Characterizing Public Cloud Resource Contention to Support Virtual Machine Co-residency Prediction

Xinlei Han[1], Raymond Schooley[2], Delvin Mackenzie[3], Olaf David[4], Wes J. Lloyd[5]

Amazon
Seattle, Washington USA
[1]hxl@uw.edu

University of Washington
Engineering and Technology
Tacoma, Washington USA
[2]ravschoo, [3]delvin,
[5]wlloyd@uw.edu

Object Modeling Systems Laboratory
Colorado State University
Fort Collins, Colorado USA
[4]odavid@colostate.edu

Abstract— **Hypervisors used to implement virtual machines (VMs) for infrastructure-as-a-service (IaaS) cloud platforms have undergone continued improvements for the past decade. VM components including CPU, memory, network, and storage I/O have evolved from full software emulation, to paravirtualization, to hardware virtualization. While these innovations have helped reduce performance overhead when simulating a computer, considerable performance loss is still possible in the public cloud from resource contention of co-located VMs. In this paper, we investigate the extent of performance degradation from resource contention by leveraging well-known benchmarks run in parallel across three generations of virtualization hypervisors. Using a Python-based test harness we orchestrate execution of CPU, disk, and network I/O bound benchmarks across up to 48 VMs sharing the same Amazon Web Services dedicated host server. We found that executing benchmarks on hosts with many idle Linux VMs produced unexpected performance degradation. As public cloud users are interested in avoiding resource contention from co-located VMs, we next leveraged our dedicated host performance measurements as independent variables to train models to predict the number of co-resident VMs. We evaluated multiple linear regression and random forest models using test data from independent benchmark runs across 96 vCPU dedicated hosts running up to 48 x 2 vCPU VMs where we controlled VM placements. Multiple linear regression over normalized data achieved R^2 =.942, with mean absolute error of VM co-residency predictions of ±1.61 VMs. We then leveraged our models to infer VM co-residency among a set of 50 VMs on the public cloud, where co-location data is unavailable. Here models cannot be independently verified, but results suggest the relative occupancy level of public cloud hosts enabling users to infer when their VMs reside on busy hosts. Our results characterize how recent hypervisor and hardware advancements are addressing resource contention, while demonstrating the potential to leverage co-located benchmarks for VM co-residency prediction in a public cloud.**

Keywords—resource contention, multitenancy, resource management and performance, Infrastructure-as-a-Service, virtualization

I. Introduction

Public Infrastructure-as-a-Service (IaaS) clouds abstract the physical placement of virtual machines (VMs) in the cloud to provide distribution transparency to end users. When horizontally scaling VMs to host multi-tier applications, co-location of too many VMs requiring similar resources on the same physical host(s) has been shown to create performance degradation. For service oriented applications, for example, a performance loss of 25% is demonstrated when too many web application servers share the same host(s) in [1]. Another study found that when launching large pools of 80 Amazon EC2 m3.medium 1-vCPU VMs, 21% were inferred to be provisioned on the same physical host. These results demonstrate the potential for unwanted performance degradation from VM co-location for VMs launched in the public cloud.

Recent innovations in virtualization have enabled hardware virtualization of nearly all VM components including the CPU, memory, network and storage I/O. These improvements have drastically reduced virtualization performance overhead as VMs now directly interface with system hardware, but these improvements do not mitigate performance losses from resource contention of co-located VMs. At the same time, cloud servers have become increasingly core dense, as systems include multiple CPUs with ~24-32 physical CPU cores each, to offer as many as 96 to 128 virtual CPU cores. To leverage available CPU capacities, cloud vendors may co-locate increasingly more VMs than ever before to amortize the cost of expensive hardware. Increased densities of VM placements across physical servers can increase public cloud resource contention.

To address unwanted resource contention from co-located cloud VMs, cloud providers now provide users access to dedicated private servers. Amazon Web Services (AWS) (Nov 2015 release), Microsoft Azure (Aug 2019 preview), and IBM Cloud all offer dedicated hosts, privately provisioned physical servers where users can place and run fully isolated VMs [2][3][4]. Google Cloud offers the same feature known as Google sole-tenant nodes (June 2018 beta) [5]. These dedicated servers provide users access to isolated hardware shared with no other users in the public cloud to help mitigate resource contention, but at a premium price, potentially ~20x more than the price of commodity VMs.

When horizontally scaling VMs for Apache Spark or Docker container clusters (e.g. Kubernetes) to run distributed workloads, spreading VM placements across separate physical hosts can disperse user workloads to maximize performance. This is especially important for embarrassingly parallel jobs deployed across VMs as they require identical resources (CPU, memory, disk, network I/O) creating contention if VMs are scaled out horizontally on the same hosts. When VMs are tightly

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

packed onto the fewest number of physical servers as with greedy VM placement algorithms, embarrassingly parallel workloads may suffer from significant CPU, disk, and/or network contention. Recently, AWS introduced spread placement groups, a feature that guarantees VMs are placed on "different (server) racks", but this feature is limited to a maximum of seven VMs per availability zone limiting its utility for larger clusters [6].

In this paper, we focus on two problems. First, we are interested in quantifying performance degradation from co-location of VMs across modern hypervisors and hardware. Given trends with increasing CPU core density of cloud host servers, VM multitenancy is a trend that is likely to increase. Second, we are interested from a user's perspective, in evaluating the quality of VM placements to ensure that low-cost commodity VMs launched in the public cloud can avoid as much resource contention as possible. By providing users with a means to assess the quality of their commodity VM placements, they can determine whether to retain or replace them based on workload or application requirements [7].

To support our research aims, we investigate performance degradation from resource contention when executing common performance benchmarks concurrently across VMs explicitly placed across dedicated cloud servers. For each iteration, we test how resource stress grows by increasing the number of co-located VMs that run the benchmark. We automate our tests with a custom test suite to coordinate parallel benchmark execution, and investigate resource contention across three generations of Amazon EC2 instances. We test: up to 32 x c3.large and c4.large (3rd & 4th generation), 48 x z1d.large (5th generation), and 96 x m5d.large (5th generation) 2 vCPUs VM instances on Amazon EC2 dedicated hosts. Our benchmarking suite profiles CPU, disk, and network resource contention using carefully selected benchmarks including, for CPU contention: sysbench and y-cruncher [8][9], for CPU+disk contention: pgbench [10], and for network contention: iPerf [11].

Benchmark performance metrics are then used as independent variables to train models to predict the number of VM tenants on shared cloud host servers. We applied multiple linear regression and random forest regression to train models to predict VM tenancy on m5d dedicated hosts with up to 48 co-located VMs. We found that multiple linear regression over normalized data explained variance of R^2=.9423 while providing VM co-location predictions of ±1.61 VMs. Performance metrics from our benchmarks can be collected for newly launched VMs to determine the quality of VM placements before committing to long term use in compute clusters (e.g. Apache Spark, Kubernetes). Combined, our four benchmarks can be run sequentially to obtain performance metrics for inferencing with use with our VM tenancy models in ~96 seconds. This fast characterization enables the state of resource contention to be sampled and resampled quickly as needed.

A. Research Questions

This paper investigates the following research questions:

RQ-1: (Resource Contention) What extent of performance degradation (CPU, disk, network) results from VM co-location when running identical benchmarks in parallel on a public cloud? How is public cloud resource contention impacted by recent advancements in virtualization hypervisors and hardware?

RQ-2: (VM Multitenancy Prediction) How effective are performance metrics derived from CPU, disk, and network benchmarks run in parallel across VMs as independent variables to predict VM multitenancy on physical hosts? How accurate are VM multitenancy predictions from multiple linear regression and random forest models trained using these variables?

B. Research Contributions

This paper provides the following research contributions:

1. We characterize the performance degradation for running identical Linux benchmarks in parallel across up to 48 co-located 3rd, 4th, and 5th generation AWS EC2 VM instances. Our benchmarks characterize CPU, disk, and network resource contention from VM multitenancy. Tested VM generations encapsulate multiple successive enhancements to hypervisors and hardware. 3rd and 4th generation VMs leverage the XEN hypervisor customized by AWS [12], while 5th generation VMs leverage Amazon's AWS "Nitro" hypervisor derived from KVM [13][14][15].
2. We provide a combined benchmark suite to orchestrate fast profiling of VMs that aggregates metrics from four performance benchmarks (sysbench, y-cruncher, pgbench, and iPerf). Our suite automates running benchmarks in parallel across variable sized pools of VMs to test VMs launched on dedicated hosts or on the public cloud. Our suite enables characterizing resource contention using four benchmarks in ~96 seconds, to provide independent variables to estimate the number of co-located VMs sharing the same physical hosts.
3. We evaluate multiple linear regression and random forest regression to construct predictive models to estimate the number of co-resident VMs sharing the same physical host server. We demonstrate accuracy within ±1.61 VMs on 5th generation EC2 dedicated hosts (m5d) which can host up to 48 co-located VMs.

II. Background and Related Work

A. VM Provisioning Variation and Resource Contention

Rehman et al. identified the problem of "provisioning variation" in public clouds in [16]. Provisioning variation is the random nature of VM placements in public clouds that generates varying multitenancy across physical cloud servers leading to performance variance from resource contention. Schad et al. further demonstrated the unpredictability of Amazon EC2 VM performance resulting from provisioning variation and resource contention from VM multitenancy [17]. Ayodele et al. demonstrated the ability to identify resource contention from VM co-location using the cpuSteal metric by running SPEC CPU2006 benchmarks in [18]. Ayodele verified this relationship on both a private cluster, and with the m3.medium Amazon EC2 VM type. Lloyd et al. investigated the use of cpuSteal thresholds to identify co-located VMs from

large VM pools that ran concurrent compute bound webservice workloads [1]. Lloyd found that webservices hosted on co-located worker VMs performing similar work concurrently underperformed by as much as 25%. By replacing undesirable VMs early on, Ou and Farley demonstrated potential to obtain performance improvements by retaining only VMs with the best CPUs when the cloud provider mixed up to five CPU types to implement the VM (m1.large) [7][19]. Liu offered a novel approach to characterize CPU utilization across public cloud hosts over long periods by analyzing CPU temperature [20] but did not consider if temperature can help detect VM tenancy.

Govindan et al. provided a practical method to predict performance interference due to shared processor caches requiring minimal software changes known as Cuanta in [21]. Their approach used cache simulation to predict performance degradation from shared processor caches (e.g. L1, L2, L3) for servers hosting multiple VMs. Cuanta required a linear number of measurements per CPU and produced a prediction error of ~4% while supporting identification of potential VM consolidations to reduce cache interference. Their approach, however, required a carefully placed probe VM to be co-located with other VMs of interest to enable profiling. As public clouds obscure the location of VMs, it would be difficult to know if probe VMs are deployed on the correct host. Kim, Eom, and Yeom developed performance models to infer to what degree application workloads can interfere with each other [22]. These models characterize last level cache misses and references to support a new VM placement algorithm called swim. Performance degradation from VMs placed by swim was found comparable to optimal VM placement, but as their work required changing the public cloud VM placement algorithm, it is not feasible for cloud users to employ as users cannot control public cloud VM placement.

Mukherjee et al. identified problems when relying on hardware counters to infer resource contention for virtualization of kernel resources or memory subsystems in [23]. They offered a software-based probe deployed across small co-resident VMs to detect network and memory resource contention. Their probe detected subtle performance losses from microbenchmarks after a careful tuning phase established specific thresholds to detect degradation with minimal resource utilization tuned on specific hardware. While their approach avoided the use of hardware counters, tradeoffs included their probe must be tuned before use on every platform, and the requirement to use separate probe VMs, consuming resources. Additionally, in the public cloud, controlled placement of probe VMs is not feasible.

B. VM Co-location Detection with Side Channels

In computer security, side channels are used to launch attacks by leveraging information gained from the implementation of computer systems. Side channels involve identification of novel exploits that designers never considered to expose implementation details. In cloud systems, side channels can be used to infer VM co-location in public clouds [24][25][26][27][28][29]. In 2009, Ristenpart et al. demonstrated the potential of VM co-residency detection by launching probe VMs for exploration to leverage a variety of side channels, raising attention to the potential to exploit cloud security [25]. Increasingly sophisticated efforts have followed.

Bates et al. described an approach to infer the location of VMs in the public cloud by launching a large number of "flooder" VMs in [26]. Flooder VMs communicate with an external client to create network traffic across a large set of physical cloud hosts creating a detectable delay that can be observed in the victim VM's network traffic flow. Flooders imprint a "watermark" signature, a detectable pattern of change in the victim's network flow, that can help identify the victim's physical host. The authors exploited the lack of network isolation of co-located VMs to offer a side channel to identify a VM's host. Our work confirms this network isolation vulnerability and quantifies the extent across 3rd, 4th, and 5th generation instances on AWS, and leverages network contention to help identify co-located VMs as part of our multi-benchmark suite. While flooder VMs may be effective to create detectable patterns to infer a VM's location, they are expensive, as users must pay for potentially large numbers of them.

Varadarajan et al. identified the potential to generate resource freeing attacks in the cloud in [29]. Attacks were launched by co-resident attacker VMs sharing the same host as victim VMs. Attacker VMs created resource contention for CPU, disk, or network resources. By generating contention, victim VMs were delayed as they bottlenecked on one or more unavailable resources effectively altering the victim's workload to free resources for the attacker. By stealing resources, Varadarajan demonstrated performance improvements up to 60% for attacker workloads on private clouds, and 13% on the AWS public cloud. Varadarajan did not explicitly address identification of the number of co-located VMs, and their approach of launching large numbers of attack VMs can be expensive.

Inci et al. proposed three approaches to detect co-located VMs on commercial IaaS clouds by using an attacker VM that works without the cooperation of other VMs on the same host in [27]. Two approaches focused on the malicious use of CPU last level caches (LLCs), while the third performed memory bus locking with atomic CPU instructions. The goal was to introduce interference to cause detectable application performance degradation across other VMs due to apparent co-location. LLC attacks were sufficient to identify co-location of VMs with some confidence, but prone to noise and only effective when VMs shared the same physical CPU socket. Memory bus locking schemes, however, were able to identify co-located VMs that shared different physical CPUs on the physical host, but detection was problematic when running applications performing few memory accesses. Inci launched 80 VM pools of 1 vCPU instances on the Amazon, Google, and Azure cloud and identified 17 of 80, 8 of 80, and 4 of 80 co-located instances respectively. Their approach focused on identifying VMs that shared the same physical CPU on multiprocessor servers. They did not address detection of VMs on adjacent CPUs that can experience significant network and/or disk resource contention. An ideally placed VM will avoid more than CPU contention.

O'Loughlin and Gillam offered a method to detect co-located VMs specific to the Xen hypervisor in [28]. They

installed the xen-utils Debian package on Amazon EC2 VMs and read the domid. On VM reboot, the Xen hypervisor increases guest domids until cycling at 65536. By carefully rebooting VMs in absence of other events, shared physical hosts can be identified as they use ids in a unique range from 0 to 65536. We recently tested this method and discovered it no longer works without uninstalling xen-utils before each reboot, and then reinstalling after each boot. This method requires a large number of time intensive reboots plus careful accounting of IDs to identify counter pacing, and only works with XEN-based VMs. It is not applicable to 5th generation AWS EC2 instances.

In this paper, we offer a new approach that *does not require explicitly placed probe VMs* as in [25][26][29], that leverages common performance benchmarks to train machine learning models to infer the number of co-located VMs. We select CPU, disk, and network benchmarks to characterize *multiple types of resource contention that span VMs sharing multi-processor hosts* improving on [27]. We design and evaluate our approach to predict the number of co-located VMs on CPU-dense hardware with 96 vCPUs capable of hosting 48 co-located 2 vCPU VMs. We are unaware of prior efforts that evaluate prediction accuracy on similar, very large servers. Our models can suggest the health of the environment with respect to CPU, disk, and network contention, as well as the number of co-located VMs. Our approach of VM profiling to collect data for model inferencing requires ~96 seconds and can be repeated to reassess the health of VMs belonging to shared clusters *providing a hypervisor agnostic faster alternative* to [28]. Strategies to test-and-replace underperforming cluster VMs before committing to run intense distributed workloads can ultimately serve to improve performance, and lower cloud computing costs extending the trial-and-better approach of [7].

III. METHODOLOGY

A. Combined Benchmarking Suite

To quantify resource contention from VMs on multi-tenant hosts, we developed a benchmarking suite that combines common system benchmarks due to their wide availability and ease of use. Y-cruncher and sysbench were selected to generate CPU stress, pgbench to generate CPU+disk I/O disk stress, and iPerf for network I/O stress. These benchmarks were deployed and run in parallel across VMs placed on Amazon EC2 dedicated hosts, private cloud hosts available for a premium price, enabling VM tenancy to be controlled. We scheduled identical benchmarks to run in parallel across co-located VMs specifically to stress the same resources. By stressing *the same resources* at *the same time*, our objective was to create and observe the *maximum performance degradation* allowed on the host. The behavior of each benchmark when run in parallel across all possible combinations of co-located VMs (e.g. n=1..48) is then harnessed as an independent variable in predictive models to estimate the number of co-located VMs. To enable fast assessment of resource contention a complete execution of our combined benchmark suite to obtain variables for model inferencing requires ~96 seconds and can run in parallel across any number of VMs in a user's account.

Sysbench provides a comprehensive tool suite encapsulating multiple operating system performance benchmarks [8]. For our experiments, we leveraged sysbench as a CPU stress test to perform the deterministic task of generating the first 2,000,000 prime numbers using 2 threads, 10 times. Sysbench as configured required approximately ~6 seconds of wall clock time on idle c4.large EC2 instances.

Y-cruncher computes Pi and other constants to trillions of digits and can be employed to generate CPU stress [9]. We leveraged y-cruncher as a deterministic workload to calculate Pi to 25 million digits using multiple threads. Y-cruncher requires ~7.2 seconds on an idle c4.large EC2 instance while generating primarily CPU stress. Post-processing and validation against pre-calculated constants extend the runtime to approximately ~7.9 wall clock seconds while adding a small amount of disk I/O. We found that to observe a significant performance variance from running multiple instances of sysbench in parallel, it was necessary to ***stop*** idle Linux VMs. This involved suspending the VM which effectively deallocates it from the hypervisor. On AWS when stopping and resuming instances, VMs retain the same instance-id, but renegotiate their public and private IP addresses. Presumably this is done because there is no guarantee the VM will resume on its original host. For y-cruncher, best-to-worst case performance variance was 12.9% without stopping idle VMs, but this performance variance grew to 32.4% when stopping idle VMs. Stopping idle VMs improved performance up to 19.5% for parallel instances of y-cruncher, and up to 20.63% for sysbench on dedicated hosts.

Pgbench is a database benchmark integrated with the PostgreSQL relational database that measures system performance for running PostgreSQL database transactions [10]. It supports running a batch of SQL queries repeatedly in multiple concurrent database sessions to calculate average system throughput in transactions per second (TPS). We disabled database vacuuming when running pgbench to avoid performance variance from automated vacuuming that could occur at random intervals. Pgbench initially populates rows randomly into 4 tables according to a scale factor. It then runs transactions with 5 simple queries (SELECT, UPDATE, and INSERT) using 10 concurrent client database connections. We configured pgbench to provision 10 worker threads, one thread for each client, to run all sessions in parallel. Pgbench ran against PostgreSQL version 9.5. In contrast to y-cruncher and sysbench, pgbench generates CPU, memory, and disk I/O stress. We measured the total number of transactions pgbench could complete in 60 seconds.

iPerf provides a standard benchmark for measuring network bandwidth [11]. We deployed iPerf to test TCP throughput between clients and servers. We used a TCP socket buffer size of 416 kilobits for maximum throughput given the network latency between the client and server VMs. We performed concurrent duplex data transfer between the client and server, and calculated total network throughput (upload+download) for 15 second test runs. Running iPerf requires two VMs, one server and one client. We allocated one dedicated host machine for client VMs, and another for server VMs.

B. Testing Infrastructure

All tests were completed in the AWS us-east-1 Virginia region using EC2 instances running Ubuntu 16.04.5 LTS

Linux. For each experiment, all instances were provisioned using a virtual private cloud and configured to share the same subnetwork and launched in the same availability zone. We profiled performance degradation of co-located parallel benchmark runs on EC2 dedicated hosts [2]. We investigated compute optimized instances from the 3rd, 4th, and 5th EC2 instance generations. Specifically, we investigated instances backed by AWS specific versions of the XEN (c3, c4) and KVM "AWS Nitro" (z1d, m5d) hypervisors [13]. Key performance indicators for these hosts are described in Table I. Dedicated hosts support controlled placement and tenancy of VMs across physical servers. To maximize the potential for resource contention, we leveraged only 2 vCPU instances (c3.large, c4.large, z1d.large, m5d.large) and provisioned the maximum number of co-located VMs allowed on each dedicated host server. We initially investigated larger co-located VMs: c4.xlarge (4 vCPUs, up to 8 VMs per host), and c4.2xlarge (8 vCPUs, up to 4 instances per host) but observed less resource contention when using larger VMs as expected. While resource contention was not eliminated, we opted to focus on using 2 vCPU instances to investigate worst-case provisioning scenarios of VMs on the public cloud.

TABLE I. KEY PERFORMANCE INDICATORS FOR CLOUD DEDICATED HOSTS – US EAST PRICING

KPI	c3	c4	z1d, m5d
Xeon CPU model	E5-2680v2	E5-2666v3	Platinum 8191(z1d), Platinum 8175m(m5d)
family/microns/yr	Ivy Bridge-EP/ 22nm/Sep2013	Haswell-EP / 22nm/Nov2014	Skylake-SP / 14nm/Jul2017
vCPUs/host	40	40	48 (z1d), 96 (m5d)
physical CPU cores/host	20	20	24 (z1d), 48 (m5d)
Base clock MHz	2800	2900	3400 (z1d), 2500 (m5d)
Burst clock MHz (single / all)	3600/3100	3500/3200 [30]	4000/4000 (z1d), 3100/3500 (m5d) [31]
Hypervisor / virtualization-type	XEN / full	XEN / full	AWS Nitro (KVM/full)
Max # of 2 vCPU instances/host	16 x c3.large	16 x c4.large	24 x z1d.large, 48 x m5d.large
Pg db storage	16 GB local shared SSD	100GB io1 EBS volume, 5k iops	75GB local shared NVMe
Network capacity/instance	"Moderate" ~550 Mbps	"Moderate" ~550 Mbps	Up to 10 Gbps
Host price/hr	$1.848	$1.75	$4.91 (z1d), $5.97 (m5d)
VM price/hr	$.1155	$.109375	$.205 (z1d), $.124 (m5d)

By default, all of our EC2 instances use default general-purpose 2 (GP2) elastic block store disks with a supported burst rate of 3,000 I/O operations/sec. To run pgbench, we opted to use alternate disks for the PostgreSQL database for all tests. For c3.large instances we used the locally provided solid-state disk (SSD) on the dedicated host. We relocated the database to a local ephemeral volume implemented on a shared physical SSD on the dedicated host. For c4.large, local disks are not available. To create disk I/O contention we provisioned a 100 GB "provisioned IOPS SSD" (io1) volume supporting 5000 IOPS. This was used for the root volume, and also to host the PostgreSQL database. For z1d/m5d instances we moved the database to a local 75GB volume on a shared NVMe SSD on the dedicated host. For pgbench, c3.large, z1d.large, and m5d.large instances allowed measurement of resource contention for co-located VMs sharing two different generations of local SSDs: c3.large (standard SSD) and z1d/m5d (NVMe SSDs). Fourth generation EC2 instances (e.g. c4/m4/r4 family) do not offer the option for local SSDs in favor of remotely hosted elastic block store (EBS) volumes [32]. The hardware configuration of local disk volumes (e.g. individual SSDs or redundant arrays) is not disclosed by the cloud provider. To generate resource contention with pgbench on 4th generation EC2 instances, we provisioned io1 volumes with a fixed number of 5,000 IOPS for a total of 80,000 IOPS for the host. For 4th generation instances with default general purpose-2 (GP2) EBS volumes, the 3,000 IOPS I/O quota effectively hides I/O resource contention when running pgbench. At the maximum IOPS rate, the host has more than sufficient bandwidth to the remote EBS storage medium. Custom provisioned IOPS EBS volumes on 4th generation instances with higher throughput rates were able to create I/O contention to the remote EBS volumes. A drawback of using provisioned IOPS volumes to generate resource contention for testing, is that they are expensive, which may help deter malicious activity.

Our test suite consists of multiple scripts written in Python3 to automate tests and shell to the operating system to execute Linux commands. Our scripts generated CSV output which was analyzed with R and Python in Jupyter notebooks. We leveraged Python packages including: re for parsing, csv for working with comma separated data, os to access Linux, and datetime for time calculations. Bash scripts supported creation and configuration of cloud infrastructure while leveraging ssh, scp, pssh, crontab, the AWS EC2 api, sed, and ntp. Crontab scheduled tests to run concurrently across all EC2 instances at synchronized times. OS clocks were synchronized using the network time protocol (ntp). All VMs were pre-provisioned in advance of running the benchmarks.

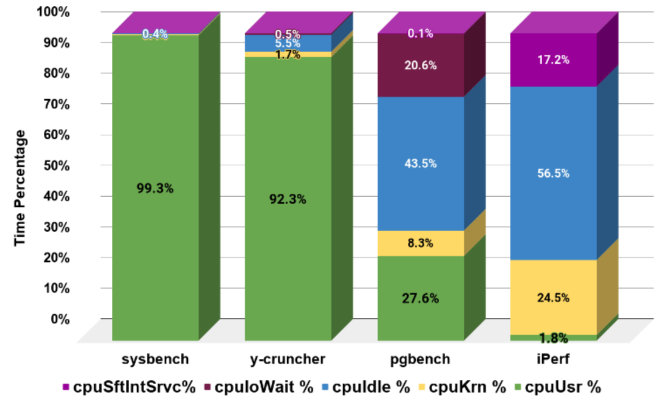

Fig. 1. Our benchmarks exhibit diverse resource utilization profiles to create a wide range of different types of resource contention. CPU time accounting metrics for benchmarks show that **sysbench** and **y-cruncher** are CPU-bound as they're dominated by cpu user mode (*cpuUsr*) time. **Pgbench** involves a blend of idle (*cpuIdle*), user mode (*cpuUsr*), IO wait (*cpuIoWait*), and kernel (*cpuKrn*) time, while **iPerf** is dominated by cpu kernel (*cpuKrn*) and cpu soft interrupt service (*cpuSftIntSrvc*) time.

IV. Experimental Results

A. Resource Utilization Profiles for Common Benchmarks

We profiled the resource utilization (CPU, memory, disk/network I/O) of our benchmarks using the ContainerProfiler tool [33]. This tool supports characterization of the resource utilization of any computational tasks run in a Docker container. Resource utilization of each benchmark for our prescribed configurations was captured, and Linux CPU time accounting variables is shown in Figure 1. The wall clock time of any workload can be calculated by adding Linux CPU time accounting variables (cpuUsr, cpuKrn, cpuIdle, cpuIOWait, cpuSftIntSrvc) and dividing by the total number of CPU cores. Figure 1 characterization resource stress generated by each benchmark.

B. CPU Resource Contention

We initially measured CPU contention on c3.large and c4.large dedicated hosts with up to 16 co-located 2 vCPU VMs on the same server. We visualize the performance of running y-cruncher in parallel on up to 16 co-located c4.large EC2 instances in Figure 2. C4.large instances offer 2 vCPUs, 3.75 GB RAM, and moderate network performance (~550Mb/s) [34]. Figure 2 depicts the average execution time for running y-cruncher in parallel across co-located c4.large instances using box plots, where the set number refers to the number of co-located VMs. For this initial test, all VMs remained online during every test while a varying number of VMs ran y-cruncher. We did not stop or shutdown VMs.

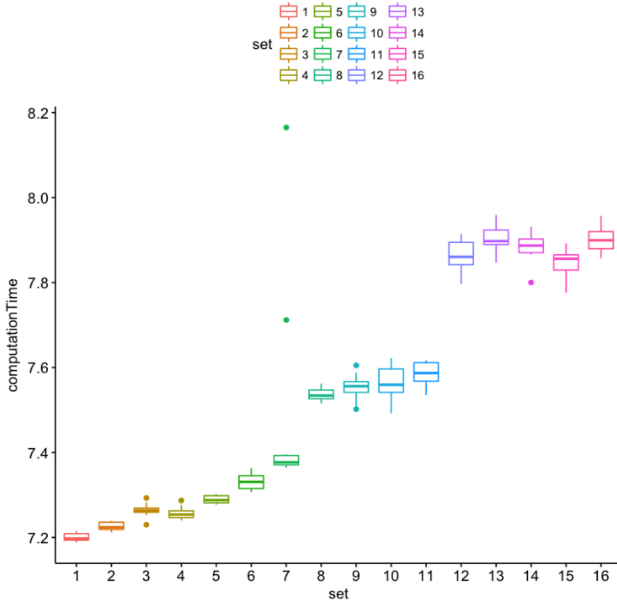

Fig. 2. This figure depicts y-cruncher average execution time when running in parallel across co-located c4.large instances using box plots. Each set includes 16 co-located VMs, where the set number refers to the number of active VMs that ran y-cruncher. All other VMs were online and idle. Box plots show the distribution of runtime for runs with a specific VM tenancy. The average runtime of y-cruncher grows with increasing tenancy by ~9.8%.

Y-cruncher runtime spanned from a minimum of 7.164 seconds with 2 co-located VMs, to a maximum of 7.896 seconds with 16 co-located VMs. **Y-cruncher on c4.large yielded a performance difference of ~9.8%.** Average performance for all configurations was 7.472 seconds, calculated using data from 16 sets of 10 identical test runs. Each test set added an additional co-located c4.large VM to the host. Y-cruncher and sysbench performance across 3rd (c3.large), 4th (c4.large), and 5th (z1d.large and m5d.large) generation instances are contrasted in section E.

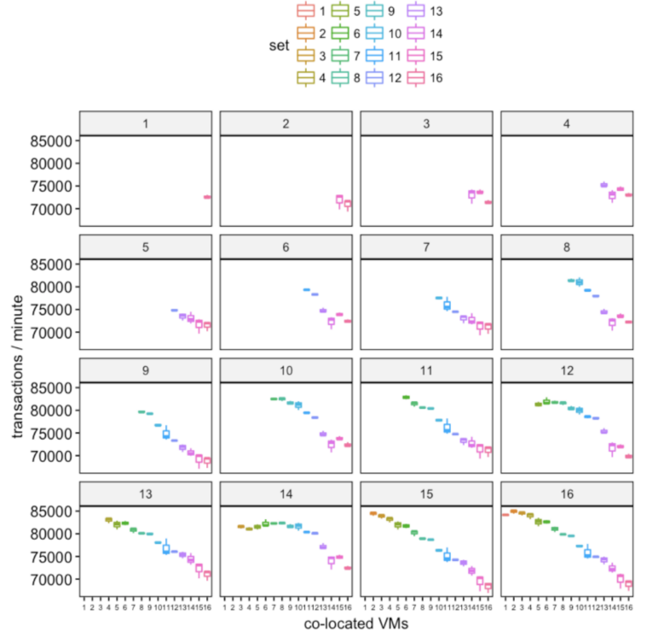

Fig. 3. This figure depicts the total number of database transactions completed in one minute by Pgbench with increasing c3.large VM tenancy. The number of transactions completed by each VM decreased by up to ~24%, relative to the number of tenants sharing the dedicated host.

C. Disk Resource Contention

We illustrate pgbench CPU+I/O contention across co-located c3.large EC2 instances in Figure 3 where postgresql was hosted on a 16GB ephemeral disk volume created on a shared SSD attached to the physical host. Box plots show the performance distribution of the average number of completed database transactions for 10 x 60-second test runs for each tenancy configuration from n=1 to 16 co-located VMs. Each graph square depicts the performance of a specific VM across the experiment. For square 1 in the upper-left corner (VM #1), pgbench ran once in parallel with all 16 VMs. For square 16 in the lower-right corner (VM #16), pgbench ran concurrently 16 times in parallel across up to 16 VMs. On VM #16-run #1, the left-most datapoint on the x-axis, ran with full isolation (e.g. no other VMs ran pgbench), whereas VM #16-run #16 ran with maximum tenancy (e.g. all VMs ran pgbench in parallel). Average performance was calculated for 10 test runs, and Figure 3 shows c3.large average performance in transactions per minute (tpm). Performance ranged from 66946 tpm to 85192 tpm, with an average performance of 76724 tpm. **Pgbench on c3.large produced a best-to-worst case performance difference of 23.8%.** We ran pgbench on c4.large instances with 100GB provisioned IOPS EBS volumes at 5000 IOPS. Performance ranged from 41474 (n=16) to 67274 (n=5), with average performance of 61069 tpm. **Best-**

to-worst case pgbench on c4.large-provisioned IOPS performance varied by 42.25%. We contrast pgbench performance across 3rd (c3.large), 4th (c4.large), and 5th (z1d.large and m5d.large) generation instances in section E.

D. Network Resource Contention

Network resource contention was created and measured by running iPerf in parallel across co-located EC2 instances. We provisioned two dedicated hosts for iPerf, one for server VMs, and the other for client VMs. Running iPerf on c3/c4 instances required a total of 32 VMs across 2 dedicated hosts. On z1d instances, 48 VMs across 2 dedicated hosts were created, and for m5d instances, 96 VMs across 2 dedicated hosts were used. We configured all VMs to share the same availability zone and subnetwork. Clients bind to the same server for each test.

The iPerf duplex option was used to perform a bidirectional test to benchmark upload/download throughput in parallel. We averaged all 20 bandwidth measurements together, 10 from client to server, and 10 from server to client. For an initial c4.large test, **individual iPerf throughput results varied 88%** from the 1-VM average, spanning from a max of 1262 Mbps (n=4) to a min of 235 Mbps (n=12), for a range of 1027 Mbps. We contrast iPerf performance across 3rd (c3.large), 4th (c4.large), and 5th (z1d.large and m5d.large) generation instances in section E.

TABLE II. BENCHMARK PERFORMANCE COMPARISON: MINIMUM VS. MAXIMUM PERFORMANCE FOR TESTS AT ALL POSSIBLE TENANCY LEVELS ON EC2 DEDICATED HOSTS

	y-cruncher *avg runtime (sec)*	**pgbench** *avg transactions*	**iPerf** *throughput MB/sec*	**sysbench** **avg runtime (sec)**
c3 min	15.01	66,973	774	9.40
c3 max	15.57	82,923	958	9.43
c4 min	7.87	66,729	684	8.48
c4 max	8.56	70,684	1,182	8.49
z1d min	4.43	175,882	1,139	6.78
z1d max	4.86	198,012	7,376	6.80
m5d min	5.67, 4.40†	124,247	1,004	9.04, 7.17†
m5d max	6.51	185,525	18,506	9.06

† - indicates test with stopped VMs

E. Cross Generation Resource Contention Comparison

In this section we contrast performance across 3rd (c3.large), 4th (c4.large), and 5th (z1d.large and m5d.large) generation instances for all benchmarks in our test suite including: y-cruncher, pgbench, iPerf, and sysbench. This work addresses **(RQ-1)**: "*How do recent advancements in hardware and virtualization hypervisors address public cloud resource contention?*" Considerable effort has been dedicated to reduce performance overhead through virtualization and hardware advancements. Recently this overhead is stated to be less than the degree of statistical noise (e.g. ~1%) [13][15]. Our goal was to quantify how *well* recent virtualization advancements in software (hypervisors) and hardware (all system components) prevent performance losses from CPU, disk, and network resource contention. A major goal of IaaS cloud platforms is to provide resource transparency, thus we argue that cloud abstractions need to address detectable performance variance from resource contention to eliminate potential side channels from being exploited as detailed in section II. We executed our benchmark suite across c3.large, c4.large, z1d.large, m5d.large 2 vCPU instances created on dedicated hosts. Table II captures the min/max raw performance values for our benchmarks. In the majority of cases minimum performance occurs with maximum tenancy (n=16, 24, or 48 co-located VMs), and maximum performance with minimum tenancy (n=1 VM).

We depict the normalized performance degradation for our benchmark suite from VM co-location for 1 to 16 x co-located 3rd generation c3.large instances in figure 4. Performance degradation is shown for n=1 to 16 x 4th generation c4.large instances in figure 5, 1 to 24 x 5th generation z1d.large instances in figure 6, and 1 to 48 x 5th generation m5d.large instances in figure 7. In all cases, network contention measured with (iPerf) exhibited the steepest decline in performance as a result of increasing VM tenancy, followed by CPU+disk contention (pgbench), and then CPU contention (y-cruncher). Figure 8 depicts the total combined change in performance for all benchmarks run across VMs from different EC2 instance generations.

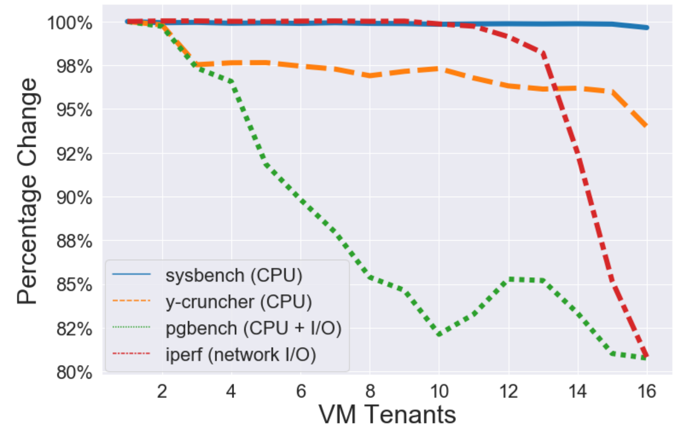

Fig. 4. 3rd generation (c3.large) Instance - Normalized Performance Degradation from VM Co-location on c3.large EC2 dedicated hosts

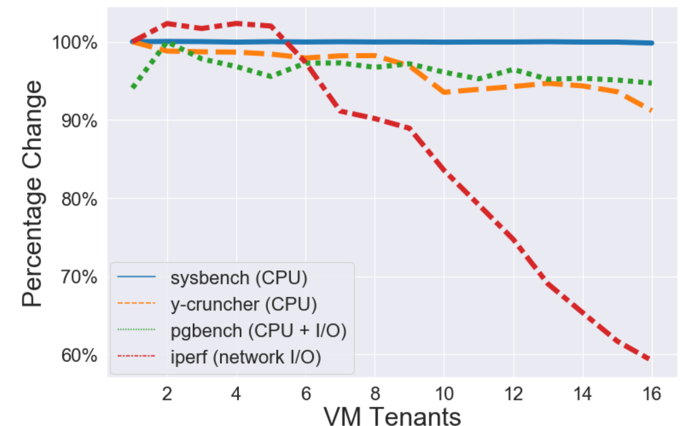

Fig. 5. 4th generation (c4.large) Instance - Normalized Performance Degradation from VM Co-location on c4.large EC2 dedicated hosts

F. *Performance Implications of Idle Virtual Machines*

When deploying our y-cruncher test on m5d.large VMs, we initially measured a performance difference of 14.96% (5.67s to 6.52s) scaling from 1 to 48 VMs. We then refactored our scripts to stop VMs after each test where previously VMs remained online. By stopping idle VMs fewer concurrent instances of the Linux OS kernel shared the physical host. Using this approach, the performance delta increased to 47.97% (4.4s to 6.51s). Running y-cruncher by shutting down idle VMs improved performance by 32.42%, providing a min/max delta 1.49x larger to help better characterize the environment. We observed a similar performance change running sysbench on m5d.large VMs in the presence of idle VMs. Running sysbench in the presence of idle VMs resulted in an indiscernible performance delta of ~0.18% from n=1 to 48 VMs. **Running sysbench by shutting down idle VMs improved performance by 20.81%, providing a delta 116x larger to help better characterize the environment.** We replicated our y-cruncher result on z1d.large instances improving performance by 4.47% by shutting down idle VMs, but could not obtain an improvement on XEN-based c3.large or c4.large instances, or with sysbench on z1d.large.

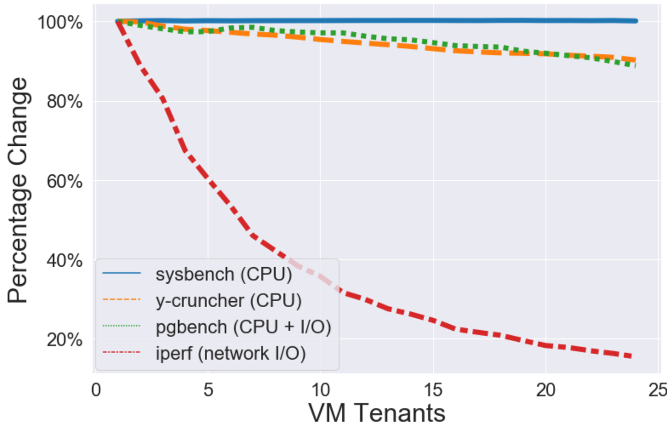

Fig. 6. 5th generation (z1d.large) Instance - Normalized Performance Degradation from VM Co-location on EC2 z1d.large dedicated hosts

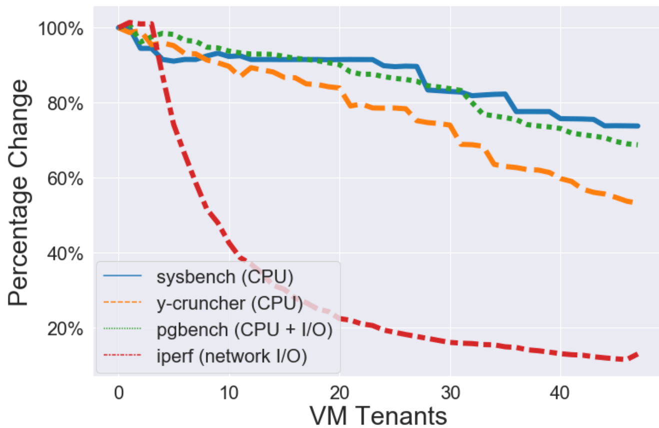

Fig. 7. 5th generation (m5d.large) Instance - Normalized Performance Degradation from VM Co-location on EC2 m5d.large dedicated hosts

We hypothesize that the larger performance delta measured by shutting down idle VMs results for Linux CPU scheduler stress. Our Linux benchmarking VM when idle runs around ~130 processes. When 48 Linux VMs share the same m5d dedicated host, this equates to 6,240 processes. The host OS runs a Linux scheduler that context switches VMs as KVM processes, and each VM runs its own Linux scheduler to context switch local processes. Y-cruncher appears more susceptible to CPU scheduler stress than sysbench on m5d.large as stopping idle VMs improved performance by a wider margin. With 24 Linux VMs on a z1d.large dedicated host, only y-cruncher showed improved performance when stopping idle VMs. Redundant operating system instances on high density cloud hosts demonstrate the need to replace traditional VMs with containers or microVMs to reduce this overhead.

In Figure 8 we show the percentage performance difference using averages for each benchmark at each VM tenancy level to calculate a total percentage difference. Figure 8 depicts a disturbing trend: **with each successive VM generation, the total observable performance degradation from resource contention has grown**. We observe a total performance difference from resource contention of 41.3% for c3.large, 56% for c4.large, 104.9% for z1d.large, and 196.4% for m5d.large. As newer instance generations support greater VM-density, the percentage performance difference for our co-located stress benchmarks running at 100% capacity of a given system resource (e.g. CPU, disk, network) has increased 4.75x. This trend can be explained by the increasing host capacity of new hardware. We observed a performance change of 84.6% running iPerf with 24 co-located z1d instances obtaining ~1.1 Gbps network throughput down from 7.4 Gbps when only one VM shared the host. The performance change increased to 94.6% for 48 co-located m5d.large instances, though these instances still sustained ~1 Gbps throughput**. Loose network quotas provide an exploitable side channel to infer VM co-location.** Network performance degradation is likely to be prominent across 5th generation instances: m5, m5a, m5ad, c5, c5n, and c5d instances that divide network capacity across many VMs. At present, 52 AWS instance types advertise a loosely constrained network capacity of "up to 10 Gbps", and 8 others "up to 25 Gbps". Cloud providers could alleviate performance variance by enacting stricter resource quotas.

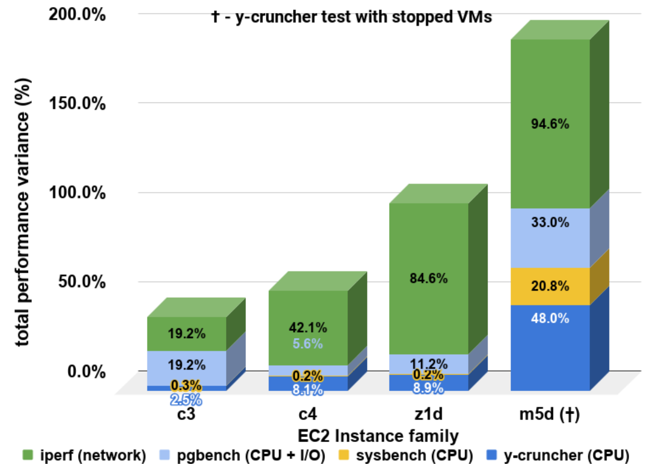

Fig. 8. Total observed performance difference from resource contention for co-located VMs running performance benchmarks across three generations of VMs on Amazon EC2.

As a result of increasing CPU-core density, new instance families (e.g. m5, m5d, r5, r5d, and i3en) support up to 48 co-located 2 vCPU instances. These families appear to provide a wide breadth of performance scores on coordinated benchmark runs. **Our results expose notable performance differences for benchmarks based on the number of VM tenants demonstrating potential to leverage benchmarks to predict the number of co-located VMs (RQ-2)**.

V. Virtual Machine Multitenancy Prediction

A. VM tenancy Model Evaluation on Dedicated Hosts

We next investigated the use of multiple linear regression (MLR) and random forest regression to predict the number of co-located VMs leveraging performance metrics from our benchmarks. We focused on predicting the number of co-located m5d instances, *the most difficult of our case studies* with high tenancy up to 48 guests. Fortunately, from a user's perspective, it is sufficient to learn an approximate number of co-located guest VMs to detect resource contention to learn when to abandon the host in search of better environment to run. We used each benchmark as an independent variable, and merged predictors into a single dataset where each row characterizes performance for a VM with a given host tenancy. Running each benchmark across 48 VMs produced 1,176 individual performance values. We ran each benchmark typically 8 to 11 times, removing the first run, and then averaging performance results for the model to produce a training set of 4 x 1,176 independent variables. We built models to predict VM tenancy using Jupyter Notebooks with an R kernel. We normalized data for MLR using the R scale() function. We obtained our training datasets and test datasets completely independent of each other by performing independent runs weeks apart on different EC2 dedicated hosts in the Virginia Region. We evaluated our independent variables using Random Forest by checking the increase in both node purity and root mean squared error by adding each variable to the model. Pgbench was the strongest predictor, followed by, y-cruncher, sysbench, and finally iPerf. Removing iPerf, reduced R^2 of random forest regression from .9755 to .9739, whereas removing pgbench dropped R^2 to .9459.

TABLE III. Independent Variables Evaluation w/ Random Forest

Independent Variable	% Increase in Node Purity	% Increase in RMSE
iPerf (throughput in MB/sec)	70725	9.268
Sysbench (seconds)	106714	31.064
Y-cruncher (seconds)	135112	49.908
Pgbench (transactions)	146221	56.220

TABLE IV. VM Tenancy Model Evaluation

Evaluation Metric	Random Forest with raw data	MLR with normalized data
R^2	.9755	.9423
Root Mean Squared Error (RMSE)	2.479	2.175
Mean Absolure Error (MAE)	1.950	1.608
Min Prediction	4.537	-0.480
Max Prediction	47.479	49.543

The utility of our predictors is further explained by the coefficient of variance (CV), which is the average divided by the standard deviation to produce a normalized percentage expression of the statistical variance of the benchmark. CV is quite high for our weakest predictor iPerf (noisy signal), and quite low for sysbench (low information) our weakest predictors.

Random forest is not sensitive to numerical precision (i.e. difference in numerical scales) allowing variables to be used directly without transformation. We applied random forest to train a model obtaining an R^2 of .9755. We then applied our random forest model to our test dataset and obtained a root mean squared error of 2.479. Our mean absolute error was ± 1.95 VMs, for a mean absolute percentage error of about 4%. Figure 10 visualizes our observed vs. predicted values using the random forest model. We then normalized training data and trained an MLR model obtaining an R^2 of .9423. We then applied our MLR model to predict VM tenancy for our test dataset and obtained a root mean squared error of 2.175. Our mean absolute error was ± 1.61 VMs, for a mean absolute percentage error of about 3.4%. Figure 11 visualizes our observed vs. predicted values using the MLR model.

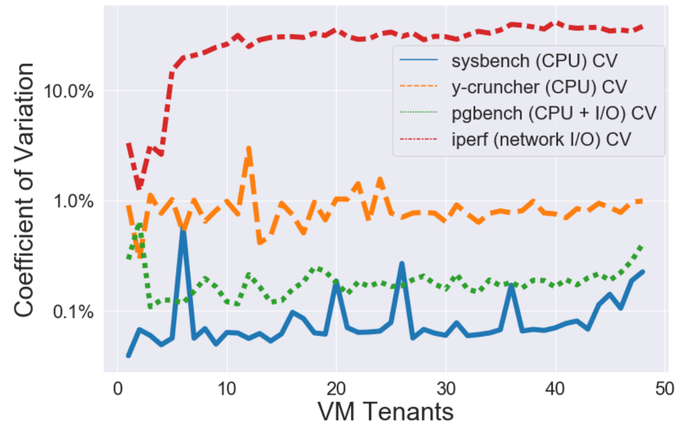

Fig. 9. This log scale graph depicts the coeffieicnet of variance (CV) providing a normalized percentage comparison of statistical variance.

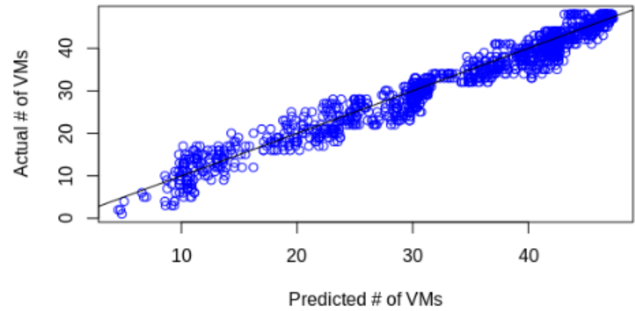

Fig. 10. Observed vs. predicted co-located VMs - random forest model

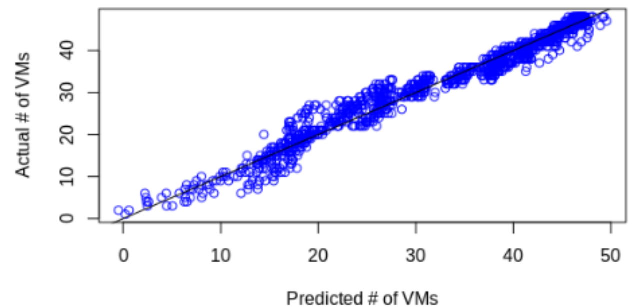

Fig. 11. Observed vs. predicted co-located VMs - random forest model

B. VM tenancy Model Application on the Public Cloud

We next applied our random forest regression model to predict VM tenancy for 50 x m5d.large spot instances using AWS. We executed ~11 runs of sysbench, y-cruncher, and pgbench across these VMs to obtain our dataset while discarding the first run of each benchmark. We then launched an additional 50 x m5d.large spot instances to serve as iPerf clients. These VMs served only as clients for the iPerf test to match our procedure used to obtain training data. We then aggregated our benchmark metrics by calculating performance averages. We used the averages and applied our random forest model to generate VM tenancy predictions. VM tenancy predictions ranged from 10 to 33, with an average of 22.66, mean of 24, and mode of 32 (15 predictions). We visualize VM tenancy predictions for our pool of 50 x m5d.large spot instances in figure 12. We note that to evaluate model accuracy knowledge of the actual public cloud VM placements is required. We expect there are false positives in our predictions, and these will cause some VMs to be predicted multiple times. A key observation is where groupings of predictions tend to cluster around 11, 17, 24, and 32 co-located VMs.

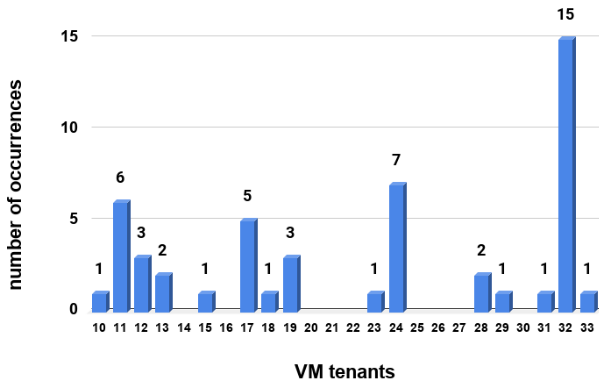

Fig. 12. Predicted number of co-located VMs for pools of 50 x ec2 m5d.large spot instances

VI. Conclusions

Despite continued improvements to virtualization hypervisors and hardware, we were able to generate considerable CPU, disk, and network performance degradation running system benchmarks in parallel on co-located VMs **(RQ-1)**. Hardware with high CPU core density produced worst case performance degradation with co-located VMs, for the m5d.large, y-cruncher: (48%), pgbench: (33%), sysbench (20.8%), and iPerf (94.6%). Total performance variance increased across instance generations from c3 (42%), to c4 (56%), to z1d (104.9%) and m5d (196.4%) trending with VM density. We discovered that shutting down idle VMs increased y-cruncher performance by 32.4% on m5d hosts and 4.5% on z1d hosts. Sysbench performance increased 20.8% on m5d hosts when shutting down idle VMs. Network performance suffered the most from resource contention averaging 60% across all VM generations. Multiple linear regression models trained with benchmark performance data predicted the number of co-located VMs on m5d dedicated hosts with up to 48 co-located VMs to within ~±1.61 VMs **(RQ-2)**.

Having access to cloud-based dedicated hosts (e.g. bare metal servers) enables employing performance benchmarks as side channels to infer VM co-location. Benchmarking performance of VMs in isolation while controlling the degree of VM tenancy helps establish expected levels of performance that can then be leveraged to determine when user VMs are co-located. By adopting Farley et al.'s trial-and-better approach [7], tenancy prediction can be used to test and replace VMs to foster better performing multi-node MapReduce, Apache Spark, and Kubernetes clusters. For running parallel workloads that scale out horizontally, perfect accuracy of VM tenancy predictions is not required to detect resource contention. Obtaining better placement of nodes in clusters is of importance to improve cloud computing performance. These advancements can be leveraged by users to better distribute workloads across cloud infrastructure, ultimately lowering costs to process big data workloads in the cloud. Scripts and resources supporting this research are available online at: [35].

VII. Acknowledgements

This research is supported by the US National Science Foundation's Advanced Cyberinfrastructure Research Program (OAC-1849970), the NIH grant R01GM126019, and by the AWS Cloud Credits for Research program.

References

[1] W. Lloyd, S. Pallickara, O. David, M. Arabi, and K. Rojas, "Mitigating resource contention and heterogeneity in public clouds for scientific modeling services," in *Proceedings - 2017 IEEE International Conference on Cloud Engineering, IC2E 2017*, 2017.

[2] "Amazon EC2 Dedicated Hosts Pricing." [Online]. Available: https://aws.amazon.com/ec2/dedicated-hosts/pricing/.

[3] "Introducing Azure Dedicated Host." [Online]. Available: https://azure.microsoft.com/en-us/blog/introducing-azure-dedicated-host/.

[4] "Dedicated hosts and dedicated instances." [Online]. Available: https://cloud.ibm.com/docs/vsi?topic=virtual-servers-dedicated-hosts-and-dedicated-instances.

[5] "Introducing sole-tenant nodes for Google Compute Engine — when sharing isn't an option." [Online]. Available: https://cloud.google.com/blog/products/gcp/introducing-sole-tenant-nodes-for-google-compute-engine.

[6] "Placement Groups - Amazon Elastic Compute Cloud." [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html#placement-groups-spread.

[7] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: Exploiting Performance Heterogeneity in Public Clouds," in *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*, 2012, pp. 1–14.

[8] "Akopytov/sysbench: Scriptable database and system performance benchmark." [Online]. Available: https://github.com/akopytov/sysbench.

[9] "y-cruncher, A Multi-Threaded Pi Program." [Online]. Available: http://www.numberworld.org/y-cruncher/.

[10] "PostgreSQL: Documentation: 10: pgbench."

[11] "iPerf - The TCP, UDP, and SCTP network bandwidth measurement tool."

[12] P. Barham *et al.*, "Xen and the art of virtualization," *ACM SIGOPS*

Operating Systems Review, vol. 37. p. 164, 2003.
[13] B. Gregg, "AWS EC2 Virtualization 2017: Introducing Nitro," *Brenden Gregg's Blog*, 2017. [Online]. Available: http://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html.
[14] A. Kivity, U. Lublin, A. Liguori, Y. Kamay, and D. Laor, "kvm: the Linux virtual machine monitor," *Proc. Linux Symp.*, vol. 1, pp. 225–230, 2007.
[15] "AWS Nitro System."
[16] M. S. Rehman and M. F. Sakr, "Initial findings for provisioning variation in cloud computing," in *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, 2010.
[17] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, pp. 460–471, 2010.
[18] A. O. Ayodele, J. Rao, and T. E. Boult, "Performance Measurement and Interference Profiling in Multi-tenant Clouds," in *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*, 2015.
[19] Z. Ou *et al.*, "Is the Same Instance Type Created Equal? Exploiting Heterogeneity of Public Clouds," *IEEE Trans. Cloud Comput.*, 2013.
[20] H. Liu, "A measurement study of server utilization in public clouds," in *Proceedings - IEEE 9th International Conference on Dependable, Autonomic and Secure Computing, DASC 2011*, 2011, pp. 435–442.
[21] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta : Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines," *Proc. 2nd ACM Symp. Cloud Comput.*, pp. 22:1--22:14, 2011.
[22] S. G. Kim, H. Eom, and H. Y. Yeom, "Virtual machine consolidation based on interference modeling," *J. Supercomput.*, 2013.
[23] J. Mukherjee, D. Krishnamurthy, J. Rolia, and C. Hyser, "Resource contention detection and management for consolidated workloads," in *Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management, IM 2013*, 2013.
[24] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, "Understanding the effects of hypervisor I/O scheduling for virtual machine performance interference," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 2012, pp. 34–41.
[25] T. Ristenpart and E. Tromer, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," ... *Conf. Comput. ...*, pp. 199–212, 2009.
[26] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, "On detecting co-resident cloud instances using network flow watermarking techniques," *Int. J. Inf. Secur.*, 2014.
[27] M. S. İnci, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "Co-location detection on the cloud," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.
[28] J. O'Loughlin and L. Gillam, "Sibling virtual machine co-location confirmation and avoidance tactics for Public Infrastructure Clouds," *J. Supercomput.*, 2016.
[29] 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 ACM Conference on Computer and Communications Security*, 2012.
[30] "CPU-World: BSP/CPU/Microprocessor/Co-processor/FPU/Micro-controller families."
[31] "Amazon EC2 z1d instances."
[32] "Amazon EBS Volume Types – Amazon Elastic Compute Cloud."
[33] "wlloyduw/ContainerProfiler: Tools to profile the resource utilization (CPU, disk I/O, network I/O) of Docker containers." [Online]. Available: https://github.com/wlloyduw/ContainerProfiler.
[34] "Testing Amazon EC2 network speed." [Online]. Available: http://epamcloud.blogspot.com/ 2013/03/testing-amazon-ec2-network-speed.html.
[35] "IaaSCloudResourceContention: Parallel resource contention experiment scripts, data, processing pipeline." [Online]. Available: https://github.com/wlloyduw/IaaS CloudResourceContention.