Proctor – Detecting and Investigating Performance Interference in Shared Datacenters

Ram Srivatsa Kannan, Animesh Jain, Michael A. Laurenzano, Lingjia Tang and Jason Mars
Department of Computer Science and Engineering
University of Michigan, Ann Arbor
Email: {ramsri,anijain,mlaurenz,lingjia,profmars}@umich.edu

Abstract—**Cloud-scale datacenter management systems utilize virtualization to provide performance isolation while maximizing the utilization of the underlying hardware infrastructure. However, virtualization does not provide complete performance isolation as Virtual Machines (VMs) still compete for non-reservable shared resources (like caches, network, I/O bandwidth etc.) This becomes highly challenging to address in datacenter environments housing tens of thousands of VMs, causing degradation in application performance. Addressing this problem for production datacenters requires a non-intrusive scalable solution that 1) detects performance intrusion and 2) investigates both the intrusive VMs causing interference, as well as the resource(s) for which the VMs are competing for.**

To address this problem, this paper introduces Proctor, a real time, lightweight and scalable analytics fabric that detects performance intrusive VMs and identifies its root causes from among the arbitrary VMs running in shared datacenters across 4 key hardware resources – network, I/O, cache, and CPU. Proctor is based on a robust statistical approach that requires no special profiling phases, standing in stark contrast to a wide body of prior work that assumes pre-acquisition of application level information prior to its execution.

By detecting performance degradation and identifying the root cause VMs and their metrics, Proctor can be utilized to dramatically improve the performance outcomes of applications executing in large-scale datacenters. From our experiments, we are able to show that when we deploy Proctor in a datacenter housing a mix of I/O, network, compute and cache-sensitive applications, it is able to effectively pinpoint performance intrusive VMs. Further, we observe that when Proctor is applied with migration, the application-level Quality-of-Service improves by an average of 2.2× as compared to systems which are unable to detect, identify and pinpoint performance intrusion and their root causes.

I. Introduction

Enterprise datacenters like VMWare, Microsoft, and Amazon often house thousands of servers to service large-scale cloud applications across the globe. Cloud computing is becoming more common every day among a diverse set of users executing high-performance computing applications [5], user-facing web service applications [12], machine learning applications [17], [18] etc. Hence, improving the utilization of cloud platforms is of critical importance in terms of improving cost and reducing the footprint of the datacenters [2], [8]–[10], [19], [26]. This has motivated users towards using cloud platforms for executing a varied class of applications ranging from

Over past few years, datacenter operators have switched to *virtualization*, a technique that encapsulates and abstracts applications from the physical hardware by creating Virtual Machines (VM), that assists sharing of physical hardware by scheduling multiple VMs on the same physical machine. State-of-the-art VM monitors, also known as hypervisors, like Xen, Hyper-V and ESXi [41], [45] reserve fragments of the physical server resources (like CPU core, DRAM storage etc) for each application separately in a virtualized environment. This abstraction leads to better hardware utilization, as multiple VMs can now be easily scheduled on the same physical machine.

However, virtualization does not provide complete performance isolation as VMs still compete for non-reservable shared resources (like caches, network, I/O bandwidth etc.), resulting in performance interference between the VMs, which can have significant and unpredictable effects on the application performance. This unpredictable performance is particularly problematic for user-facing applications that have strict Quality of Service (QoS) requirements, forcing the datecenter operators to disable co-location, reducing datacenter utilization. Therefore, data center operators need to achieve the best of both worlds - satisfy strict QoS requirements while also keeping server utilization high.

A suitable solution to mitigate the interference problem so that it can take corrective measures later to meet QoS requirement while achieving good server utilization, needs to perform two major tasks at *runtime – Detection* and *Investigation*. First, when a performance intrusive VM is colocated with an application having strict QoS requirement and negatively impacts its performance, the technique should be able to **detect** this performance degradation. Second, once this performance intrusion is detected, it is necessary to **investigate** the source of this contention (both the performance-intrusive VM and the contended shared resource) to undertake useful remediation.

Due to the increasing usage of cloud services, along with new server paradigms (e.g., colocating storage with compute in HyperConvergence [44]), there are three major challenges that arise while tackling the problem of mitigating interference.

1) **Absence of Apriori Application Profile.** There are new applications getting executed in the cloud infrastructure, for which the datacenter operators do not have any prior performance profile. This makes the Detection task challenging as there is no baseline performance to compare against to detect a change in the QoS metric.

2) **Multiple Sources of Contention.** Different applications put stress on different susbsystems of the stack (one application might only stress network while other application might have large number of IO requests stressing IO system stack), requiring Investigation task to handle multiple sources of contention.
3) **Low Runtime Overhead.** The technique needs to perform both these tasks with very low performance overhead in order to quickly adapt to the application runtime environment.

Prior relevant body of work solves these challenges partially. Bubble-up [27] and Cuanta [13] require a priori knowledge of application behavior restricting its applicability in the Detection task. While Application Slowdown Model (ASM) [39], Geiko [37] and Seawall [35] detect performance degradation, the are unable to identify the source of contention, restricting its applicability in the Investigation task. Finally, a third category of work, Deepdive [31] and *CPI*^{2} [48], have very high overhead in performing these two tasks, making it difficult to deploy them at runtime systems.

To tackle these challenges, we present *Proctor*, a runtime system that continuously monitors, automatically detects and investigates a wide range of performance issues directly affecting the Quality of Service of VMs running in a cloud scale datacenter, with high accuracy and low performance overhead. For Detection, Proctor employs a Performance Degradation Detector (PDD), that continuously monitors the performance metric of the executing VMs, looking for abrupt changes in the QoS. PDD uses state-of-the-art noise removal technique (median filtering algorithm) and step detection to detect a performance anomaly, as opposed to previous work that requires a priori knowledge. For Investigation, Proctor employs Performance Degradation Investigator (PDI), that identifies the source of contention for a performance anomaly at runtime using online statistical correlation analysis. The challenge here lies in performing investigation quickly as this process is laborious and requires querying a database consisting of large amounts of VM monitoring data. To tackle this challenge, PDD uses a robust sub-sampling technique that reduces the amount of the data that needs to be queried while *accurately* detecting the source of contention.

The specific contributions of this paper are as follows:

1) **Performance Degradation Detector** – Accurately detects sudden performance anomaly using HW performance counters, without any apriori application profile.
2) **Performance Degradation Investigator** – Statistical correlation analysis and robust sub-sampling technique that greatly reduces the footprint of the telemetry data that needs to be queried, to quickly and accurately identify the exact source of contention.
3) **Runtime System** – Proctor, a runtime system, continuously monitors, automatically detects and identifies the sources of contention, with low overhead and high accuracy. We envision Proctor as a guide, that can direct the corrective measures for mitigating interference.

We perform a thorough evaluation of our platform on real systems across a wide range of applications and commonly contended shared resources, demonstrating its effectiveness in diagnosing performance issues at runtime, improving the performance of the applications running in datacenter by up to 2.2×.

II. Background and Motivation

In this section, we provide the background for the performance interference for different sources of contention, followed by the limitations of the prior work in solving the problem of mitigating interference.

A. Sources of Contention

Although virtualization reserves fragments of machine resources for each application individually, the VMs can still experience performance interference when multiple VMs are colocated on the same physical machine. This happens because there are a number of non-reservable resources that can be shared among VMs, that can have significant and unpredictable effect on the VM performance. In datacenters, there are mainly four such shared resources - *I/O* [14], [40], *CPU core* [29], [38], [49], *Network* [36] and *Last Level Cache* [21], [22], [27], [30], [39].

As an example, I/O contention can occur when guest operating system within each VM is oblivious to the virtual nature of underlying disk and the existence of neighboring VMs on the same machine. Under such situations, a single badly behaved application that continuously issues frequent I/O requests to a disk array can disrupt the latency/throughput of every other application running over that array, negatively impacting the performance of other VMs. Similarly, such performance intrusive behavior can happen at other hardware resources like CPU core, Network and Last Level Cache.

B. Limitations of Prior Work

There exists several prior approaches that are specifically designed to mitigate the effects of contention when multiple VMs are consolidated in a shared datacenter. However, these approaches have some limitations that restrict their deployability in a commercial datacenter. We have broadly classified them under the following three categories based on their limitations towards solving the Detection and Investigation problems.

1) **Require A Priori Application Profile.** Prior approaches like Bubble-Up and Cuanta [13], [27] have been shown to be effective at generating a precise estimation of performance degradation at co-located execution scenarios. However, these techniques require a priori knowledge of application behavior restricting their deployability in datacenters that encounter unknown applications on a regular basis (for eg., private datacenters and public clouds). Additionally, these techniques are incapable of investigating the root cause of performance

intrusion. Therefore, this category is unsuitable to perform Detection and Investigation tasks in datacenters that encounter unknown applications.

2) **Incapable of Investigating Root Cause.** Second category of prior approaches [14], [39], that do not require a priori knowledge, focus on investigating a particular source of contention, unable to detect and mitigate the performance interference caused by other shared resources. In addition, the overhead incurred by these techniques in detecting performance degradation is high because their methodology perturbs the execution of VMs periodically for brief periods of time in order to profile application execution. Therefore, this class of prior work is also unsuitable because of its high overhead in performing Detection task and their disability to execute the Investigation task.

3) **Performs VM Migration/Cloning.** A third class of approaches, identifies performance intrusion as well as its root causes. However, these techniques perform frequent VM migration and cloning, resulting in many drawbacks. First, copying huge data across machines is time consuming and introduces additional contention on the computing resources. Second, the overhead with respect to the number of additional servers required to perform these techniques is very high. Therefore, this category of prior work is also not suitable in shared datacenters as they incur high overheads while executing the Detection and Investigation tasks.

III. Overview of the Proposed Approach

To this end, we present Proctor, a runtime system that utilizes a two step methodology to solve the Detection and Investigation tasks respectively. In this section, we provide a high level overview of our technique along with the challenges in designing Proctor components.

A. Goals and Challenges

Performing Detection. Proctor utilizes *Performance Degradation Detector (PDD)* for this purpose. In contrast to prior approaches which affect the execution of application by utilizing synthetic benchmarks like smashbench, PDD is an extremely low overhead continuous monitoring infrastructure that observes individual VM QoS metric to detect drastic variation in the numerical range of the QoS metrics. This change would be an indication of an event that signifies performance degradation of the application. To detect drastic variation in numerical range of metrics, we employ step detection – a signal processing technique that is utilized to find abrupt changes in time series signals [32].

Challenges and Approach – However, the time series data obtained from system software tools and performance counters is highly corrupted due to noise. The most straightforward solution for such problems is to perform curve smoothing. However, the most commonly used curve smoothing techniques, like exponential moving average and Kalman filter [43], are not effective in highlighting drastic changes in the time series data. This is because they project drastic changes in QoS measurements as a slow cumulatively occurring event, making it hard to detect the abrupt changes. Hence, we used a technique called *median filtering* designed specifically to cater to the step detection problem.

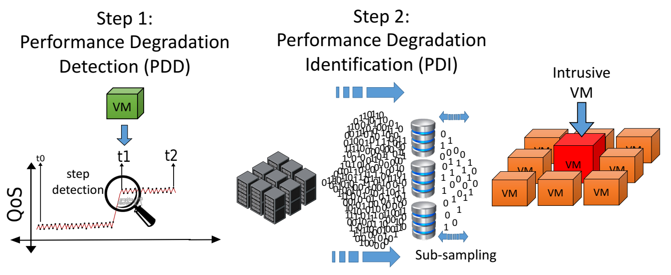


Fig. 1: Proctor System Architecture - a two-step process performing Detection and Investigation to identify the root cause of performance interference

Performing Investigation. Once, PDD establishes the existence of performance degradation, we utilize the *Performance Degradation Investigator (PDI)* to pinpoint the exact source of contention (both VM and the shared resource the applications are competing for). PDI uses correlation analysis for this purpose, finding correlation between the hardware counter metrics of all the co-running VMs and the primary QoS metric of the affected VM (as detected by PDD). High value of correlation co-efficient for a particular metric provides sufficient evidence that the co-running VM and the resource corresponding to that metric is the root cause of performance degradation.

Challenges and Approach – However, performing correlation analysis on large amounts of HW performance counter data, which is collected at a second level granularity, is computationally intensive, resulting in high performance overhead. To tackle this problem, we sub-sample the performance counter data, reducing the amount of data that is to be utilized to find the source of contention. A random sub-sampling method can be utilized for this purpose. However, it becomes crucial that the obtained sample should be a good representation of the population from which it is drawn, as biased samples can lead to inaccuracy in performing Investigation. To address this challenge, we validate each sample by utilizing hypothesis testing techniques. As our time series measurements do not follow the guassian curve, we use a non-parametric statistical hypothesis testing technique called χ^2 test to ensure that the sub-sampled data is a good representation of the original performance counter data [48].

IV. Proctor Architecture

Proctor is a dynamic runtime system that automatically detects performance intrusive VMs in the datacenters, their victims and the shared resource that is causing contention, with high accuracy and low overhead. In order to achieve this,

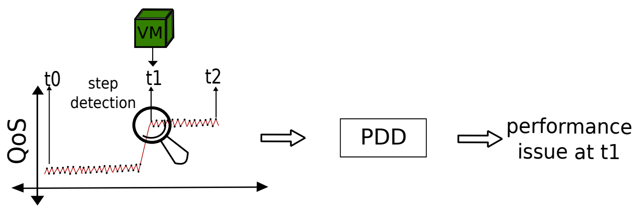


Fig. 2: PDD detects abrupt performance variations in the application telemetry data

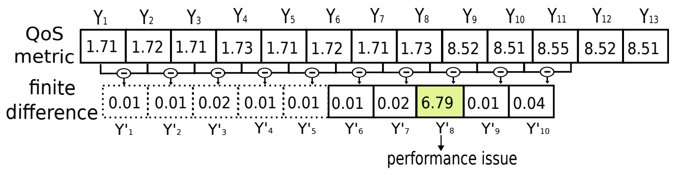
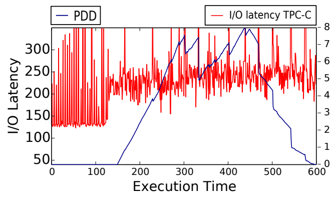
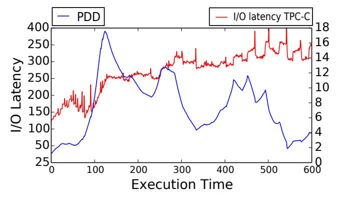


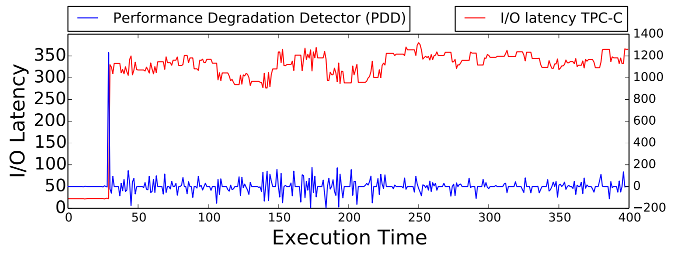
Fig. 3: PDD Step Detection using Finite Difference Method



(a) no noise removal



(b) exponential moving average



(c) median filtering

Fig. 4: Comparison of detection accuracies (a) without noise removal, (b) with exponential moving average and (c) with median filtering for the application TPC-C. Median filtering algorithm detects abrupt changes in performance

Proctor utilizes a two step approach as shown in Figure 1. The first step, PDD, detects performance degradation caused due to performance intrusive VMs. The second step PDI, pinpoints the root cause by identifying the exact VM that is responsible for the performance intrusion and the corresponding metric for which there is contention. This section elaborates in detail the key components present in Proctor's design.

A. *Performance Degradation Detector*

Proctor utilizes (PDD) that operates in parallel with applications, continuously monitoring and looking for performance anomalies in the dataceters at runtime. It utilizes time series measurements of the primary QoS metric of each application executing inside a VM to detect drastic variation in the numerical range of metrics. This drastic variation acts as an indication of an event that the performance of the application has degraded significantly.

PDD employs a signal processing technique called *step detection* to detect these abrupt changes in the application performance [32], [34]. However, time series performance data of an application has high amount of noise, causing many false alarms if step detection is applied naively. We use *Median filtering* algorithm [7] to reduce the noise in the telemetry data, making PDD accurate in detecting performance anomalies. In the next two subsections, we will elaborate on the step detection and median filtering techniques.

1) Step Detection: Step detection is a process of finding abrupt changes in a time series signal [32], [34]. Using the time series measurements of the primary QoS metrics, we try to identify the exact timestamp at which abrupt changes occur in the numerical quantity of primary QoS metric. An abrupt change is statistically defined as a point in time where the statistical properties before and after this time point differ significantly. This is clearly illustrated by Figure 2 where we can see a sharp increase in the QoS metric at time t1. The role of PDD here is to detect such abrupt changes at runtime and identify the exact timestamp at which such abrupt changes occur. We utilize *finite difference method* for this purpose.

The fundamental hypothesis of finite difference method towards identifying abrupt changes is based on the fact that the absolute difference between subsequent time series measurements is very high at the exact point where the abrupt changes occur. This can be utilized to highlight the timestamp at which these abrupt changes occur.

Mathematically, finite difference of a time series signal is the rate of change in the individual elements in the time series. We implement finite difference method by performing pair wise difference of subsequent elements present in the time series using the following formula :-

$$Y' = \frac{Y_{j+1} - Y_j}{2\Delta T} \qquad Y'_j = Y_j \ (for\ 1 < j < n-1)$$

where Y_j is the j^{th} points present in the time series, n being the number of points, ΔT being the difference between the X values of adjacent data points (difference in the number of timestamps for time series values). The result highlights the drastic change by showcasing a high value for Y'. This is clearly illustrated by Figure 3 where we can see a sharp increase in the QoS metric at time t1 at the point Y_9. Its corresponding finite differential value is very high at point Y'_9, which is utilized to indicate performance degradation at that timestamp t1.

2) Noise Reduction: Naively applying step detection leads to large number of false positives because of the noise in the time series measurements of QoS metric. For example, we directly apply the step detection algorithm for TPC-C benchmark and show the detected performance anomalies in Figure 4a. The figure shows that there are large number of false alarms.

In order to eliminate the noise present in the raw time series measurements, we tried to utilize the state-of-the-art curve smoothing techniques like exponential moving average and kalman filter [25]. However, these techniques still show significantly high number of false positives. This is because these techniques end up smoothing out drastic changes in time series measurements, projecting them as a slow and cumulatively occurring event as shown in Figure 4b, failing to detect the drastic performance degradation.

To tackle this problem, we use *median filtering* for noise reduction as this technique preserves drastic changes. Our implementation of median filter consists of a moving window that selectively discard elements that are significantly higher than the median within that window. This preserves drastic changes while also removing noise from the time series measurement. Finally, Figure 4c shows the effectiveness of applying median filtering for noise reduction, reducing number of false alarms and making PDD highly accurate.

3) Obtaining QoS Measurements: The presence of virtualization in datacenter infrastructures introduces challenges towards obtaining application specific QoS metrics. Applications often run as performance black-boxes and adaptive services must infer application performance from low-level information or rely on system-specific ad hoc methods. Although this is not a challenge for CPU intensive batch applications and I/O intensive applications as their respective QoS metrics can be obtained through performance counters and system software tools, a class of user facing latency critical applications that run as performance black-boxes, provide very little information about their current performance and no information about their performance goals (eg. 99th percentile tail latency). The primary goal in such situations is to offload the responsibility of providing time series measurements corresponding to the QoS metrics of an application to the user. For this purpose we utilize the the Application Heartbeats framework [20] which provides a simple, standardized way for applications to report their performance/goals to external observers. These are enabled through API calls consisting of a few functions that can be called from applications or through system software. This is being utilized to track the progress of any executing application which is fed into our proposed PDD for identifying performance intrusion during runtime.

B. Performance Degradation Investigator

Once PDD establishes the existence of performance degradation, *Performance Degradation Investigator* (PDI) is invoked for further analysis which pinpoints performance intrusive VMs and the major server resource that is causing the performance degradation.

1) Correlation Based Root Cause Identification: PDI identifies performance intrusive VMs and the major server resource causing contention by utilizing a correlation based root cause identification technique. The primary objective of correlation based root cause identification is to highlight the root cause VM and the metrics corresponding to it that correlate highly with the primary QoS metric of the affected VM. In order to obtain that, PDI utilizes the time series measurements from each low level metric corresponding to the co-running VM and tries to correlate them with the time series measurements of the affected VM's primary QoS metric. The metrics having the highest value of correlation coefficient are the most highly likely indicators of resource contention and its corresponding VMs are the most likely culprits for creating performance intrusion. The list of metrics that we try to correlate is enumerated in Table I. Our implementation of correlation tries to obtain Pearson's correlation coefficient [6]. However, performing correlation analysis on the complete telemetry data causes high performance overhead. Therefore, we sub-sample the complete dataset and reduce the time to find the source of contention.

Name	**Description**
load	Input load of application
CPU util	CPU utilization of app
page-faults	Page faults per sec of app
context-switches	Context switches per sec of app
n/w throughput	Total bytes sent and received by network
cache-misses	Total cache misses (L1,L2 and LLC)
I/O requests	Total I/O requests (read + write)
branch-misses	No. of branch mispredictions of app

TABLE I: List of metrics utilized for performing correlation with the primary QoS metric to identify source of contention

2) Real Time Sub-sampling: One of the key challenges faced by Proctor while realizing a real time solution is the large amount of telemetry data that needs to be queried, resulting in high performance overhead. Hence, instead of performing correlation analysis on full telemetry data, we utilize a sub-sampling technique where a sample from a large data is utilized as input to PDI.

The key objective to be satisfied while realizing a sub-sampling technique is that the statistical characteristics of the sample should be consistent with that of the population. For example, measurements obtained from system software tools are bound to contain extreme values (spikes) at a very low frequency. The sub-sample that we collect should include these events as well. To ensure that, we perform a hypothesis testing to check whether the random sample that we select is representative enough of the population. If not, our hypothesis testing techniques repeats the process by randomly selecting a sample till it is representative enough of the population.

Most widely used hypothesis testing techniques assume population to be normally distributed. However, based on our experiments we have observed that measurements that come from system software tools and performance counters are highly deviated from being normally distributed. Therefore, widely used parametric hypothesis testing techniques like t-test and F-test are not suitable for our purpose.

Hence, we use non-parametric hypothesis testing approaches that are capable of testing samples irrespective of their nature (being normally distributed). Unlike parametric statistics which primarily utilize mean and variance for this purpose, non-parametric statistics make no such assumptions

Processor	Microarchitecture	Kernel	Hypervisor
Intel Xeon E5-2630 @2.4 GHz	Sandy Bridge-EP	3.8.0	KVM-QEMU v2.0
Intel Xeon E3-1420 @3.7 GHz	Haswell	3.8.0	KVM-QEMU v2.0

TABLE II: Experimental platform where Proctor is evaluated

	Application	Description	Benchmark Suite	QoS Metric
CPU / LLC	lbm	Fluid Dynamics	SPEC CPU2006	IPC
	libquantum	Quantum Computing		
	omnetpp	Discrete Event Simulation		
	sphinx3	speech recognition		
CPU / LLC	Naive Bayes	Big data classification	Big Data Bench	IPC
	Sort	Sort words from text		
	Grep	Search words from text		
	Word Count	Count words from text		
	Kmeans	Processing facebook network		
I/O	YCSB	Querying from Yahoo dataset	OLTP bench	I/O latency and throughput
	TPC-C	Querying from retail database		I/O latency
	TPC-H	Querying from business database		I/O latency
	Twitter	Querying from tweets		I/O latency and throughput
Network	Redis	Key value store	Redis	Tail Latency
	netperf	Network packet generator	netperf	network throughput

TABLE III: Benchmarks which have been used to evaluate Proctor and its descriptions

	Work Load ID	App 1 - Main app	App 2 - Colo app	App 3 - Colo app	App 4 - Colo app	App 5 - problematic app
Network	WL1	Redis	Search	lbm	Sort	netperf
Disk I/O	WL2	Twitter	lbm	Redis	Sort	YCSB
	WL3	TPC - C	libquantum	Redis	Grep	Random I/O
	WL4	YCSB	sphinx3	Redis	Word Count	TPC - H
	WL5	TPC - H	lbm	Redis	K-Means	YCSB
CPU	WL6	Naive Bayes	libquantum	Redis	lbm	Page Rank
	WL7	Grep	TPC-C	Redis	sphinx3	Sort
	WL8	lbm	TPC-H	Redis	Sort	libquantum
LLC	WL9	omnetpp	TPC-H	Redis	Word Count	lbm
	WL10	libquantum	Random I/O	Redis	Grep	povray
	WL11	Redis	povray	Redis	povray	libquantum

TABLE IV: Workload scenarios that have been created from the benchmarks to evaluate Proctor

on the probability distributions of the variables being assessed. Therefore, we utilize Pearson's Chi-Squared test for testing whether a sample is representative of a population [47].

Chi-square χ^2 test is a statistical test used to examine differences within categorical variables [47]. For time series data, we have taxonomized categories as numerical ranges within which measurements from system software tools and performance counters can fall into. In other words, we segregate the population data into different categories where each category refers to a specific range of numerical quantities. Subsequently, we classify the sample data also into the same categories as the population. We now obtain the frequency of elements present in each category for both the sample and population data. For the sample data to be acceptable, the frequency of elements of the sample data in each category should be close to the frequency of elements of the population data in the same category. Chi squared test, compares the frequency of elements of sample and population data in every category to determine the sample's acceptability

Input. Frequencies of population measurements and sample measurements lying in each range.

Output. Accept/Reject sample to be representative of a population.

Methodology. We undertake the following steps to perform Chi-square χ^2 test.

1) We identify the frequency of entities that belong to every range for the sample distribution.
2) To compare the frequency per range of the sample and population distribution, we adopt the following methodology.
   Null Hypothesis H_0: Sample and Population distributions are similar
   Hypothesis Test:
$$\chi^2 = \frac{(Population - Sample)^2}{Sample}$$
3) We assess the significance level based on the size of the sample to accept/reject the null hypothesis. Hence, if the null hypothesis is rejected we repeat the same test with a different sample.

V. Evaluation

A. Methodology

Infrastructure. We evaluate Proctor on two commodity multicore processors summarized in Table II. We use system software tools `iostat` and `netstat` to obtain network and disk specific performance metrics and linux `perf` tool to measure HW counters. Performance telemetry is collected at a second level granularity using HW counters.

Applications. Table III enumerates the applications, their description, input, application domain and the respective suite from which they is obtained. We evaluate Proctor on *workloads*, where each workload is a mix of 5 applications. We design these workloads in a careful manner to study different types of resource contention. 4 out of 5 applications in a workload are chosen in a manner that they put stress on the four shared resources - I/O, network, CPU core and LLC. Once these four applications are executing, arrival of fifth application now causes contention for the resource it uses heavily. Table IV illustrates the workload mixes that we have considered in our evaluation. Workloads are executed for a period of one hour where each application is introduced after a period of 12 mins.

B. Proctor Accuracy

We first evaluate end to end accuracy of Proctor in detecting and investigating the source of contention. In this experiment, we execute all the workloads and check whether Proctor is

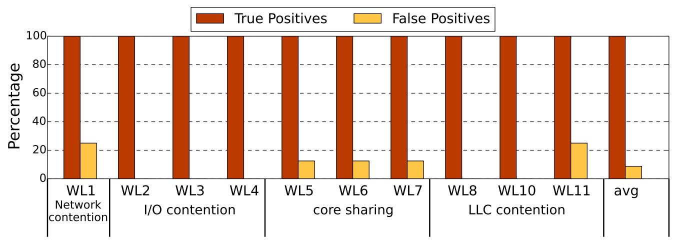


Fig. 5: Percentage of true and false positives while utilizing Proctor to detect performance issue and identify its root cause.

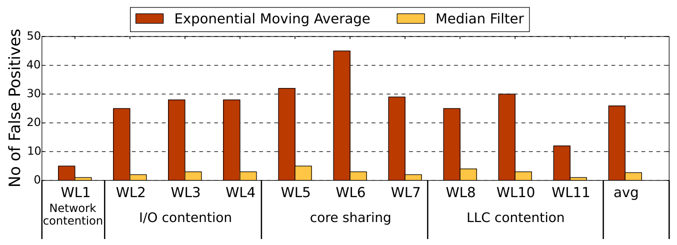


Fig. 6: Number of falsely identified performance degradation scenarios when exponential moving average/median filtering is utilized to remove noise before step detection

able to detect and identify the source of contention correctly. The findings of this experiment are presented in Figure 5, showing the true positive and the false positive rates across our workloads.True positives are the situations during which Proctor identifies performance intrusion when it exists. False positives are the situations during which Proctor identifies performance intrusion when there aren't any. In this graph, false positives represent the percentage of falsely identified metrics compared to the total number of metrics present.

We observe that Proctor detects the interference and investigates its root cause for all the workloads, whenever a performance intrusive VM is introduced into the system, as shown by 100% true positives. In additional, Proctor shows low false positive rate with an average of 8% across our workloads. This experiment show that Proctor is *accurate* in detecting and investigating the source of a performance anomaly, and is fully capable of guiding the remediation techniques for mitigating the performance interference. We now show evaluate the two components of Proctor in more detail.

C. Detection of Performance Interference

In this section, we evaluate the accuracy and performance of Proctor Performance Degradation Detector (PDD).

Accuracy. One of the main reasons for Proctor's accuracy is its robustness in performing the Detection task by PDD. The median filtering technique is highly effective in minimizing the false positives in detecting performance intrusion. Here, we compare the false positive rate for median filtering against state-of-the-art exponential moving average curve smoothing technique. In this experiment, we measure the false positives for both the techniques just for detecting the performance anomaly across all our workloads. The findings of this experiment are presented in Figure 6, showing the number of false positives for both the techniques.

The figure shows that the average number of false positives is around 10× lesser for median filtering as compared to exponential moving average. This is because exponential moving averages are highly affected by extreme values, as described in Section IV-A2, misinterpreting such noisy events as the performance degradation events. However, median filtering discards such extreme values, thereby reducing the error rate.

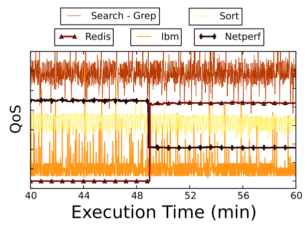
Performance. The computational time required for PDD is extremely negligible. The functionalities can be broken down into performing two tasks :- 1) a single subtraction per second per VM for performing step detection and 2) sorting and discarding outliers once in every 30 seconds per VM for performing median filtering. Therefore, PDD has minimal performance impact on VM performance.

D. Investigating the Performance Degradation

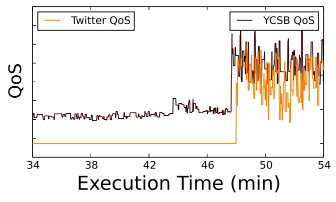
In this section, we evaluate the efficiency of Proctor's Performance Degradation Investigator, in pinpointing the root cause of performance degradation, towards identifying both the VM causing the performance degradation (referred to as contentious VM) and the shared resource for which the applications are competing. In this experiment, we execute each workload with the Proctor runtime system, enabling PDI to investigate performance anomalies. Here, we first show how the QoS metric of VMs affected on the arrival of a contentious VM. The QoS metric of the contentious VM would correlate with the QoS metric of the affected VM. Second, we enumerate the correlation coefficients obtained from correlating the HW performance counter measurements of the contentious VM and the QoS metric of the affected VM. Due to lack of space, we show the results for only 4 workloads, covering the four shared resources most commonly contended in datacenters - Network, I/O, CPU and Last level cache (LLC). The findings of this experiment are presented in 7. We show all the five applications only for workload WL1, but show only the contentious and affected VMs for the rest of the workloads for clarity. We now present the evaluation for each source of contention in detail.

Network Contention. We use setup present in WL1 to study network contention, where contentious VM is executing netperf and the affected VM is executing the application redis. Therefore, we expect a high correlation between the QoS metric of VMs executing redis and netperf. Figure 7a illustrates this correlation, showing the QoS metrics for all the five applications in the workload. We observe that the QoS lines represented by Redis and netperf are highly correlated having a correlation coefficient of 0.97, while the correlation coefficient of the QoS metric of redis with other the QoS metric of the other CPU bound applications is very low.

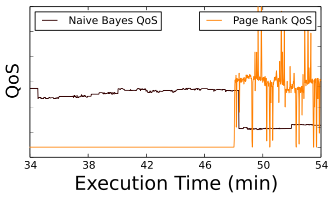
Further, Figure 8a shows the correlation coefficients obtained by correlating the QoS metric of redis, the affected VM with HW performance counter measurements collected for the contentious VM netperf. Since netperf puts significant stress



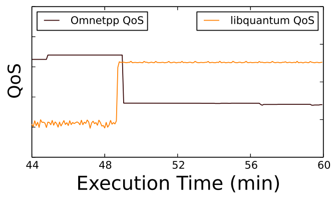
(a) WL1 – Root cause corr 0.97, others corr 0.13



(b) WL2 – Root cause corr 0.87

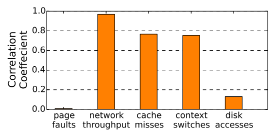


(c) WL6 – Root cause corr 0.83

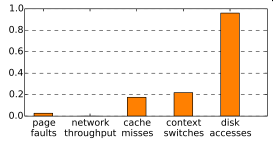


(d) WL9 – Root cause corr 0.93

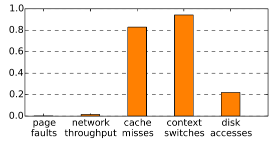
Fig. 7: Correlation between primary QoS of affected VM and other co-running VMs



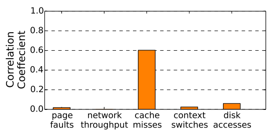
(a) WL1



(b) WL2



(c) WL6



(d) WL9

Fig. 8: Root cause metrics identified by Proctor.

on the network, we observe that the correlation coefficient for network throughput is highest, giving substantial evidence that network is the shared resource for which the two VMs are competing for.

Interestingly, we also observe high correlation for the cache misses and the context switches. Upon further investigation, we found the when netperf starts executing, its CPU based telemetry like cache misses and context switches start giving non-zero measurements compared to zero measurements when it was idle. This directly correlates with the primary QoS metric of the affected VM. As Proctor only looks at the most correlated metric (network throughput in this case), these false positives are ignored while performing the investigation.

I/O Contention. We use the scenario exhibited by WL2 to study Disk I/O contention. Here, Twitter, an I/O latency critical application, is being affected and Yahoo Cloud Serving Benchmark (YCSB) is the contentious application both running in virtualized environments. Therefore, we expect the QoS metric of YCSB to correlate with QoS metric of Twitter application.

We show this correlation in Figure 7b. YCSB, being an I/O intensive application, increases the latency of the Twitter drastically. This is because the I/O requests of the throughput intensive I/O applications pollute the I/O queue present in the disk, increasing the access time of the latency critical I/O applications. Therefore, we observe high correlation coefficient of 0.87 between the QoS metrics of YCSB and Twitter application.

Since both are I/O critical applications, sending a large number of disk requests, we expect the I/O to the be shared resource that VMs are competing for. Figure 8b shows this investigation where the disk accesses are highly correlated with the QoS of the Twitter application. In this manner, PDI correctly identifies the contentious VM and the shared resource for I/O intensive applications.

CPU Core Sharing. We use the setup present in WL6 for studying contention due to CPU core sharing. In this workload, Naive Bayes is the affected VM and Page Rank is the contentious VM. When a VM executing Naive Bayes is consolidated with a VM executing Google Page Rank in the same physical core, the IPC of Naive Bayes is affected as both of them are CPU intensive and end up time sharing the CPU core. In this case, we expect a high correlation between the QoS metrics of Naive Bayes and Page Rank applications. We illustrate this interference in Figure 7c, showing a high correlation between the QoS metric of Naive Bayes and Google Page Rank. We observe a correlation coefficient of 0.83 in this case.

Similarly, Figure 8c shows the metrics correlating with Naive Bayes' QoS when it shares the CPU core with Page Rank algorithm. We observe that context switches, a by-product of CPU core contention, show high correlation.

Interestingly, we also observe that the cache misses show high correlation. This is because when VMs share physical cores, in addition to core resources, they share all private and shared caches as well. This leads to a high correlation between primary QoS of the affected VM with the cache misses of the contentious VM. Again, PDI only looks at the shared resource with the highest correlation and ignore cache misses.

LLC Contention. We use the experimental setup present in WL9 to study LLC contention, where omnetpp is the affected application and lbm is the contentious application. In this scenario, both the applications are cache sensitive and compete for last level cache. Figure 7d shows the effect of the arrival of lbm on the QoS of omnetpp application. We observe that when omnetpp is consolidated with a VM executing libquantum in the same server, its primary QoS metric (IPC) drops substantially, resulting in a very high correlation coefficient of 0.93.

Further, we use PDI to investigate the source of contention. Figure 8d shows that cache misses of contentious VM have a high correlation with the QoS of affected VM. This is expected as both the applications are cache intensive. PDI's correlation coefficient is able to tell that the cache misses of LLC for libquantum correlates with primary QoS metric of omnetpp.

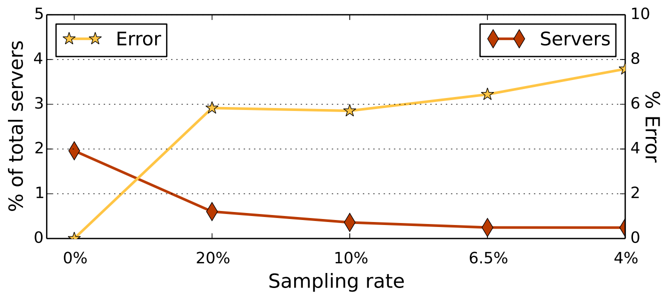


Fig. 9: No. of Proctor servers required to handle 12800 VMs

No Contention. Another interesting experimental setup was conducted to verify if PDD is successful in disregarding false positives when there is no contention. WL10 illustrates a scenario where all the applications do not interfere with each other's performance. In such scenarios, PDD did not trigger a performance degradation at all. This shows the robustness of our technique in disregarding false positives.

These experiments show that PDI is accurate in investigating the source of contention across a wide range of shared resources.

E. Scalability

One of the key goals of Proctor is to provide a datacenter wide solution towards identifying performance intrusion. In this section, we study how Proctor scales in a large datacenter. In particular, we evaluate the benefits of subsampling when scaled and show that there is a minimal loss in the accuracy of detecting performance intrusive VMs when a sub-sampled data is utilized by Proctor.

For this evaluation, we simulate an environment similar to a datacenter setup capable of executing up to 12800 VMs simultaneously while utilizing 2560 nodes. For this experiment, we collect telemetry data obtained from multiple executions runs for the workload scenarios enumerated in III. We then extrapolate the telemetry to obtain data nearly equivalent to the amount of data that is being collected at large-scale data centers [33]. PDI then queries the large-scale telemetry data to identify the source of contention. In this experiment, we start with no sampling and then increase the rate of subsampling, calculating the number of servers required to address the PDI requests from 12800 VMs. The findings of this experiment are presented in Figure 9, showing the impact of subsampling on datacenter resources (left y-axis).

Our baseline utilizes live telemetry (no sampling) to investigate the root cause of performance intrusion. We observe that the size of telemetry data for 12800 VMs that have been executing for an hour is around 91 GB. The baseline requires 50 servers (2% of production datacenters) to keep up with the requests of 12800 VMs. To reduce the amount of data required for the investigation, PDI uses a robust subsampling technique, that significantly reduces the server resource requirements. As shown in the figure, Proctor at 20% sampling requires only 15 machines, as compared to 50 machines with no sampling.

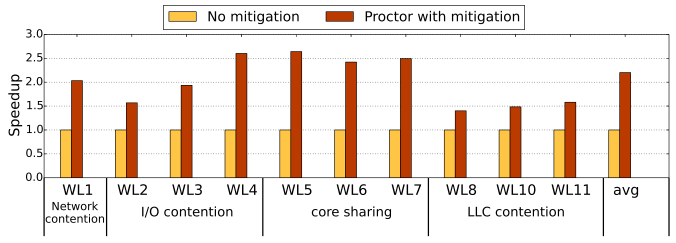


Fig. 10: Performance improvement due to Proctor runtime system

This number reduces to just 6 machines with 4% sampling.

However, aggressive sampling can result in inaccurate results. We show the effect of sampling rate on accuracy error in Figure 9 (right y-axis), where accuracy error is measured from the difference between the correlation coefficients obtained by querying the sampled data and corrletaion coefficient obtained from the original data. As shown in the Figure, no sampling has zero error. We observe that subsampling results in low error in the investigation process, increasing the error to just 5% and 8% for 20% and 4.5% samples respectively. In addition, this error gets masked because the VM or the metric having maximum correlation coefficient stays the same before and after sampling. We observe diminishing benefits with more aggressive sub-sampling rate. Hence, we utilize 6.5% sampling as a final parameter for our experiments as it was the sweet spot optimizing for low error and server count overhead.

F. Putting It All Together

The key use case of Proctor's detection and investigation technique is mitigate VMs that are subjected to performance intrusion. In this section, we demonstrate the benefits brought by Proctor towards this regard. For this study, we couple Proctor's detection and investigation methodology with a simple mitigation technique that migrates the contentious application to another core/physical disk/network channel if Proctor detects a performance anomaly. Our baseline is a system with no performance degradation detection and identification mechanism. Speedup is calculated as the ratio of QoS of the application when its performance is degraded with the QoS of the application after Proctor mitigates the performance intrusive VM at the point when PDD detects intrusion. The findings of this experiment are presented in Figure 10, showing the speedup achieved by Proctor for the affected VM as compared to baseline.

We are able to see that in every situation, the presence of Proctor is able to improve the QoS of the affected VMs. For example, Proctor improves the performance of I/O and network intensive workloads on an average of about 2×. This is due to the fact that the latency of I/O intensive workloads are highly affected in many cases due to intrusion. In situations when CPU cores are being shared, IPC is affected minimum 2×. This is primarily due to context switch overhead when two applications share the same CPU core. On an average, we observe that the presence of Proctor improves the performance of datacenters by 2.2×.

VI. Related work

In this section we discuss work relevant to Proctor in the areas of detecting problematic application/VM behavior and diagnosing its root causes. We also present related work that mitigates I/O and network contention.

VM Management: State-of-the-art VM management tools such as vSphere [16], XenServer [45] or resource management tools utilized in IaaS public clouds like Microsoft Azure [26] and Amazon Web Services [2] performs VM placements naively using primitive factors and metrics. For example, VMware's Distributed Resource Management (DRM) [15] takes into account factors like load balancing and power management as factors for optimal placement of VMs. However, this is agnostic towards performance issues due to disk failures, congestion in the network or contention by neighboring VMs. This can create several issues like performance problems, resource unavailability and in some cases also resulting in security vulnerabilities like denial of service [11], SQL injection [3] etc. Proctor can complement such systems by informing datacenter providers information pertaining to problematic VMs and its root causes. This can motivate smart VM placement strategies.

Contention Detection Techniques Major classes of contention detection techniques focus on a particular aspect present in the system rather than providing an integrated approach. Zhuravlev et al.extended the CPU scheduler to alleviate the degree of interferences in a native system. The goal of this work is to schedule the threads by evenly distributing the load intensity to caches [49]. Shieeh et.al [35] tries to eliminate disk contention by utilizing a log-structured design for disk arrays. Parda [14] and IOFlow [40] tries to address contention at the disk level by observing latency of I/O requests and reordering disk queues. Seawall [36], EyeQ [23] and Hadrian [4] focus mainly on isolating interference in network in multi-tenant environments. However, all these techniques fail to provide an integrated solution for hyperconverged environments where contention exists at storage, network and in CPUs.

A Priori Knowledge Another class of applications observe correlation between various system parameters to detect performance issues in runtime and its root causes [1] [42] [46] [28] [24]. Typically, these techniques leverage baseline performance from a set of training applications and provide predictive solutions at runtime for unknown applications. However, the hyper-parameters present in current day systems are too complex to create a generalized model for prediction. Moreover, in current generation datacenters, we observe system configurations to be highly dynamic which is directly reflected on the application's performance. Hence, in addition to being agnostic towards the nature of the application, it becomes mandatory for our solution to be adaptable towards changing characteristics of system as well as newer systems.

VII. Conclusion

We have presented Proctor, a real time performance monitoring infrastructure that is able to detect performance intrusion and identify the root cause VM and resources causing contention. Proctor is based on a robust statistical online learning approach that requires no special profiling phases or assumptions about system and hardware configuration, standing in stark contrast to a wide body of prior work that assumes pre-acquisition of application or system level information prior to its execution. We implement proctor as a real time system that can identify performance issues for VMs at a very low overhead. This, in turn, can improve the application perfromance by $2.2\times$ on an average by identifying smart co-location scenarios.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[2] Amazon Inc. Amazon Elastic Compute Cloud(EC2).

[3] S. Avireddy, V. Perumal, N. Gowraj, R. S. Kannan, P. Thinakaran, S. Ganapthi, J. R. Gunasekaran, and S. Prabhu. Random4: An application specific randomized encryption algorithm to prevent sql injection. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1327–1333, June 2012.

[4] H. Ballani, D. Gunawardena, and T. Karagiannis. Network sharing in multi-tenant datacenters. Technical report, February 2012.

[5] S. Benedict. Performance issues and performance analysis tools for hpc cloud applications: a survey. *Computing*, 95(2):89–108, Feb 2013.

[6] J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[7] D. R. K. Brownrigg. The weighted median filter. *Commun. ACM*, 27(8):807–818, Aug. 1984.

[8] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. *SIGOPS Oper. Syst. Rev.*, 51(2):17–32, Apr. 2017.

[9] 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.

[10] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[11] C. Delimitrou and C. Kozyrakis. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2017.

[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, Mar. 2012.

[13] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 22:1–22:14, New York, NY, USA, 2011. ACM.

[14] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[15] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[16] F. Guthrie, S. Lowe, and K. Coleman. *VMware vSphere Design*. SYBEX Inc., Alameda, CA, USA, 2nd edition, 2013.

[17] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. *SIGARCH Comput. Archit. News*, 43(3):27–40, June 2015.

[18] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '15, New York, NY, USA, 2015. ACM. Acceptance Rate: 17

[19] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 786–799, New York, NY, USA, 2017. ACM.

[20] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM.

[21] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[22] A. Jain, M. A. Laurenzano, L. Tang, and J. Mars. Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[23] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. *REM*, 1005(A1):A2, 2013.

[24] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 4–4, Berkeley, CA, USA, 2010. USENIX Association.

[25] A. J. Lawrance and P. A. W. Lewis. An exponential moving-average sequence and point process (ema1). *Journal of Applied Probability*, 14(1):98?113, 1977.

[26] H. Li. *Introducing Windows Azure*. Apress, Berkely, CA, USA, 2009.

[27] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[28] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[29] V. Nagarajan, R. Hariharan, V. Srinivasan, R. S. Kannan, P. Thinakaran, V. Sankaran, B. Vasudevan, R. Mukundrajan, N. C. Nachiappan, A. Sridharan, K. P. Saravanan, V. Adhinarayanan, and V. V. Sankaranarayanan. Scoc ip cores for custom built supercomputing nodes. In *2012 IEEE Computer Society Annual Symposium on VLSI*, pages 255–260, Aug 2012.

[30] V. Nagarajan, K. Lakshminarasimhan, A. Sridhar, P. Thinakaran, R. Hariharan, V. Srinivasan, R. S. Kannan, and A. Sridharan. Performance and energy efficient cache system design: Simultaneous execution of multiple applications on heterogeneous cores. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 200–205, Aug 2013.

[31] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, Berkeley, CA, USA, 2013. USENIX Association.

[32] E. S. Page. A test for a change in a parameter occurring at an unknown point. *Biometrika*, 42(3/4):523–527, 1955.

[33] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[34] B. M. Sadler and A. Swami. Analysis of multiscale products for step detection and estimation. *IEEE Transactions on Information Theory*, 45(3):1043–1051, Apr 1999.

[35] A. Shieh, S. K, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *In NSDI*, 2011.

[36] A. Shieh, S. Kandula, A. G. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[37] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 285–297, San Jose, CA, 2013. USENIX.

[38] V. Srinivasan, R. Basu Roy Chowdhury, E. Forbes, R. Widialaksono, Z. Zhang, J. Schabel, S. Ku, S. Lipa, E. Rotenberg, W. Davis, and P. D. Franzon. H3 (heterogeneity in 3d): A logic-on-logic 3d-stacked heterogeneous multi-core processor, 11 2017.

[39] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75, Dec 2015.

[40] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[41] VMWare. Vmware esxi and esx.

[42] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.

[43] G. Welch and G. Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995.

[44] Wikipedia. Hyper-converged infrastructure — wikipedia, the free encyclopedia, 2017. [Online; accessed 5-May-2017].

[45] D. E. Williams. *Virtualization with Xen(Tm): Including XenEnterprise, XenServer, and XenExpress: Including XenEnterprise, XenServer, and XenExpress*. Syngress Publishing, 2007.

[46] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[47] F. Yates. Contingency tables involving small numbers and the χ 2 test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934.

[48] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

[49] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, Mar. 2010.