Mitigating Interference in Cloud Services by Middleware Reconfiguration

[Research Paper]

Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi
Purdue University
West Lafayette, IN, USA
{amaji,mitra4,bzhou,sbagchi}@purdue.edu

Akshat Verma
IBM Research
New Delhi, India
akshatverma@in.ibm.com

ABSTRACT

Application performance has been and remains one of top five concerns since the inception of cloud computing. A primary determinant of application performance is multi-tenancy or sharing of hardware resources in clouds. While some hardware resources can be partitioned well among VMs (such as CPUs), many others cannot (such as memory bandwidth). In this paper, we focus on understanding the variability in application performance on a cloud and explore ways for an end customer to deal with it. Based on rigorous experiments using CloudSuite, a popular Web2.0 benchmark, running on EC2, we found that interference-induced performance degradation is a reality. On a private cloud testbed, we also observed that interference impacts the choice of best configuration values for applications and middleware. We posit that intelligent reconfiguration of application parameters presents a way for an end customer to reduce the impact of interference. However, tuning the application to deal with interference is challenging because of two fundamental reasons — the configuration depends on the nature and degree of interference and there are inter-parameter dependencies. We design and implement the IC^2 system to address the challenges of detection and mitigation of performance interference in clouds. Compared to an interference-agnostic configuration, the proposed solution provides upto 29% and 40% improvement in average response time on EC2 and a private cloud testbed respectively.

Keywords

Cloud performance, interference, dynamic configuration

1. INTRODUCTION

In the brief history of cloud computing, unpredictable application performance has been one of the two key issues prevening widespread adoption of the cloud paradigm. In a recent survey of IT buyers, about 40% cited application performance as a key concern [14]. Operational support for critical applications was another key concern with 40% IT buyers, making performance-related issues two of the top five concerns for cloud customers.

Performance issues in cloud are often attributed to misconfigurations of virtual machines (VMs), storage, and networks [24, 11, 26]. Another key reason for performance issues, which has not received adequate attention, is imperfect isolation of hardware resources across multiple VMs. Some resources, such as CPU and memory can be partitioned among VMs with little interference. However, current hypervisors do not isolate low level hardware resources, such as cache and memory bandwidth. Contention for these shared hardware resources leads to variable performance across VMs. Partitioning low-level hardware resources in software (hypervisor) will introduce significant overheads and we do not envision that processor caches and memory bandwidth will be isolated on a per-VM basis in the forseeable future.

Interference due to contention of shared resources can lead to severe performance degradation [28, 29, 12]. [28] reports that contention between two network intensive VMs can increase benchmark runtime upto 2X, while disk-to-disk and cache-to-cache contention can increase runtimes by 4.5X and 5.5X respectively. We found similar results in our experiments, where a cache intensive benchmark can increase average response time of a web server from a fraction of a second (10^{-1}) to several seconds. Existing work on handling interference in clouds are driven primarily from a private cloud perspective. The key idea is to either schedule interfering VMs at different points in time on the same host (e.g., [7]) or place interfering VMs on different hosts (e.g., [4, 20]). The first approach is limited in terms of the choices of VMs available on a host that can be co-scheduled. The second approach requires frequent live migrations, which is very resource intensive, especially when the source server is highly loaded [30]. Live migration in such a scenario is often long drawn and fails frequently. Further, it significantly impacts application performance during the migration. So, it is not suitable to deal with short-lived interference, which we observe is prevalent in EC2. Finally, these approaches are application-oblivious and can not accurately judge the real impact on application performance.

Our solution approach. In this paper, we present a complementary approach of handling interference by application reconfiguration. We argue that an application can mitigate

WOODSTOCK '97 El Paso, Texas USA

the ill effects of short-term interference — rise in response time and drop in throughput — by deploying an intelligent configuration manager. This configuration manager continuously monitors for interference and when it is observed, reconfigures the application and/or middleware (e.g., web server, database) to reduce contention for the bottlenecked resources. Our solution gives power in the hands of the application owners, and does not rely on the infrastructure provider making prompt changes to help the application with its periods of interference. This is also important because interferences in public clouds are often short-lived, less than a minute, and therefore application reconfiguration, which can be more agile than infrastructure reconfiguration, is particularly well suited.

We selected web applications as our preferred application domain for two primary reasons. First, web applications constitute a large portion of cloud workloads. [25] reports that nearly 25% of all IP-addresses in a portion of Amazon EC2 host a public website. Second, web applications and middleware components typically have a large number of tunable parameters with known performance benefits. We selected Apache as the web server primarily for its popularity—Apache has 53.32% market share of top million busiest sites as on May 7, 2014 [19].

In this paper, we make the following key contributions:
1. We rigorously study the performance variability of web-based applications in a public cloud environment. In this study, we run the CloudSuite [6] benchmark in Amazon's EC2 for 100 hours over a 5-day period. We then compare the statistics obtained from these runs with sample runs of CloudSuite in a private cloud testbed. We observe that CloudSuite has much longer response time distribution in EC2 (ranging upto 5.5s) than in the local testbed (upto 0.42s only) with identical resource configurations. This validates our hypothesis, that public clouds have high degree of performance uncertainty.
2. We conduct a study to understand if applications can be configured to deal with interference. We observed that an ideal operating configuration for Apache web server depends on the type and degree of interference. Further, parameters in different elements of the software stack depend on each other and the inter-dependency changes with the degree of interference; and finally, the application performance curves with the configuration values are discontinuous in places, making traditional control-theoretic approaches for parameter tuning [5] ineffective. Specifically we found three parameters corresponding to the degree of concurrency and the time to live of existing connections to be particularly significant.
3. We present a *simple, heuristic-driven* configuration manager, IC^2 to reconfigure the application upon interference. IC^2 solves three key challenges for dynamic reconfiguration—first, it presents a machine learning based technique for detecting interference; second, it uses a heuristic-based controller for determining suitable parameter values during periods of interference; and finally, it reduces the cost of reconfiguration of standard Apache distributions by implementing an online reconfiguration option in the Httpd server. A prototype implementation of IC^2 was deployed both in EC2 and our private testbed. The experiments show that IC^2can recapture lost response time by upto 29% in EC2 and 40% in our private testbed.

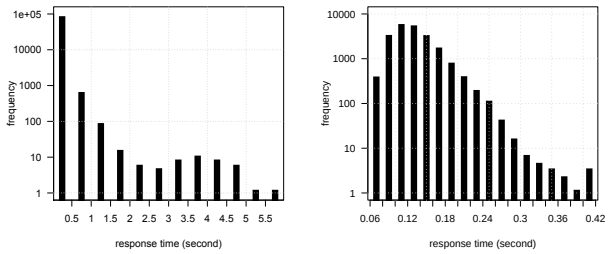
The rest of the paper is organized as follows. In Section 2, we verify the presence of interference in cloud platforms and present a quantitative evaluation of performance degradation. We next show the impact of interference in a private cloud testbed and identify how interference changes optimal configuration values for applications. The design of our proposed solution and performance improvement achieved by it are highlighted in Sections 4 and 5. We finally compare our work with current research and conclude the paper.

2. IS INTERFERENCE REAL?

We performed an experimental study to see if the performance concerns due to interference are real. Our objectives here are to answer two questions: i) Does an application suffer from unpredictable latencies in EC2? ii) What happens when a co-located VM starts accessing memory very fast? To answer the first question, we ran an application benchmark on Amazon EC2 with a constant workload setting and collected periodic performance data over 100-hours. We then analyzed the collected data to detect outliers and see how much performance variability there is. The application benchmark we selected for our experiments is CloudSuite, a popular web application benchmark [6] (more details on the application benchmark are in the next section). The web server and database of CloudSuite were installed on separate EC2 VMs each of type m1.large instances (equivalent to 2 vcpus or 4EC2 compute units, and 7.5GB memory).
Observations. We see that as a result of interference, there is significant variance in the performance of Olio on EC2 with regard to the response time (Figure 1(a)), and correspondingly, the throughput. The histogram is plotted such that the value represents response time between the two marks on the X-axis, e.g., there are 539 measurement intervals with response times between 0.5 and 1 sec. In contrast, for a similar experiment on local testbed (measurements taken over 60 hours) we found the response time was always < 0.5s. The response time distribution in EC2 has a much longer tail indicating periods of unpredictable performance. In EC2, we also measured the duration of interference using an outlier detection method as shown in Equation 1. Our results indicate that there are several instances when interference lasted for 30s or longer, the longest duration being 140s. While these interference instances are a small portion of the total number of requests, there are two conditions that suggest we need to deal with them—they are unpredictable and therefore, worst-case provisioning for performance critical applications suggests we must put in place mechanisms to deal with them; when interferences do happen, they cause pathologically poor behavior of the application and may push the application into a "death spiral". Evidence of death spiral in applications due to transient degradation has been given in the past, such as, due to overfilling of application queues [23].

$$| P_i - P_{N/2} | > C \times median(| \{P_i\}_{i=1}^{N} - P_{N/2} |) \quad (1)$$

To answer the second question raised earlier, we ran another set of experiments both on EC2 and private cloud testbed. The results indicate cache-intensive interference from co-located VMs can increase response time of a web server by an order of magnitude. Our experiments in following sections substantiate this point.



(a) (b)

Figure 1: Distribution of response time of Olio running on (a) Amazon's EC2 (b) Private cloud. VM resource settings and workload intensity are indentical in both cases.

3. INTERFERENCE IMPACTS OPTIMAL CONFIGURATION VALUES

In this section, we endeavor to understand the relationship between optimal configuration values of middlewares and interference. To do so, we ran an extensive set of experiments with Cloudsuite in a private cloud testbed. We first describe our experimental setup and then highlight our findings.

3.1 Experimental Setup

Hardware. Our private cloud testbed consisted of three Poweredge 320 servers with Intel Xeon E5-2440 processors. Each server has 6 cores (or 12 hardware threads with hyperthreading enabled), 15 MB cache and 16 GB memory. We installed the KVM hypervisor on these machines. We co-located our custom interference VMs on the same host as the web server, while database VM was run on a separate physical machine. In this work we focus only on web server performance and over-provisioned the DB VM to eliminate any DB bottleneck. The database (approx. 1.6GB) was also loaded in memory to reduce disk contention. The third machine of our setup was used to run the benchmark driver and rest of the client emulators. All the computers were connected via a dedicated 1 Gbps switch. Table 1 lists the values of different configurations for each experiment presented in this paper. It is to be noted that, we never created contention for CPU and memory on the physical server. With two interference threads running and 4 vcpus for WS, the physical server's CPU utilization was at the 50% mark or lower for all experiments. Similarly, memory utilization of the host was never an issue. Our maximum WS memory utilization was well below 3GB for all workloads.

Application Benchmark. The application benchmark we selected for our experiments is CloudSuite, a popular web application benchmark [6]. CloudSuite internally uses Olio, a social event calendar application as the base package. Throughout the rest of this paper we use the terms "CloudSuite," "Olio," and "Application benchmark" interchangeably. In our setup, we hosted Olio on a multi-threaded Apache server (*apache-worker* v2.4) and used Php Fastcgi Process Manager (*php-fpm*) for dynamic content generation. Our setup closely resembles a typical three-tier application with *php-fpm* v5.3 as the busines logic (BL) tier. We use identical CloudSuite setup in all our experiments—homogeneous VMs with Ubuntu 12.04/Apache 2.4/php-fpm 5.3/Java 1.7. CloudSuite uses the Faban harness to emulate clients. Client emulation is done using a pre-defined distribution (negative exponential) of think times and operation mixes as defined in [21]. Workload size is given in terms of `#concurrent_clients`.

Interfering Application. We emulated interference from co-located VMs by running two different benchmarks–LLCProbe and Dcopy–on two VMs (also referred to as inteference VMs). Dcopy is an application under the BLAS [1] benchmark suite, which copies contents of a source array to a destination array. LLCProbe [28] creates an LLC (Last Level Cache) sized array in memory and then accesses each cache line very frequently. Both Dcopy and LLCProbe are cache intensive, however, rate of cache access is higher in LLCProbe than in Dcopy. Moreover, by using Dcopy with a large array size we can also emulate memory bandwidth contention. Interferences of this type may arise in reality if a co-located VM runs data mining applications like Hadoop or even under periodic consolidation operations. Earlier work has shown [28, 30] that such interferences are a routine occurence in present-day cloud infrastructures.

Parameter Selection. In our experiments and subsequent evaluations, we consider three key configuration parameters – `MaxClients` (MXC) and `KeepaliveTimeout` (KAT) from Apache web server[1] and `pm.max_children` from Php runtime. These parameters greatly impact Apache's web application performance [5]. `MaxClients` captures the maximum number of parallel threads the web server employs to serve requests. This is typically configured based on the workload intensity, number of hardware threads available on the physical server, and its RAM capacity. `KeepaliveTimeout` indicates how long a web server would keep an idle client connection in its connection pool (typically occupying a thread). `pm.max_children` defines the maximum number of threads used by the Php interpreter. We refer to `pm.max_children` as `PhpMaxChildren` in this paper. It is pertinent to note that these parameters are generic thread-pool management parameters and have their counterparts in most commercial server distributions making our study applicable to most enterprise middleware (e.g. thread pool size in glassfish).

Metric Collection. CloudSuite (Olio) uses Faban harness to emulate clients and generates high level benchmark metrics for each run (Response Time and Throughput). For each data point in our plots (i.e. a given setting of configuration values or workload) we consider an average of three runs. Each run lasted for 10 minutes (excluding ramp-down) of which last 5 minutes were considered as steady state and reported. The experimental VMs were rebooted after each run to clear any state. For monitoring hardware performance counters we started oprofile [22], a low overhead profiler, on the hypervisor of the web server VM. We use the observation that a guest VM in KVM is represented as a qemu process in the hypervisor. We used oprofile to monitor the hardware events corresponding to the qemu process of the WS VM. We next report some of our key experimental results.

3.2 Impact of interference on middleware configurations

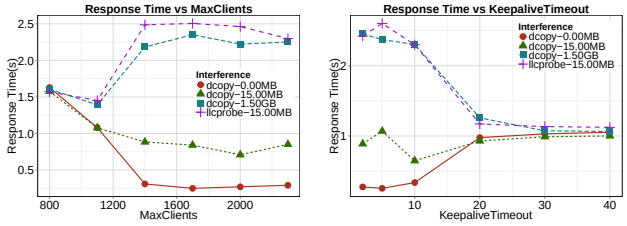
[1]We use the terms Apache web server and Httpd synonymously to identify the Apache web server.

Experiment	# Vcpus	Memory(GB)	MaxClients	KeepaliveTimeout	PhpMaxChildren	Load Size
Sec. 3.2.1	4	4.5	Variable	5	1000	1500
Sec. 3.2.2	4	4.5	1700	Variable	1000	1500
Sec. 3.3	4	4.5	Variable	Variable	1000	1500
Sec. 2,5	2	7.5	650*	5*	50*	550

Table 1: Summary of WS VM config. and parameters during different experiments. Values with asterisk are reconfigured with IC2.

In this experiment, we evaluate the impact of interference on the choice of optimal values for the three parameters—`MaxClients`, `KeepaliveTimeout` in Apache and `PhpMaxChildren` in Php-fpm. For each of these parameters, we ran the web server with different interference intensity - LLCProbe with array size of 15MB and Dcopy with array sizes 15MB and 1.5GB. Due to its fast cache access, LLCProbe emulates a strong interference, while Dcopy 15MB emulates a low interference. With Dcopy size of 1.5GB we emulate contention for both cache and memory bandwidth, and its overall effect is that of a moderate interference. Here a Dcopy size of 0.0MB implies a run where no interference benchmark was run (baseline). For each interference intensity, we varied one parameter of Apache while the other was set to an observed good value. Run configurations for each experiment can be seen from Table 1. For all the experiments, we kept the workload intensity (`#concurrent_clients`) to a fixed value of 1500 which was found to be lower than the saturation point of the web server[2]. Note that although we have a constant number of concurrent clients, Faban may generate bursty traffic in some intervals due to is stochastic "wait time".

3.2.1 *Effect on* `MaxClients`



(a) Response Time vs. MXC (b) Response Time vs. KAT

Figure 2: Choice of optimal parameter values with varying Dcopy and LLCProbe. For all experiments, `#concurrent_clients` is 1500, chosen default values are $MXC = 1700$, $KAT = 5$, and $PHP = 1000$. In each experiment, one of the parameters are varied while others are kept constant at their default values.

Figure 2(a) show the choice of optimal `MaxClients` (MXC) values for different interference intensities. In the baseline case (Dcopy 0MB), best response time can be obtained by setting MXC to 1700. However, the optimal value reduces to 1100 for Dcopy-1.5GB and LLC-15MB. Interestingly, with a smaller interference of Dcopy-15MB, the optimal value increased to 2000 (although the gain in response time was small compared to at 1700). Even though all the curves show concave nature before saturation, they diverge from each other significantly clearly highlighting a change in the operating environment.

It can also be seen that interference causes the response time of WS to go up from fraction of a second (< 0.5s) to several seconds. If we keep MXC constant at the baseline optimal value of 1700 (refer Fig. 2(a)), with LLCProbe it increases upto 2.5s. However, with a different MXC value (1100), this degradation can be limited to only 1.5s. One may argue that we can always keep MXC fixed at 1100, but this wastes server resources (e.g. baseline throughput at 1100 is 13% lower than that at 1700). A better alternative is to configure it for the dominant case (no-interference) and to reconfigure when interference is detected.

3.2.2 *Effect on* `KeepaliveTimeout`

We found similar results for variable `KeepaliveTimeout` (KAT) which suggests different optimal KAT values for varying interference intensity (refer Fig. 2(b)). For this experiment, we kept the MXC value fixed at the optimal baseline MXC value of 1700 and varied KAT from 2 to 40 seconds. The curves show very different patterns with varying interference intensity. In the baseline case, increasing KAT beyond 5s increases response time. On the other hand, with strong interferences increasing KAT reduces response time significantly. Based on this, one may argue that we can always keep KAT fixed at a high value (e.g. 20). We see from the plot that such a choice is suboptimal for no-interference; it also shows poor throughput. As a general rule we found that interference from co-located VMs increases the optimal KAT value. This emphasizes that a web application needs to reconfigure its KAT value in the presence of interference and finding the optimal is a non-trivial problem.

Due to space limitations we only present the key findings of varying `PhpMaxChildren` (PHP). In general, increasing PHP had almost no impact in no-interference response time. We therefore choose a low PHP value with lower memory footprint as the no-interference optima. With interference, optimal response time is seen for `PhpMaxChildren`= 800 or higher although the performance improvement is smaller compared to `MaxClients`. Interested readers may find the details in [16].

Table 2 presents a summary of our observations about optimality of parameters and the relationship with interference. Each cell in this table summarizes the impact of interference on the optima of a given parameter (whether it increases or decreases) and the degree of impact this parameter has on performance (high or low).

3.3 Change in inter-parameter dependency

In this section, we answer a commonly asked question on configuration management–are two parameters independent? We verify this with the specific example of two parameters that had the most profound effect on the performance of our benchmark applications, namely, `MaxClients`

[2]We define saturation point to be the minimum workload intensity when the web server exhausts its cpu or memory capacity.

Table 2: Summary of our experiments on evaluating the impact of interference on optimal parameter values.

Application runtime	Apache and Php
Operating context changes	Cache, memory bandwidth pressure

General impact on optimal configuration values			
Context	`MXC`	`KAT`	`PMC`
No interference Initial value	High	Low	Low
With Interference Performance Impact	Decrease High	Increase High	Increase Low
Memory Pressure Performance Impact	Decrease High	Increase High	Decrease High

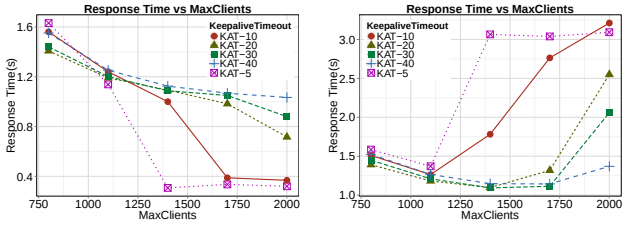
and `KeepaliveTimeout`. We find that dependency does exist between these parameters and it changes with interference.

For this experiment, we varied both MXC and KAT for the Apache server under two scenarios. In the first, we ran the web server with no interference, while in the second, we ran it with LLCProbe-15MB. We found that the nature of curves changed significantly across these experiments (refer figures 3(a) and 3(b)). For the case with no interference, the curves generally have a negative slope, while with interference, the curves display both positive and negative slopes. Choice of optimal KAT for a given MXC is significantly different in the two. As a general observation, we find that lower KAT is better at baseline while higher KAT is better during interference.

One may argue that the following simple equation suffices to determine KAT value for a given MXC:

`KAT` = `MXC/#new_connections/sec`

However, this does not work well during interference. For example, during interference, if we reduce MXC then according to this formula we should also reduce KAT to maintain a constant connection rate. But such an action would further increase load on the server. Due to shorter KAT, a larger fraction of established client connections would time out, necessitating new connection establishment. A better alternative is to be aware of interference and select a different value for `#new_connections`. This emphasizes the need for the tuning algorithm to be context aware. Depending on the presence or absence of interference it must select a different optimal KAT value for a given MXC.



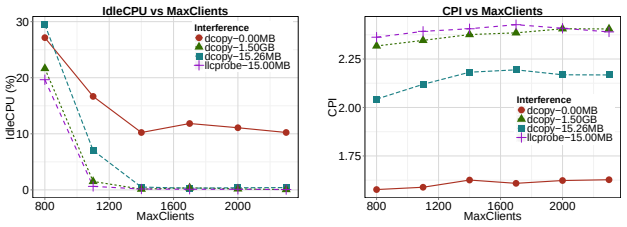
(a) Response Time vs. MXC with varying KAT (no interference)

(b) Response Time vs. MXC with varying KAT (with LLCProbe)

Figure 3: (a–b) Depedendency between `MaxClients` and `KeepaliveTimeout` changes with interference.

3.4 Interference and web server capacity

In the previous section, we found that interference has a significant impact on the resposne time of a web server. In this section, we ask ourselves what is the root cause for such increase? To answer this question, we analyzed the system metrics obtained from the previous experiments (Sec 3.2). We also evaluate the impact of interference with varying workload sizes. Our observations are presented below.



(a) Increase in CPU utilization

(b) CPI of WS-VM

Figure 4: Effects of interference. Here we identify system level bottlenecks that causes response time to increase by an order.

Interference increase CPU utilization of WS-VM. We found that for a memory allocation of 4.5GB, the WS was never constrained for memory. But the cpu utilization (Fig. 4(a)) values showed significant botleneck. It can be seen that for a given choice of `MaxClients`, the IdleCPU values for with-interference curves are lower than baseline. Note that the IdleCPU values are virtual utilization measured inside the WS VM. Intuitively, for a constant workload this should remain fixed irrespective of the functioning of a co-resident VM. To understand this behavior, we measured the CPI (cycles per instruction retired) values for the WS VM with varying degrees of interference. Due to the large number of cache misses induced by the interference VM, the WS VM uses more of its cpu cycles fetching data from memory to cache and consequently the CPI increases. It can be seen from Fig. 4(b) that the CPI values for the WS with interference is between 2 and 2.25, whereas, baseline CPI is only 1.5. It implies that, on average, a WS thread takes longer time to finish execution. The overall effect is that a larger fraction of the WS VM's time slice is occupied by some busy thread. This is reflected as increased cpu utilization inside the guest VM.

Interference increases active memory of WS-VM. Similar to Section 3.4, we found active memory of the web server increased during interference. This happens since with interference, Apache threads are active for a longer duration on average (higher response time). Note that an active Apache thread has larger memory footprint than in idle one (in Apache terminology active threads includes the `request pool`, a large block of memory for storing the request and response data, in addition to `server` and `configuration pools`), therefore longer response time implies increased active memory. We found that this observation becomes even more significant if the web server is under memory pressure. In such a case, if a web server's memory footprint is just below capacity in a baseline case, with interference it is likely to start swapping. This again has catastrophic impact on performance. We verified this hypothesis by running the WS VM with 2GB RAM in a separate experiment and found evidence of swapping with interference

even though no swapping happened in a baseline run. Due to space limitation, we omit the details. Based on Fig. 4(a), we conclude *interference reduces the capacity of a web server*.

4. DESIGN AND IMPLEMENTATION

In the previous section, we found that the choice of optimal configuration values for web services middleware depend significantly on interference created by co-located VMs. Here we propose the design of a configuration manager that is aware of the *operating context* [26] of the web server VM. Although most of our implementation focuses on mitigating impact of cache-intensive interference, the same principles can be applied for other types of interferences (e.g. network). To design an interference-aware configuration manager we need to answer three important questions:
i) How do we detect a web server is suffering from interference?
ii) Which parameters can be configured to mitigate interference?
iii) For the parameters determined in step (ii), how should their values be set as a function of the degree of interference?
We answer each of these questions in rest of this section.

Fig. 5 presents a high level system architecture of the proposed solution. IC^2 consists of two primary modules: a) *Performance monitor*, and b) *Config manager*. For all the VMs that are part of a web application (e.g. web server, database and mail server) and managed by IC^2, performance monitor collects performance data at three levels. At the application level it collects aggregate response time and throughput measurements, whereas at system level, it collects utilization values for CPU, memory, IO, and network. If hardware performance counters are available (on our local testbed, but not on AWS), it also collects CPI (cycles per instruction) and CMR (last level cache-miss rate) data for the monitored VMs. Based on the collected data, config manager can detect if any system context has changed. This can either be a change in workload, VM resource allocation or presence of interference. There are several existing solutions that can handle workload and resource changes [34, 5] and these can run concurrently with our solution.

A high level functioning of IC^2 is shown in Fig. 6. After collecting metrics, IC^2 tries to detect if the web server is under interference. Based on the detection result it maintains a state machine for the web server. The state machine, in turn, is used to decide when reconfigurations are needed. Finally, the config controller actuates the reconfiguration action. Details of this configuratin loop is presented below.

Interference detection. Any interfering VM that is accessing large amounts of memory, such as our two experimental interference VMs running DCopy or LLCProbe, will ultimately cause a pressure on the shared cache on the physical machine. We find empirically that a sharp increase in CMR is a *leading* indicator of interference. For example, during our experiments in Section 3.2, we found the CMR of the web server VM was always $< 5\%$ in a no-interference run, whereas, during interference it increased to 15% or higher (depending on the degree of interference). In our local testbed implementation of IC^2 we used increased CMR as conclusive proof of interference. This approach, however, cannot be used in public clouds due to the policy of disallowing access to hardware counters. For our experiments on EC2, we used a sharp rise in CPU, reduction in throughput (THPT) , and increase in response time (RT) of the application VM as secondary evidence of interference.

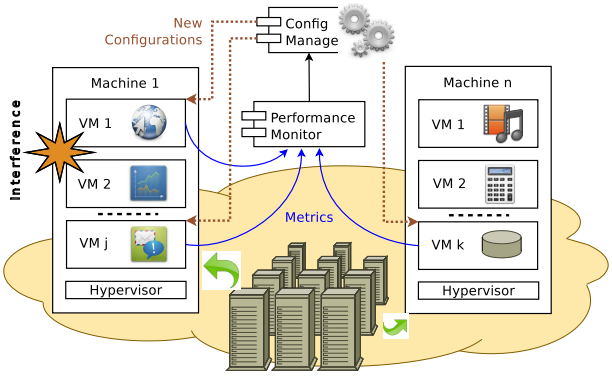


Figure 5: System architecture of IC^2

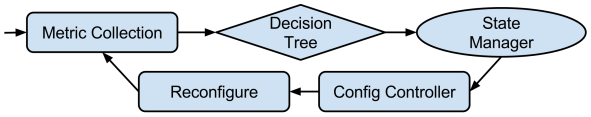


Figure 6: High level functioning of IC^2

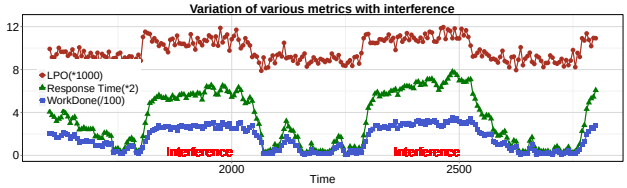


Figure 7: Interference impacts load per operation (LPO) and work done (WorkDone) by a web server. These, together with response time, can be used as metrics for detecting interference. The values are normalized by the factors shown in figure for better visualization.

Instead of using raw CPU utilization which may show sharp fluctuations due to stochastic nature of request arrivals, we use a normalized metric Load Per Operation (LPO). LPO is defined as $LPO = \frac{CPU_{util}}{Throughput}$. We also define another derived metric $WorkDone = RT * THPT * CPU_{util}$. Intuitively, Workdone approximates the number of CPU cycles spent to serve all the requests during current measurement interval. Without interference, assuming the server is not saturated, Workdone is small since $RT < 1s$ even though throughput is high. With interference, however, workdone is large as response time increases significantly even though throughput reduces. To determine applicability of LPO and Workdone for interference detection, we ran CloudSuite in EC2 for multiple 1-hour runs. During these runs we periodically start Dcopy on a co-located VM at fixed intervals of 8 minutes (4 minutes of interference followed by 4 minutes of no-interference). The collected metrics for one 1-hour run is shown in Fig. 7. It can be seen from the figures that both LPO and Workdone form distinct clusters with and without interference. It may be argued that interference detection

based on CPU utilization may fail to detect small interferences that does not increase utilization above threshold. In our experiments, we found such interferences have minimal impact on response time. Here, our primary focus is to detect pathological cases that saturate server resources.

Decision Tree for Detecting Interference. To detect interference in EC2 we built a Decision Tree classifier using the attributes LPO, Workdone, and Response time. A decision tree generates a finite set of "tests" on attribute values to determine the class of a given sample. Our choice of decision tree is due to its two key advantages: i) simplicity—it is easy to visualize the rules in a decision tree ii) customizability—an administrator can manually change the thresholds of various attribute values based on expert knowledge or QoS requirements. Our classifier consists of 3 classes: Interference, No-interference, and Transient. The Transient class is introduced to capture temporary fluctuations in performance (e.g. immediately after starting or stopping of emulated interference). Based on current observation values the classifier tries to pedict if the web server is suffering from interference or interference has gone away.

A key challenge in building the decision tree is to deal with changes in parameter values. When IC^2 reconfigures a web server during interference, its performance metrics change and therefore can lead to misclassification. A possible solution for this is to collect metrics with various combination of parameter values, with and without interference. However, collecting training data for all combinations of parameter values is time consuming and may even be impractical. We therefore select an alternate bootstrapping approach where the classifier is trained in 3 phases. Each training phase consists of 10-hour run of Cloudsuite and is done offline. In phase I, we run Cloudsuite with baseline optimal setting (IC^2 disabled), periodically generating interference. The collected data is used for training the phase I (base) classifier. In Phase II, we repeat the experiment with IC^2 enabled and use the base classifier for interference detection. The metrics collected approximate measurements with random parameter combinations. We use 50% data from Phase I and 50% data from phase II measurements to train the phase II classifier. Finally, in phase III we use phase II classifier and collect more data with IC^2 enabled. The data collected in phase III is used for training the final classifier. Note that, during training we use only Dcopy with varying array sizes and intensity (`#dcopy threads`) as our interference benchmark, while in evaluations we use both Dcopy and LLCProbe to test our detection module. We used the Weka [10] toolkit to create the decision tree.

Configuration Controller. IC^2 internally uses a simple state machine to keep track of current operating context of the web server and generate reconfiguration triggers (Fig. 8). In local testbed, the state machine consists of only two states and interference detection is merged with the state machine. We use response time in the trigger to ignore cases where response time was within QoS values, this prevents the server from incurring reconfiguration overheads during less-intense interferences. Self loop in the state diagram are the negation of the trigger condition on outgoing edge. In EC2, however, we use a 5-state machine, two representing interference and two representing normal (no-interference) runs, and one for the transient phase described in the previous paragraph. The transition labels are classifier outputs based on recent observations. Our choice of 5-states instead of two serves two purposes: i) Due to ambient interference in EC2, state changes may be short lived. Reconfiguring frequently in such cases may impact throughput. Our design forces IC^2 to reconfigure only after it has seen two successive periods under interference or no-interference (assuming the current phase will last a while). ii) This hides classifier false positives. For example, if the server is under no-interference but the classifier predicts interference, it would take at least three successive misclassifications for a reconfiguration (No-interference -> Transient -> Interference -> Interference), the probability of which is much smaller than the classifier error rate. IC^2 performs reconfiguration actions when the server enters the states `I2` or `NI2` as shown in Fig. 8.

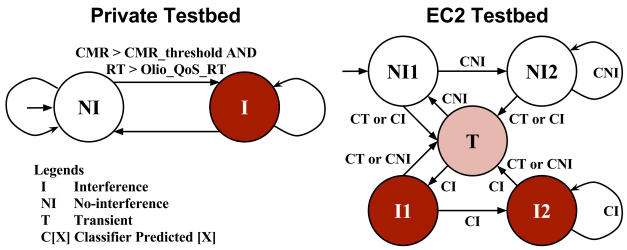


Figure 8: State transitions of IC^2. In EC2, reconfiguration is done when the server enters I2 or NI2.

Reconfiguration actions. Our reconfiguration actions in IC^2 are currently implemented as a heuristic backed by a knowledge base (refer Table 3). This knowledge base directs IC^2 *which* parameters to reconfigure when a trigger is detected. It does not include precise values of the parameters but instead specifies a set of rules. The knowledge base can be created in two ways: i) with the help of a domain expert, ii) analyzing performance logs from training runs. Note that most commercial web applications go through load testing phase before going to production. A systematic variation of critical middleware parameters (as in Section 3.2) during these tests can generate insights about application performance. Our current implementation deals with row 3 of Table 3, i.e. increased CMR. Our earlier experiments suggest that the actions MXC↓, KAT↑, and PHP↑ can improve application performance during phases of cache interference. We reconfigure all three parameters simultaneously.

Table 3: Knowledge base for web server reconfiguration

Context Change	Configuration Heuristic
Increased Workload (High Idle Memory)	MXC↑ and PHP↑
Increased Virtual/Physical CPU ratio	MXC↓ and KAT↑
Increased LLC Miss Rate	MXC↓, KAT↑, PHP↑
Increased Host Memory Contention	MXC↓ and PHP↓
Increased Page Faults (Active Memory Low)	PHP↓

Update functions. The quantitative update functions for the three parameters are shown in Algorithm 1. The updation objective for MXC is to reduce CPU demand of the web server. We therefore decrease it proportional to the increase in CPU utilization (approximated by δ_{LPO}). We restrict the

Algorithm 1 Parameter update functions for IC^2

1: **procedure** RECONFIGURE_FOR_INTERFERENCE()
2: $\delta_{MXC} \leftarrow ((MXC * \frac{LPO - LPO_{nointf\_median}}{LPO}))$
3: $\delta_{MXC} \leftarrow checkBounds(\delta_{MXC})$
4: $\delta_{KAT} \leftarrow (\delta_{response} * C_{KAT})$
5: $\delta_{KAT} \leftarrow checkBounds(\delta_{KAT})$
6: **update_params**$(\delta_{MXC}, \delta_{KAT}, (400 - PHP))$
7: **end procedure**

new value to be within a min-max bound so that throughput does not degrade alarmingly. A similar objective function can be realized for a memory constrained web server by considering memory utilization. An underlying assumption here is that the server's CPU utilization is dominated by the Apache Httpd server. In our setup, though Php-fpm was used for dynamic content generation, we found the impact of `PhpMaxChildren` on response time/throughput was much smaller than `MaxClients`. This likely indicates that the effect of `PhpMaxChildren` on CPU utilization of the VM was marginal.

On the other hand, increase in response time implies the server's average request cycle time (response time + wait time) is increased. We increase KAT proportional to the $\delta_{response\_time}$ to offset increased cycle time, i.e., to keep a connection alive for longer since the server is taking longer time to respond to client requests. During experiments in Fig. 2(b), it was found that the increase in optimal KAT value ($KAT_{opt}^{intf} - KAT_{opt}^{nointf}$) during interference is several times larger than $\delta_{response\_time}$. Therefore, a constant multiplicative factor (C_{KAT}) is used with $\delta_{response\_time}$ to come up with the change in the KAT value. We empirically determined the value of C_{KAT} to be 3. For `PhpMaxChildren`, we found performance improvement beyond a certain value (400, for a VM with 2vcpus) is negligible. We therefore select two constant values of PHP for interference and no-interference scenarios.

Implementation. IC^2 currently has been implemented as a Java application which combines the functionalities of Performance Monitor and Config Manager described in Fig. 5. One instance of IC^2 is designed to handle an application group as shown in Fig. 5, e.g. an application group may consist of web server, database server, and e-mail server. In current implementation, we focus on managing the web server. IC^2 uses remote scripts to fetch performance metrics from various levels of the monitored systems. It uses Faban logs to collect application level metrics (in periodic intervals of 5s), and uses `sysstat` utilities (inside WS VM) for cpu and memory utilization. In the local testbed, we also collect hardware counters from the hypervisor. Based on the collected data and configured threshold values, it detects if an interference has started or stopped. It then sends reconfiguration commands to the Apache and Php servers. A separate program was implemented to start and stop the interference benchmarks in periodic intervals.

Redesigning Httpd. During initial testing with IC^2, we found that CloudSuite had significant increase in response time and decrease in throughput immediately after a reconfiguration. This transient phase lasted between 30-60s and was determined to be a limitation of Apache Httpd server. In order to update configuration values, the server has to restart all child proceses. This is essential because Httpd internally assumes that configuration values are never changed (read-only)—doing so allows it to avoid synchronization overhead during request processing.

To avoid the penalty of restarting Httpd, we implemented an online reconfiguration option for Httpd. The online reconfiguration option enables Httpd to gracefully change over from old parameter values to new parameter values without needing to shut down and restart all worker processes. We noted that `MaxClients` is used only in Httpd master process to control the number of worker threads. The children processes (workers) are oblivious of MXC. Therefore MXC can be updated in master (and subsequently propagated to children) without requiring restart. KAT value is read at the end of every request processing, therefore any change to it is reflected in the next request. Assuming a relaxed consistency model, we can modify KAT in master and propagate the changes to children later.

We implemented a custom signal handler (`SIGUSR2`) and an online reconfiguration command (`apachectl reconfigure`) in the Httpd server (`worker mpm`) to initiate online reconfiguration of these two parameters. The signal is delivered to the master process by `apachectl` and later propagated to children via Httpd's Pipe of Death (POD) implementation. We also updated the Scoreboard structure to store runtime values of MXC and KAT. The reconfiguration decisions are implemented in `server_main_loop()` in master and `child_main()` in children. Our implementation involved adding/modifying 500 lines of code in current Apache codebase (v2.4.3). With our implementation of online Httpd, the server showed significantly less overhead of reconfiguration as explained in the next section. The modified version of Apache can be downloaded from [15].

5. EVALUATION

In this section, we evaluate the effectiveness of IC^2 in detecting and remediating interference. The high level objective here is to reduce the response time for the web server during periods of interference. Therefore, if the average response time after reconfiguration is lower than that before reconfiguration, we consider IC^2 to have achieved its objective. More specifically, we ask ourselves the following questions, individually for the local testbed and Amazon EC2:
i) Can IC2 successfully detect interference?
ii) How much improvement in response time can be obtained by running IC^2?
iii) What is the overhead of reconfiguration in IC^2?

To quantify the performance change due to IC^2, we compare IC^2 with the performance of an interference-agnostic controller. We assume that an interference-agnostic controller is able to achieve optimal parameter setting under normal runs and does not react to interferences. We found that for resource configurations equivalent to EC2 `m1.large` instances, the optimal parameter values for no-interference runs were $< MXC = 650, KAT = 5, PMC = 50 >$. For all the experiments, we consider a CloudSuite workload with 550 concurrent clients which is below the saturation point of the web server.

Interference Emulation. To emulate interference we started the interference benchmarks with varying array sizes at different instants in time. To simplify implementation, we consider a periodic interference behavior as opposed to a stochastic behavior. Due to transient behavior of httpd-

basic immediately after reconfiguration, we found it difficult to precisely evaluate benefits of IC^2 with a bursty interference. For our evaluations the interference benchmarks are on for 240s followed by an off period of 240s. We selected emulated interference to evaluate IC^2 instead of natural interference in EC2 primarily because of two reasons: 1) Interferences occur infrequently enough to make statistically significant results difficult within a reasonable experimental time. 2) The nature (intensity and duration) of interference may change every time making it hard to draw comparable results. We run LLCProbe with an array size of 20MB, Dcopy with 20MB (also referred to as Dcopy-low) and Dcopy with 1.5GB (Dcopy-high). On EC2, this synthetic interference happened in addition to ambient interference in the environment. On the local testbed, we ensured that no other VM, extraneous to our experiment, was running.

Co-location in EC2. In order to evaluate IC^2 in EC2 we needed to co-locate some of our VMs on the same machine as the WS VM. This is necessary to emulate interference on the web server. We iteratively started 10 EC2 instances in batches (as described in [25]) and were able to successfully co-locate 2VMs after some trial and error. We found that the co-located instances had sequential `domid`s and were able pass messages among themselves using `xenstore` (write in one VM and read from another). We used this as verifying evidence that co-location was achieved. Our results, in themselves, are also secondary validation of co-location since we found noticeable impact of interference on WS performance. The co-located VMs on EC2 were hosted on a Xeon-2650 machine having 8(16) physical(logical) cores and 20MB L3.

Baseline Formation. To form baseline observations for both private testbed and EC2 we first configured their corresponding web servers to the no-interference optimal settings. These settings, with IC^2 disabled, emulate an interference-agnostic controller. We then used Faban to generate client requests for a 1-hour run. During the run we started our interference controller described above to generate periodic interferences. The application metrics for CloudSuite (response time and throughput) were collected at intervals of 5s. These metrics when plotted against time axis represent performance of one baseline run. In general, we found that interference had more performance impact in EC2 than in local testbed. For this experiment, we reconfigured the WS VM on the local testbed to match Amazon EC2's m1.large instances. To achieve noticeable impact, we had to use 4 threads of the interference benchmark on the local testbed compared to 2 in EC2. We found that with this utilization of the local server (6 of 12 hardware threads) the effects of interference in local and EC2 were of comparable magnitude.

Similar to baseline measurements, we also ran CloudSuite with IC^2 enabled. In both testbeds, we evaluate IC^2 under two scenarios: one where Apache is reconfigured with traditional `apachectl -k graceful` command (httpd-basic) and the other where our instrumented version of Apache is reconfigured online (httpd-online). We iteratively start 1-hour of baseline run followed by 1-hour of IC^2 with httpd-online, and 1-hour of httpd-basic. This was repeated 16 times for a total runtime of 48 hours (3×16). We restart the web server between each 1-hour run.

5.1 Results

5.1.1 Improvement in Response Time

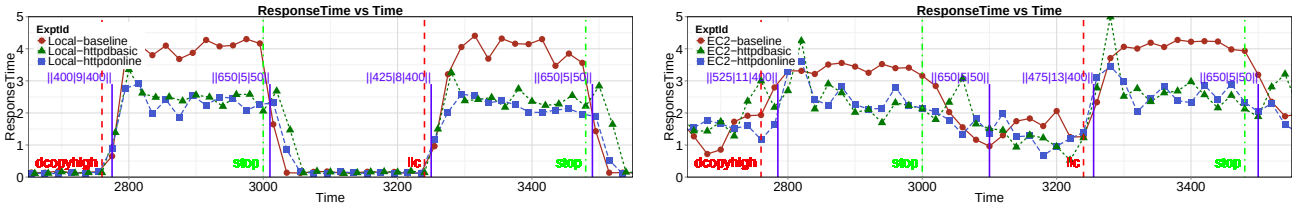
Fig. 9 shows the variation in Response time with Time in local testbed and in EC2 for a set of representative runs. In each plot, red vertical lines show the point on time axis when an emulated interference is started and green vertical lines show when interference is stopped. The blue vertical lines show the point when IC^2 reconfigured with httpd-online. New parameter settings at each reconfiguration point is annotated as the three tuple $|MXC|KAT|PHP|$. It can be seen that in general both httpd-online and httpd-basic are able to reduce response time during interference. In case of httpd-basic, there is a spike in response time following a reconfiguration, an indication that Apache is restarting all of its child processes. With httpd-online this spike is nearly eliminated, although, some overhead remains due to updation of `PhpMaxChildren`. Interference detection is faster in Local than EC2 since we use cache miss rate. Another interesting fact is that the effect of interference persists longer in EC2 even after emulated interference is stopped. This happens for two reasons, i) ambient interference in EC2, ii) max throughput in EC2 is lower than in the local testbed, hence queued requests persist for longer in EC2.

To quantify improvement in response time, we analyze response time during interference in two halves—a) From onset of interference (red line in Fig. 9) upto 60 seconds is considered first half. This is the period when interference detection and reconfiguration take place effectively showing overhead of IC^2, specially in case of httpd-basic. b) From 60s after interference to stopping of interference (green line in Fig. 9) is considered the second half. This is the steady state performance of IC^2 during interference.

We found that, across different interference types in EC2, httpd-basic degraded response time by 5-10% in the first half, but httpd-online improved response time by 3-10%. This proves that the online version of Apache is able to reconfigure faster. In local testbed, during the first half, httpd-basic showed improvement between 5 and 19% while httpd-online showed improvement between 20 and 25%. The measurements are better in Local testbed compared to EC2 since interference detection happens faster. In steady state or second half (60-240s from onset of interference), httpd-online showed improvements of 21-29% in EC2 and 32-40% in Local testbed (refer Fig. 10). The numbers for httpd-basic are 18-22% in EC2 and 34-40% in Local. The steady state performance of httpd-basic and httpd-online are comparable in local testbed, although httpd-online outperforms httpd-basic in EC2. Overall IC^2 showed higher improvement in response time in local testbed since it was able to compute δ_{MXC} and δ_{KAT} more precisely (no ambient interference as in EC2). We find that the response time improvements are significant considering the simplicity of our controller. It further establishes our point that, in a cloud deployment, an application configuration manager must be interference-aware. A summary of our results can be found in Table 4.

5.1.2 Detection Latency

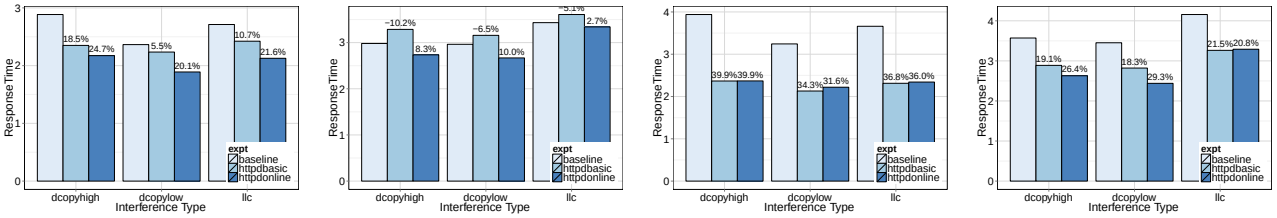
From the collected metrics we also measured how long it takes for IC^2 to detect interference in EC2 and in Local testbed. We define detection latency to be time from the starting or stopping of an emulated interference to the time when IC^2 reconfigures Apache server. In Fig. 9 these are the times between a red line and the next blue line (we call this



(a) On Private Testbed.

(b) On EC2 Testbed.

Figure 9: IC^2 improves response time of a web server during phases of interference. Red vertical bars show when an emulated interference is started and green vertical bars show when interference is stopped. The blue vertical bars show the point when IC2 reconfigured with httpd-online. New parameter setting at each reconfiguration point is annotated as the three tuple `MXC,KAT,PHP`. Baseline run implies IC^2 is disabled.



(a) Firsthalf RT Private Testbed

(b) Firsthalf RT EC2

(c) Secondhalf RT Private Testbed

(d) Secondhalf RT EC2

Figure 10: Response time with IC^2 for various interferences. Numbers represent percentage improvement from baseline RT.

Table 4: Summary of IC^2 Results. Response time numbers are %change from baseline runs across interference benchmarks. FH:=first half, SH:=second half, INTF:=interference, NI:=no-interference

	Response Time (%change)				Detection Latency	
	httpd-online		httpd-basic			
	FH	SH	FH	SH	INTF	NI
Local	20-25↓	32-40↓	6-19↓	34-40↓	15s	10s
EC2	3-10↓	21-29↓	5-10↑	18-22↓	20s	65s

interference detection latency) or time between a green line and the next blue line (no-interference detection latency). We found that in local testbed, median values for interference and no-interference detection latencies are 15s and 10s respectively. In comparison, IC^2 detects interference in EC2 with a median latency of 20s. Detection of no-interference in EC2 takes much longer—a median value of 65s. This happens since effect of interference persists much longer in EC2 as described in the previous section (Fig. 9(b)). Our future work includes finding ways to reduce detection latency even further in both testbeds.

5.1.3 *Classifier Accuracy*

To measure the accuracy of our classifier we apply it on the data collected from our experiments in Section 5. For each experiment type (httpd-online and httpd-basic), we create a test set comprising measurements collected by IC^2 in that experiment. Due to space constraints, we present only the first type here. We label the test data based on its timestamp and our knowledge of when an emulated interference is started and stopped. Data from the start(stop) of an interference upto 30s is labeled Transient, rest are labelled according to which interval it is (Interference or No-interference). Note that this labeling does not take into account ambient interference in EC2 and therefore may manifest as poorer precision, although the classifier works well in practice as seen in results from Section 5.1.1. We found the Transient class had significant overlap with both Interference and No-interference in the training data, as a result it had very low precision. But since IC^2 does not perform any reconfiguration in this state, the cost of misclassification is zero. We therefore ignore the results for Transient and focus primarily on interference detection.

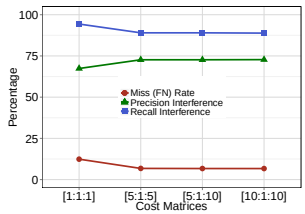


Figure 11: Accuracy of Interference Detection with varying cost matrices. The cost values $5:1:10$ are used in production.

During training phase, it was found that with default cost for misclassification, the decision tree had significant number of False Negatives (FN). This causes IC^2 to perform badly, e.g. using baseline parameter setting while in interference may have significant performance degradation and vice versa. We tried several combinations of cost matrices and selected the one with lowest miss rate. Fig. 5.1.3 shows the interference detection rate of our classifier with different cost matrices for httpd-online test data. Here a cost value of $\{a:b:c\}$

represent a 3×3 cost matrix, a is the cost of No-interference (NI) detected as Interference (I), b is the cost of Transient misclassified as any other class, c is the cost of (I) classified as (NI), and cost of correct prediction is 0. We define Miss Rate as (I classified as NI + NI classified as I)/Total Samples. It can be seen that, initially with default cost of $1:1:1$, miss rate was 12%, but with higher cost values miss rate reduced to 6%. We used the cost values of $5:1:10$ in our production runs based on the fact that response time penalty for misconfiguration in (I) is much higher than cost of misconfiguration in (NI). The largest percentage of FNs (98.7%) arise from NI being detected as I (with cost $10:1:10$). This happens since impact of interference persists longer in EC2 as seen in Fig. 9(b) (but our labeling does not account for this). This also manifested as lower precision of (I) and lower recall of (NI). In general, our interference detection achieved 89% recall and 73% precision.

5.1.4 Cost of IC^2

The cost of IC^2 can be defined in terms of two metrics—(a) Apache performance immediately after a reconfiguration, and (b) execution cost of IC^2. We already found in Section 5.1.1 that IC^2 with httpd-online improves response time during first half, both in EC2 and in private testbed. This indicates httpd-online is able to reduce cost of reconfiguration significantly (compared to httpd-basic, response time improved by upto 17%). Note that IC^2 is trained offline, therefore it does not have any runtime cost for building the classifier. Since the classifier has only 3 attributes, the tree has a simple structure and classification decision is made in the order of 10 comparisons. This is insignificant compared to our measurement period of 5s, which is also the frequency at which the classifier is invoked. Therefore, the execution cost of IC^2 is negligible.

5.2 Discussion

How generic is the knowledge base in IC^2?
We believe the knowledge base (KB) in Table 3 to be applicable to thread-pool based server architectures (e.g. Apche, Glassfish, WebSphere). In a small scale experiment with Glassfish, we found that it has sensitivity to thread-pool size (similar to MXC). Our KB, however, is not applicable to event-driven architectures (e.g. Nginx). We are currently conducting further experiments to understand relevant configuration parameters for such event-driven servers.

How expensive is it to generate the knowledge base?
The knowledge base in IC^2 can be created empirically by systematically varying important parameters as described in Sec. 3.2. It can be done in parallel with the load testing phase of web applications. Note that the KB does not include precise values of parameters, rather IC2 can figure out the parameter values depending on runtime conditions, including interference. Once created, it can be used for a given application and middleware distribution irrespective of deployment (assuming similar architectures, e.g. x64 or x86).

Can IC^2 handle other types of interference?
Network interference is another major problem that seriously affects performance of cloud applications. IC2 can also be useful in mitigating some of the effects of network interference through application reconfiguration. In our preliminary experiments, we simulated an environment where bandwidth available to the WS-VM became constrained (by upto 20%) due to a co-located VM using up a major share of the network. We observed, as the level of network interference increases (i.e. the available bandwidth to WS-VM reduces) the response time of the webserver sharply degrades. IC2 can improve response time by employing an *admission control* mechanism, which is equivalent to reconfiguring the `MaxClients` parameter in Apache to a lower value. We empirically verified that optimal MXC setting with network interference is lower than no-interference optima [16]. IC2 can be trained to use response time along with packets pruned from send-buffer as a trigger to detect such network interference.

6. RELATED WORK

The issue of interference in virtualized environments has been pointed out by several researchers [28, 8, 27, 17] and some efforts have been made for providing better resource isolation [17, 9, 27]. However, due to the intrusive nature of these changes and the impact on performance, today's production virtualized environments still do not provide isolation for cache usage and memory bandwidth, which are relevant to the results that we presented here. Existing solutions primarily try to reduce the probability of occurrence of interference by using better scheduling or consolidation techniques [7, 4, 20, 18]. All of these are beyond the control of an end-user. In this paper, we look at the problem of interference from a customer's perspective and try to suggest simple solutions to mitigate it in a non-intrusive manner; by tuning application level parameters in enterprise middleware.

In one of the early works on tuning of Apache servers [13], Liu *et al.* showed that `MaxClients` exhibits a concave upward behavior on response time—an observation that led to the design of a tuning agent using hill-climbing algorithms. Our results also highlight this behavior of `MaxClients`, but we find that the gradients of the curves change frequently due to interference and dependence on other parameters. In another work [5], Diao *et al.* presented a multi-input multi-output (MIMO) feedback control for optimizing web server performance, however, their controller only considers workload intensity—one of the many challenges we present in our paper. A more recent work [35] looks into automatic generation of configuration files in multi-tier web servers. All of these operate in non-virtualized environments.

The question of how to performance tune applications that are executing in virtualized environment has been addressed by several prior works. Such approaches were applied to configuration of software systems like Apache server [32], application server [33], database server [31] and online transaction services [3, 35]. Some of these consider coordinated tuning of resources allocated to VMs and associated application configurations [31, 2], an approach which requires changes to the hypervisor.

We believe we are the first show the challenges of application configuration in the presence of interference and present mitigation actions. Our evaluations points out the non-linear, discontinuous nature of the performance space and the difficult-to-model dependencies between configuration parameters and therefore it is unlikely that an existing configuration solution will work for all environments in a dynamic cloud environment.

7. CONCLUSION AND FUTURE WORK

In this paper, we investigated one of the major sources of performance variability in clouds, namely, interference and presented ways in which an end-customer can mitigate its ill-effects. More specifically, we evaluated the frequency and impact of interference in public clouds like Amazon EC2. Our experiments suggest that performance anomaly due to interference is a reality. We designed and evaluated an interference-aware application configuration manager (IC^2), which is able to detect interference and find suitable parameter values during these phases. Our solution reduced application response time by upto 29% in EC2 and 40% in a private cloud testbed during periods of cache interference.

Our future work is geared towards achieving two objectives: i) Reducing detection latency for reconfiguration so that IC^2 can deal with smaller durations of interferences gracefully, and ii) Dealing with various other kinds of interferences.

8. REFERENCES

[1] Basic Linear Algebra Subprograms. http://www.netlib.org/blas.

[2] X. Bu, J. Rao, and C.-Z. Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *Parallel and Distributed Systems, IEEE Transactions on*, 24(4):681–690, 2013.

[3] I.-H. Chung and J. K. Hollingsworth. Automated cluster-based web service performance tuning. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 36–44. IEEE, 2004.

[4] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[5] 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 *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 219–234.

[6] EPFL. CloudSuite. `http://parsa.epfl.ch/cloudsuite/cloudsuite.html`.

[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 *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 126–136, New York, NY, USA, 2007. ACM.

[8] A. Gulati, A. Merchant, and P. J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–7. USENIX Association, 2010.

[9] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 342–362. Springer-Verlag New York, Inc., 2006.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[11] R. J. F. Inc.). More details on today's outage, 2011. `https://www.facebook.com/note.php?note\_id=431441338919\&id=9445547199`.

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

[13] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh. Online response time optimization of apache web server. In *Proceedings of the 11th international conference on Quality of service*, IWQoS'03, pages 461–478, Berlin, Heidelberg, 2003. Springer-Verlag.

[14] R. P. Mahowald and M. Rounds. It buyer market guide: Cloud services. In *IDCReport*, July, 2013.

[15] A. K. Maji. Httpd with Online Reconfiguration, 2014. `https://github.com/amaji/httpd-online-2.4.3.git`.

[16] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma. IC2: Interference-aware Application Configuration in Clouds, May 2014. Technical Report, School of Electrical and Computer Engineering, Purdue University, `http://docs.lib.purdue.edu/ecetr/`.

[17] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–18. USENIX Association, 2007.

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

[19] Netcraft. May 2014 Web Server Survey, 2014. `http://news.netcraft.com/archives/2014/05/07/may-2014-web-server-survey.html`.

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

[21] Olio. The Workload. `http://incubator.apache.org/olio/the-workload.html`.

[22] OProfile. OProfile. `http://oprofile.sourceforge.net/about/`.

[23] Oracle. Jdk-6558100 : Cms crash following parallel work queue overflow, 2011. `http://bugs.sun.com/view_bug.do?bug_id=6558100`.

[24] K. T. (PCWorld). Thanks, Amazon: The Cloud Crash Reveals Your Importance, 2011. `http://www.pcworld.com/article/226033/thanks\_amazon\_for\_making\_possible\_much\_of\_the\_internet.html`.

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

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

[27] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: performance isolation for cloud datacenter networks. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud)*, pages 1–6. USENIX Association, 2010.

[28] 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 *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

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

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

[31] L. Wang, J. Xu, and M. Zhao. Application-aware cross-layer virtual machine resource management. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 13–22, New York, NY, USA, 2012. ACM.

[32] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, volume 6, pages 1–14, 2004.

[33] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296. ACM, 2004.

[34] C.-Z. Xu, J. Rao, and X. Bu. Url: A unified reinforcement learning approach for autonomic cloud management. *Journal on Parallel and Distributed Computing (JPDC)*, 72(2):95–105, Feb. 2012.

[35] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 219–229, New York, NY, USA, 2007. ACM.