at the compromised VM is about 8-10%. The results are similar for other contentions. While interference-aware thresholds could help in this case, this would require a relationship between threshold, type, and amount of interference. Since ICE does not estimate interference, the threshold value is static.

Queue-length based strategies. Queue-length or load-based



(a) TagSearch for LLC contention. (b) TagSearch for CPU contention.

Fig. 15. Comparison of DIAL with other LB heuristics.

strategies send traffic to the VM that has the lowest load. In particular, we consider the Least Connections (LC) strategy that directs the next incoming request to the application VM that has least number of active connections; Amazon leverages LC for its elastic load balancer [31] for this purpose. Under interference, the outstanding requests for the compromised VM will be higher, resulting in fewer additional requests being sent to it under LC.

Figure 15 shows the reduction in T_{90} afforded by DIAL over LC (and other heuristics that we discuss next) for the TagSearch request class under CPU and LLC contentions; results are similar for other classes and for NET and DISK contention. We see that DIAL lowers response time significantly, by as much as 70-80%, when compared to LC (red dashed line). The improvement is greater at higher contentions. The reason for this improvement is that the compromised VM does not just have lower capacity, but also requires (non-linearly) *more time* to serve each request. The response time-minimizing weights under DIAL take both these into consideration, as opposed to LC that only addresses the former.

Weighted load balancing strategies. We now compare DIAL with other weighted load balancing heuristics, such as Weighted Round Robin (WRR) and Weighted Least Connections (WLC). For WRR and WLC, we use proportional interference-aware weights, as discussed in Section II-C. Note that DIAL is essentially WRR with theoretically optimal weights ($\vec{p^*}$, via Eq. (3)).

Figure 15 shows the reduction in T_{90} afforded by DIAL over WRR (blue solid line) and WLC (black dotted line). We see that DIAL lowers response time considerably when compared to these heuristics. It is interesting to note that WRR is typically worse than WLC under CPU contention, but better than WLC under LLC contention; this observation reaffirms the fact that the impact of interference depends on the type of resource under contention.

V. Prior Work in the Context of DIAL

Interference detection: Recent work has emphasized the need for user-centric interference detection [1, 2, 32, 33]. IC^2 [2] employs decision trees using VM-level statistics to detect interference at the cache; this information is then used to tune the configuration of web servers in co-located environments. Casale et al. [33] focus on CPU interference and present a user-centric technique to detect contention by analyzing the CPU steal metric. CRE [34] makes use of collaborative filtering to detect interference in web services by monitoring the response times. While we also monitor response time, unlike CRE, we

go beyond detection and also estimate the amount of interference. CPI^2 [26] employs statistical approaches to analyze an application's CPI metric to detect and mitigate processor interference between threads of different jobs. While CPI^2 can be used in virtual environments, public cloud VMs (e.g., AWS) do not always expose performance counters. Bubble-Up [35] presents a similar approach for the memory subsystem.

There have also been many studies on interference detection from the perspective of the hypervisor (such as ILA [5], TRACON [9], and DejaVu [7]). While effective, such techniques require hypervisor access for monitoring host-level metrics (e.g., global CPU usage from Dom0 or hardware performance counters), which are not always feasible for cloud users.

Interference-aware performance management: ICE [1] proposes interference-aware load balancing by limiting the CPU utilization of the affected VM below a certain threshold. While effective, we find, via experiments (see Section IV-F), that this strategy is not adaptive to different levels of interference. Tarcil [13] and Quasar [14] profile workload classes and carefully colocate workloads that do not significantly impact each others' performance due to their specific resource requirements. By contrast, DIAL does not control colocation (since VM placement is typically not in the user's control); instead, DIAL globally adjusts the LB policy of the fg application to reroute some of the requests directed at affected VMs.

VI. Conclusion

We presented DIAL, a user-centric Dynamic Interference-Aware Load balancer that can be employed directly by cloud users to reduce tail response times during interference. DIAL works by leveraging two important components: (i) An accurate user-centric, response time-monitoring based interference detector, classifier, and estimator, and (ii) A framework for deriving theoretically optimal load balancer weights under interference. Our experimental results using CloudSuite and WikiBench web applications, under interference from benchmarks such as Memcache and Hadoop, on OpenStack and AWS clouds demonstrate the benefits of DIAL.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Dr. Lydia Chen, for their helpful suggestions. This work was supported by NSF grants CNS-1617046 and CNS-1464151.

References

[1] A. Maji et al., "ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services," in *Proceedings of ICAC 2015*, pp. 91–100.

[2] ——, "Mitigating Interference in Cloud Services by Middleware Reconfiguration," in *Proceedings of Middleware 2014*, pp. 277–288.

[3] C. Wang et al., "Effective Capacity Modulation as an Explicit Control Knob for Public Cloud Profitability," in *Proceedings of ICAC 2016*, pp. 95–104.

[4] Y. Xu et al., "Small is Better: Avoiding Latency Traps in Virtualized Data Centers," in *Proceedings of SOCC 2013*, pp. 7–16.

[5] X. Bu et al., "Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters," in *Proceedings of HPDC 2013*, pp. 227–238.

[6] D. Novakovic et al., "Deepdive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," in *Proceedings of USENIX ATC 2013*, pp. 219–230.

[7] N. Vasic et al., "DejaVu: Accelerating Resource Allocation in Virtualized Environments," in *Proceedings of ASPLOS 2012*, pp. 423–436.

[8] M. Zaharia et al., "Improving MapReduce Performance in Heterogeneous Environments," in *Proceedings of OSDI 2008*, pp. 29–42.

[9] R. C. Chiang et al., "TRACON: Interference-Aware Scheduling for Data-Intensive Applications in Virtualized Environments," in *Proceedings of SC 2011*, pp. 1–12.

[10] R. Nathuji et al., "Q-clouds: Managing Performance Interference Effects for QOS-Aware Clouds," in *Proceedings of EuroSys 2010*, pp. 237–250.

[11] C. Delimitrou and C. Kozyrakis, "QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon," *ACM Transactions on Computer Systems*, vol. 31, no. 4, 2013.

[12] S. Nathan et al., "Towards a Comprehensive Performance Model of Virtual Machine Live Migration," in *Proceedings of SOCC 2015*, pp. 288–301.

[13] C. Delimitrou et al., "Tarcil: High Quality and Low Latency Scheduling in Large, Shared Clusters," in *Proceedings of SOCC 2015*, pp. 97–110.

[14] ——, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of ASPLOS 2014*, pp. 127–144.

[15] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

[16] K. Leonard, *Queueing Systems, Volume 2*. New York: Wiley-Interscience, 1976.

[17] "HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer," http://www.haproxy.org.

[18] Q. Zhang et al., "A Regression-based Analytic Model for Capacity Planning of Multi-tier Applications," *Cluster Computing*, vol. 11, no. 3, pp. 197–211, 2008.

[19] T. Horvath et al., "Multi-mode Energy Management for Multi-tier Server Clusters," in *Proceedings of PACT 2008*, pp. 270–279.

[20] M. I. Reiman and B. Simon, "An Interpolation Approximation for Queueing Systems with Poisson Input," *Operations Research*, vol. 36, no. 3, pp. 454–469, 1988.

[21] M. Boon, E. Winands, I. Adan, and A. van Wijk, "Closed-form waiting time approximations for polling systems," *Performance Evaluation*, vol. 68, no. 3, pp. 290 – 306, 2011.

[22] J. Mukherjee et al., "Resource Contention Detection in Virtualized Environments," *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 217–231, 2015.

[23] M. Ferdman et al., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of ASPLOS 2012*, pp. 37–48.

[24] E.-J. Van Baaren, "WikiBench: A distributed, Wikipedia based web application benchmark," Master's thesis, Vrije Univesiteit Amsterdam, the Netherlands, 2009.

[25] "Google Cloud Platform: Setting Up HTTP(S) Load Balancing," https://cloud.google.com/compute/docs/load-balancing/http.

[26] X. Zhang et al., "CPI^2: CPU Performance Isolation for Shared Compute Clusters," in *Proceedings of EuroSys 2013*, pp. 379–391.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[28] X. Zhang et al., "CPI^2: CPU Performance Isolation for Shared Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013, pp. 379–391.

[29] David Lo et al., "Heracles: Improving Resource Efficiency at Scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[30] X. Chen et al., "CloudScope: Diagnosing and Managing Performance Interference in Multi-tenant Clouds," in *Proceedings of MASCOTS 2015*, pp. 164–173.

[31] "AWS: How Elastic Load Balancing Works," http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/how-elb-works.html.

[32] S. A. Javadi et al., "UIE: User-centric Interference Estimation for Cloud Applications," in *Proceedings of IC2E 2016*, pp. 119–122.

[33] G. Casale et al., "A Feasibility Study of Host-level Contention Detection by Guest Virtual Machines," in *Proceedings of CloudCom 2013*, pp. 152–157.

[34] Y. Amannejad et al., "Detecting Performance Interference in Cloud-based Web Services," in *Proceedings of IFIP IM 2015*, pp. 423–431.

[35] J. Mars et al., "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in *Proceedings of MICRO 2011*, pp. 248–259.