ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services

Amiya K. Maji, Subrata Mitra, Saurabh Bagchi
Purdue University, West Lafayette, IN
Email: {amaji, mitra4, sbagchi}@purdue.edu

***Abstract*—Performance degradation due to imperfect isolation of hardware resources such as cache, network, and I/O has been a frequent occurrence in public cloud platforms. A web server that is suffering from performance interference degrades interactive user experience and results in lost revenues. Existing work on interference mitigation tries to address this problem by intrusive changes to the hypervisor, e.g., using intelligent schedulers or live migration, many of which are available only to infrastructure providers and not end consumers. In this paper, we present a framework for administering web server clusters where effects of interference can be reduced by intelligent reconfiguration. Our controller, ICE, improves web server performance during interference by performing two-fold autonomous reconfigurations. First, it reconfigures the load balancer at the ingress point of the server cluster and thus reduces load on the impacted server. ICE then reconfigures the middleware at the impacted server to reduce its load even further. We implement and evaluate ICE on CloudSuite, a popular web application benchmark, and with two popular load balancers - HAProxy and LVS. Our experiments in a private cloud testbed show that ICE can improve median response time of web servers by upto 94% compared to a statically configured server cluster. ICE also outperforms an adaptive load balancer (using least connection scheduling) by upto 39%.**

***Keywords*-interference; cloud performance; load-balancer; dynamic configuration;**

I. Introduction

Performance issues in web service applications are notoriously hard to detect and debug. In many cases, these performance issues arise due to incorrect configurations or incorrect programs. Web servers running in virtualized environments also suffer from issues that are specific to cloud, such as, interference [1], [2] or incorrect resource provisioning [3]. Among these, performance interference and its more visible counterpart performance variability cause significant concerns among IT administrators [4]. Interference also poses a significant threat to the usability of Internet-enabled devices that rely on hard latency bounds on server response (imagine the suspense if Siri took minutes to answer your questions!). Existing research shows that interference is a frequent occurrence in large scale data centers [1], [5]. Therefore, web services hosted in the cloud must be aware of such issues and adapt when needed.

Interference happens because of sharing of low level hardware resources such as cache, memory bandwidth, network etc. Partitioning these resources is practically infeasible without incurring high degrees of overhead (in terms of compute, memory, or even reduced utilization). Existing solutions primarily try to solve the problem from the point of view of a cloud operator. The core techniques used by these solutions include a combination of one or more of the following: a) Scheduling, b) Live migration, c) Resource containment. Research on novel scheduling policies look at the problem at two abstraction levels. Cluster schedulers (consolidation managers) try to optimally place VMs on physical machines such that there is minimal resource contention among VMs on the same physical machine [6]. Novel hypervisor schedulers [7] try to schedule VM threads so that only non-contending threads run in parallel. Live migration involves moving a VM from a busy physical machine to a free machine when interference is detected [8]. Resource containment is generally applicable to containers such as LXC, where the CPU cycles allocated to batch jobs is reduced during interference [1], [9]. Note that all these approaches require access to the hypervisor (or kernel in case of LXC), which is beyond the scope of a cloud consumer. Prior work indicate that, despite having (arguably the best of) schedulers, the public cloud service, Amazon Web Service, shows significant amount of interference [2].

We therefore need to find practical solutions that do not require modification of the hypervisor. One existing solution that looks at this problem from a cloud consumer's point of view is IC^2 [2]. IC^2 mitigates interference by reconfiguring web server parameters in the presence of interference [2]. The parameters considered are `MaxClients` (MXC) and `KeepaliveTimeout` (KAT) in Apache and `pm.max_children` (PHP) in Php-fpm engine. The authors showed that they could recapture lost response time by upto 30% in Amazon's EC2. However, the drawback of this approach is that web server reconfiguration usually has high overhead (the web server need to spawn or kill some of the worker threads). Moreover, IC^2 alone cannot improve response time much lower without drastically degrading throughput. We note that the key goal in IC^2 is to reduce the load on the web server (WS) during periods of interference. We also observe that implementing admission control at the gateway (load balancer) is a direct way of reducing load to the affected web server. An out-of-box load balancer is agnostic of interference and therefore treat the WS equally. We aspire to make this context aware, and evaluate how much performance gain can be achieved.

Solution Approach. The proposed solution, called ICE, uses hardware counter values to detect the presence of interference and therefore provides fast detection. Access to hardware counters does not always require hypervisor (root) access as these may be virtualized [10]. Our evaluations showed that ICE could detect and reconfigure the WS cluster with a median latency of 3s. This is better than existing techniques which would incur a detection latency of 15-20s [2]. When interference is detected, ICE performs two-level reconfigurations of the web-server cluster. The first level, geared towards agility, is to reconfigure the load balancer to send fewer requests to the affected WS VM. We found that this provides the maximum benefit in terms of reduced response time. The second level of reconfiguration, which configures Apache and Php-fpm as described in IC^2, is activated only if the interference lasts for a long time. The second level reconfiguration ensures that the WS does not incur overhead of idle threads (note that since the LB has been reconfigured this WS VM will receive fewer client requests). Our key contributions in this paper are as follows:

1. We present the design and implementation of a two-level reconfiguration engine for web server clusters to deal with interference in cloud. Our solution, called ICE, includes algorithms for: a) detecting interference quickly (primarily cache and memory bandwidth contention), and b) predicting new weights for impacted server to reduce load on them.
2. We deploy our solution on a popular web application benchmark called CloudSuite. Our evaluation experiments show that on a combination of Apache+Php+HAProxy middleware, ICE can reduce median response time of web servers by upto 94%. We evaluate ICE for two different scheduling policies (weighted round-robin, and weighted least-connection) in HAProxy and find that it improves response time across both scheduling policies (upto 94% and 39% respectively). Median interference detection latency was 3s.
3. To evaluate the generalizability of our framework, we also ran some experiments with the Darwin media streaming server running with LVS load balancer. Our results show: i) reconfiguring server weight in LVS can be used to reduce the inter-frame delay, and ii) optimal `num_threads` parameter in Darwin is vastly different with and without interference indicating WS reconfiguration is also beneficial.

II. Overview of Load Balancers

Load balancers are used to scale websites to serve larger number of users by distributing load among multiple servers. Existing research on load balancing can be primarily categorized into two orthogonal problems: i) **Routing:** How to redirect or route traffic for a web server to its internal "real" servers *transparently* and *efficiently*? ii) **Scheduling:** How to decide which server would get the current request so that the servers have equal load?

There are many load balancing solutions that are available in the market. A majority of these operate at the network layer and tries to balance network load between servers. Among the open source software load balancers, HAProxy [11] and Linux Virtual Server (LVS) [12] are popular choices. Both these load balancers implement four major scheduling policies. These are:

Round Robin (RR): Servers are picked in a circular order.

Least Connection (LC): Server with the least number of active connections are chosen. This is better than round robin in general since it considers server state ($num\_connections$).

Weighted Round Robin (WRR): This policy helps when servers have unequal capacity. The scheduler picks a server based on its weight. For example if servers $S_1, .., S_n$ have weights $w_1, .., w_n$, then S_k is chosen if $w_k = max(w_1, .., w_n)$. Weight w_k is reduced every time a server is selected, and when all the servers reach weight 0 they are reset to their initial values.

Weighted Least Connection (WLC): Similar to LC, but server weight is also considered for scheduling decision. If W_i is the weight of server i and C_i is the current number of connections to it, then the scheduler picks server j for the next request such that, $C_j/W_j = min\{C_i/W_i\}$, i=$[1, .., n]$. In rest of the paper we use the terms RR and WRR (LC and WLC) synonymously since the concept of weight is often implicit. Apart from these there are also load balancing policies that are based on source address of a client. In this case, an appropriate hash function is used to map the source address to a server.

Session stickiness: Once a client session has been established, the load balancer remembers which server is processing requests originating from that client. This is called session persistence and it is required since most web servers maintain sessions and relevant application state locally.

III. Interference Degrades WS Performance

In this section, we present an empirical evaluation of the impact of interference on web server (WS) performance. We use the average response time of the WS as the metric of interest as this directly relates to customer satisfaction.

A. Experimental Setup

Testbed. Our experiments were conducted in a private cloud testbed consisting of three PowerEdge 320 servers (Intel Xeon E5-2440 processor, 6 cores–12 threads with hyperthreading, 15MB L3 cache and 16GB RAM). The servers are managed by the popular KVM hypervisor. Each of our experimental virtual machines (VM) were configured with 2 vcpus and 3GB RAM except the DB VM. The DB VM was provisioned with 4 vcpus and 4GB RAM to eliminate any DB bottleneck. All the machines were connected by a 1 Gbps switch.

Application Benchmark. For our experiments with web servers, we use the popular CloudSuite [13] web application benchmark. CloudSuite emulates a social event calendar application written in Php. It is an example of a dynamic web application, where most of the responses are generated by executing Php scripts and running database queries. The benchmark also includes a custom workload generator based on real-life distribution of user requests.

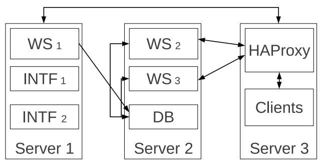
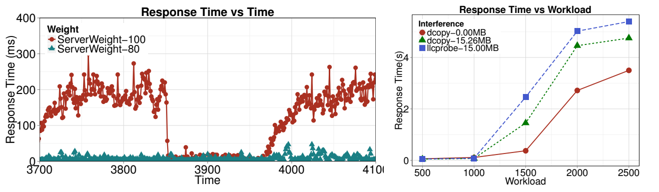


Figure 1. Layout of virtual machines in private cloud testbed.

VMs and Middleware. We used a multi-threaded Apache web server (*Apache-worker*) to host CloudSuite, while Php scripts were interpreted using the Php-Fastcgi Process Manager (*Php-fpm*). Note that one instance of Apache and Php-fpm runs in a single virtual machine (called WS VM). We replicated the WS VM three times to create a web server cluster with 3 virtual servers. These were distributed across 2 physical machines (refer Fig. 1). The third machine was used to emulate clients. Note that the physical machine with one WS VM was chosen to run interference to limit the impact of interference across the cluster. Unless otherwise specified, we refer to *this* WS VM as the monitored web server (or impacted web server, also denoted WS_X). All our measurements and analysis below pertain to this single WS VM instance (WS_X). We used the popular HAProxy load balancer to distribute requests equally among the WS VMs. The CPU and Memory utilizations of all physical servers (as well as the HAProxy and Client VM) were always well below their provisioned capacities.

Interference Benchmarks. We use the LLCProbe and Dcopy benchmarks described in [2] for generating interference on the monitored WS VM. LLCProbe creates an LLC Sized array in memory and then accesses each cache line very frequently. On the other hand, Dcopy from the linear algebra suite (BLAS) copies contents of one array to another. While LLCProbe is an example of a cache read benchmark, Dcopy is an example of cache read+write benchmark. We found that in general LLCProbe has greater cache access frequency than Dcopy and therefore emulates a stronger interference.

Configuration Parameters. We consider several parameters in Apache, Php and HAProxy for reconfiguration tasks in ICE. The HAProxy parameter that ICE manages is the `server weight` parameter which determines what fraction of all client requests go to a particular WS VM. In Apache, we automatically reconfigure the `MaxClients` (MXC) and `KeepaliveTimeout` (KAT) parameters for improving response times. While MXC indicate the maximum number of worker threads that an Apache server can spawn, KAT determines how long a client connection is persisted in idle state before terminating it. In Php, we consider the `pm.max_children` (PHP) parameter which indicates the maximum number of worker threads in the Php-fpm engine. Note that although our current experiments



(a) (b)

Figure 2. (a) Response time of a WS VM during interference with different LB weights. LB scheduling policy is round robin. (b) Increase in response time during interference with varying workload sizes. The plot shows interference has greater performance impact with larger workloads and the impact varies across interference benchmarks.

are geared towards Apache+Php+HAProxy setup, similar parameters exist for most threadpool-based web services middleware and load balancers. Therefore, our design principles can be easily adapted for other web servers. We show this with an example of the Darwin media streaming server with LVS load balancer in Section VI.

B. Interference Increases Response Time

In this experiment, we run CloudSuite with a given load size (`#concurrent_clients`) for several 1-hour runs. Note that all the clients direct their requests to the HAProxy load balancer which distributes the requests among WS VMs using WRR scheduling. During each run, LLCProbe benchmarks were started and stopped in the interference VMs periodically. We show the response times of WS VM over time during two such runs (one with high load, plot ServerWeight-100; and another with lower load, plot ServerWeight-80) in Fig. 2(a). It can be seen that with interference the response time (RT) of the impacted WS VM (WS_X) can increase sharply from 10x ms to 100x ms (plot ServerWeight-100 in Fig. 2(a)). Such sharp rise in RT is due to the fact that an impacted WS may saturate with fewer clients than in a no-interference scenario. This shows that a load-balancer which has no knowledge of which WS VM is suffering from interference, would continue to send the same number of requests to the impacted server(s), as to the other WSs (assuming same default weight). We therefore need to make the load-balancer aware of dynamic situations (interference) and make intelligent scheduling decision.

C. Interference vs. Load

In a separate experiment, we ran CloudSuite on a standalone WS VM (no HAProxy) with varying workload sizes and interference types. Unlike the previous experiment, each run here indicates a 15-minute run of CloudSuite with a specific combination of (workload x interference). The interference benchmark is executed for the entire duration of a run. The results from our experiments are displayed in Fig. 2(b). Note that each data point in the plot indicate the average of all observations during a 15-minute run. An interference of type Dcopy-0.0MB indicate no interference was run. It can be seen that the rise in response time due to interference, for a given interference, varies with workload size. For example, with very low workload (< 1000), the impact of interference is negligible and all the plots overlap with each other. However, as the workload size increases (> 1000), the slopes of the curves increase. In other words, rise in response time (RT) depends on the server load. Therefore, if the load of a WS can be reduced sufficiently, then the impact of interference can be reduced (or nearly eliminated). In a load-balanced WS setup, reducing load on an impacted WS VM implies forwarding fewer client requests to that server. It can be achieved by reducing the scheduling weight of the corresponding server in the load-balancer configuration. This can be validated from plot ServerWeight-80 in Fig. 2(a) where the scheduling weight of the monitored WS (WS_X) is set to 80. This shows that the average response time of WS_X reduces quite significantly while using a lower weight during interferences. In the following section, we describe how we automate the reconfiguration actions in ICE.

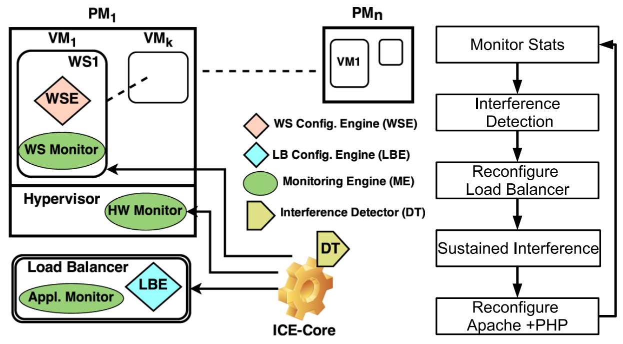
IV. Design and Implementation

Overview

ICE primarily consists of four components: i) monitoring engine (ME), ii) interference detector (DT), iii) load balancer configuration engine (LBE), and iv) web server configuration engine (WSE) as shown in Fig. 3(a). These components are coordinated by ICE-Core. Fig. 3(b) shows the high level functioning of this configuration engine. First the monitoring engine collects performance metrics from the web server cluster. These are then fed as inputs to the subsequent modules. In the second stage, the interference detector analyzes observed metrics to find which WS VM(s) are suffering from interference. Once the affected VM(s) are identified, the load-balancer configuration engine then reduces the load on these VMs by diverting some traffic to other VMs in the cluster. We chose the LBE as the first level configuration engine since reconfiguring load balancer has much less overhead than reconfiguring web servers (the first one involves updating global variable(s), whereas, the second one involves creation or destruction of processes). If interference lasts for a long time, the web server configuration engine reconfigures the web server parameters (MXC, KAT, and PHP) to improve response time (RT) further. Finally, when interference goes away the load balancer and web server configurations are reset to their default values. Fig. 3(a) shows how these components are distributed across the web server cluster and their dependencies. Note that the exact deployment of a given component is flexible and choices are driven by the need to reduce network and computation overhead. As an example, our ICE core is deployed in the hypervisor of the monitored WS VM. We present the details of each module in the following sections.

Monitoring

ICE monitors the performance metrics of WS VMs at three levels of the system. The sensor placements are shown in Fig. 3(a). The hardware counter sensor collects values for cycles per instruction (CPI) and cache miss rate (CMR)



(a) Components of ICE and their deployment. (b) ICE workflow.

Figure 3. High level overview of ICE.

for all monitored VMs, whereas, CPU utilization (CPU) of each VM is monitored inside the VM. The application level sensor at HAProxy gives us the average response time (RT), and requests/second (RPS) metrics for each VM. While the hardware counter values are primarily used for interference detection, system and application metrics are used for reconfiguration actions. We chose a monitoring interval of 1 second for all the metrics. During training, we found that system and application metrics give better accuracy with an interval of 5 sec (refer paragraph on Estimating $\xi()$). Therefore, we also maintain a periodic average of 5 sec for those metrics.

Interference Detection

We found that the CPI and CMR values of affected WS VMs increase significantly during phases of interference. This is shown in Fig. 4 where the red vertical lines (start of interference) are followed by a sudden rise in CPI and CMR. Both these metrics are leading indicators of interference (as opposed to RT which is a lagging indicator) and therefore allow us to reconfigure quickly and prevent the web sever from going into a death spiral (i.e., one where the performance degrades precipitously and does not recover for many subsequent requests). Prior work has also shown that distribution of CPI values can be used to detect interference in web server clusters [1]. In our experiments, we found that CPI values had larger variance than CMR, hence, the latter provided better accuracy for interference detection. Interference detection in ICE is performed by a Decision Tree (DT) classifier which uses CPI and CMR as the attributes. Our choice of classifier is primarily due to the simplicity of the classification rules generated by DT. The classifier is trained using sample runs of the CloudSuite benchmark. We postpone the description of training runs until the next section since the same data is also used to train the LB reconfiguration engine. It was found that although both CPI and CMR were used as attributes for training the DT, the final classifier included thresholds on CMR only. This happens because CMR alone is able to classify most samples correctly. The classifier showed a detection accuracy (TP+TN) of 99.12% on the training data using 10-fold cross-validation. In contrast, a DT constructed with CPI only,

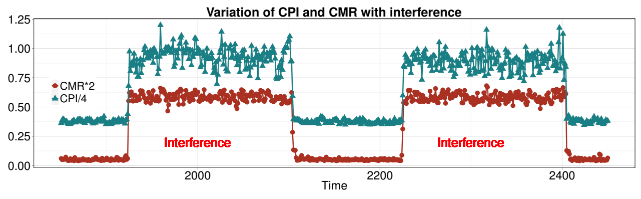


Figure 4. Variation in CPI and CMR with interference. Metrics are measured with a periodic interval of 1 sec. Actual values are scaled with the factors shown in labels for clarity.

showed detection accuracy of 98.15%. We used the first classifier (with CMR) for our evaluation experiments.

Load-balancer Reconfiguration

In Section III, we found that interference creates greater performance degradation during high server load since it can easily saturate the CPU [2]. Therefore, the goal of ICE is to reduce traffic to the affected WS so that it does not cross a CPU utilization setpoint (u_{thres}). We achieve this by updating the weight of the corresponding WS VM in the load balancer configuration. Note that, $\#requests$ forwarded to a WS is a function of its scheduling weight. Formally,

$$r = f(w, T),$$

where r is the requests per second (RPS) received by a WS, w is the scheduling weight, and T is total requests received at front-end (HAProxy).

Similarly, it is traditionally known that the CPU utilization (u) of a WS is a function of its load, i.e., $u = g(r)$. [2] also showed that u depends on interference. The degree of interference on a WS can be approximated by its CPI measurements (c). Therefore, $u = h(c)$. We approximate the dependence of u on r and c with the empirical function

$$u = \xi(r, c)$$

During our experiments we also found that CPU utilization u_t at time t is often dependent on the utilization u_{t-1} at time $t-1$. This happens because execution of a task may often last multiple intervals. For example, a request that started executing just before taking measurement for interval t may be served in interval $t+1$. Therefore, the measurements in intervals t and $t+1$ are not independent. This dependence is captured by taking into account the CPU utilization at the previous interval (we denote this as OldCPU or o) in our empirical function. The final function for estimating u is represented as

$$u = \xi(o, r, c)$$

The LBE works as shown in Algorithm 1:

Assume that CPI, RPS and CPU values at time t are c_t, r_t, and u_t. When the DT detects interference it sends a reconfiguration trigger to LBE. The LBE then computes the predicted CPU utilization ($\hat{u}_t$), given the current metrics c_t, r_t, and u_{t-1}. Notice that we use the estimated CPU utilization $\hat{u}_t$ to compare with setpoint u_{thres} since rise in u often lags behind rise in c. If $\hat{u}_t$ is found to exceed the setpoint, we predict a new RPS value $\hat{r}_t$ s.t. the CPU utilization falls below setpoint. To predict a new load-balancer weight we first compute the percentage reduction in RPS (δ) that is required to achieve this. This is then used to reduce weight w proportionately. Note that during periods of heavy load, the estimated δ may be very high, practically marking the affected server offline (very low w_{new}). To avoid this, we limit the change in w within a maximum bound (40% of its default weight). In the following sections, we present how to train the estimator function $\xi()$.

Algorithm 1 LB reconfiguration function for ICE

```
 1: procedure UPDATE_LB_WEIGHT()
 2:     if DT detected interference then
 3:         Estimate $\hat{u}_t \leftarrow \xi(u_{t-1}, r_t, c_t)$
 4:         if $\hat{u}_t > u_{thres}$ then
 5:             Estimate $\hat{r}_t$, s.t. $u_{thres} = \xi(u_t, \hat{r}_t, c_t)$
 6:             Compute $\delta \leftarrow (r_t - \hat{r}_t)/r_t$
 7:             Set weight $w_{new} \leftarrow (1-\delta)w_{current}$
 8:             Check Max-Min Bounds ($w_{new}$)
 9:         end if
10:     else
11:         Reset default weight
12:     end if
13: end procedure
```

Collecting Training Data

To collect training data for estimating the function $\xi()$, we ran CloudSuite under various scenarios. Each run involved running the Faban workload emulator for 90 minutes with a constant number of concurrent clients. During a run, we periodically emulated various interference benchmarks (LLCProbe and Dcopy) (interference for 3 min., followed by no-interference for 2 min.). To emulate various degrees of interference, we varied the number of interference threads (between 2 and 4) and array sizes (between 8MB and 750MB per thread). This gave us sufficient variance in CPI and CMR. To collect measurements with varying server loads, we changed the HAProxy weights of the monitored server between 100, 80 and 60 across runs. This allowed us to collect measurements with sufficient variance in RPS values. The benchmark was run for a total of 9 runs. Note that we used this data for training both the DT and the Estimator. For building the DT, we labeled the observations based on whether an interference benchmark was running (class label: IF) at that time or not (class label: NI). Since the WS has different performance characteristics during interference and no-interference, we use only the data collected during interference for training the estimator (note that the estimator is used only when interference is in effect).

Estimating $\xi()$

We approximate $\xi()$ using multi-variate regression on variables r, c, and o, where o is a time-shifted version of the vector u (observed CPU utilization). Among the observed metrics, we found that RPS had significant number of outliers, hence standard IQR (inter-quartile range) based outlier detection techniques were applied on the dataset. The thresholds for the selected observations were chosen as $r > 1Q - 1.5 * IQR$ and $r < 3Q + 1.5 * IQR$, where $1Q$ and $3Q$ are first and third quartile values and $IQR = 3Q - 1Q$.

The metrics were also scaled to a value between 0 and 1 by normalizing with their range ($max - min$). The final dataset was fed to the `lm` regressor in R. We measure the accuracy of a regression model by its coefficient of determination or R^2 score. The R^2 score is defined as

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}},$$

where SS_{res} is the residual sum of squares and SS_{tot} is the total sum of squares (proportional to variance of the dataset). Intuitively, R^2 measures how much of the variance in the dataset is captured by the predicted model. Therefore, a higher R^2 score implies a better model. Initially, we found that the metrics collected with 1 sec interval had high variance in RPS and CPU and therefore resulted in a lower R^2 score. On the other hand, using a larger measurement interval (5 sec) averages most of the noises, thereby, giving a better model fit. Based on this observation, we used the measurements collected at 5 sec period as input to the estimator.

To check if a higher degree polynomial in $o, r,$ and c produces a better fit for the training data, we also ran regression with various degrees. The R^2 scores for models with varying degrees of polynomials are shown in Fig. 5. We found that models that include dependence of u with past CPU utilization (i.e., $u \sim (o, r, c)^i$) have better fit than those not considering this dependence ($u \sim (r, c)^i$). It can also be seen from Fig. 5, although higher degrees give a better fit, the improvements in R^2 values for degrees > 1 are very low. Moreover, in a linear model, the estimated values of u and r can be computed easily and the model can be updated over time with relative ease. We therefore choose a linear model in (o, r, c) as our estimator. The final prediction function $\xi()$ was computed as:

$$u = 0.06 + 0.64 * o + 0.21 * r + 0.24 * c$$

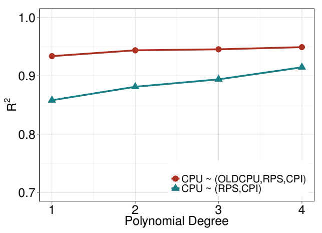


Figure 5. Accuracy of regression with varying degrees of polynomials. The metrics used for regression are shown in the plot labels.

WS Reconfiguration The final level of reconfiguration in ICE is performed by updating parameters of web server middleware (Apache, Php-fpm, Darwin etc.) hosting an application. Note that reconfiguring middleware parameters is costly since it often involves process creation or destruction. Also, the effects of parameter changes are only visible after a lag (usually 5-10 sec). Due to this, ICE performs Web Server reconfiguration only if an interference lasts for a long time ($>$ 20 sec). Our WS reconfiguration decisions are based on a knowledge base described in earlier research. [2] found that during interference the following reconfiguration actions are necessary to improve WS RT: $MXC \downarrow, KAT \uparrow, PHP \uparrow$, i.e., the optimal KAT and PHP increases during interference, while the optimal MXC decreases. New values for the parameters are computed based on proportional increase in CPU utilization (MXC) or response time (KAT). We use the controller described in [2] as the WS reconfiguration engine in ICE.

ICE is currently implemented as a Java program (ICE-Core) and a collection of shell scripts distributed across the WS cluster (refer Fig. 3). Logically, ICE-Core combines the functionalities of DT, LBE, and WSE in a multi-threaded application. We used the Weka toolkit for generating the DT, while R was used to compute the estimator function $\xi()$. Monitoring and reconfiguration actions are performed with the help of shell scripts and are also driven by ICE-Core.

V. Evaluation

We hypothesized earlier that the response time (RT) of a web server can be improved during periods of interference by taking two measures: a) by diverting traffic away from the impacted VM and b) by reconfiguring the web server parameters. These are the core design principles of ICE. Therefore, if we find that the average RT of a WS VM during interference is reduced by using ICE, we may conclude that our hypothesis is validated. More specifically, to evaluate the benefits of ICE, we ask the following questions:

i) How much improvement in RT is achieved by using ICE over a statically configured load-balancer?

ii) Does the performance of ICE vary across selection of scheduling policies (WRR and WLC)? If so, how much?

iii) How fast can ICE reconfigure in presence of interference?

We answer each of these questions in the following sections.

Setup. We use the same CloudSuite setup described in Sec. III to run our evaluation experiments. Only one of the VMs in the web server cluster was subjected to interference as before and we use the metrics collected from the impacted VM to perform our analysis. To quantify the RT improvement with ICE, we compare it against two scenarios: i) when the WS cluster is statically configured (i.e. the LB weight and WS parameters never change), we call this the baseline run, and ii) when the LB is statically configured, but the WS VMs are reconfigured using IC^2. While the improvement over baseline shows the overall benefit of ICE, its comparison against IC^2 shows how ICE outperforms existing solutions. The experiments are repeated with different scheduling policies at HAProxy to answer question (ii) earlier.

Interference Emulation. We emulate interference by running Dcopy and LLCProbe in a periodic manner. The array sizes chosen for our evaluation were LLCProbe (32MB), Dcopy-Low (30MB), and Dcopy-high (3GB) with 4 threads of interference. Note that only a subset of these interferences (LLCProbe 32MB and Dcopy-high 3GB, 4 threads) were present in our training set. To analyze results across interference types, only one type of interference is run at any given point in time. Each run of CloudSuite is an emulation

Table I

SUMMARY OF ICE RESULTS. NUMBERS INDICATE % CHANGE IN MEDIAN RT FROM BASELINE RUNS FOR DIFFERENT INTERFERENCES. ARROWS INDICATE WHETHER RT DECREASES (↓) OR INCREASES (↑).

Scheduling Policy	LLCProbe		Dcopy-High		Dcopy-Low	
	IC^2	ICE	IC^2	ICE	IC^2	ICE
WRR	29%↓	94%↓	18%↓	92%↓	59%↑	82%↓
WLC	5%↑	39%↓	15%↑	21%↓	35%↑	25%↓

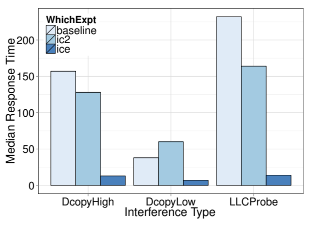
lasting 1 hour, during which interferences are run with a period of 8 minutes. Within each period, an interference benchmark runs for 4 minutes followed by 4 minutes of no-interference.

Data Collection. Note that our experiments involve measuring RT across various combinations of scheduling policies (WRR, WLC) and configuration controllers (Baseline, IC^2, and ICE). Each such combination is considered one experiment. Within one experiment, we run CloudSuite as described above for 5 X 1 hour runs. This gives us enough measurements for each interference type (LLCProbe, Dcopy-Low, and Dcopy-High) to have statistically significant results. The cumulative runtime of the evaluation experiments was more than 30 hours. The metrics observed by ICE during each run are stored in a log file and analyzed offline. Below, we present the results from our experiments.

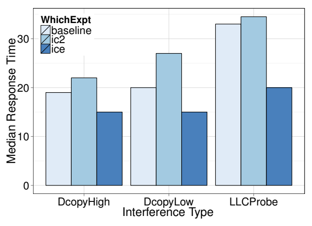
A. Improvement of Response Time due to ICE

The performance (RT) of ICE compared to baseline and IC^2 is shown in Fig. 6. The starting(stopping) points of interferences are shown with red(green) vertical lines. The magenta(blue) vertical lines show the points when HAProxy(WS) is reconfigured. The texts associated with the blue lines indicate new parameter values for reconfiguration. It can be seen from Figures 6(a) and 6(b) that the WS RT show very different characteristics depending on scheduling policy. With round-robin (RR), RT of the impacted WS always goes up and the degradation lasts as long as the interference. On the other hand, with least connection (LC), RT shows occasional spikes. Since LC scheduling implicitly accounts for server state (`#busy_connections`), an out-of-box load balancer with LC can mitigate the effects of interference to a large extent. However, in both cases, ICE outperforms a baseline run. It can also be seen in Fig. 6(b) that although the RT spikes in baseline are not very high or long-lasting, they persist throughout the entire duration of interference. In contrast, with ICE, after LB reconfiguration these spikes are much fewer. The performance of IC^2 falls midway between a baseline run and ICE. Although IC^2 is able to reduce the RT during stronger interferences, the reduction is not as significant as ICE.

Fig. 7 shows the median RT of the monitored WS VM across interference types with various scheduling policies. We find that, when using a WRR scheduler, ICE shows an improvement of 82-94% over baseline RT while improvement with IC^2 is 20-30%. Across interference types, LLCProbe, which is the strongest interference, shows the



(a) Scheduling: Round-robin



(b) Scheduling: Least Connection

Figure 7. Median Response time (RT) with IC^2 and ICE for various interferences against a statically configured load balancer (baseline). Note that baseline LC is able to reduce RT significantly compared to baseline RR. RT with IC^2 increases in LC (also with Dcopy-Low) due to overhead of dropped connections.

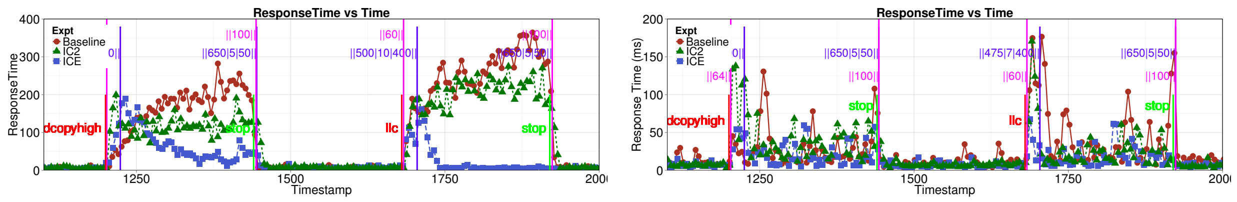
most performance benefit. Interestingly, with Dcopy-Low IC^2 shows performance degradation. With WLC scheduling the performance benefits are significantly less compared to RR. WLC by itself is able to direct some traffic away from the congested WSs and therefore help mitigate interference to a significant extent. However, using ICE alongside WLC improves RT even further. Across interference types this improvement varies between 21 and 39%. However, using IC^2 alongside WLC shows poorer performance compared to baseline (degradation of 5-35%). This is primarily because of the fact that, even though the WS is reconfigured to handle fewer clients the LB has no knowledge of it and it continues sending nearly the same number of requests to the WS as before. This results in increased overhead of queuing or connection dropping, thereby, increasing processing times over existing connections.

B. ICE has low detection latency

To measure how fast ICE can mitigate the effects of interference, we computed the detection latency of ICE from our evaluation runs. Note that detection latency defines the duration between the onset of an interference and the first reconfiguration action at HAProxy in response to that interference. This includes the performance of the DT classifier (how quickly and accurately it can detect interference) as well as any lag between detection and the corresponding reconfiguration action. Since HAProxy is only configured in the runs with ICE enabled, only these samples are used for measurement of detection latency. We found that across interference types, ICE had a median detection latency of 3 sec (the maximum being 4s). We observe that this is a very fast response considering most interferences in public clouds last for 10s of seconds. This is also much faster compared to IC^2 where reconfiguration happened with a delay of 15-20 sec. In IC^2, the authors didn't have access to HW counter data, hence had to rely on CPU Utilization and Response Time which increases with a delay. In ICE, use of hardware counters allows us to have much lower detection latency.

Performance of Classifier

To evaluate the accuracy of the classifier we also ran it on the data collected during our evaluation experiments. The collected data was labeled based on whether an interference



(a) Scheduling policy is round-robin (RR)

(b) Scheduling policy is least connection (LC)

Figure 6. Response time (RT) over time. ICE improves RT significantly compared to baseline and IC^2, with RR scheduling. With LC the lines are not clearly distinguishable, however, median RT is best with ICE.

benchmark was running. We found that the decision tree showed an accuracy (TP+TN)/(Total Samples) of 99.6%. Notice that the decision tree generated in our evaluations of ICE has ranges on a single attribute (CMR) (IV). It is therefore identical to a threshold-based detection, therefore the cost of classification is negligible.
We found that ICE incurred about 10% CPU overhead in the HAProxy load balancer for calculating average RT. Note that the total CPU utilization of the HAProxy was less than 60%. Therefore, our monitoring overhead does not skew the results. This can be eliminated by moving the RT calculation from HAProxy to a dedicated ICE VM. The monitoring overhead of other metrics (sysstat + ocount) was negligible.

So far, we have described the implementation of ICE only in the context of CloudSuite setup. However, the general concepts used here can also be used to adapt to other server setups and other types of workloads. We validate this by showing a set of simple experiments on the Darwin streaming media server in the next section. Darwin and Apache+Php are two very different server frameworks (the first one optimized for delivery of media streams, i.e. static content, while the second one is optimized for generating dynamic responses) and they have different workload characteristics (the former has long client sessions, while the latter is usually short). Therefore by validating ICE's design principles on both, we provide strong evidence of generalizability.

VI. Streaming Server Evaluation

We evaluated ICE's capabilities for widely used open source **Darwin streaming server** [14] (originally developed by *Apple*), which shares the same code base as **QuickTime** streaming server.

A. Monitoring and performance metrics

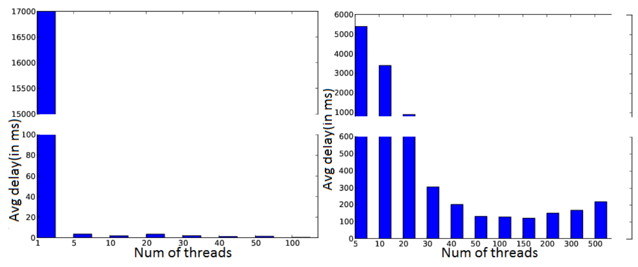
Video streaming servers do not follow a strict request-response pattern during its operation. Typically clients request for videos and some protocol messages are exchanged before the server starts streaming the videos to client devices. We define a metric called *frame-delay* to quantify the performance of the streaming server. While streaming, the server sends video frames to multiple clients. Based on the *frames per second (fps)* playing rate of the videos and clients' remaining buffer size, the server calculates an *expected* sending time for each frame. The time difference between when a frame was actually sent by the server to its expected sending time, is called the frame-delay. Under normal circumstances, this delay should be non-positive, i.e. the server will send the frame no later than its expected time for sending. When the server is overloaded or in the presence of interference, server will not be able to sustain its performance and frame-delay will increase.

B. Experimental setup

We set up two Streaming Server(SS) VMs in two physical machines and put a load-balancer in front. Each SS VM was allocated 2-vcpus and 2GB of RAM. SS typically uses RTSP for control messages and RTP to send data. Since often RTP is implemented on top of UDP, we use Linux Virtual Server (LVS) as our load-balancer as it can handle both UDP and TCP traffic. Specifically we used the Direct Routing (DR) configuration for LVS for our load-balancing purpose. We used CloudSuite's media streaming benchmark [13] to emulates realistic user requests through client browsers. Videos are requested at different resolutions with varying frequencies and realistic usage patterns [15], where some clients abruptly terminates the connection after arbitrary duration. We emulated 10,000 clients in a separate physical machine and LVS was configured to divide client load equally by default (i.e. each SS VM had a default weight of 100). Thus each SS gets request for videos from 5000 clients. We ran Darwin with `maximum_connections` and `maximum_bandwidth` set to `-1` (i.e., unlimited). All other parameters were set to their default values.

C. Experiments

a) Variation of frame-delay with number of threads: without interference
In this experiment, we find the optimum number threads that is sufficient for handling 5,000 clients connected to each SS. We modified the `run_num_threads` (number of threads created by the server to handle concurrent connections) configuration parameter. We used 600 seconds of steady state period for the client emulation engine. We found that SS is CPU bound and memory foot-print is not significant (less than 1 GB). As shown in Fig. 8(a) , when only 1 thread is used, the server experiences massive frame-delay. But the delay quickly drops and we found 10 is the optimum number of threads giving almost negligible delay. For higher number of threads, frame-delay remained negligible and we did not observe any significant increase due to context switching overhead.



(a) Frame-delay: normal condition (b) Frame-delay: with LLCProbe

Figure 8. Variation of frame-delay with number of threads: (a) under no interference. With just 1 thread, the server gets overloaded leading to long expected delay for the frames. With 10 optimum number of threads, the server shows negligible delay. (b) under LLC interference. With 150 optimum number of threads, the server shows minimum frame-delay.

b) Variation of frame-delay with number of threads: with LLC interference

Next, we find how the streaming server behaves under cache-interference (LLCProbe) and the optimum number of threads under such situation. We found LLC interference causes a significant disruption in the service. As shown in Fig. 8(b), even with optimum `run_num_threads`, i.e. 10, found under normal usage, the SS experiences significant frame-delay (3,200 msec). As we increase the number of threads, we found 150 is the optimum number where we observed the least frame-delay. Beyond that we observed slight increase in frame-delay. Thus, in the presence of interference, the optimum number of threads differs from normal operating environment.

c) Variation of frame-delay with reconfiguration of the load-balancer

Similar to WS, interference in media streaming servers can be detected early from the change in CPI and CMR. Once interference is detected, weights in the load-balancer can be dynamically reconfigured for reducing frame-delay. We experiment how frame-delay improves as we decrease the relative weight of the affected SS. We ran 2 sets of experiments, one with optimal `run_num_threads` (10) found under normal condition and another with optimal `run_num_threads` (150) found during interference. As shown in Fig. 9, as we decrease the relative weight of the affected SS (i.e. move the clients to the other server), the frame-delay metric improves. For relatively higher weights (100 and 90), using optimum number of threads for interference scenario gives better performance. But with further decrease in weight, performance of the Streaming Server converges for both thread counts. With a relative weight of 70, the frame-delay of the affected SS becomes almost negligible. Thus, the same design principle of ICE applies to two very different kinds of servers (web server and media streaming server) to improve on the metrics of interest.

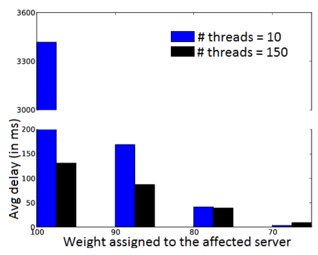


Figure 9. Change of frame-delay with load-balancer weights - when one real-server is under LLC interference. Two plots show how delay changes for optimum number of threads calculated for both interference and non-interference cases

D. Discussion: advanced streaming techniques and how ICE can help

While naive load-balancing technique by redirecting new requests for videos away from the affected server will bring down the frame-delay in a highly dynamic (i.e. lot of new requests coming in every second, and old streams end) streaming service. The responsiveness of the mitigation mechanism would work best if a significant percentage of the streams are short lived, which is the case in reality [15]. However, our technique will work best (for both short and long lived streams) if state-of-the-art streaming techniques such as Dynamic Adaptive Streaming over HTTP (DASH) [16] is used. By augmenting basic DASH protocol (as done by [17], [18]), it is possible to request the next *slices* of videos from a different server which is not affected by interference. Thus, even for long running streams we can effectively migrate the load to a different server which in turn will improve the frame-delay of the affected server as shown in Fig. 9. We leave implementation of such augmented DASH based load-balancing as our future work.

Summary:

From the experimental evaluations in Sections V and VI, we have the following three key takeaways:

1. An off-the-shelf Load Balancer can mitigate the effects of interference in VMs significantly by using the Least Connections scheduling strategy.

2. ICE improves the WS performance in the face of interference further, by detecting interference quickly and dynamically adjusting the weights of the Load Balancers.

3. The technique can be generalized as we applied it to a very different kind of server (media streaming server) with a different metric of interest.

VII. Related Work

Effect of interference on application performance has been evaluated by many researchers over the years [19], [20], [21], [22], [23]. In terms of contended resources, the papers can be classified as either related to memory interference (cache, memory-bandwidth) [19], [20], [21], or network interference [22], [23], [24]. In most cases, the performance degradation observed is severe enough to encourage implementation of sophisticated mitigation strategies. Among the mitigation strategies, better scheduling [25], [7], [6] and live migration [8], [21] are most common. However, each of these solution strategies has its shortcomings. Firstly, a VM's resource usage pattern may change over time, often unpredictably. A consolidation manager cannot foresee such usage changes without knowing of the applications running within the VM. Secondly, both types of solutions operate at a level which is beyond the scope of an end-customer. Solutions using VM live migration are slow compared to ICE, resource intensive, and may fail frequently when the source server is highly loaded [26]. In this work, we try to mitigate the effects of performance interference using intelligent reconfiguration of load balancers and web servers. The prior work closest to our solution is [2] which showed that it is feasible to mitigate interference using web server reconfiguration. However, the work shows limited benefit in a system that employs LB due to its overhead of reconfiguration. Our solution, ICE present a much more agile controller for managing large web server clusters. Our work is also partially related to research on building adaptive web servers using intelligent configuration engines [27], [28], [29], [30], [31], [32]. However, most of these are evaluated in a non-virtualized environment and the reconfiguration actions are performed in the order of minutes. This is unsuitable in a cloud environment, where transient interferences may render the choice of a given parameter value sub-optimal.

VIII. Conclusion

In this paper, we have presented a two-level configuration manager for web server clusters which can mitigate effects of performance interference in cloud. Our solution called ICE consists of a decision engine, a load-balancer configuration engine, and a web server configuration engine. We found that the decision engine can detect and reconfigure very fast ($\sim$ 3s). The combined configuration controller improves the median response time of the WS by upto 94% compared to a static configuration. We find that ICE also gives better response time by upto 39% compared to an adaptive load balancer (least-connection). We also show the applicability of ICE to a media streaming server (Darwin). Our future work involves identifying useful reconfiguration parameters for a server or a LB automatically and handling different forms of interference under one uniform configuration engine.

References

[1] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, “Cpi2: Cpu performance isolation for shared compute clusters,” in *EuroSys*, 2013.

[2] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, “Mitigating interference in cloud services by middleware reconfiguration,” in *Middleware*, 2014.

[3] B. Sharma, P. Jayachandran, A. Verma, and C. Das, “Cloudpd: Problem determination and diagnosis in. shared dynamic clouds,” in *DSN*, 2013.

[4] R. P. Mahowald and M. Rounds, “It buyer market guide: Cloud services,” in *IDCReport*, July, 2013.

[5] J. Dean and L. A. Barroso, “The tail at scale,” *CACM*, Feb. 2013.

[6] C. Delimitrou and C. Kozyrakis, “Paragon: Qos-aware scheduling for heterogeneous datacenters,” in *ASPLOS*, 2013.

[7] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, “Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms,” in *VEE*, 2007.

[8] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, “Deepdive: Transparently identifying and managing performance interference in virtualized environments,” in *ATC*, 2013.

[9] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, “Stay-away, protecting sensitive applications from performance interference,” in *Middleware*, 2014.

[10] “PAPI on Virtualization Platforms,” http://icl.cs.utk.edu/projects/papi/wiki/ PAPITopics:PAPI_on_Virtualization_Platforms.

[11] “HAProxy,” http://www.haproxy.org/.

[12] W. Zhang *et al.*, “Linux virtual server for scalable network services,” in *Ottawa Linux Symposium*, 2000.

[13] “CloudSuite,” http://parsa.epfl.ch/cloudsuite/cloudsuite.html.

[14] “Darwin Streaming Server,” http://dss.macosforge.org/.

[15] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, “Youtube everywhere: Impact of device and infrastructure synergies on user experience,” in *IMC*, 2011.

[16] T. Stockhammer, “Dynamic adaptive streaming over http –: Standards and design principles,” in *MMSys*, 2011.

[17] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang, “Qdash: A qoe-aware dash system,” in *MMSys*, 2012.

[18] A. Raha, S. Mitra, V. Raghunathan, and S. Rao, “Vidalizer: An energy efficient video streamer,” in *IEEE WCNC*, 2015.

[19] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, “Resource-freeing attacks: improve your cloud performance (at your neighbor’s expense),” in *CCS*, 2012.

[20] A. Verma, P. Ahuja, and A. Neogi, “pmapper: Power and migration cost aware application placement in virtualized systems,” in *Middleware*, 2008.

[21] R. Koller, A. Verma, and A. Neogi, “Wattapp: An application aware power meter for shared data centers,” in *ICAC*, 2010.

[22] S. K. Barker and P. Shenoy, “Empirical evaluation of latency-sensitive application performance in the cloud,” in *MMSys*, 2010.

[23] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, “Enforcing performance isolation across virtual machines in xen,” in *Middleware*, 2006.

[24] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, “Sharing the data center network.” in *NSDI*, 2011.

[25] R. Nathuji, A. Kansal, and A. Ghaffarkhah, “Q-clouds: Managing performance interference effects for qos-aware clouds,” in *EuroSys*, 2010.

[26] A. Verma, G. Kumar, R. Koller, and A. Sen, “Cosmig: Modeling the impact of reconfiguration in a cloud.” in *MASCOTS*, 2011.

[27] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh, “Online response time optimization of apache web server,” in *IWQoS*, 2003.

[28] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury, “Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server,” in *NOMS*, 2002.

[29] W. Zheng, R. Bianchini, and T. D. Nguyen, “Automatic configuration of internet services,” in *EuroSys*, 2007.

[30] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, “A smart hill-climbing algorithm for application server configuration,” in *WWW*, 2004.

[31] L. Wang, J. Xu, and M. Zhao, “Application-aware cross-layer virtual machine resource management,” in *ICAC*, 2012.

[32] X. Bu, J. Rao, and C.-Z. Xu, “Coordinated self-configuration of virtual machines and appliances using a model-free learning approach,” *IEEE Transactions on Parallel and Distributed Systems*, 2013.