http://www.diva-portal.org

This is the published version of a paper published in *Computing*.

Citation for the original published paper (version of record):

Taheri, J., Zomaya, A Y., Kassler, A. (2017)
vmBBProfiler: A BlackBox Profiling Approach to Quantify Sensitivity of Virtual Machines to Shared Cloud Resources
Computing, 99(12): 1149-1177
https://doi.org/10.1007/s00607-017-0552-y

Access to the published version may require subscription.

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-64566

Computing (2017) 99:1149–1177
DOI 10.1007/s00607-017-0552-y

vmBBProfiler: a black-box profiling approach to quantify sensitivity of virtual machines to shared cloud resources

Javid Taheri[1] · **Albert Y. Zomaya**[2] ·
Andreas Kassler[1]

Received: 22 August 2016 / Accepted: 15 March 2017 / Published online: 25 March 2017
© The Author(s) 2017. This article is an open access publication

Abstract Virtualized Data Centers are packed with numerous web and cloud services nowadays. In such large infrastructures, providing reliable service platforms depends heavily on efficient sharing of physical machines (PMs) by virtual machines (VMs). To achieve efficient consolidation, performance degradation of co-located VMs must be correctly understood, modeled, and predicted. This work is a major step toward understanding such baffling phenomena by not only identifying, but also quantifying sensitivity of general purpose VMs to their demanded resources. vmBBProfiler, our proposed system in this work, is able to systematically profile behavior of any general purpose VM and calculate its sensitivity to system provided resources such as CPU, Memory, and Disk. vmBBProfiler is evaluated using 12 well-known benchmarks, varying from pure CPU/Mem/Disk VMs to mixtures of them, on three different PMs in our VMware-vSphere based private cloud. Extensive empirical results conducted over 1200 h of profiling prove the efficiency of our proposed models and solutions; it also opens doors for further research in this area.

Keywords Performance degradation · Virtualization · Cloud computing

Mathematics Subject Classification 68M20 · 68M14

✉ Javid Taheri
javid.taheri@kau.se

Albert Y. Zomaya
albert.zomaya@sydney.edu.au

Andreas Kassler
andreas.kassler@kau.se

1 Karlstad University, Karlstad, Sweden

2 University of Sydney, Sydney, Australia

1 Introduction

The demand for cloud computing has been constantly increasing during recent years. Nowadays, Virtualized Data Centers (vDCs) accommodate thousands of Physical Machines (PMs) to host millions of Virtual Machines (VMs) to fulfill today's large-scale web applications and cloud services. Many organizations even deploy their own private clouds to manage their computing infrastructure [1]; it is shown that more than 75% of current enterprise workloads are currently running on virtualized environments [2]. Despite the massive capital investments however, their resource utilization rarely exceeds 20% of their full capacity [2,3]. This is because, alongside its many benefits, sharing PMs also leads to performance degradation of sensitive co-located VMs and could undesirably reduce their quality of service (QoS) [4].

Figure 1 shows relative throughput (with regard to their isolated run) of eight high resource demanding VMs when co-located with another VM running a Mem + Disk intensive application (unzipping large files). All VMs had 2vCPU, 2 GB of RAM, and 20 GB of Disk. For each test, VMs were pinned on the same set of CPUs/Cores and placed on the same disk to compete for CPU cycles, conflict on L1/L2/L3 memory caches, and interfere with each others' disk access. As can be inferred, despite being classified as "resource demanding", five of these applications could be safely co-located with the background resource intensive application (Mem + Disk), while a conservative view would have separated all VMs to allocate on separate PMs. This simple example shows that conservative methods could be unnecessary for many high demanding VMs; it also justifies the importance of understanding performance degradation to identify VMs that can be safely co-located with minimum interference to each other.

This work is a major step toward quantifying such interferences through profiling a variety of benchmarks under different working scenarios. Such profiles are then used to identify sensitivity of each VM to its allocated resources, and consequently to identify/link the importance of each resource to its actual throughput. We used 12 well-known benchmarks with different resource usage signatures (CPU/Mem/Disk intensive and various combinations of them) to run on three different PMs. Results were collected and used to model sensitivity of each VM to its allocated resources. We finally aligned our results with actual throughput of these benchmarks to show accuracy of our approach: VM Black-Box Profiler (vmBBProfiler).

Our contributions in this work are: unlike all available similar approaches, vmBBProfiler (1) only uses Hypervisor level metrics to identify sensitivity of a VM to its allocated resources. No code/agent is required to be developed, installed, and/or executed inside VMs; (2) provides a systematic approach to calculate sensitivity of VMs to their resources; (3) uses a wider range of benchmarks to calculate sensitivity values; and (4) produces a sensitivity number for each resource instead of binary-labeling it as either 'sensitive' or 'insensitive'.

The remainder of this paper is structured as follows. Section 2 reviews the related work. Section 3 explains the architecture of vmBBProfiler. Section 4 demonstrates vmBBProfiler's procedures. Section 5 lays out our experimental setup. Results are discussed and analyzed in Sect. 6, followed by conclusion and future directions in Sect. 7.

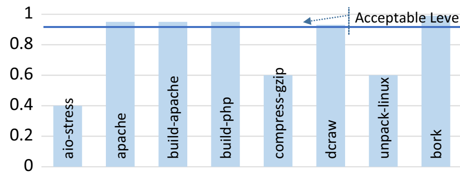

Fig. 1 Relative performance of eight applications when co-located with a Mem + Disk (unzipping large files) intensive application. The "acceptable-level" is an arbitrary threshold that a service provider can tolerate when delivering its services

2 Related work

The ever increasing popularity of virtualization [5] in vDCs is one of the most significant shifts in the IT industry for the twenty first century. Through virtualization, PM resources are partitioned for VMs to run cloud services. Running a highly efficient vDC is however not a trivial task. Firstly, vDCs are envisaged to be able to properly partition resources and run several VMs on each PM. Though resources like CPU and Network seem to be fairly partition-able, Mem and Disk are proven to be much more cumbersome. Last Level Cache (LLC) in particular has been the focus of many approaches because of its profound, yet unpredictable effect on co-located VMs [3]. Secondly, vDCs need to accurately provide online operational information to both administrator and users so that functionality of deployed services can be monitored, controlled, and ensured at all times. Management systems are also required to dynamically (re)organize—via cold or live migration—VM placements. Because optimization techniques to perform efficient allocations are heavily linked to understanding the true behavior of co-located VMs under contention, to date, VM (re)allocation decisions are either made manually based on administrators' experience, or automatically based on very few parameters such as CPU load of VMs and PMs [5]. Thirdly, vDC management systems must be able to identify performance bottlenecks in virtualized environments to make certain that all services are able to perform in such complex digital ecosystems. This demands the ability to accurately predict the performance of different VMs in various working scenarios; ie, isolated or co-located as well as under or free of resource contentions. This concern, in particular, seems to be more important than the other two because it can directly lead to significant increase of vDCs' productivity.

To date, many approaches are proposed to solve the performance degradation of VMs in vDCs, although none actually models and/or proposes a metric/criterion to measure/reflect it; they can be categorized into the following four main themes.

2.1 Agent based approaches

Approaches in this group use foreign agents/codes to constantly monitor throughput of applications and report them to a central decision making system for further analysis and decision making. They rely on developing a tailor-made module/agent for each application, installing it in the VM, and giving it enough system privileges to collect and send out performance data.

Xu et al. [6] proposed two Fuzzy based systems (global and local) to monitor resource utilization of workloads in VMs. The local system—injected into a VM—compares its performance with the desired SLA, and request or relinquish resources (eg, CPU share) if required. The global controller receives all local requests and decides what VMs should get more resources in cases of contention. Rao et al. [7] proposed VCONF, an auto-configuration RL-based approach, to automatically adjust CPU and Memory shares of VMs to avoid performance degradation. Other approaches [8–12], including Q-cloud [8] and TRACON [10], control applications' response times inside VMs only through adjusting their CPU-shares. Bartolini et al. [12] proposed AutoPro to take a user-defined metric and adjust VMs' resources to close the gap between their desired performances and their current ones. AutoPro uses a PI controller to asymptotically close this gap and can work with any metric (eg, frame/s) as long as developers can provide it. Delimitrou and Kozyrakis [13] proposed iBench to inject/run a series of 15 snippets into a VM to pressure its running application and identify the amount of contention it can tolerate before violating its QoS.

Agent based approaches are generally more accurate than others because they use direct measurements from applications inside VMs to adjust control variables. Their usage however could be very limited, because (1) they all reply on inside tailor-made agents to report the exact throughput of applications/VMs, and (2) their focus is to improve performance of VMs rather than model and/or measure resource contention.

2.2 Model based approaches

Approaches in this group perform engineered experiments to model performance degradation of co-located VMs. They co-locate and run VMs in isolated environments to observe how they clash to access resources.

Akoush et al. [14] co-located VMs 2-by-2 and proposed a semi-linear performance model for consolidated VMs; they identified three sources for contention: visible resources (eg, CPU), invisible resources (eg, shared cache), and Hypervisor overheads. Doyle et al. [15] used various queuing models to estimate response time of services under various working scenarios; their model assumes a fair amount of knowledge about the application, and response times of both servers and storage services. Bennani et al. [16] studied application-aware systems to predict throughput and response time of both online and batch workloads. Auto-regressive models [17] and Fuzzy logic [6] were also proposed to model/link performance of VMs to their CPU-allocations. More complicated approaches—such as Auto-regressive moving average model [18] and Artificial Neural Network (ANN) [5]—modeled/linked performance of applications to multiple parameters. Dai et al. [19] proposed a multi-linear regression technique to measure how a 'background' application/VM can influence performance of a 'forward' one. Performing a large number of experiments to build a rich library, they could predict performance degradation of an unknown application/VM through weight-averaging its similarity with the ones from this library. Chen et al. [20] used a few Hypervisor metrics to model/calculate slowdown of VMs and migrate them according to these slowdown values.

Model based approaches are generally very accurate for the exact applications for which models are produced. Their three main drawbacks are: (1) they need to prepare isolated environments to put VMs in direct competition with each other, (2) their results usually lack generality, and (3) their focus is to improve performance of VMs rather than model and/or measure resource contention.

2.3 Classification based approaches

Approaches in this group first classify applications into several groups and then decrease performance degradation through co-locating VMs from different groups. These approaches do not identify, model, and/or measure the exact sources/causes of contention, but ways to avoid it.

Calgar et al. [21,22] proposed two approaches to classify performance of soft-real-time applications from Google trace data. Using ANN [21] or K-means [22] to investigate the effect of several inputs, they packed VMs onto PMs to reduce performance degradation. Qian et al. [23] proposed a Fuzzy based performance interference system to model all sorts of resource contention between co-located VMs; an influence matrix is then produced to (re)allocate VMs. Using five network metrics, Hayashi et al. [24] deployed machine learning to train several classes and measure performance degradation for Apache servers. This is probably among the very few approaches that uses a non-intrusive approach to gauge performance of a system although they added many servers/PCs to just collect these metrics.

Classification based approaches usually lead to contention free environments, and thus can directly improve performance of a vDC as a whole. These approaches, however, have no intention to understand, model, and/or measure contention, but to effectively avoid it. In fact, they reduce contention only because they reduce the chance of clashing for resources by allocating VMs from different classes on each PM.

2.4 Last Level Cache (LLC) based approaches

Other approaches are also proposed to solve contention from very specific angles. For example, many approaches try to predict/model performance degradation of VMs according to LLC metrics for the Memory; this is mainly because LLC has shown to have a significant impact on the performance of co-located VMs. These approaches mostly focused on reducing conflicts in cache rather than directly modeling/relating LLC to performance degradation.

Efficient resource partitioning [25], throttling [26], and adaptive cache replacement policies [26] are examples of such methods. Because most of these approaches unrealistically demand physical access and/or ability to change hardware designs, more recent ones mostly focus on practical software partitioning techniques [27]. For example, Govindan et al. [25] proposed Cuanta to measure performance degradation of co-located applications (VMs) due to their LLC contention. Microsoft researchers, Roytman et al. [28], introduce PACman to consolidate VMs on PMs considering both their energy consumption and mutual performance degradation. They too identified

cache and memory bandwidth as the two most important metrics to reduce performance degradation of VMs.

The major flaw with these approaches is that they are generally too narrow. In fact, LLC based approaches are not very accurate for general purpose VMs, such as most CPU and disk intensive applications that are usually insensitive to LLC cache misses.

After close examination of many techniques presented to date, we have noticed the following shortcomings. Firstly, many techniques require an agent/code to be injected to a VM to either report its throughput or put pressure on its shared resources. The need to have access to VMs and permission to run tailor-made foreign codes is neither acceptable not practical in most general cases. Secondly, many techniques aim to identify contention so that they can avoid it, not to model and/or measure it. No approach, to the best of our knowledge, has aimed to calculate quantitative sensitivity values for VMs. To date, sensitivity is either defined as binary or the level of pressure a VM can tolerate before violating its QoS. Finally, most approaches do not target multidimensional resource demands.

To address these shortcomings, we designed vmBBProfiler to not only identify, but also quantify sensitivity of VMs to shared resources. vmBBProfiler is an application-agnostic non-intrusive approach that does not require access to VMs to run foreign agents/codes. Using Hypervisor level controls/metrics, it systematically pressures working VMs to model their behavior under pressure.

3 Architecture of vmBBProfiler

The key idea of our solution, vmBBProfiler, is to identify how a VM behaves under resource contention. Its two main components (Fig. 2) are vmProfiler and vmDataAnalyser. The vmProfiler—consists of vmLimiter and vmDataCollector in turn—commands a Hypervisor to impose resource limits to a VM, and collects/records its behavior under the imposed limitations. To further detail vmBBProfiler's components, we refer to our actual deployment using our private VMware [29] based cloud. Other systems such as Xen, KVM, etc. can easily adopt our detailed descriptions to perform similar procedures. They only need to collect similar metrics as we collected through VMware. For example, the '`virsh nodecpustats -percent <vm-name>`' command in KVM will report the cpu utilization of a VM. More comprehensive tools/commands such as '`virt-top --csv <file-name>`' can also be used to collect and store online statistics of running VMs to machine readable CSV (Comma Separated Values) files. vmDataAnalyser of vmBBProfiler can then be launched to use this collected CSV file instead of the one collected by vmProfiler currently designed for VMware.

vmProfiler aims to emulate 'contention' through 'limitation'. That is, instead of challenging a VM to compete with other co-located VMs to access/use shared resources (CPU, Mem, and/or Disk), the vmLimiter limits resource usage of a VM so that it reveals its behavior under hypothetical contentions. For example, vmLimiter commands a Hypervisor to restrict a VM to use only up to 25%, 50%, and 75% of its allocation CPU, Mem, and Disk allowances, respectively. Although resource starvation under contention is fundamentally different than starvation under limitation,

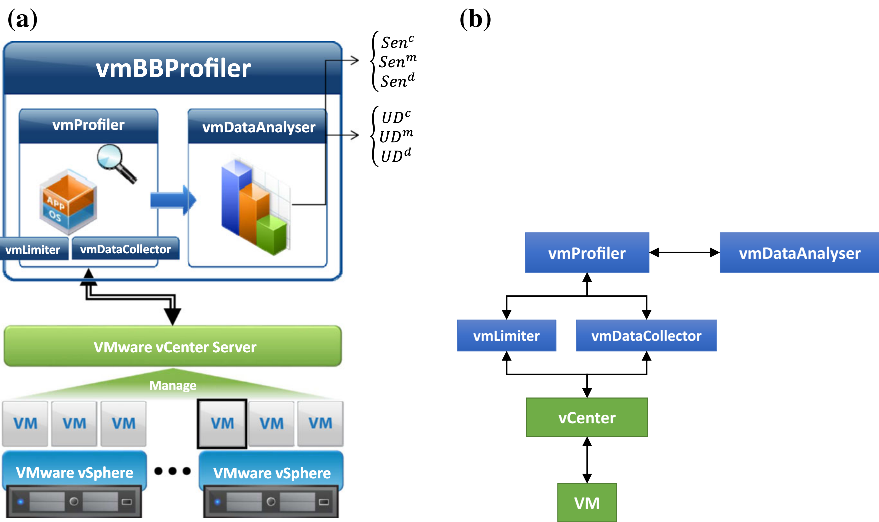

Fig. 2 **a** Architecture of vmBBProfiler and **b** connections among its components

we empirically show that (1) ‘limitation’ can fairly emulate ‘contention’, and (2) it relieves us from encountering too many unknowns, such as the exact usage profile of all other VMs, the exact cache architecture of PMs, etc.

Hypothesis 1 Based on our extensive experiments, we observed and then hypothesized that behavior of a VM in ‘contention’ with other VMs is fairly similar to when it is ‘limited’ to access shared resources. It is based on the fact that an application/VM always needs specific amounts of CPU cycles, Memory bandwidth, and Disk I/O to perform its computation. Any shortage of resources, for any reason, would lead to its performance degradation according to its internal processes. We observed that regardless of why a VM cannot access enough resources, it would always behave the same when performing its internal processes.

To validate this hypothesis, we performed a series of engineered experiments to starve various VMs of resources and reveal their behavior under true contention. Each VM’s behavior is then aligned with its behavior when it was limited to access that specific amount of resources. Figure 3 shows our setup in which VM1 is running a benchmark (eg, aio-stress), while VM2 is producing a predefined amount of load; we used Sys-Bench [30] to consume specific amounts of CPU and Mem, and FIO [31] to perform specific amount of Disk I/O. Both VMs are pinned on the same CPUs/Cores and placed on the same hard-disk to compete for CPU cycles, clash on Memory caches at all levels (L1/L2/L3), and clash on disk buffers. To reflect one of our experiments, Fig. 4 overlaps CPU/Mem/Disk utilization of VM1 for 5 min when running the aio-stress benchmark, while it is using only 25% of its CPU, Mem, and Disk allocations. Lines marked as ‘Limitation’ are to show when resources were available but VM1 was limited to use only 25% of them; lines marked as ‘Contention’ are to show when VM2 is using 75% of resources, and thus VM1 is forced to use only 25%.

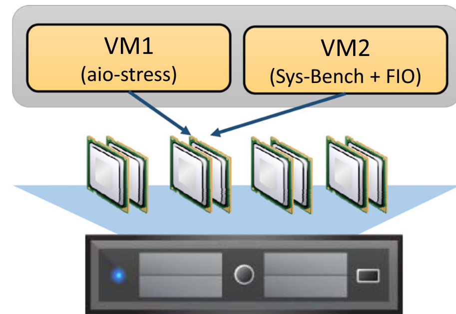

Fig. 3 Limitation versus contention validation setup

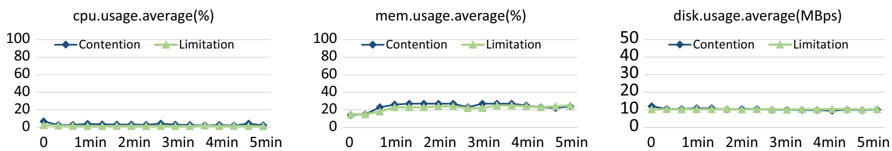

Fig. 4 Resource utilization of aio-stress under 'Limitation' versus 'Contention'

Figure 5 shows the final throughput of aio-stress under other contention scenarios. Performing the same procedure to validate our hypothesis for all benchmarks we used in this article, we observed that VMs' behavior under 'limitation' is always very similar to their behavior under 'contention' because they always produce fairly similar graphs such as the ones in Figs. 4 and 5; in fact, their final throughput differed less than 5% (on average) for all cases. Based on our experiments, we conclude that limitation can fairly represent a VM's behavior under contention, and thus we hypothesize/believe that our deductions in this work can be extended to true contention scenarios. □

It is worth noting that the limitation cannot always emulate contention when applications inside VMs have specific resource usage patterns that can significantly benefit from running inside non-virtualized operating systems; for example, (1) two cache sensitive cpu-intensive applications or (2) two disk-intensive applicators with mostly sequential reads/writes. Our complementary results—not shown here due to page limitation—proves that in both cases, the gap between the final throughput of a VM under limitation to its throughput under contention could in fact raise up to ∼20% for VMs performing sequential read/writes and ∼10% when performing cache-sensitive computations.

vmLimiter `(vmName,cpuLimit,memLimit,diskLimit)` in Fig. 2 directly communicates with a Hypervisor and sets specific limits for each resource so that its under-stressed behavior could be monitored and analyzed. `vmName` is a unique ID to identify a VM; eg, 'VirtualMachine-vm-12' in VMware. `cpuLimit` $\in [0, 1]$ sets the percentage of CPU that the VM can use; if a VM, for example, has two 2.4 GHz vCPUs, `cpuLimit=0.25` would limit its CPU usage to $0.25 \times 2 \times 2.4 = 1.2$ GHz. `memLimit` $\in [0, 1]$ imposes a similar limit to its memory usage. `diskLimit` $\in [0, 1]$ limits the number of IOPs (I/O operations per second); the maximum number of IOPs is related to the storage technology, its buffer size, and its block size. For our experiments with VMware ESXi 5.5, and VMFS version 5.61 on HDDs, we measured

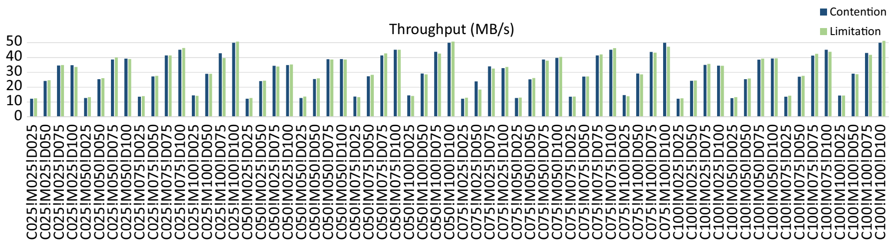

Fig. 5 Throughput of aio-stress under 'Limitation' versus 'Contention'. `Cx!My!Dz` implies using x% of CPU, y% of Memory, and z% of its Disk allowances

Table 1 Sample Hypervisor metrics for a VM

Timestamp	CPU (%)	Mem (%)	Disk (KBps)
6-Jun-2016 17:16:00	0.47	1.65	0.00
6-Jun-2016 17:16:20	10.05	18.32	5.43
6-Jun-2016 17:16:40	24.47	33.62	4.43
…	…	…	…
6-Jun-2016 17:45:20	0.20	3.23	4.23

that up to 2000 IOPs (≈50 MBps disk read/write) can be performed on our HDDs, and thus `diskLimit=0.25` would limit a VM to use up to 500 IOPs for our experiments.

vmDataCollector `(vmName, startTime, finishTime)` collects/records performance of a VM under imposed limitations through polling several Hypervisor level metrics (eg, CPU utilization): it neither demands nor needs any specific metric from the VM itself. This makes vmBBProfiler unique when compared with many other similar approaches that either ask developers to provide specific metrics to reflect a VM's performance [5,6,8,11–13,21], or use peripheral systems to collect external metrics to reflect such performance [24]. Because, the vmBBProfiler remains fully agnostic to the actual internal processes of a VM, we refer to our approach as a 'Black-Box' technique. `startTime` and `finishTime` define the measurement period. Table 1 shows a sample dump for one of our experiments with the default measurement interval of 20 s in VMware-vSphere [29,32].

vmDataAnalyser is invoked upon profiling behavior of a VM under several limitation profiles to analyze the collected data (eg, Table 1) and calculate sensitivity of a VM to its CPU, Memory, and Disk allowances; they are respectively named Sen^c, Sen^m, and Sen^d and are aimed to reflect generalizable conclusions about a VM's sensitivity to cloud resources.

4 Procedures of vmBBProfiler

Figure 6 shows procedural steps of profiling an unknown application/VM to identify its sensitivity to CPU, Mem, and Disk. Profiling can be performed in two modes: offline and online. In the offline mode, it is assumed that an application can be repeatedly

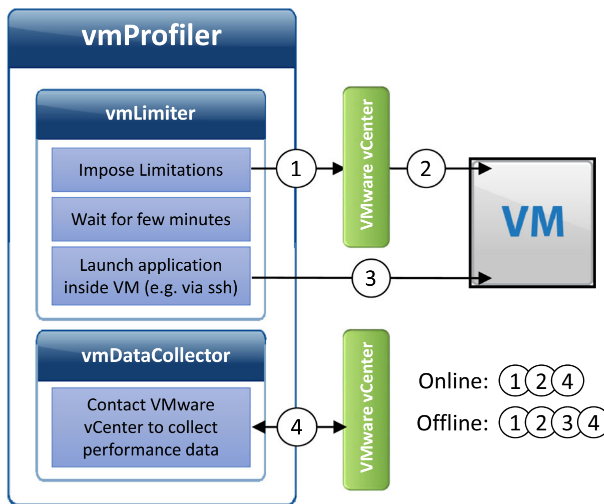

Fig. 6 Procedural steps of vmBBProfiler

started and stopped to perform a set of predefined tasks. In the online mode, the application is assumed running and limitations are imposed to it while it is still in service. The online mode is well aligned with many online services where interruption might not be acceptable. Regardless of the mode, vmBBProfiler needs to challenge a VM to work under pressure so that it can quantify its sensitivity. In fact, accurate results can only be achieved when VMs are pushed to their limits for delivering their services. Thus, for the online mode where service interruption is unacceptable, redundant services must certainly be switched on before using vmBBProfiler.

In the offline mode however, because profiling is performed in a controlled environment, its impact on any cloud/virtualized service is minimal. Nevertheless, unlike similar systems, vmBBProfiler does not demand an isolated PM to perform its profiling. It only needs a PM that is able to comfortably accommodate a target VM: ie, not on over-provisioned PMs that can barely allocate resources to any VM. For example, all our experiments in this work were performed in a production cloud environment where PMs were shared among 50 VMs. We only made certain that, in no circumstances, the average CPU, Mem, and Disk utilization of our PMs exceeded 70% of their capacity (mostly about 40–50%) during our experiments. The safe margin of 70% was enforced because of direct VMware recommendations in practical guidelines [29].

Additional notes: in this article, (1) we will elaborate on the online method first, and then explain how similar procedures could be performed for the offline mode; and (2) we use the words ‘run’ and ‘experiment’ interchangeably.

4.1 Profiling VMs

To profile a VM, a set of profile limitations is first designed and then imposed on a VM one after another. Table 2 shows a sample profiling table in which `cpuLimit∈`

Table 2 Limitation scenarios to stress a VM

Run #	cpuLimit	memLimit	diskLimit
1	c_1	m_1	d_1
2	c_1	m_1	d_2
…	…	…	…
nd	c_1	m_1	d_{nd}
$nd + 1$	c_2	m_1	d_1
…	…	…	…
$nc \times nm \times nd$	c_{nc}	m_{nm}	d_{nd}

$\{c_1, \ldots, c_{nc}\}$, memLimit$\in \{m_1, \ldots, m_{nm}\}$, and diskLimit$\in \{d_1, \ldots, d_{nd}\}$ to produce a total number of $nc \times nm \times nd$ profiling scenarios. Upon spending a specific amount of time on each scenario, vmDataCollector is invoked to collect Hypervisor metrics (eg, Table 1). vmLimiter is invoked again to enforce the next set of limitations, followed by invoking vmDataCollector. This procedure is repeated for all rows of Table 2 to finalize the "profiling" stage.

4.2 Analyzing collected data

After profiling a VM and collecting its metrics for all $nc \times nm \times nd$ different profiling scenarios, vmDataAnalyser is invoked to analyze collected information and produce Sen^c, Sen^m, and Sen^d to reflect sensitivity of a VM to its allocated CPU, Mem, and Disk allocation, respectively. The following steps elaborate on how to calculate Sen^c when cpu/mem/diskLimit $\in \{0.25, 0.50, 0.75, 1.00\}$ (64 scenarios) for one of our benchmarks; $\text{Sen}^{m/d}$ can be calculated following very similar procedures.

4.2.1 Group experiments/runs

The first step is to group different runs based on their cpuLimit. Figure 7 shows such a grouping when running the 'apache' benchmark on a VM (Ubunto 14.04) with 2vCPU (2 × 2.4 GHz), 2 GB of Mem, and 20 GB of Disk on the AMD machine in Table 3. The first column of these sub-figures shows CPU, Mem, and Disk utilization profile of all 64 runs: one run per limitation set according to Table 2. The second column (C025) overlays 16 runs when cpuLimit = 0.25; the third (C050), fourth (C075), and fifth (C100) columns each overlays 16 runs when cpuLimit = 0.50,0.75,1.00, respectively. Overlaying C025, C050, C075, and C100 columns would produce the first column. The x-axis in all sub-figures is time in seconds.

A close look at these sub-figures reveals that 'apache' uses all its allocated CPU allowance, while using very little of memory and disk for all its runs—regardless of its CPU demand. This is why runs are perfectly grouped according to their cpuLimit. Figure 8 shows grouping of these 64 runs according to their memLimit. A visual analysis on sub-figures in Figs. 7 and 8 makes us to speculate that 'apache' is most probably very sensitive to its CPU-share while insensitive to its Memory-shares. This

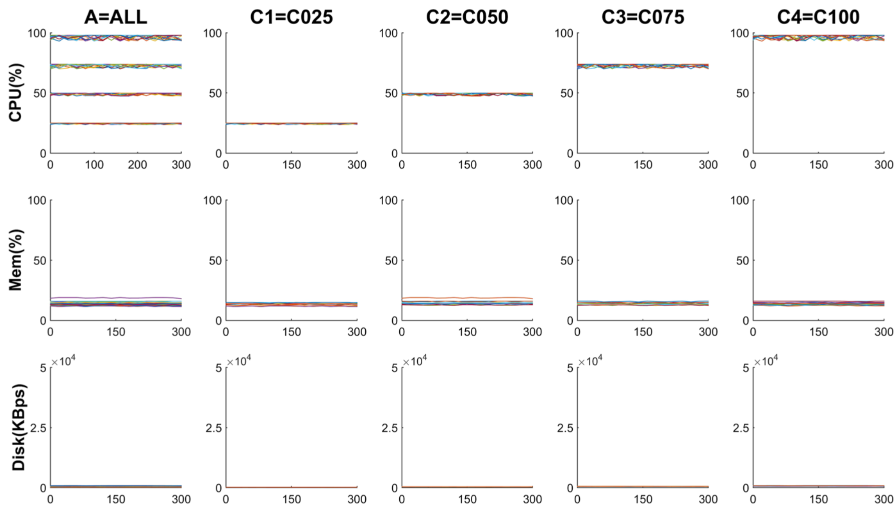

Fig. 7 Grouping 'apache' runs based on `cpuLimit`; x-axis represents time in seconds

Table 3 Characteristics of used physical machines

PM name	CPU family	# Cores (speed)	Memory	Cache (L1/L2/L3)
AMD	AMD Opteron 6282 SE	64 (2.599 GHz)	256 GB	(768 KB/16 MB/16 MB)
DELL	Intel i7-3770	8 (3.40 GHz)	16 GB	(256 KB/1 MB/8 MB)
SGI	Intel Xeon(R) E5420	8 (2.493 GHz)	32 GB	(256 KB/12 MB/–)

is because unlike Fig. 7, different `memLimit` could not perfectly group runs; very similar sub-figures were obtained when grouping runs based on their `diskLimit`. Thus we expect a high Sen^c and low Sen$^{m/d}$; we will confirm our expectation in Sect. 6. It is also worth noting that applications/VMs that are sensitive to multiple resource types (eg, 'bork' in Table 4) grouping runs would not result in clear-cut behavioral patterns as seen for 'apache'.

4.2.2 Calculate coefficient of variance margins

After grouping runs, Coefficient of Variance (CV) margins are calculated for each group. Figure 9 illustrates such CV margins for sub-graphs of Fig. 7. The x-axis represents the normalized times; the y-axis represents the normalized CPU/Mem/Disk utilization.

To calculate CV margins, assume that K runs, $\{r_1, \ldots, r_K\}$, belong to a group (K = 16 in Fig. 9). Also assume that each run, r_x, is composed of three time series: $\{r_x^c, r_x^m, r_x^d\}$ to respectively represent the time series of its CPU, Mem, and Disk utilization. The average (μ), the standard deviation (σ), and the CV (cv) of these K runs are then calculated at specific time slots. For example, $\mu^c_{C025}(t)$, $\sigma^c_{C025}(t)$, and $cv^c_{C025}(t)$ for the C025 group is calculated as:

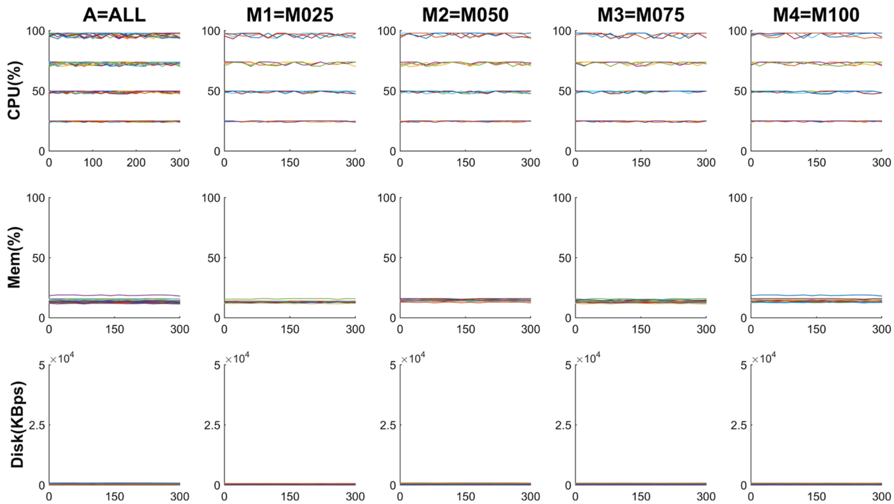

Fig. 8 Grouping 'apache' runs based on `memLimit`; x-axis represents time in seconds

Table 4 Benchmark list

Benchmark	Util.[a]	Short description
apache	H/–/–	Requests per second an apache server can sustain
john-the-ripper	H/–/–	A fast password cracker to detect weak Unix passwords
n-queens	H/–/–	Solves the N-queens problem on a 18 × 18 board
build-apache	H/–/–	Builds an Apache HTTP Server
build-php	H/–/–	Build a PHP-5 with the Zend engine
dcraw	L/–/–	Converts high-resolution RAW NEF image files to the PPM image format
x264	L/–/–	x264 H.264/AVC encoder on a CPU with disabled OpenCL
unpack-linux	L/L/L	Extract the .tar.bz2 Linux kernel package
blogbench	–/H/L	Replicate load (read/write) on a blog with fake content and pictures
bork	–/L/L	A java based utility to encrypt files for long-term storage
compress-gzip	–/L/H	Compress files using Gzip
aio-stress	–/–/H	An a-synchronous I/O benchmark created by SuSE

[a] The utilization (util.) profiles are in CPU/Mem/Disk format for H high, L low, and–negligible

$$\mu^c_{C025}(t) = \frac{1}{K}\sum_{k=1}^{K} r^c_k(t)$$

$$\sigma^c_{C025}(t) = \sqrt[2]{\frac{1}{K-1}\sum_{k=1}^{K}\left(r^c_k(t) - \mu^c_{C025}(t)\right)}$$

$$cv^c_{C025}(t) = \frac{\sigma^c_{C025}(t)}{\mu^c_{C025}(t)}$$

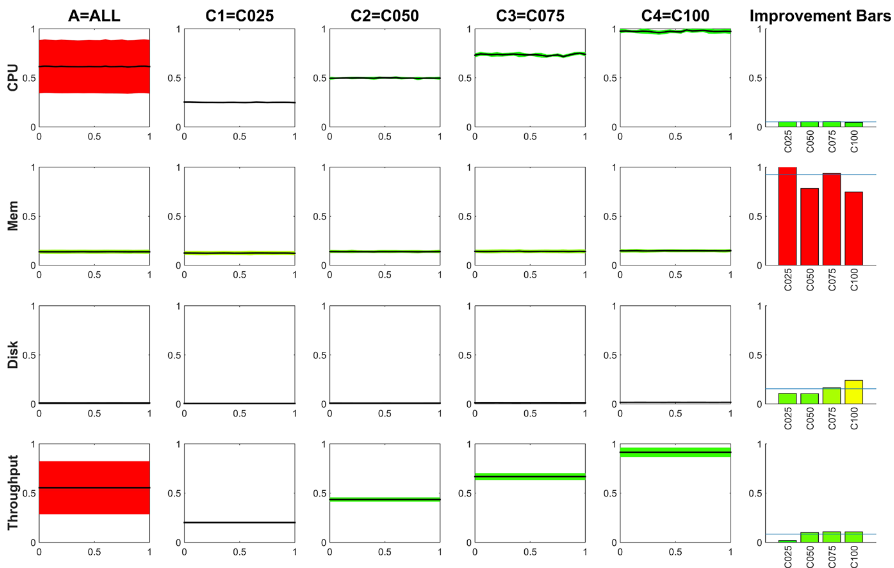

Fig. 9 Color-coded CV margins for 'apache' from *green* (insensitive) to *red* (very-sensitive) (color figure online)

where $t \in [0, 1]$ is the normalized time, $r_k^c(t)$ is the CPU utilization value of the k-th run at time 't'; $r_k^c(t)$ is either directly calculated from original measurements, or interpolated using immediate preceding and subsequent measurements. All values of $\mu_{Cxx}^c(t)$, $\sigma_{Cxx}^c(t)$, and $cv_{Cxx}^c(t)$ for $Cxx \in \{\text{C025, C050, C075, C100, ALL}\}$ can be calculated using similar formulas.

If 't' is incremented in 'Δt' steps; ie, $T = \{0, \Delta t, 2\Delta t, \ldots, 1\}$, then:

$$\mu_{Cxx}^c = \frac{1}{|T|}\sum_{t\in T}\mu_{Cxx}^c(t) \qquad \sigma_{Cxx}^c = \frac{1}{|T|}\sum_{t\in T}\sigma_{Cxx}^c(t) \qquad cv_{Cxx}^c = \frac{1}{|T|}\sum_{t\in T}cv_{Cxx}^c(t)$$

4.2.3 Calculate improvement bars

After calculating all values of μ_{Cxx}^c, σ_{Cxx}^c, and cv_{Cxx}^c for each group, their relative improvement bars are calculated to reflect the "effect" of each grouping based on its resource limitation. The last column of sub-figures in Figs. 9 and 10 show improvement bars for each group. Before delving into mathematical formulas, it is essential to explain the hypothesis/rationale behind them so that they can be better perceived.

Hypothesis 2 It is a fact that if a VM is so sensitive to one of its resources, then starving it of that specific resource should have a major impact on its performance. We hypothesize that through gauging similarity between different runs of an experiment we can detected such phenomenon.

This hypothesis is made after scrutinizing and aligning characteristics of various grouped runs (eg, Figs. 9, 10) with each other. We observed that when a benchmark is

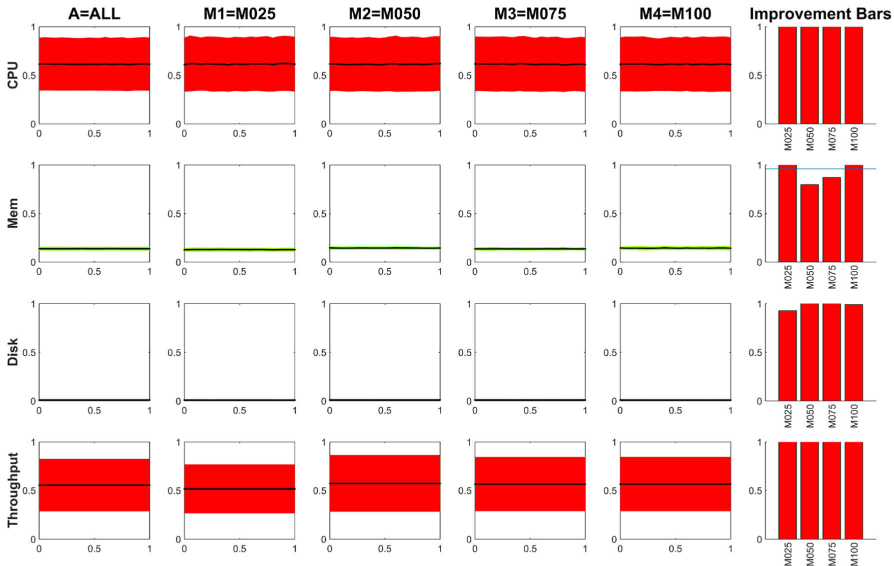

Fig. 10 Memory color-coded CV margins for 'apache'

sensitive to one of its resources, there would be a great similarity between characteristics of grouped runs for that specific resource with the actual throughput of the VM. For example, the first row of sub-graphs in Fig. 9 shows that all runs in each group have the same throughput as can be seen in the last row of these sub-figures. In fact, the more sensitive a VM is to one of its resources (CPU, Mem, Disk) the more similar (bundled) is the overall characteristics of grouped runs based on that specific resource. We used cv to capture the level of such similarity/boundlessness in each group. For 'apache' as an instance, the cv of runs in C025, C050, C075, and C100 (16 runs in each group) groups is much smaller than the average cv for ALL with 64 runs. Such an observation is well aligned with our prior knowledge from the Phoronix Test Suites [33] that 'apache' is very sensitive to its CPU. For better explanation, by comparison, observing Fig. 10 shows that 'apache' is insensitive to its Mem allocation. This can also be observed from cv values in different groups: unlike the previous case, the average cv of all runs in M025-M100 groups does not differ much from the ALL. In other words, variety/irregularity of runs in the M025 groups is not better than the overall variation of all runs at all situations. As a result, 'apache' does not care much about its memory allocation.

It is also worth mentioning that the actual throughput of applications is usually inaccessible because of the need to install agents in VMs. Nevertheless, we could have access to these values for our experiments because Phoronix benchmarks do in fact provide such numbers at the end of their runs. Taking advantage of this opportunity, we used such directly reported throughput values to discover, in a reverse engineer fashion, significant relations and only to build our hypotheses and produce formulas in this articles. For actual deployments, vmBBProfiler does not need to have access to actual throughput of VMs, and thus vmBBProfiler perceives "throughput" as a normalized number, ie $\text{Thr} \in [0, 1]$, where $\text{Thr} = 1$ implies the maximum performance. □

To mathematically capture such behavior and produce the improvement bars (shown in last columns of sub-figures in Figs. 9, 10), we designed the following formula to compute point-by-point improvement of cv for each group.

$$B^c_{Cxx} = \left(\frac{1}{|T|} \sum_{t \in T} \frac{cv^c_{Cxx}(t)}{cv^c_{ALL}(t)} \right) \tag{1}$$

In Sect. 6 we will show that defining improvement bars as Eq. 1 would result in accurate sensitivity values for all sorts/types of VMs.

4.2.4 Calculate performance values and indicators

Using B^c_{Cxx} values, we designed the following formulas to reflect critical performance values and indicators for each benchmark.

$$\text{Sen}^c = 1 - \left(\frac{1}{|Cxx|} \sum_{Cxx} B^c_{Cxx} \right) \tag{2}$$

$$\begin{aligned} \text{UD}^c_{0\sigma} &= \frac{1}{|T|} \sum_{t \in T} \left(\mu^c_{C100}(t) \right) \\ \text{UD}^c_{1\sigma} &= \frac{1}{|T|} \sum_{t \in T} \left(\mu^c_{C100}(t) + \sigma^c_{C100}(t) \right) \\ \text{UD}^c_{2\sigma} &= \frac{1}{|T|} \sum_{t \in T} \left(\mu^c_{C100}(t) + 2 \times \sigma^c_{C100}(t) \right) \end{aligned} \tag{3}$$

$\mathbf{Sen^c} \in [0, 1]$ (Eq. 2) highlights the overall sensitivity of a VM to its CPU allocation. $\text{Sen}^c = 0$ implies that a VM does not show any behavior change when its CPU allowance is changed/limited; $\text{Sen}^c = 1$ implies that the VM shows extremely different resource usage patterns according to its CPU allowance; other values of Sen^c show other levels of sensitivity: the higher the more sensitive. In Sect. 6, we will show that considering sensitivity values for VMs, when co-locating, would have a great impact on the overall performance degradation of all VMs.

$\mathbf{UD^c_{x\sigma}} \in [0, 1]$ (Eq. 3) is to reflect the "usual CPU demand" for a benchmark so that it can comfortably work under different situations and provide its intended services. This value is calculated based on runs in the C100 group (16 runs in Fig. 9) where there is no restriction on the CPU. $\text{UD}^c_{0\sigma}$, $\text{UD}^c_{1\sigma}$, and $\text{UD}^c_{2\sigma}$ respectively compute the average (μ), one standard deviation above the average ($\mu + \sigma$), and two standard deviations above the average ($\mu + 2\sigma$) CPU demands of a VM. In Sect. 6, we will show that reserving resources according to these usual demands yields negligible performance degradation for VMs.

5 Experimental results

To validate vmBBProfiler, we ran ≈1200 h of actual running and profiling benchmarks on our VMware-based private cloud. We used three different PMs (Table 3) and profiled 12 benchmarks (Table 4), varying from pure CPU/Mem/Disk intensive to various combination of CPU + Mem + Disk intensive ones.

5.1 Benchmark selection

Performance and accuracy of vmBBProfiler is evaluated using the Phoronix Test Suite [33] as one of the most comprehensive testing and benchmarking platform; it contains ≈450 test profiles and ≈100 test suites to effectively carry out both qualitative and quantitative benchmarks. Table 4 lists the 12 benchmarks (out of 168 available ones in v5.2.1) we used for our experiments. We deliberately picked benchmarks with different intensities of resource usage (CPU, Mem, and Disk) to cover realistic applications. In this table 'H', 'L', and '–' respectively mean High, Low, and Negligible resource utilization. From the 12 benchmarks, eight run CPU intensive, five run Memory intensive, and five run Disk intensive processes.

5.2 Experimental results

Table 5 shows experimental results of using vmBBProfiler; three rows exist for each benchmark: one row for each PM in Table 3. For our experiments: PowerShell [34] scripts were developed to systematically perform each task and minimize human errors; PowerCLI [35] commands are used to make directly communicate with the VMware-vCenter [32] and poll/record various metrics for VMs. Results are saved into separate csv files; MATLAB (2014b) and Microsoft Excel (2013) are used to read csv files for each benchmark, analyze their data, and produce the figures/graphs in this article.

6 Discussion and analysis

To better analyze data in Table 5, as one of our many tables produced after collecting data during weeks of experiments, we highlight the most stimulating ones in this section.

6.1 Validity of $\text{Sen}^{c/m/c}$

Because no other method exists in the literature with which to directly compare our results, we used statistical analysis on our own collected data to show accuracy as well as validity of vmBBProfiler's calculations. To this end, we first define Sen_τ^c and Corr_τ^c to respectively reflect the sensitivity of an application to its actual throughput and the correlation between actual CPU utilization of an application to its actual throughput. We then show that there are great relations between Sen^c, Sen_τ^c, and Corr_τ^c. Note that Sen_τ^c and Corr_τ^c can only be measured for our benchmarks in this article because

Table 5 Rresults of vmBBProfiler on the selected benchmarks in Table 4

Benchmark	PM	$Sen^c/Sen^m/Sen^d$	$UD^c_{0\sigma}/UD^m_{0\sigma}/UD^d_{0\sigma}$	$UD^c_{1\sigma}/UD^m_{1\sigma}/UD^d_{1\sigma}$	$UD^c_{2\sigma}/UD^m_{2\sigma}/UD^d_{2\sigma}$	$Sen^c_\tau/Sen^m_\tau/Sen^d_\tau$
apache	AMD	0.95/0.00/0.00	0.97/0.14/0.01	0.99/0.16/0.01	1.00/0.18/0.02	0.91/0.00/0.00
	DELL	0.97/0.00/0.00	0.97/0.19/0.02	0.98/0.20/0.02	0.99/0.22/0.03	0.83/0.00/0.00
	SGI	0.97/0.03/0.00	0.97/0.19/0.01	0.98/0.21/0.01	0.99/0.23/0.02	0.97/0.00/0.00
john-the-ripper	AMD	0.93/0.00/0.00	0.86/0.04/0.00	0.89/0.05/0.00	0.92/0.06/0.00	0.79/0.00/0.00
	DELL	0.96/0.00/0.00	0.90/0.03/0.00	0.92/0.04/0.00	0.94/0.05/0.00	0.98/0.00/0.00
	SGI	0.96/0.00/0.00	0.83/0.04/0.00	0.84/0.05/0.00	0.86/0.06/0.00	0.97/0.00/0.00
n-queens	AMD	0.95/0.00/0.00	0.98/0.02/0.00	0.99/0.03/0.00	1.00/0.04/0.00	0.98/0.00/0.00
	DELL	0.97/0.00/0.00	0.97/0.04/0.00	0.99/0.05/0.00	1.00/0.06/0.00	0.99/0.00/0.00
	SGI	0.97/0.00/0.00	0.97/0.03/0.00	0.98/0.04/0.00	1.00/0.04/0.00	0.99/0.00/0.00
unpack-linux	AMD	0.19/0.10/0.40	0.34/0.26/0.29	0.40/0.28/0.36	0.47/0.30/0.42	0.19/0.04/0.40
	DELL	0.21/0.09/0.25	0.30/0.31/0.51	0.39/0.36/0.61	0.49/0.42/0.72	0.15/0.05/0.20
	SGI	0.18/0.09/0.35	0.37/0.30/0.35	0.45/0.33/0.45	0.54/0.36/0.55	0.17/0.09/0.30
build-apache	AMD	0.94/0.00/0.00	0.89/0.17/0.02	0.92/0.18/0.03	0.95/0.20/0.03	0.91/0.00/0.00
	DELL	0.96/0.00/0.00	0.91/0.19/0.04	0.94/0.23/0.05	0.96/0.27/0.06	0.94/0.00/0.00
	SGI	0.96/0.04/0.00	0.87/0.19/0.02	0.88/0.22/0.02	0.90/0.24/0.03	0.97/0.00/0.00
build-php	AMD	0.95/0.02/0.00	0.97/0.25/0.01	0.99/0.28/0.02	1.00/0.31/0.03	0.91/0.00/0.00
	DELL	0.96/0.00/0.00	0.94/0.25/0.03	0.96/0.28/0.04	0.99/0.31/0.05	0.89/0.00/0.00
	SGI	0.97/0.07/0.00	0.96/0.25/0.01	0.98/0.27/0.02	0.99/0.30/0.02	0.94/0.00/0.00
dcraw	AMD	0.54/0.00/0.00	0.44/0.19/0.07	0.47/0.21/0.09	0.49/0.22/0.11	0.78/0.00/0.00
	DELL	0.55/0.00/0.00	0.42/0.22/0.12	0.46/0.24/0.15	0.51/0.25/0.18	0.75/0.00/0.03
	SGI	0.48/0.04/0.00	0.42/0.25/0.07	0.45/0.26/0.08	0.47/0.27/0.10	0.73/0.00/0.00
x264	AMD	0.33/0.01/0.00	0.56/0.27/0.01	0.89/0.31/0.02	1.00/0.34/0.03	0.36/0.18/0.00
	DELL	0.39/0.00/0.00	0.76/0.26/0.02	0.79/0.28/0.06	0.81/0.30/0.11	0.35/0.00/0.00
	SGI	0.41/0.02/0.00	0.62/0.22/0.01	0.88/0.24/0.03	1.00/0.27/0.05	0.44/0.22/0.00

Table 5 continued

Benchmark	PM	$Sen^c/Sen^m/Sen^d$	$UD^c_{0\sigma}/UD^m_{0\sigma}/UD^d_{0\sigma}$	$UD^c_{1\sigma}/UD^m_{1\sigma}/UD^d_{1\sigma}$	$UD^c_{2\sigma}/UD^m_{2\sigma}/UD^d_{2\sigma}$	$Sen^c_\tau/Sen^m_\tau/Sen^d_\tau$
blogbench	AMD	0.09/0.74/0.16	0.18/0.89/0.22	0.40/1.00/0.33	0.61/1.00/0.44	0.04/0.50/0.16
	DELL	0.00/0.75/0.20	0.05/0.70/0.55	0.08/1.00/0.74	0.11/1.00/0.93	0.01/0.48/0.05
	SGI	0.11/0.81/0.18	0.25/0.89/0.13	0.61/1.00/0.18	0.96/1.00/0.22	0.17/0.47/0.20
bork	AMD	0.00/0.47/0.18	0.14/0.45/0.62	0.19/0.72/0.92	0.23/0.99/1.00	0.00/0.59/0.00
	DELL	0.00/0.45/0.09	0.03/0.44/0.31	0.04/0.70/0.45	0.05/0.96/0.59	0.00/0.30/0.00
	SGI	0.00/0.53/0.20	0.07/0.62/0.56	0.11/0.94/0.84	0.14/1.00/1.00	0.00/0.31/0.00
compress-gzip	AMD	0.00/0.00/0.55	0.20/0.30/0.67	0.24/0.31/0.72	0.28/0.33/0.77	0.03/0.00/0.50
	DELL	0.00/0.00/0.45	0.04/0.40/0.21	0.05/0.67/0.24	0.06/0.93/0.27	0.00/0.00/0.50
	SGI	0.00/0.00/0.47	0.13/0.61/0.74	0.21/0.94/0.89	0.28/1.00/1.00	0.00/0.00/0.46
aio-stress	AMD	0.00/0.31/0.84	0.07/0.70/0.66	0.10/0.74/0.73	0.12/0.79/0.79	0.00/0.03/0.75
	DELL	0.00/0.32/0.91	0.02/0.47/0.78	0.03/0.70/0.83	0.04/0.93/0.89	0.00/0.00/0.84
	SGI	0.00/0.30/0.80	0.04/0.74/0.36	0.05/0.80/0.38	0.06/0.86/0.40	0.00/0.10/0.84

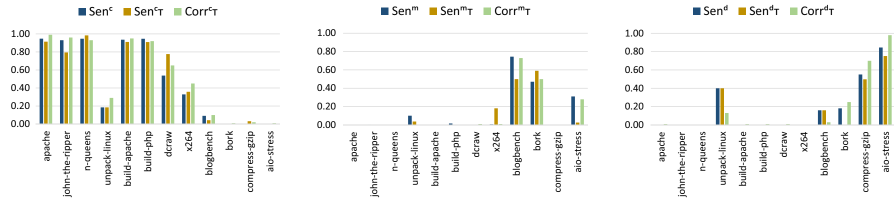

Fig. 11 Bar graph alignment of $\text{Sen}^{c/m/d}$ versus $\text{Sen}_{\tau}^{c/m/d}$ versus $\text{Corr}_{\tau}^{c/m/d}$ for AMD

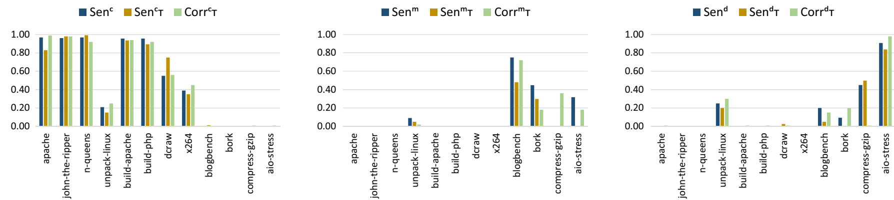

Fig. 12 Bar graph alignment of $\text{Sen}^{c/m/d}$ versus $\text{Sen}_{\tau}^{c/m/d}$ versus $\text{Corr}_{\tau}^{c/m/d}$ for DELL in Table 3

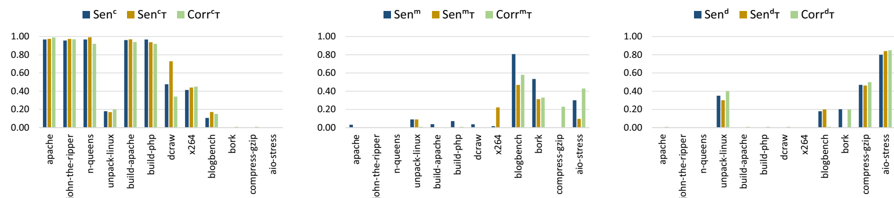

Fig. 13 Bar graph alignment of $\text{Sen}^{c/m/d}$ versus $\text{Sen}_{\tau}^{c/m/d}$ versus $\text{Corr}_{\tau}^{c/m/d}$ for SGI in Table 3

Phoronix provides the actual throughput of each benchmark. Nevertheless, by showing that they are fairly aligned with Sen^c, which its calculation is independent of the actual throughput measurement of applications, we validate the procedures for computing Sen^c in Eq. 2. Similar to Eqs. 1–2 we defined:

$$B_{Cxx}^{\tau} = \left(\frac{1}{|T|} \sum_{t \in T} \frac{cv_{Cxx}^{\tau}(t)}{cv_{ALL}^{\tau}(t)} \right) \tag{4}$$

$$\text{Sen}_{\tau}^{c} = 1 - \left(\frac{1}{|Cxx|} \sum_{Cxx} B_{Cxx}^{\tau} \right) \tag{5}$$

$$\text{Corr}_{\tau}^{c} = \frac{E[(X - \mu_X) \times (Y - \mu_Y)]}{\sigma_X \sigma_Y} \tag{6}$$

where $\overline{r_k^c} = \frac{1}{|T|} \sum_{t \in T} r_k^c(t)$; $\overline{r_k^\tau} = \frac{1}{|T|} \sum_{t \in T} r_k^\tau(t)$; $X = \left[\overline{r_1^c}, \overline{r_2^c}, \ldots, \overline{r_K^c}\right]$; and $Y = \left[\overline{r_1^\tau}, \overline{r_2^\tau}, \ldots, \overline{r_K^\tau}\right]$. $\mu(\cdot)$, $\sigma(\cdot)$, and $E(\cdot)$ respectively represent the average, the standard deviation, and the expected value of a vector.

Figure 11 illustrates and aligns bar graph representation of $\text{Sen}^{c/m/d}$, $\text{Sen}_{\tau}^{c/m/d}$, and $\text{Corr}_{\tau}^{c/m/d}$ for all benchmarks in Table 4 for the AMD machine in Table 3. As can be

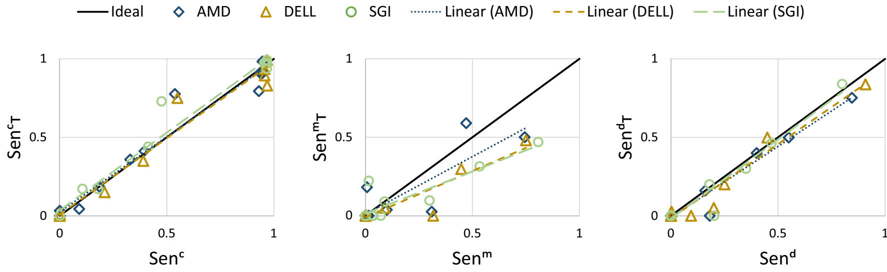

Fig. 14 Point-by-point alignment of $\text{Sen}^{c/m/d}$ versus $\text{Sen}_\tau^{c/m/d}$

observed, there is a very tight alignment between these values across all benchmarks and all resource types for this PM (Figs. 12, 13). Note that although $\text{Sen}_\tau^{c/m/d}$, and $\text{Corr}_\tau^{c/m/d}$ are only obtainable for our benchmarks in this articles, their very tight alignment with $\text{Sen}^{c/m/d}$ conductivity proves not only usefulness but also the accuracy of Eqs. 1–2 to quantify the true sensitivity signature of applications/VMs to shared resources. Very similar alignments/results were observed for DELL and SGI in Table 3; their graphs are not shown because of page limitation. Figure 14 provides a point-by-point representation of aligning $\text{Sen}^{c/m/d}$ with $\text{Sen}_\tau^{c/m/d}$ to also prove that these two values are fairly aligned with each other.

6.2 Accuracy/usefulness of $\text{Sen}^{c/m/d}$

6.2.1 Accuracy

Figure 11 show different levels of accuracy for different resource types. Knowing that $\text{Sen}_\tau^{c/m/d}$ truly reflect sensitivity of an application to resources, the closer the values of $\text{Sen}^{c/m/d}$ to their $\text{Sen}_\tau^{c/m/d}$ counterparts, the more accurate they are. Figure 14 also depicts the "Ideal" line where all ($\text{Sen}^{c/m/d}$, $\text{Sen}_\tau^{c/m/d}$) pairs must be laid on to have a perfect alignment and achieve the highest accuracy. According to sub-figures of 14, we can conclude that $\text{Sen}^{c/d}$ are fairly aligned with their $\text{Sen}_\tau^{c/d}$ counterparts. Table 6 reflects the average, minimum, and maximum distance of all points to the "Ideal" line in each category. Sen^m showed less accuracy across all benchmarks. Scrutinizing the results and diving deeper into possible sources of this inaccuracy, we identified the following two reasons.

The first reason is related to the caching structure (L1/L2/L3) of PMs. Memory structure has always been seen as one of the most influential resource types in causing performance degradation for co-located VMs. Having less accurate Sen^m—as compared with $\text{Sen}^{c/d}$—is well aligned with other findings in this field and is the first source of such inaccuracy. Figure 14b shows Sen^m for the AMD is slightly better than the other two, most probably because it has larger L1/L2/L3 caches (Table 3). The second reason is related to the virtualization itself. To illustrate this reason, we used our validation setup in Fig. 3 in which VM1 was switched off and VM2 was programmed to use 75% of CPU (5.1 GHz), Memory (1.5 GB), and Disk (1500 IOPs, 37.5 MBps)

Table 6 Euclidean dist. of (Sen^c, Sen^c_τ) points to the "Ideal" line in Fig. 14

PM	$\|(Sen^c, Sen^c_\tau), Ideal\|_2$			$\|(Sen^m, Sen^m_\tau), Ideal\|_2$			$\|(Sen^d, Sen^d_\tau), Ideal\|_2$		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
AMD	0.04	0.00	0.17	0.05	0.00	0.20	0.02	0.00	0.13
SGI	0.03	0.00	0.14	0.05	0.00	0.22	0.03	0.00	0.11
DELL	0.03	0.00	0.18	0.07	0.00	0.24	0.02	0.00	0.14

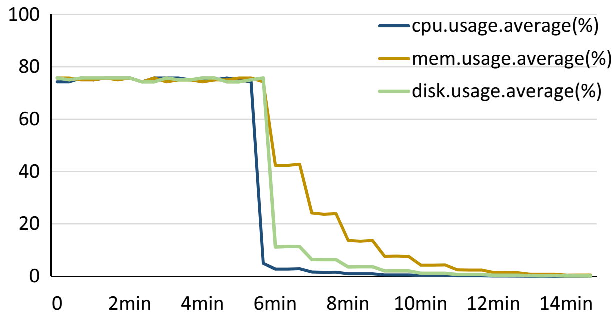

Fig. 15 Cool down process of DELL after 5 min of heavy CPU, Mem, and Disk usage

for the duration of 5 min. Figure 15 depicts CPU, Mem, and Disk utilization of VM2 during and after this 5 min. As can be seen in Fig. 15, VMware-vSphere [29] treats memory differently than CPU and Disk because CPU and Disk utilization of VM2 are immediately dropped to ≈5–10%, while it takes almost 5 min for its Memory to cool down. This simple experiment shows that the VMware-ESXi Hypervisor is very conservative when taking memory from VMs—as compared with CPU and Disk. According to VMware documentation and guidelines, this conservative process would eventually result in higher productivity of VMs because it results in fewer cache-misses for VMs. This conservative view also leads to less accurate measurements for actual memory usage of applications/VMs. This is because Hypervisor metrics still record memory cleaning-up procedures of a VM as "activity", while its inside application is actually not using any memory.

It is also worth noting that vmBBProfiler is designed to profile sensitivity of in-service VMs with fairly steady load. That is, if the targeted VM is an 'apache' server for example, its request rate per second should stays fairly steady (eg, ≈1000 requests per second) during the whole profiling period. To check this concern, we checked vmBBProfiler for unsteady loads also and noticed that it can tolerate up to 15% load deviation during the profiling phase.

6.2.2 *Usefulness*

To show the usefulness of using $Sen^{c/m/d}$, we conducted a series of contention scenarios similar to the one in Fig. 3; here seven VMs with different sensitivity profiles

Table 7 Consolidation scenarios

Scenario	VMs	$\sum \text{Sen}^c / \sum \text{Sen}^m / \sum \text{Sen}^d$	$\sum \text{CPU} / \sum \text{Mem} / \sum \text{Disk}$	APD
1	3 × dcraw, 2 × unpack-linux, 1 × compress-gzip, 1 × aio-stress	1.63/1.69/1.67	2.51/2.99/2.40	0.50
2	3 × dcraw, 2 × blogbench, 2 × unpack-linux, 1 × compress-gzip	1.99/0.51/2.20	2.26/ 2.10/2.30	0.62
3	3 × dcraw, 3 × unpack-linux, 1 × aio-stress	2.18/0.61/2.05	2.40/2.06/1.92	0.71

APD average performance degradation

and resource demands are pinned on the same CPU/Cores and placed on the same hard-drive to compete with each other to access CPU cycles, clash on all levels of cache, and buffer reads/writes to the same disk. Table 7 shows three scenarios to place these seven VMs on the AMD machine while their collective CPU, Mem, and Disk utilization were respectively capped to 2×2.6 GHz, 2 GB, and 50 MBps.

Scenario #1 shows a balanced view of co-locating VMs with $\sum \text{Sen}^c \approx \sum \text{Sen}^m \approx \sum \text{Sen}^d \approx 1.65$. Here, the total summation of normalized CPU, Mem, Disk demands from the Hypervisor is respectively 2.51 ($2.51 \times 2 \times 2.6$ GHz), 2.99 (2.99×2 GB), and 2.40 (2.40×50 MBps) times the capacity of this PM. In other words, these seven VMs reflect an overbooking of 2.51, 2.99, and 2.40 on CPU, Mem, and Disk, respectively. Similar to throughput, Performance Degradation (PD)—also known as the 'Slow-Down' factor—is defined as a normalized number to reflect how a VM's functionality is effected. $\text{PD} = 0$ implies no performance degradation when compared with throughput of an isolated VM; $\text{PD} = 1$ implies absolute performance degradation with almost no productivity of a VM; $\text{PD} = 0.5$ implies the VM is performing at almost half of its full capacity. The last column of this table shows that the Average Performance Degradation (APD) of all VMs in this scenarios is 50% of what could have been achieved if these VMs were placed in contention-free environments. Scenario #2 shows a slightly skewed placement where $\sum \text{Sen}^c = 1.99$ and $\sum \text{Sen}^d = 2.20$ are greater than $\sum \text{Sen}^{c/d}$ in Scenario #1, while $\sum \text{Sen}^m = 0.51$ is less than $\sum \text{Sen}^m$ in Scenario #1. As can be seen, although the total normalized summation of CPU, Mem, and Disk is less than those of the placement in Scenario #1, they suffer from a higher APD (12% more). Scenario #3 presented an even more skewed version where $\sum \text{Sen}^{c/m/d}$ values are more imbalanced. In this case, APD is even worst than Scenario #2, although VMs in this placement still collectively demand less resources than those of Scenario #1.

This simple example empirically showed that different placements could ask for the same amount of resources, while encounter different levels of performance degradation. Using sensitivity values in this study, we showed the importance of considering VMs' $\text{Sen}^{c/m/d}$ values when making placement decision. It is worth noting that VM placement is an NP-Complete problem and is beyond the scope of our work in this article. In a follow up study, we showed that, for the same amount of power, using sensitivity values to place VMs can lead to $\sim$10% performance improvement in cloud data centers [36].

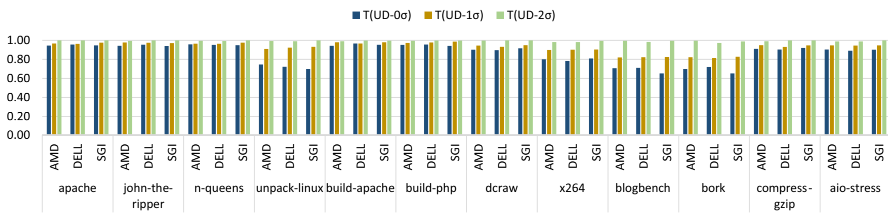

Fig. 16 Relative throughput of benchmarks when resources are reserved according to $\mathrm{UD}_{0\sigma}^{c/m/d}$, $\mathrm{UD}_{1\sigma}^{c/m/d}$, and $\mathrm{UD}_{2\sigma}^{c/m/d}$

6.3 Accuracy/usefulness of $\mathbf{UD}_{0\sigma}^{c/m/d}$, $\mathbf{UD}_{1\sigma}^{c/m/d}$, and $\mathbf{UD}_{2\sigma}^{c/m/d}$

Besides calculating resource sensitive values for VMs, vmBBProfiler also computes three "usage demand" values for each VM. These were calculated to reflect three different levels of conservatism to assure minimum performance degradation for VMs. To show that reserving resources according to these values would in fact lead to minimum performance degradation, we conducted another series of engineered experiments using our validation setup in Fig. 3. Figure 16 reflects the actual throughput of all applications when hypervisor reserves $UD_{0\sigma/1\sigma/2\sigma}^{c/m/d}$ amount of resources for VM1 (in Fig. 3), while running a background load in VM2 (90% CPU, 90% Mem, and 90% Disk).

Figure 16 shows that reserving resources according to $\mathrm{UD}_{0\sigma}^{c/m/d}$ would lead to an average throughput of 86% for all benchmarks. In this case, while throughput of most benchmarks is above 90% of their isolated runs, a few benchmarks such as blogbench (65%) could significantly underperform. Reserving resources according to $\mathrm{UD}_{1\sigma}^{c/m/d}$ values is more conservative, and consequently has greater throughput values across all benchmarks: 93% on average with the minimum of 80%. The most conservative reservation recommendation of $\mathrm{UD}_{2\sigma}^{c/m/d}$ leads to 99% of average throughput with the minimum of 97%.

Besides their overall usage to reserve resources to ensure minimum performance degradation of applications/VMs, these values also highlight how resources could be traded to assure desired throughput values per VM. For example, allocating 20–25% more CPU to blogbench could significantly improve its performance from 80 to 97%, while it would not improve performance of other benchmarks such as aio-stress. It is worth noting that making resource reservation decisions according to $\mathrm{UD}_{0\sigma/1\sigma/2\sigma}^{c/m/d}$ to achieve VM-by-VM performance assurance is also another NP-Complete problem and beyond the scope of this study as well.

6.4 The effect of vmBBProfiler on running applications

In this section, we elaborate on how performance of a running application is affected during the profiling procedures of vmBBProfiler. Table 8 reflects the average throughout of each benchmark during the whole profiling phase of vmBBProfiler. As can be expected, VMs have different levels of throughput when going through differ-

Table 8 vmBBProfiler effect on benchmarks

Benchmark	PM	Throughput		
		Average	Min	Max
apache	AMD	0.58	0.21	1.00
	DELL	0.63	0.25	1.00
	SGI	0.65	0.26	1.00
john-the-ripper	AMD	0.69	0.23	1.00
	DELL	0.71	0.31	1.00
	SGI	0.77	0.36	1.00
n-queens	AMD	0.60	0.22	1.00
	DELL	0.63	0.25	1.00
	SGI	0.63	0.25	1.00
unpack-linux	AMD	0.75	0.36	1.00
	DELL	0.59	0.29	1.00
	SGI	0.59	0.29	1.00
build-apache	AMD	0.66	0.26	1.00
	DELL	0.63	0.25	1.00
	SGI	0.67	0.27	1.00
build-php	AMD	0.61	0.23	1.00
	DELL	0.65	0.25	1.00
	SGI	0.64	0.26	1.00
dcraw	AMD	0.83	0.41	1.00
	DELL	0.81	0.37	1.00
	SGI	0.83	0.47	1.00
x264	AMD	0.64	0.09	1.00
	DELL	0.66	0.24	1.00
	SGI	0.58	0.16	1.00
blogbench	AMD	0.74	0.40	1.00
	DELL	0.75	0.17	1.00
	SGI	0.53	0.25	1.00
bork	AMD	0.47	0.09	1.00
	DELL	0.73	0.26	1.00
	SGI	0.45	0.09	1.00
compress-gzip	AMD	0.87	0.64	1.00
	DELL	0.64	0.42	1.00
	SGI	0.54	0.13	1.00
aio-stress	AMD	0.61	0.25	1.00
	DELL	0.60	0.24	1.00
	SGI	0.77	0.46	1.00

ent limitation scenarios (Table 2). Sometimes, vmBBProfiler has no effect on their throughput, while they could significantly struggle at other times. Nevertheless, as explained before, accurate computations can only be achieved if applications/VMs are pushed/challenged to perform under severe limitations. Therefore, if performance degradation of a profiling VM is absolutely unacceptable, parallel services/VMs must be switched on during the whole profiling phase to minimize downside effects of vmBBProfiler.

In practice, vmBBProfiler did not significantly impact the average relative throughput ($\approx$60%) of any application/VM during its profiling phase of $\approx$6 h:30 m ($64 \times 5\,\text{m} + 63 \times 1\,\text{m} = 383\,\text{m} \approx 6\,\text{h:}30\,\text{m}$ (64 limitation scenarios plus 1m transition between two consecutive scenarios). We also tested both shorter and longer profiling periods to gauge how the accuracy of vmBBProfiler would have been affected should a different number of profiling points were collected/used. Using the default measurement interval of 20 s in VMware-vCenter [29,35], vmBBProfiler used $5\,\text{m}/20\,\text{s} \times 64 = 15 \times 64 = 960$ measurement-points for each metric per benchmark. We also noticed that using more measurement-points has little effect on our results. This is mainly because VMs are usually under steady loads during operations, and thus even 5 min of observing is quite enough to discover their behavior under pressure, and consequently judge their sensitivities. To make certain about our claim however, we also all benchmarks for 1, 2, 3, 4, 10, 15, and 20 m intervals as well. We found that profiling periods less than 3 m usually leads to fragile situations where the smallest fluctuations during profiling could lead to outliers and untrustworthy measurement-points; profiling periods of 10 m and above were absolutely unnecessary because they lead to identical results to those computed with 5 m profiling periods. Thus, we conclude that a profiling period of 5 m is probably the shortest possible period to not only produce accurate results, but also detect measurement outliers should they occur during profiling.

6.5 Transferability of results across PMs

One of the most valuable attributes of vmBBProfiler is its ability to detach findings for a VM from its underlying PM. Table 5 and Figs. 11, 12 and 13 show the striking similarity of results despite their totally different PMs: one AMD, one Intel Xeon, and one Intel-i7. Results look more similar between the two Intel machines (DELL and SGI) than across the Intel and the AMD machine. Although it is perfectly justifiable because of their internal CPU architecture, it is still very desirable that even results from profiling on an AMD machine can be fairly generalized on an Intel machine, and vice versa. This phenomenon also proves that procedures of vmBBProfiler so efficiently target the application inside the VM that a change of PM has a little effect on them.

6.6 vmBBProfiler in the offline mode

For the offline mode, where an application with predefined tasks can be repeatedly launched, an extra step to launch an application inside a VM needs to be added (Fig. 6). To compare results, we also launched vmBBProfiler in the offline mode for all bench-

marks in Table 4; results were astonishingly similar to those in Table 5 that we decided to only reflect results of the online mode in this article.

6.7 vmBBProfiler limitations

vmBBProfiler in its current design has a few limitations that need to be carefully considered to compute accurate sensitivity values for VMs. Firstly, the VM under the profiling procedure must remain under a fairly steady workload. This is rational, because if a VM changes its behaviour during the profiling phase, it is inherently changing its nature as well. For example, assume a VM is responding to three different types of requests: 20% type A, 40% type B and 40% type C. In this case, as long as these requests are sent with a more or less same distribution, it would have negligible affect on the profiling. However, if the VM receives %100 type A for few minutes/hours followed by %100 type B for another few minutes/hours, then it can be inferred that the VM is actually performing a different activity; this would be similar to having a different VM altogether. Secondly, resources must be fairly available for the PM that is hosting the under profiling VM. This is because, the observed performance degradation of a VM cannot be confidently related to any specific resource otherwise. For example, if a PM is using 95% of its CPU during the profiling phase of one of its hosted VMs, we cannot confidently identify the source of performance degradation for the VM as it could also be the result of CPU saturation of the PM.

7 Conclusion and future directions

In this work, we presented vmBBProfiler to identify and quantify the sensitivity of general purpose VMs to their allocated CPU, Memory, and Disk resources. vmBBProfiler consists of two parts: vmProfiler and vmDataAnalyser to respectively impose resource limitations on VMs and analyze their collected profiled data. After validating that contention can be fairly emulated by limitation, we imposed many variations of resource limitations to VMs to disclose their behavior under contention.

Using 12 well known benchmarks to cover all sorts of possible applications, vmBBProfiler managed to successfully identify and quantify sensitivity of VMs to their CPU, Memory, and Disk allocations. vmBBProfiler was implemented on our VMware-vSphere based private cloud and proved its efficiency across 1200 h of empirical studies. vmBBProfiler is the first Black-Box Profiler, to the best of our knowledge, that uses only basic Hypervisor level metrics for its very systematic calculations. Besides sensitivity values for each resource, vmBBProfiler also produces three usage demand values to guide cloud administrators how to reserve resources for important VMs. Through engineered experiments we also showed the importance of considering (1) sensitivity values in allocating/packing VMs on PMs, and (2) usage demands to guarantee minimum performance degradation of VMs.

To continue this work, we would like to (1) design smart strategies to check just enough limitation scenarios—e.g., not all 64 scenarios in our case—to significantly shorten vmBBProfiler's profiling time, (2) design algorithms/heuristics to place and/or migrate VMs according to their sensitivity profiles—our preliminary results show

at least 20% performance improvement for small vDCs, (3) add other dimensions (eg, network) to vmBBProfier; considering vmBBProfiler's very modular structure, it would be very straight forward, yet needs careful planning as every dimension introduces its own characteristics/complexities, and (4) extent vmBBProfiler to also profile linked/cascaded systems (eg, 3-tier web applications and Hadoop clusters).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Banga G, Druschel P, Mogul JC (1999) Resource containers: a new facility for resource management in server systems. In: Proceedings of the third symposium on operating systems design and implementation, OSDI '99, pp 45–58
2. Chen H, Wang S, Shi W (2011) Where does the power go in a computer system: experimental analysis and implications. In: 2011 International green computing conference and workshops, pp 1–6
3. Mars J, Tang L, Hundt R, Skadron K, Soffa ML (2011) 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, pp 248–259
4. Tang L, Mars J, Vachharajani N, Hundt R, Soffa ML (2011) The impact of memory subsystem resource sharing on datacenter applications. In: Proceedings of the 38th annual international symposium on computer architecture, pp 283–294
5. Kundu S, Rangaswami R, Dutta K, Zhao M (2010) Application performance modeling in a virtualized environment. In: The sixteenth international symposium on high-performance computer architecture, pp 1–10
6. Xu J, Zhao M, Fortes J, Carpenter R, Yousif M (2008) Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. Clust Comput 11(3):213. doi:10.1007/s10586-008-0060-0
7. Rao J, Bu X, Xu CZ, Wang L, Yin G (2009) VCONF: a reinforcement learning approach to virtual machines auto-configuration. In: Proceedings of the 6th international conference on autonomic computing, pp 137–146
8. Nathuji R, Kansal A, Ghaffarkhah A (2010) Q-clouds: managing performance interference effects for QoS-aware clouds. In: Proceedings of the 5th European conference on computer systems, pp 237–250
9. Watson BJ, Marwah M, Gmach D, Chen Y, Arlitt M, Wang Z (2010) Probabilistic performance modeling of virtualized resource allocation. In: Proceedings of the 7th international conference on autonomic computing, pp 99–108
10. Chiang RC, Huang HH (2011) TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In: 2011 International conference for high performance computing, networking, storage and analysis (SC), pp 1–12
11. Wang L, Xu J, Zhao M, Fortes J (2011) Adaptive virtual resource management with fuzzy model predictive control. In: Proceedings of the 8th ACM international conference on autonomic computing, pp 191–192
12. Bartolini DB, Sironi F, Sciuto D, Santambrogio MD (2014) Automated fine-grained CPU provisioning for virtual machines. ACM Trans Archit Code Optim (TACO) 11(3):27
13. Delimitrou C, Kozyrakis C (2013) iBench: quantifying interference for datacenter applications. In: 2013 IEEE international symposium on workload characterization (IISWC), pp 23–33
14. Akoush S, Sohan R, Rice A, Moore AW, Hopper A (2010) Predicting the performance of virtual machine migration. In: 2010 IEEE international symposium on modeling, analysis and simulation of computer and telecommunication systems, pp 37–46
15. Doyle RP, Chase JS, Asad OM, Jin W, Vahdat AM (2003) Model-based resource provisioning in a web service utility. In: Proceedings of the 4th conference on USENIX symposium on internet technologies and systems, vol 4, p 5

16. Bennani MN, Menasce DA (2005) Resource allocation for autonomic data centers using analytic performance models. In: Second international conference on autonomic computing (ICAC'05), pp 229–240
17. Liu X, Zhu X, Singhal S, Arlitt M (2005) Adaptive entitlement control of resource containers on shared servers. In: 2005 9th IFIP/IEEE international symposium on integrated network management, 2005. IM 2005, pp 163–176
18. Padala P, Shin KG, Zhu X, Uysal M, Wang Z, Singhal S, Merchant A, Salem K (2007) Adaptive control of virtualized resources in utility computing environments. SIGOPS Oper Syst Rev 41(3):289. doi:10.1145/1272998.1273026
19. Dai Y, Yang L, Xing H, Zhang B (2013) Predicting performance interference of application in virtualized environments. In: Sixth international conference on machine vision (ICMV 2013), pp 90672B–16
20. Chen X, Rupprecht L, Osman R, Pietzuch P, Franciosi F, Knottenbelt W (2015) CloudScope: diagnosing and managing performance interference in multi-tenant clouds. In: 2015 IEEE 23rd international symposium on modeling, analysis, and simulation of computer and telecommunication systems, pp 164–173
21. Caglar F, Shekhar S, Gokhale A (2014) Towards a performance interference-aware virtual machine placement strategy for supporting soft real-time applications in the cloud. In: Proceedings of the 3rd international workshop on real-time and distributed computing in emerging applications, pp 15–20
22. Caglar F, Shekhar S, Gokhale A (2013) A performance interferenceaware virtual machine placement strategy for supporting soft realtime applications in the cloud. Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, Tech. Rep. ISIS-13-105
23. Qian Z, Tung T (2012) A performance interference model for managing consolidated workloads in QoS-aware clouds. In: 2012 IEEE fifth international conference on cloud computing, pp 170–179
24. Hayashi T, Ohta S (2014) Performance degradation detection of virtual machines via passive measurement and machine learning. Int J Adapt Resil Auton Syst (IJARAS) 5(2):40
25. Govindan S, Liu J, Kansal A, Sivasubramaniam A (2011) Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In: Proceedings of the 2nd ACM symposium on cloud computing, pp 22:1–22:14
26. Jaleel A, Hasenplaugh W, Qureshi M, Sebot J, Steely Jr. S, Emer J (2008) Adaptive insertion policies for managing shared caches. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques, pp 208–219
27. Blagodurov S, Zhuravlev S, Fedorova A, Trans ACM (2010) Contention-aware scheduling on multicore systems. Comput Syst 28(4):1
28. Roytman A, Kansal A, Govindan S, Liu J, Nath S (2013) PACMan: performance aware virtual machine consolidation. In: Proceedings of the 10th international conference on autonomic computing (ICAC 13), pp 83–94
29. VMware-vSphere (2016) Vmware-vsphere (www.vmware.com/products/vsphere/)
30. Sys-Bench (2016) Sys-bench (www.manpages.ubuntu.com/manpages/utopic/man1/sysbench.1.html)
31. FIO (2016) Fio (www.manpages.ubuntu.com/manpages/natty/man1/fio.1.html)
32. VMware-vCenter (2016) Vmware-vcenter (www.vmware.com/products/vcenter-server)
33. Phoronix (2016) Phoronix test suite (www.phoronix-test-suite.com/)
34. PowerShell (2016) Microsoft powershell (msdn.microsoft.com/en-us/mt173057.aspx)
35. VMware-PowerCLI (2016) Vmware-powercli (www.vmware.com/support/developer/powercli/)
36. Nasim R, Taheri J, Kassler AJ (2016) Optimizing virtual machine consolidation in virtualized datacenters using resource sensitivity. In: 2016 IEEE international conference on cloud computing technology and science (CloudCom), pp 168–175