User-Centric Interference-Aware Load Balancing for Cloud-Deployed Applications

Seyyed Ahmad Javadi and Anshul Gandhi

Abstract—VMs deployed in cloud environments are prone to performance interference due to dynamic and unpredictable contention for shared physical resources among colocated tenants. Current provider-centric solutions, such as careful co-scheduling of VMs and/or VM migration, require a priori profiling of customer VMs, which is infeasible in public clouds. Further, such solutions are not always aware of the user's SLO requirements or application bottlenecks. This paper presents DIAL, an interference-aware load balancing framework that can directly be employed by cloud users without requiring any assistance from the provider. 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 dynamically shift load away from compromised VMs without violating the application's tail latency SLOs. We implement DIAL for web and online analytical processing applications, and show, via experimental results on OpenStack and AWS clouds, that DIAL can reduce tail latencies by as much as 70 percent compared to existing solutions.

Index Terms—Cloud computing, performance interference, load balancing

1 Introduction

The benefits of cloud computing are undeniable – low cost, elasticity, and the ability to pay-as-you-go. Not surprisingly, many online services and applications are now hosted on the cloud, on virtual machines (VMs). Despite its popularity, however, applications deployed in the cloud can experience undesirable performance effects, the most severe of which is *interference*. Performance interference is caused by contention for physical resources, such as CPU or last-level cache, among colocated VM users/tenants.

Interference is an undesirable side-effect of a fundamental design principle of the cloud, namely, *multi-tenancy* (sharing of a physical server among users). While certain resources, such as CPU, can be partitioned among colocated VMs by cloud providers, other resources, such as processor caches, are notoriously hard to partition [1]. Nonetheless, partitioning of resources among tenants can adversely impact cloud resource utilization. Further, resource contention depends on the workload of all colocated tenant VMs, and is thus *dynamic and unpredictable* [2]; as a result, static partitioning is not a useful solution.

Prior work on interference mitigation has typically focused on provider-centric solutions. A popular approach is to profile applications and co-schedule VMs that do not contend on the same resource(s) [3], [4], [5]. 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 [6] and can incur overheads [7], especially for stateful applications [8].

A key challenge that has not been addressed with regards to interference is the *lack of visibility and control between the provider and the tenant*, especially in public clouds [9]. Specifically, tenant VMs in a public can not, or should not, be profiled a priori by the provider due to privacy concerns [10]. Further, providers are not always aware of the cloud user's Service Level Objective (SLO) requirements or the user application's bottleneck resources.

In this paper, we make the case for a *user-centric* interference mitigation approach which can be employed by the tenant without requiring any assistance from the provider or hypervisor. Such user-centric solutions empower the tenant to have greater control on their application performance, which is often the most important criteria for users. Further, user-centric solutions are, by definition, aware of the user application and its SLOs, and can appropriately react to the onset or termination of interference.

We present DIAL, a dynamic solution for mitigating interference in load-balanced cloud deployments. We consider a generic cloud-deployed application that has a tier of worker nodes hosted on multiple VMs and experiencing unpredictable interference from colocated VMs (owned by other cloud tenants), as shown in Fig. 1. The incoming load is distributed among the worker nodes via one or more load balancers. This generic model is widely applicable, for example, for web applications (where workers are web or application servers), online analytical processing (OLAP) systems like Pinot [11], etc.

The key idea behind DIAL is to *infer* contention in colocated VMs. Specifically, by monitoring its own application performance, the user can estimate the colocated load that can induce the observed level of performance degradation, 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

• *The authors are with the PACE Lab; Stony Brook University, Stony Brook, NY, 11794 USA. E-mail: {sjavadi, anshul}@cs.stonybrook.edu.*

Manuscript received 28 Aug. 2018; revised 12 June 2019; accepted 10 Sept. 2019. Date of publication 24 Sept. 2019; date of current version 8 Mar. 2022.
(Corresponding author: Seyyed Ahmad Javadi.)
Recommended for acceptance by A. Deshpande.
Digital Object Identifier no. 10.1109/TCC.2019.2943560

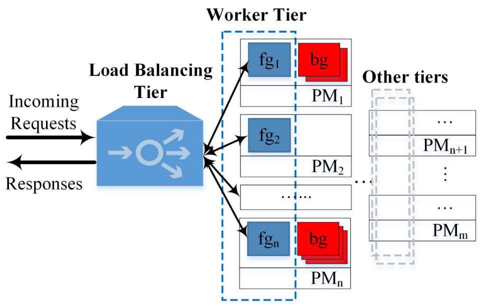


Fig. 1. Illustration of a generic cloud application containing LBC and processing tier deployed on multiple foreground (fg) VMs experiencing interference from background (bg) VMs.

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 [12], [13], to understand the impact on performance of contention at shared physical resources. DIAL then optimizes the time-varying load distribution among worker VMs to reduce tail latency.

We implement and experimentally evaluate DIAL for two specific application classes:

1) *Web applications:* We implement DIAL on HAProxy [14], and evaluate DIAL's benefits using two 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 percent compared to interference-oblivious load balancers. Further, compared to existing interference-aware solutions, DIAL reduces tail response times by as much as 48 percent.
2) *OLAP Systems:* We implement DIAL for a popular and open-source OLAP system called Pinot that has being used in production clusters at LinkedIn and Uber. Our experimental results on a KVM cluster show that DIAL can reduce 95%ile query completion times by 16-40 percent under CPU and LLC contention.

A preliminary version of this paper appeared in the ICAC 2017 conference, but only focused on web applications [15]. This version extends the performance modeling, optimization, implementation and evaluation of DIAL for the case of OLAP systems, such as Pinot [11]. Further, this version provides the analytical proof sketches for the theoretical results that guide DIAL.

2 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.

2.1 Problem Statement and Scope

Fig. 1 illustrates a typical multi-tier cloud deployed application. The worker nodes process the incoming requests, and are illustrated as a tier of VMs. Incoming requests are distributed among the worker nodes using an application-specific scheduler or dispatcher or load balancer; we abstract this entity as a Load Balancing Tier (LBT). Our focus in this paper is on the worker nodes and the LBT; specifically, we propose a new technique to dynamically infer the interference on worker nodes and adjust the load balancing weights for the worker VMs in the LBT. We assert that the LBT is ideally suited to mitigate the effects of volatile interference on worker VMs as the LBT acts at the front-end for the worker tier.

The worker tier is hosted on multiple foreground (fg) VMs, each of which is hosted on a physical machine (PM); we highlight the worker tier fg VMs in Fig. 1. Each PM may also host background (bg) VMs that do not belong to the fg user, as shown in Fig. 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.

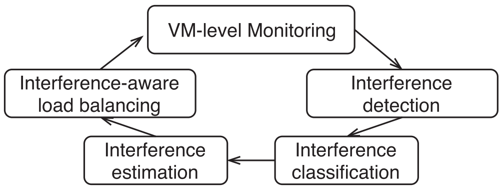


Fig. 2. Illustration of DIAL's control flow.

2.2 DIAL Overview

Our solution, DIAL, is a user-centric dynamic Interference-Aware Load Balancing framework. 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 2.3)
(ii) Given this information, how should users dynamically distribute load among their VMs to minimize tail latencies in the presence of interference? (Section 2.4)

The key idea in DIAL is to estimate, from within a user VM, the amount of interference being induced by colocated VMs, and then adapt the incoming load intensity for each user VM accordingly. Fig. 2 shows a high-level overview of DIAL's control flow. DIAL monitors performance metrics from within the VMs and signals interference if tail latency goes above a certain threshold (detection, see Section 2.3.1). DIAL then determines if the detected event is a load change for the application or a resource contention event (classification, see Section 2.3.2). Depending on the source of contention, DIAL quantifies, or infers, the severity of resource contention (estimation, see Section 2.3.3). Based on this quantification, DIAL determines the theoretically-optimal load balancing weights that the user application should employ to mitigate the impact of contention (see Section 2.4). The above steps are continually employed at runtime, enabling DIAL to respond dynamically to contention.

2.3 User-Centric Estimation of Interference

We define *amount of interference* as the fraction of available physical resources that are in use by colocated background VMs. In the context of Fig. 1, the amount of interference is the fraction of physical resources on a PM that are in use by

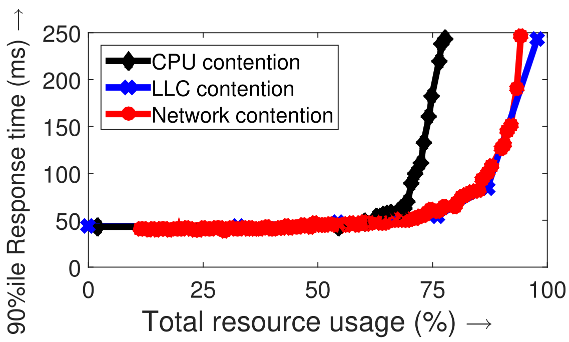


Fig. 3. Performance of an OpenStack-deployed Apache web server under interference from colocated VMs running microbenchmarks.

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 non-trivial as it requires classification and modeling of interference.

2.3.1 Impact of Interference on Tail Latencies

Interference is known to impact application response times [3], [16], [17]. 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.

Fig. 3 shows the impact of different types of resource contention on the 90%ile response time 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 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. DIAL uses the first observation to *detect* when the fg application VM is under interference. Specifically, from Fig. 3, we see that application response times, or T_x, 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*. Thus, DIAL signals interference when T_x goes beyond the 95 percent *confidence intervals* (around the mean of periodically monitored tail latencies) observed during no or low interference.

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.

2.3.2 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. Our key idea in classification is to observe the impact of interference on user metrics, such as CPU utilization and I/O wait time, which can be easily obtained from within the VM via the `/proc` subsystem.

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 4.3.2).

Distinguishing Interference from Workload Variations. An application's response time can degrade for various reasons, such as workload surges, in addition to interference. While our detection methodology detailed in Section 2.3.1 does not distinguish between interference and workload variations, DIAL makes this distinction at the classification stage by again leveraging the decision tree classifier. Specifically, 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.

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.

2.3.3 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 2.4).

From Fig. 3, 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 [12]. Mathematically, $T_x \sim 1/(1-\rho_{fg})^{\alpha}$, 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. Prior theoretical work has also shown that a quadratic term in the denominator can result in better predictability under high loads [20]. 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 resource utilization. We thus approximate x%ile response time as:

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

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 (linear term in denominator) and high load (quadratic term in denominator) regimes.

To determine the coefficients, we train the model in Eq. (1) by creating different levels of resource usage and monitoring the T_x of fg VMs (see Section 2.5.1). We then use multiple linear regression over this training data to derive the resource-specific coefficients. While Eq. (1) is inspired by queueing models, it can accurately track the relationship between tail response times and resource usage for realistic web applications, as we show in Section 4.3.

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_x and ρ_{fg}, and solving Eq. (1) for ρ_{bg}.

2.4 Interference-Aware Load Balancing

Interference-aware load balancing is the key component of DIAL. When there is no interference, balancing the load among 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 4.3.

2.4.1 Minimizing Tail Response Times for Web Applications

To minimize application tail response times under interference, we again employ queueing theory. We first consider the case where any VM can serve an incoming request, as in the case of a web application tier. 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 load balancer (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 x%ile response time, T_x.

To obtain a simple closed-form expression for the theoretically optimal p_is, we model each VM as an M/M/1 system. By focusing on the dominant resource that is causing interference, as classified using the decision tree, we employ the M/M/1 model to represent the contention at the dominant resource. While this is an oversimplification, the resulting closed-form tail latency expression enables the optimization and determination of theoretically-optimal load balancer weights. We note that the resulting p_is are only optimal under the M/M/1 model; we refer to these as the "theoretically optimal" weights in the rest of the paper.

For the M/M/1 model, the response time is known to follow an Exponential distribution [12]. We can thus obtain any tail probability of response time by using the CDF of the Exponential distribution. Under the M/M/1 assumption, T_x for a cluster of n VMs is approximated as:

$$T_x \approx \sum_{i=1}^{n} p_i \cdot \frac{-\ln(1-x/100)}{r_i - a\cdot p_i}, \tag{2}$$

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.

Eq. (2) above works for all percentiles of response time. For example, if $x = 90$, meaning we focus on the 90%ile response time, then the term in the numerator becomes $-\ln(1-0.9) = \ln 10$. For 95%ile response times, the numerator becomes $\ln 20$. Interestingly, the optimization for p_is discussed below does *not* depend on the numerator value (since it is independent of p_i), and thus *our results apply, as-is, for any percentile of response times*, including the median.

Given a (monitored at the LB) and r_i (derived as discussed above using interference estimation from Section 2.3), T_x 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_x in Eq. (2) via calculus, as presented below.

Lemma 1. *The theoretically optimal load split for minimizing T_x for a cluster of n VMs with total arrival rate a and individual VM throughputs r_i is given by:*

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

Proof. The proof proceeds via mathematical induction on n. We first prove the base case for $n=2$. Let the probability of sending a request to VM 1 (with throughput r_1) be p; thus, arrival rate into VM 1 is $a\cdot p$. Then, under the M/M/1 queueing model [12], the response time for VM 1 is distributed as $Exp(r_1 - a\cdot p)$. Based on this, the x%ile response time is $\frac{-\ln(1-x/100)}{r_1-a\cdot p}$. Likewise, the x%ile response time for VM 2 (with arrival rate $a\cdot(1-p)$) is $\frac{-\ln(1-x/100)}{r_2-a\cdot(1-p)}$. We now approximate T_x for the 2-VM system as:

$$T_x \approx p\cdot\frac{-ln(1-x/100)}{r_1-a\cdot p} + (1-p)\cdot\frac{-ln(1-x/100)}{r_2-a\cdot(1-p)}.$$

We now derive the optimal value of $0 \le p \le 1$ that minimizes T_x. Taking the derivative of T_x w.r.t. p, we get:

$$p_1^* = p^* = \frac{r_1\sqrt{r_2} - r_2\sqrt{r_1} + a\sqrt{r_1}}{a(\sqrt{r_1}+\sqrt{r_2})}.$$

Now assume that the above expression for p^* is true for $n=k$. Then, for $n=(k+1)$, we partition the $(k+1)$ VM system into a single VM with request probability p_n and a k-VM system with request probability $(1-p_n)$. For the k-VM system (with primed variables) with request rate

$a' = a \cdot (1 - p_n)$, by the inductive hypothesis, we have:

$$p_i'^* = \frac{r_i \sum_{j=1}^{k} \sqrt{r_j} - \sqrt{r_i} \sum_{j=1}^{k} r_j + a' \sqrt{r_i}}{a' \sum_{j=1}^{k} \sqrt{r_j}}.$$

The approximate x%ile response time for the $(k+1)$-VM system can then be written as:

$$T_x \approx p_n \cdot \frac{-ln(1 - x/100)}{r_n - a \cdot p_n} + (1 - p_n) \cdot \sum_{j=1}^{k} p_i'^* \cdot \frac{-ln(1 - x/100)}{r_i - a' \cdot p_i'^*} \tag{4}$$

Note that T_x is itself a function of p_n since request rate for the k-VM system is $a' = a(1 - p_n)$). We now derive the theoretically optimal p_n^* by differentiating Eq. (4) to get:

$$p_n^* = \frac{r_n \sum_{j=1}^{n} \sqrt{r_j} - \sqrt{r_n} \sum_{j=1}^{n} r_j + a \sqrt{r_n}}{a \sum_{j=1}^{n} \sqrt{r_j}}. \tag{5}$$

The remaining k theoretically optimal probabilities can then be derived by noting that $p_i^* = (1 - p_n^*) \cdot p_i'^*$. □

Note that p_i^* depends on the estimates of r_i, thus necessitating the interference estimation of Section 2.3. 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.

2.4.2 Minimizing Tail Response Times for OLAP Applications

We now extend the above analysis to the case where only a subset of workers (replicas) can serve an incoming request due to data locality, as in the case of OLAP systems. Let the number of replicas be c. Let us first consider the case where one worker, say w, out of n, is under interference. Let the non-interference throughput be r and that of w be $r_w < r$.

In the absence of interference, $1/c$ fraction of requests that have a replica on w would be sent there by the LB; further, the arrival rate to w would be a/n, assuming a fair distribution of replicas among workers. In the presence of interference, let the fraction sent to w be p, and so the fraction sent to the remaining $(c-1)$ replicas is $\frac{1-p}{c-1}$. Thus, the arrival rate into w is now $\frac{a}{n} \cdot \frac{p}{1/c} = \frac{a}{n} \cdot p\, c$, and the fraction of *all* requests that go to w is $q = \frac{p\, c}{n}$. Likewise, arrival rate into *each* non-interference worker is $\frac{a}{n} + \frac{a}{n} \cdot \frac{1-p\, c}{(n-1)} = a \cdot \frac{1-q}{n-1}$, and the fraction of all requests that go to each non-interference worker is $\frac{1-q}{n-1}$.

Using the M/M/1 model [12], we have, similar to Eq. (2):

$$T_x \approx q \cdot \frac{-\ln(1 - x/100)}{r_w - a \cdot q} + (n-1) \cdot \frac{1-q}{n-1} \cdot \frac{-\ln(1 - x/100)}{r - a \cdot \frac{1-q}{n-1}}. \tag{6}$$

Observe that Eq. (6) is exactly the same as Eq. (2) when $r_1 = r_w$ and $r_i = r$ for $i \neq 1$, except that p_1 is replaced by q. Thus, the theoretically optimal solution, via Eq. (3), is $q^* = p_1^*$, and thus the theoretically optimal split for the worker under interference is $q^* \cdot \frac{n}{c} = p_1^* \cdot \frac{n}{c}$. Intuitively, this result says that more load needs to be placed on the interference worker in case of OLAP when compared to web applications; this makes sense as there are fewer alternative workers in case of OLAP applications as opposed to web applications, i.e., $(c-1)$ as opposed to $(n-1)$. We can similarly obtain the theoretically optimal split when more than one worker is under interference.

2.5 The DIAL Control Flow

The control flow for our DIAL implementation (for web and OLAP applications) is as follows:

0) Monitoring: DIAL monitors the fg application's T_x, arrival rate, a, 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_x exceeds its 95 percent confidence bounds for successive monitoring intervals.
2) Classification: DIAL next employs the decision tree to identify the dominant resource under contention.
3) Estimation: DIAL then uses the T_x and $\rho_{fg,i}$ values in Eq. (1), with the dominant resource-specific coefficients, to estimate the interference, $\rho_{bg,i}$. The interference-aware throughput for fg VM i is adjusted 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. When T_x returns to normal (for successive intervals), we reset the LB weights.

2.5.1 Training the DIAL Controller

DIAL requires some model training to build the decision tree (Section 2.3.2) 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. In some public clouds, such as Amazon, dedicated hosts can be rented. We use these options for training the DIAL controller, as discussed in Sections 4.3 and 5.4.

2.5.2 Assumptions for DIAL Controller Training

The above-described DIAL control flow and training makes certain implicit assumptions about the incoming workload. Specifically, by training at different load intensities, DIAL assumes that (i) the workload request mix does not change significantly at runtime, and (ii) the distribution of inter-arrival times does not change significantly at runtime. When the request mix changes, for example, to a more database-heavy request mix, then the workload will have a greater sensitivity to disk I/O contention. This will thus require a retaining of the DIAL controller to infer the new model parameters. Note that the mean arrival rate and/or the number of workers may change dynamically, and this is already monitored by DIAL and is taken into account when determining the theoretically optimal load balancing weights via the a and n parameters, respectively. We show, in Section 5.4.3, that DIAL works well even under an abrupt request rate change and a change in the number of workers.

3 Evaluation Methodology

To evaluate the efficacy of DIAL, we implement it for realistic applications and study the reduction in tail latency when worker nodes face interference. This section describes the fg and bg application setup we employ, and our resource monitoring approach for worker nodes.

3.1 Foreground (fg) Applications

3.1.1 Web Applications

We consider multi-tier web applications where the application server tier is treated as the worker tier. The incoming requests are distributed among application servers via a load balancer.

We employ two multi-tier web benchmarks as our fg application, CloudSuite [22] and WikiBench [23]. Unless specified otherwise, we use CloudSuite in foreground.

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, e.g., 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 tier VMs. We use the default Round Robin policy, unless stated otherwise. (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, and (iii) three VMs running the MediaWiki application (the same application that hosts wikipedia.org), and (iv) a MySQL database to store the Wikipedia database dump.

3.1.2 OLAP Applications

We use the open-source Pinot [11] system as our representative OLAP application. Pinot is a low-latency, scalable, distributed OLAP data store that is used at LinkedIn and Uber for various user-facing functions and internal analysis. The Pinot architecture consists of three main components: (1) controller, (2) broker, and (3) historical worker nodes. The controller is responsible for cluster-wide coordination and segment (data shard) assignment to worker nodes. The broker(s) receives queries from clients, distributes them among workers, and integrates the results from the workers and sends the final result back to clients. The brokers act as our load balancing tier (LBT), see Section 2.1. The historical worker nodes host data segments and respond to queries that originate from the broker. The worker nodes constitute our worker tier. Historical workers store data in the form of an index called *segment*; every table has its own segments.

For all the above fg applications, we use the suggested default configuration values, resulting in average CPU utilization of about 25 percent for CloudSuite, 34 percent for WikiBench, and 62 percent for Pinot, without interference. Recent studies, including those at Azure [24] and Alibaba [25], reported average CPU utilizations of about 20 percent for fg VMs.

3.2 Background (bg) Workloads

In our experiments, we emulate interference by employing several bg workloads to create contention for the fg application. 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.

3.3 Resource usage Monitoring

We study resource contention for four resources: (i) network (NET), CPU, Last-Level-Cache (LLC), and disk. We now explain how we monitor fg and bg resource usage, from within the VMs, for our model training.

- NET: We use the dstat Linux tool to monitor the used network bandwidth for bg and fg VMs. We then normalize their sum by the peak bandwidth.
- CPU: We consider fair-sharing of the possibly overcommitted PM cores among VMs. If a PM has n cores available and all VMs together require m cores, then the CPU usage of each VM is normalized by $max\{m, n\}$.
- 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 experiment bandwidth. Finally, we normalize this difference by peak bandwidth to estimate LLC usage.
- DISK: Disk usage typically depends on the access pattern (sequential versus random). We thus use the same approach as for LLC, but with sysbench instead of RAMspeed, for estimating DISK usage.

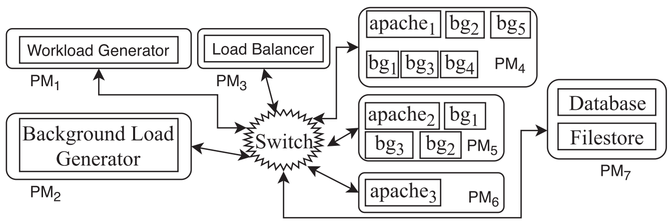


Fig. 4. Illustration of our OpenStack cloud setup.

4 DIAL FOR WEB APPLICATIONS

We first explain our DIAL implementation, and then present evaluation results for CloudSuite and WikiBench.

4.1 DIAL Implementation

For DIAL web application deployment, we implement the 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 (from within the VM) 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 are only leveraged periodically 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 percent. ough interval.

4.2 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. Fig. 4 depicts our experimental setup. We use an OpenStack Icehouse-based private cloud with several dedicated Dell C6100 physical machines, referred to as *PMs*. Each PM has 2 sockets with 6 cores each, and 48GB memory. The host OS is Ubuntu 14.04. 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 50 MB/sec (using sysbench), respectively.

AWS-based Public Cloud. We rent 10 c4.large instances (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.

4.3 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 4.3.6 we compare DIAL against existing interference-aware techniques that are popularly employed.

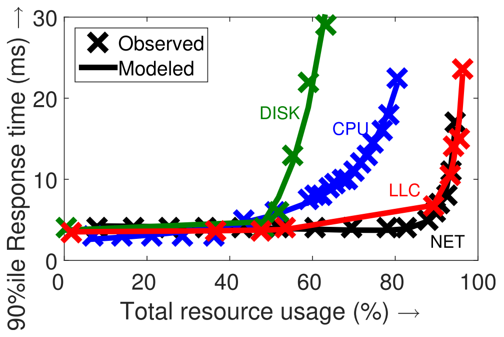


Fig. 5. Observed and modeled response times for CloudSuite under resource contention via microbenchmarks. Average modeling error: 6.1 percent.

4.3.1 Evaluating Detection, Classification, and Estimation

Detection. The crosses in Fig. 5 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 Fig. 5 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 percent confidence intervals around the mean (see Section 2.3.1) 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 percent.

Prior work has employed similar techniques to detect and analyze interference using hardware performance counters such as CPI [27], MIPS [5], 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].

Classification. We monitor the user space CPU utilization, usr, the system 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

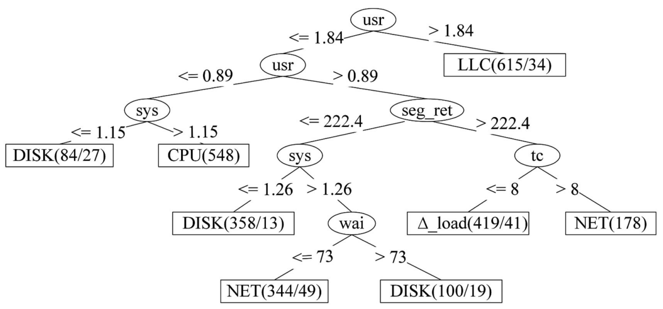


Fig. 6. Our trained decision tree. Leaves represent the contention classification with numbers in the leaves representing the total classification instances (left) and the number of misclassified ones, if any (right).

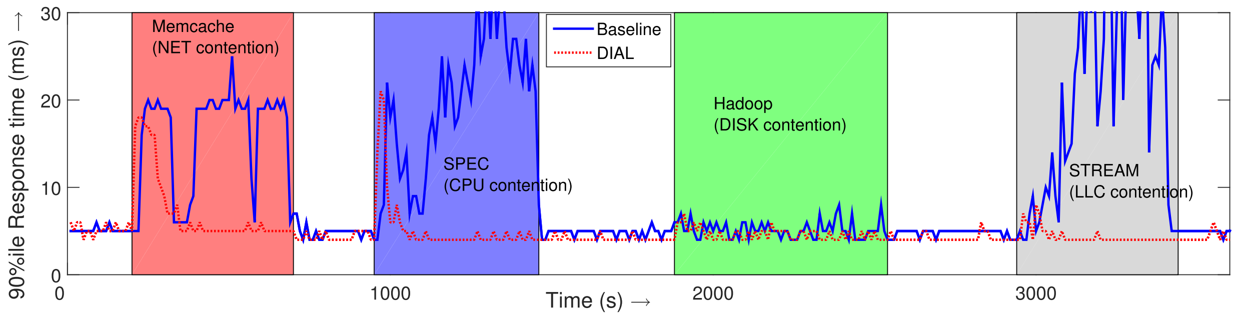


Fig. 7. Performance comparison between DIAL and baseline for test background workloads. The red, blue, green, and gray regions represent NET, CPU, DISK, and LLC contention, respectively. DIAL reduces 90%ile response time during these contentions by 39.1, 56.3, 16.2, and 59.2 percent.

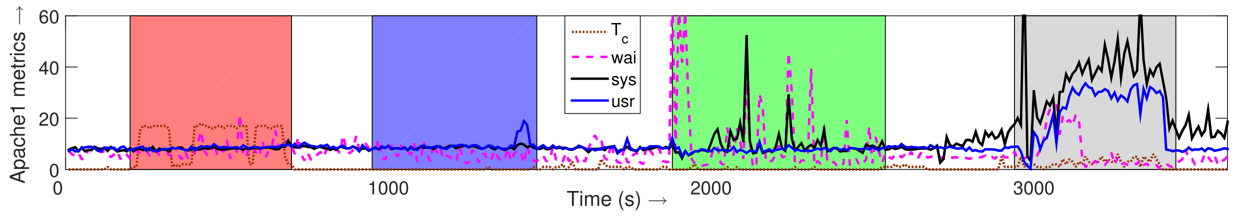


Fig. 8. T_c, wai, sys, and usr metrics for apache1 application layer VM for the experiments in Fig. 7. apache1 VM experiences NET, DISK, and LLC contention, and shows an increase in relevant metrics under those contentions.

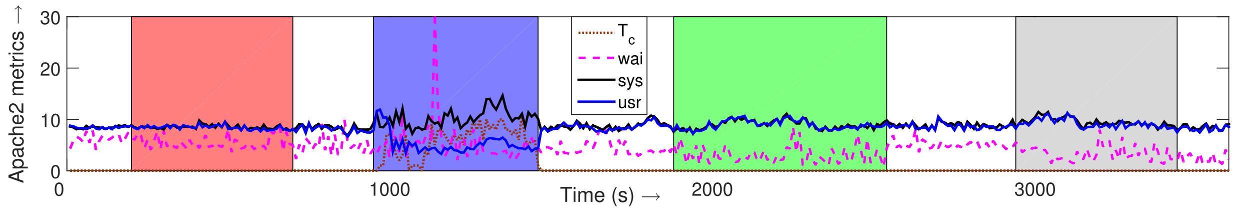


Fig. 9. T_c, wai, sys, and usr metrics for apache2 application layer VM for the experiments in Fig. 7. apache2 VM experiences only CPU contention, and consequently shows an increase in relevant metrics under CPU contention.

application VM, T_c (via HAProxy logs). Note that all metrics are monitored from within the VMs, to comply with the user-centric design of DIAL. We normalize usr and sys using predicted values to distinguish from workload variations, as discussed in Section 2.3.2. 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 Fig. 6. 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, as illustrated by our classification results for the Pinot OLAP application in Section 5.4.1. For example, we expect that LLC interference will lead to an increase in CPU usage.

Our 10-fold cross-validation error is *7.8 percent*. 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 Fig. 6).

We also evaluate our classifier using test workloads that were *not* seen during classifier training. 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. Our decision tree successfully classifies 44 of the 50 test instances; the misclassifications are observed for change in workload and DISK contention. 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 due to its complex and dynamic resource needs.

Estimation. The solid lines in Fig. 5 show our modeling results for CloudSuite interference estimation (see Section 2.3.3) under different resource contentions via training. Our average modeling error across all contentions is 6.1 percent. 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 percent. 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.

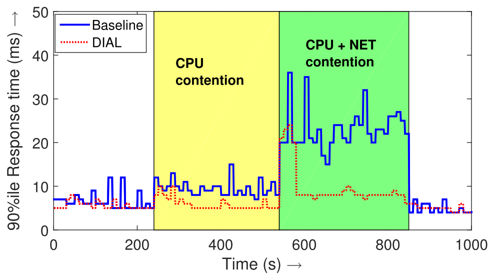


Fig. 10. DIAL reduces the response time of all request classes by 37 and 56 percent under CPU and combined CPU + NET contention, respectively.

Experimentally, we find that shorter interval lengths, such as 1s or 5s, lead to inaccurate classification and estimation due to system noise and load fluctuations. On the other hand, intervals larger than 10s do not significantly improve accuracy. For these reasons, we choose an interval length of 10s; prior work has also reported such reaction times to avoid rash decisions [2], [26], [28].

4.3.2 Evaluating DIAL under Real Workloads

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

We see that DIAL significantly reduces tail response times, when compared to the baseline, under all contentions; the reduction ranges from 16 percent under DISK contention to 59 percent under LLC contention. The relatively low improvement under DISK contention is because Hadoop intermittently utilizes 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.

Figs. 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 Fig. 9 is intentionally smaller to focus on the rise in the sys metric. For apache1, 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 Fig. 6, even though the classifier was trained on

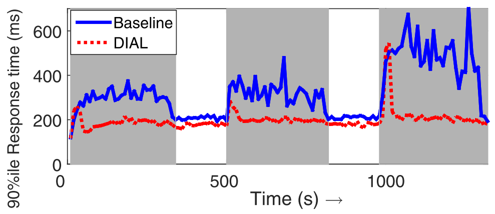


Fig. 11. Performance under LLC contention for fg WikiBench. DIAL reduces response times by ~23.6 percent during contention (gray regions).

microbenchmarks and not on these test workloads. This highlights the efficacy of our classifier.

For Memcache, the server is hosted on a bg VM and is driven by mutilate clients (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; our classifier suggests DISK contention.

4.3.3 Evaluating DIAL under Multiple Contentions

DIAL is capable of dynamically responding to multiple compromised VMs. The optimization in Section 2.4.1 provides estimates for LB weights, via Eq. (3), for all VMs. This is different from the case of multiple resource contentions on the *same* VM, which is beyond the scope of this paper.

Fig. 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 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).

4.3.4 Evaluating DIAL for the WikiBench fg Application

Fig. 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 percent 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 percent, respectively, under interference; this is in agreement with our decision tree classifier.

4.3.5 Evaluating DIAL in the AWS Setup

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

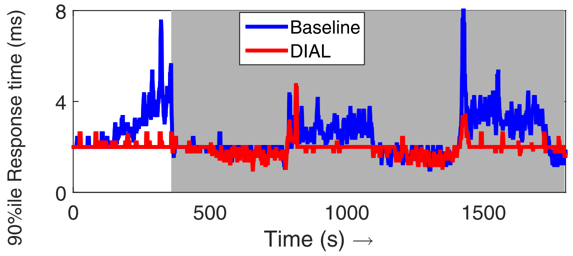


Fig. 12. Performance under LLC contention for AWS setup. DIAL reduces response times by around 22.3 percent.

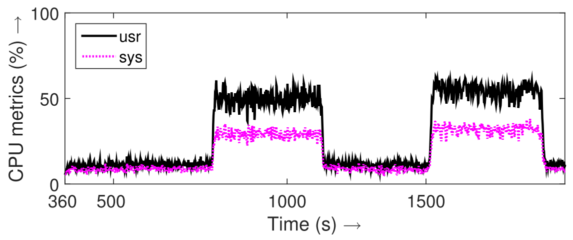


Fig. 13. usr and sys metrics for the gray region in Fig. 12. These metrics clearly increase during contention.

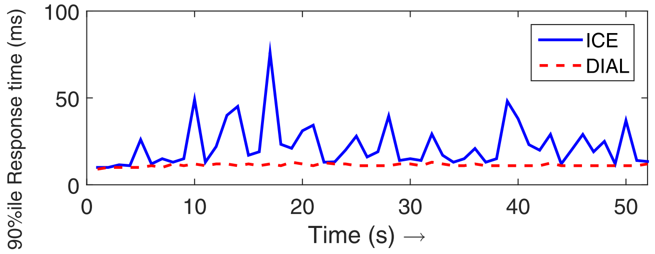
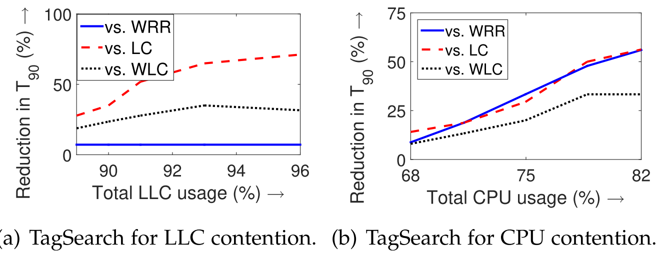


Fig. 14. Comparison of DIAL with ICE under CPU contention. DIAL reduces T_{90} by 25-48 percent for all request classes.



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

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

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 percent when compared to the baseline. This shows that *even one* compromised VM (out of 10) can considerably impact the overall response time.

Fig. 13 shows the usr and sys metrics for the shaded region in Fig. 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.

4.3.6 Comparison with Existing user-Centric Techniques

*Utilization-based Strategies.*Fig. 14 shows our experimental results for high CPU contention under DIAL and under ICE [1]. Similar to DIAL, ICE is an interference-aware load balancer 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 percent CPU utilization works well for moderate CPU interference under ICE, but does not work well for high CPU interference, as shown in Fig. 14. Under DIAL, response time is significantly lower, and the observed CPU usage at the compromised VM is about 8-10 percent; results are similar for other contentions.

Queue-Length based Strategies. Queue-length or load-based strategies send traffic to the VM that has the lowest load. We consider the Least Connections (LC) strategy that directs the next incoming request to the VM that has the least number of active connections. Under interference, the outstanding requests for the compromised VM will be higher, resulting in fewer additional requests being sent to it under LC. Fig. 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 percent, 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 weights under DIAL take both these into consideration, as opposed to LC that only addresses the former.

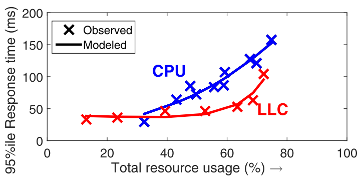
Weighted Load Balancing Strategies. We now compare DIAL with other weighted load balancing heuristics, such as Weighted Round Robin (WRR) and Weighted Least Connections (WLC). For WRR and WLC, we use proportional interference-aware weights, as discussed in Section 2.4. Fig. 15 shows the reduction in 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.

5 DIAL for Pinot

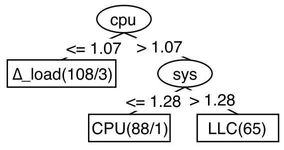
We now present our implementation of DIAL and its evaluation for a widely used OLAP solution, Pinot [11].

5.1 DIAL Implementation

The Load Balancing Tier (LBT) for Pinot consists of the Broker nodes (see Section 3.1.2), that distribute queries to the back-end workers nodes. The Brokers rely on routing tables, stored in Broker memory, to determine which worker nodes host the data segments that are needed to serve the incoming query. Each routing table is a map from every segment

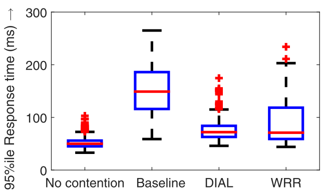


(a) Interference estimation.

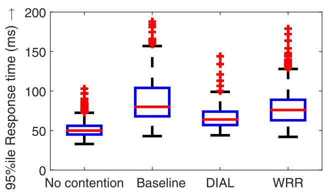


(b) Decision tree classifier.

Fig. 16. Interference modeling and classification for Pinot.



(a) CPU contention



(b) LLC contention

Fig. 17. Comparison of DIAL with other heuristics for Pinot.

to one worker node; since each segment is stored on several replicas, numerous unique routing tables can be generated. By randomly selecting a routing table for each query, the Brokers balance load among the worker nodes.

We implement DIAL on the Broker using ~300 lines of Java code. Once interference is detected, DIAL updates the routing tables to remap segments that were initially assigned to the worker(s) under interference to other replicas, based on the theoretically-derived optimal fractions, q^* (see Section 2.4.2). Likewise, once interference ceases, the routing tables are updated to the default balanced weights.

5.2 Cloud Environment

We use several blade servers (PMs) from a HP Proliant C7000 Chassis. Each PM has 2 sockets with 4-core CPUs each, and 32 GB memory. The host OS is Ubuntu 16.04. The servers are connected through 1Gb/s network links. We use KVM (on top of Ubuntu 16.04) to deploy VMs on these PMs. We deploy 6 Pinot worker nodes on 1 vCPU, 16GB memory VMs; each VM is on a separate PM. We deploy the Pinot Controller and 2 Pinot Brokers using VMs with 8 vCPUs and 16GB memory, on different PMs.

We experiment with CPU and LLC contention for Pinot. For CPU contention, we use a 1 vCPU bg VM that is statically pinned to the same core as the fg VM (via hyperthreading). For LLC contention, we use a 3 vCPU bg VM that is pinned to the remaining 3 cores of the 4-core socket that hosts the fg VM; in this way, we do not share the same core as the fg VM to avoid CPU contention.

5.3 Workload and Benchmark

We implement a query generator for Pinot based on our tables. For each table, we create several realistic queries. An example query for the ProfileView table is "SELECT COUNT (*) FROM ProfileView WHERE ViewedProfileID=*ID*", where *ID* is a (randomized) query parameter. Our table and query design is based on LinkedIn's Pinot deployment [11]. Our benchmark is implemented in ~2000 lines of code, and is open-sourced [30].

5.4 Evaluation

We use a warm-up time of 120s for all our experiments in this section. We focus on 95%ile response times for Pinot.

5.4.1 Evaluating Detection, Classification, and Estimation

Detection. The crosses in Fig. 16a show the impact of CPU and LLC contention on Pinot response times. The detection rule of $T_{95} > 61ms$ is obtained based on the discussion in Section 2.3.1. We run several experiments using the bg test workloads and find that our detection rule results in a low false positive rate of 3.3 percent.

Classification. We monitor total CPU usage, cpu, and the system space CPU utilization, sys; we normalize these values using predicted values to distinguish from workload variations, as discussed in Section 2.3.2. Our decision tree for Pinot is shown in Fig. 16b. The classification rules in Fig. 16b closely resemble those for the web application in Fig. 6. Our 10-fold cross-validation error is 2.3 percent.

We also evaluate our classifier using test workloads that were *not* seen during training. We run 10 experiments each for SPEC (CPU contention) and STREAM (LLC contention), and 10 experiments under varying Pinot workload. Our decision tree classifier is able to accurately classify all instances, except one CPU interference instance which is misclassified as LLC interference. Our classification accuracy based on these 30 experiments is 96.7 percent.

Estimation. The solid lines in Fig. 16a show our modeling results for Pinot interference estimation (as discussed in Section 2.3.3) under different resource contentions via training. Our average modeling error is 11.5 percent.

5.4.2 Evaluating DIAL for Pinot under Real bg Workloads

Fig. 17 shows our experimental results for Pinot under our KVM setup for CPU and LLC contentions created using test workloads SPEC and STREAM, respectively. Here, the request rate for Pinot is set to 200 queries/sec, which results in a CPU load of about 60 percent. We consider 6 worker nodes with a replication factor of 3. The contention is created in bg VMs on one of the six PMs hosting the Pinot worker VMs. We show the tail response time values for no contention, baseline (with contention), DIAL, using theoretically optimal interference-aware weights from Section 2.4.2, and Weighted Round Robin (WRR), which uses proportional interference-aware weights, as discussed in Section 2.4. The response time is the query completion time monitored at the Broker, and depends on the performance of all workers.

We see that DIAL significantly improves tail response times when compared to baseline; the average reduction in 95%ile response times for CPU and LLC contention is 40.5 and 25.8 percent, respectively. Compared to WRR, DIAL provides an average reduction in 95%ile response times for CPU and LLC contention of 16.1 and 16.5 percent, respectively.

5.4.3 Evaluating DIAL for Pinot under Dynamic Conditions

Fig. 18 shows the 95%ile response time (tail latency) for Pinot under DIAL and baseline for our dynamic workload experiment. Here, we start with a load of 200 queries/s (or, qps) and no interference; as before, we have 6 Pinot workers and a replication factor of 3. Then, in the next

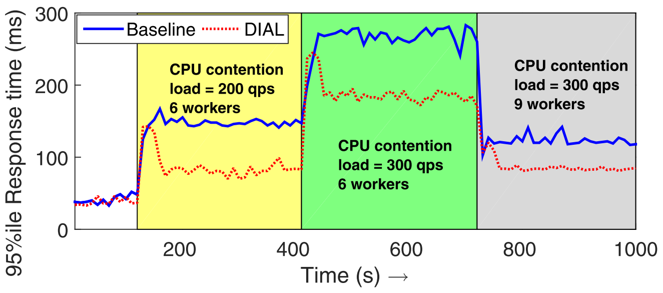


Fig. 18. Performance of DIAL and baseline using Pinot under CPU contention and under dynamic workload and cloud conditions.

phase (yellow shaded region), one of the fg worker VMs experiences CPU interference due to a colocated VM running SPEC. DIAL responds, after monitoring and detection, by setting the theoretically optimal load balancing weights for the 3 replicas of segments hosted on the under-interference worker (see Section 5.1). For this experiment, the theoretically optimal weights in this phase are $\{0.16, 0.42, 0.42\}$, obtained via the analysis discussed in Section 2.4.2. By setting these weights, the tail latency lowers from about 147ms under the baseline to 88ms (39.9 percent improvement).

In the next phase (green shaded region), Pinot experiences an increase in load to 300 qps, severely impacting tail latency. DIAL detects this load change via request rate monitoring (see Section 2.5), and updates the load balancing weights to $\{0.2, 0.4, 0.4\}$, thus lowering tail latency from 266ms under the baseline to 189ms (28.2 percent improvement). Our theoretically derived weights from Section 2.4.2 already take request rate into account (via the a parameter), and thus the updated weights can be easily obtained.

To handle the increased load, Pinot eventually scales-out by adding 3 new workers and redistributing data segments across all workers. We assume that the scale-out and data segment mapping is handled by an external autoscaling entity (e.g., MLscale [30] or other similar works [31], [32]). With the additional workers, the tail latency of Pinot decreases, as seen in the last phase (gray shaded region). DIAL again updates the weights for this new configuration, by updating the n parameter (that represents the number of workers), resulting in a further lowering of tail latency from about 123ms under the baseline to 87ms (28.8 percent improvement).

We repeated the experiments for a total of 5 runs. The results were qualitatively similar to Fig. 18, with the average improvement in 95%ile response time across all runs afforded by DIAL in the three shaded phases being about 33.1, 29.8, and 30.7 percent. We also repeated the experiment with LLC contention, and obtained qualitatively similar results with an average improvement of up to 20 percent.

6 Prior Work in the Context of DIAL

Interference Detection. Recent work has emphasized the need for user-centric interference detection [1], [2], [33], [34]. 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 [35] makes use of collaborative filtering to detect interference in web services by monitoring response times. While we also monitor response time, 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 different jobs. While CPI^2 can be used in virtual environments, public cloud VMs (e.g., AWS) do not always expose performance counters.

There have also been prior works on hypervisor-centric interference detection (e.g., ILA [3]). While effective, such techniques require hypervisor access for monitoring host-level metrics, making them infeasible 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 4.3.6), that this strategy is not adaptive to different levels of interference. Mukherjee et al. [34] propose a tenant-centric interference estimation technique that employs a software probe periodically on all tenant VMs, and compares the performance of the probe at runtime versus that in isolation to quantify interference. The authors later extended this work to PRIMA [37], which is an interference-aware load balancing and auto-scaling technique that leverages the above-described probing technique to make load balancing decisions. However, PRIMA only focuses on mean response time (as opposed to the more practical tail response time metric) and limits itself to network interference. Bubble-Up [37], Tarcil [7], Quasar [8], and ESP [38] 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 (VM placement is not in the user's control); instead, DIAL globally adjusts the fg LB policy to reroute some of the requests directed at affected VMs.

7 Conclusion

We presented DIAL, a user-centric dynamic Interference-Aware Load Balancing framework that can be employed directly by cloud users without requiring any assistance from the hypervisor or cloud provider 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 for web and OLAP applications on several cloud platforms, under interference from realistic benchmarks, demonstrate the benefits of DIAL.

Acknowledgments

We thank the anonymous reviewers for their helpful suggestions. This work was supported by NSF grants CNS-1750109, CNS-1717588, and CNS-1617046.

References

[1] A. Maji, et al., "ICE: An integrated configuration engine for interference mitigation in cloud services," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2015, pp. 91–100.

[2] A. Maji, et al., "Mitigating interference in cloud services by middleware reconfiguration," in *Proc. 15th Int. Middleware Conf.*, 2014, pp. 277–288.

[3] X. Bu, et al., "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 227–238.

[4] R. Nathuji, et al., "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 237–250.

[5] C. Delimitrou and C. Kozyrakis, "QoS-aware scheduling in heterogeneous datacenters with paragon," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, 2013, Art. no. 12.

[6] S. Nathan, et al., "Towards a comprehensive performance model of virtual machine live migration," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 288–301.

[7] C. Delimitrou, et al., "Tarcil: High quality and low latency scheduling in large, shared clusters," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 97–110.

[8] C. Delimitrou, et al., "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. 19th Int. Conf. Architectural Support Programm. Languages Operating Syst.*, 2014, pp. 127–144.

[9] A. Gandhi, et al., "The unobservability problem in clouds," in *Proc. Int. Conf. Cloud Autonomic Comput.*, 2015, pp. 13–20.

[10] D. Novakovic, et al., "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2013, pp. 219–230.

[11] J.-F. Im, et al., "Pinot: Realtime OLAP for 530 million users," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 583–594.

[12] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge, U.K.: Cambridge Univ. Press.

[13] K. Leonard, *Queueing Systems*, vol. 2, Hoboken, NJ, USA: Wiley, 1976.

[14] "HAProxy: The reliable, high performance TCP/HTTP load balancer," 2019. [Online]. Available: http://www.haproxy.org

[15] A. Javadi and A. Gandhi, "DIAL: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2017, pp. 135–144.

[16] C. Wang, et al., "Effective capacity modulation as an explicit control knob for public cloud profitability," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2016, pp. 95–104.

[17] Y. Xu, et al., "Small is better: Avoiding latency traps in virtualized data centers," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 7–16.

[18] Q. Zhang, et al., "A regression-based analytic model for capacity planning of multi-tier applications," *Cluster Comput.*, vol. 11, no. 3, pp. 197–211, 2008.

[19] T. Horvath, et al., "Multi-mode energy management for multi-tier server clusters," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, 2008, pp. 270–279.

[20] M. I. Reiman and B. Simon, "An interpolation approximation for queueing systems with poisson input," *Operations Res.*, 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," *Perform. Eval.*, vol. 68, no. 3, pp. 290–306, 2011.

[22] M. Ferdman, et al., "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. 17th Int. Conf. Architectural Support Programm. Languages Operating Syst.*, 2012, pp. 37–48.

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

[24] E. Cortez, et al., "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proc. 26th Symp. Operating Syst. Principles*, 2017, pp. 153–167.

[25] "Cluster data collected from production clusters in Alibaba for cluster management research," 2018, [Online]. Available: https://github.com/alibaba/clusterdata

[26] X. Zhang, et al., "CPI^2: CPU performance isolation for shared compute clusters," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 379–391.

[27] M. Hall, et al., "The WEKA data mining software: An update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[28] David Lo, et al., "Heracles: Improving resource efficiency at scale," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, 2015, pp. 450–462.

[29] "PACELab/pinot," 2019. [Online]. Available: https://github.com/PACELab/pinot

[30] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "Using machine learning for black-box autoscaling," in *Proc. 7th Int. Green Sustainable Comput. Conf.*, 2016, pp. 1–8.

[31] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. Kozuch, "SOFTScale: Scaling opportunistically for transient scaling," in *Proc. 13th Int. Middleware Conf.*, 2012, pp. 142–163.

[32] A. Gandhi, P. Dube, A. Kochut, L. Zhang, and S. Thota, "Autoscaling for hadoop clusters," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 109–118.

[33] G. Casale, et al., "A feasibility study of host-level contention detection by guest virtual machines," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*,2013, pp. 152–157.

[34] J. Mukherjee, et al., "Subscriber-driven interference detection for cloud-based web services," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 1, pp. 48–62, Mar. 2017.

[35] Y. Amannejad, et al., "Detecting performance interference in cloud-based web services," in *Proc. IFIP/IEEE Int. Symp. Integrated Netw. Manage.*, 2015, pp. 423–431.

[36] J. Mukherjee and D. Krishnamurthy, "Subscriber-driven cloud interference mitigation for network services," in *Proc. IEEE/ACM Int. Symp. Quality Service*, 2018, pp. 1–6.

[37] J. Mars, et al., "Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 248–259.

[38] N. Mishra, et al., "ESP: A machine learning approach to predicting application interference," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2017, pp. 125–134.

Seyyed Ahmad Javadi received the PhD degree in computer science from Stony Brook University, in 2019. He is a researcher with the Department of Computer Science and Technology, University of Cambridge. He is currenly interested in to do research on performance challenges and privacy concerns in cloud computing environments.

Anshul Gandhi received the PhD degree in computer science from Carnegie Mellon University, in 2013. He is an assistant professor of Computer Science at Stony Brook University. His current research interests are in Performance Modeling and Cloud Computing.