vCPU as a Container: Towards Accurate CPU Allocation for VMs

Li Liu
George Mason University
Fairfax, Virginia, USA
lliu8@gmu.edu

Haoliang Wang
Adobe Research
San Jose, California, USA
hawang@adobe.com

An Wang
Case Western Reserve University
Cleveland, Ohio, USA
axw474@case.edu

Mengbai Xiao
The Ohio State University
Columbus, Ohio, USA
xiao.736@osu.edu

Yue Cheng
George Mason University
Fairfax, Virginia, USA
yuecheng@gmu.edu

Songqing Chen
George Mason University
Fairfax, Virginia, USA
sqchen@gmu.edu

Abstract

With our increasing reliance on cloud computing, accurate resource allocation of virtual machines (or domains) in the cloud have become more and more important. However, the current design of hypervisors (or virtual machine monitors) fails to accurately allocate resources to the domains in the virtualized environment. In this paper, we claim the root cause is that the protection scope is erroneously used as the resource scope for a domain in the current virtualization design. Such design flaw prevents the hypervisor from accurately accounting resource consumption of each domain. In this paper, using virtual CPUs as a container we propose to redefine the resource scope of a domain, so that the new resource scope is aligned with all the CPU consumption incurred by this domain. As a demonstration, we implement a novel system, called VASE (**v**CPU **as** a contain**er**), on top of the Xen hypervisor. Evaluations on our testbed have shown our proposed approach is effective in accounting system-wide CPU consumption incurred by domains, while introducing negligible overhead to the system.

CCS Concepts • **Social and professional topics → Pricing and resource allocation**; • **Software and its engineering → Virtual machines**; **Scheduling**; **Cloud computing**.

Keywords CPU Accounting, Virtual I/O, Scheduling, Cloud Computing

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313814

ACM Reference Format:
Li Liu, Haoliang Wang, An Wang, Mengbai Xiao, Yue Cheng, and Songqing Chen. 2019. vCPU as a Container: Towards Accurate CPU Allocation for VMs. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313814

1 Introduction

The adoption of cloud computing has become increasingly popular among various Internet services. Cloud computing enables the flexible provisioning and sharing of computing resource between multiple tenants. The underlying virtualization technologies provide an isolated protection mechanism for the states and executions of each virtual machine (VM, or domain). However, such isolation is not currently well-considered when it comes to the resource usage incurred by the guest domains. Consequently, some guest domains may be able to consume significantly more resource than allocated, or *resource overuse*, as we refer to it in this paper.

Among various types of resource, the CPU is the most important one and its accurate allocation and management directly affects the operations and the revenue of the cloud providers like Amazon and Google. According to [16], each physical CPU core sells for a maximum potential annual revenue of $900. However, as previous works [8, 29] have observed and we will further demonstrate in Section 3.1, a guest domain can consume up to 70% more of its allocated CPU time, preventing the cloud providers to sell those 70% overused CPU to other clients and resulting in a noteworthy monetary loss. The resource overuse issue may also potentially degrade the performance of neighbor domains [3, 9, 15, 25, 26, 30, 34, 38] and increase the energy consumption of the host machines [14, 37].

Similar concerns over the resource overuse issue have been raised previously in the context of non-virtualized environment for processes [4] and for containers [22]. In this work, we investigate the problem in the virtualized environment, which both imposes unique challenges compared to previous works and also provides new opportunities to

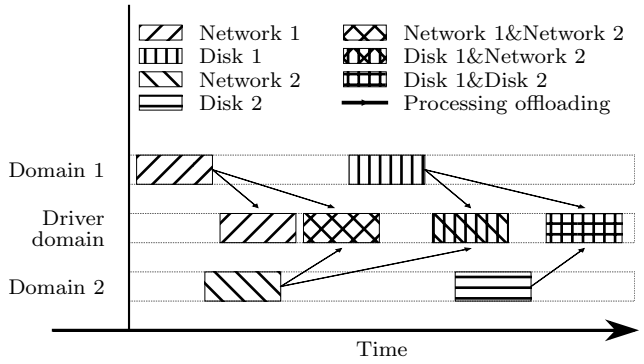


Figure 1. Illustration of software-based I/O virtualization: much of I/O processing is offloaded to the driver domain, which is not accounted to its source domain (domain-1/domain-2). Such offloading processing: 1) is asynchronous with processing in the source domain; and 2) interleaves with each other in the driver domain.

enable a more accurate and lightweight solution compared to existing ones in non-virtualized environments.

In virtualized environment, one major contributing factor of the resource overuse problem is the use of *offloading* in software-based I/O virtualizations. An illustrative example is shown in Figure 1. With the software-based I/O virtualization, I/O devices are managed by the driver domain (or the hypervisor) and guest domains share those I/O devices through the driver domain. When guest domains perform I/O operations, a significant portion of the I/O processing workload is offloaded to the driver domain. However, with the current CPU resource accounting scope, the CPU usage incurred by those offloaded processing in the driver domain is not correctly accounted to its source domain. As a result, through burdening the driver domain, the guest domains may effectively consume more CPU resource. The case also applies to the shared intrusion detection system [24] between VMs, where a similar offloading mechanism is used.

Solving the resource overuse problem relies on an accurate accounting of the offloaded processing, which is a challenging task, especially in virtualized environment. The semantic gap between the hypervisor and domains and the asynchronous nature of the offloaded processing pose significant challenges for either the hypervisor or the domains to accurately measure the offloaded CPU usage. Previous works attempted to overcome such semantic gap using VM-introspection techniques [21, 24, 31], at the cost of complicated kernel tracing and heavy runtime overhead which limits its usage in modern cloud systems. Others [19, 32] attempted to circumvent these challenges by estimating, instead of accurately measuring, the offloaded CPU time. Such estimation is based on the assumption that offloaded CPU usage for the same workload is always the same, which, as we will show in Section 3.2, is not necessarily true and therefore the estimation-based approach also fails to produce accurate accounting result.

In this paper, we claim that the root cause of this problem lies in the design of virtualization systems: *the protection scope of a domain is erroneously used as its resource scope during resource accounting and management.* The protection scope of a domain isolates its states and executions from other domains, while the resource scope of a domain should contain all the resource consumption incurred by this domain. In many cases, for instance the I/O offloading, these two scopes are not aligned with each other. Such coincidence in the current design prevents hypervisor from correctly allocating resource to each domain.

In this work, we aim to tackle the problem in the virtualized multi-tenant cloud environment by re-aligning the CPU resource scope of a domain with its actually-incurred CPU usage, so that accurate resource allocation can be enforced for all guest domains. Specifically, we redefine the resource scope for a domain, so that all the offloaded CPU consumption is included within its resource scope. The new resource scope for a guest domain is comprised of a combination of virtual CPUs from not only that domain but also the driver domain. In the driver domain, all the offloaded processing from a source domain is contained and encapsulated in the corresponding vCPUs, which are contained in the resource scope of that source domain. Therefore, the resulting resource scope of a domain contains all the incurred CPU consumption and can be used by the hypervisor to accurately manage the CPU resource per domain.

To demonstrate our proposed approach, we implement *VASE System*, a novel and light-weight solution built on top of the Xen hypervisor. The evaluations in various settings show that our approach is able to effectively manage the system-wide CPU consumption incurred by the guest domains with virtually no overhead.

To summarize, the contribution of our paper is three-fold:

- We distinguish the resource scope from the protection scope in virtualized systems and redefine the resource scope of a domain using existing vCPU abstraction, enabling accurate resource management per domain.
- Our solution encapsulates all the CPU consumption incurred by a domain to designated vCPUs contained inside its resource scope. The asynchronous and interleaved offloaded processing in the driver domain can be accurately measured and debited to its source.
- By exploiting existing vCPU abstraction, our approach eliminates explicit communications between the hypervisor and domains, and hence its associated overhead compared to approaches like kernel tracing or VM-introspection. The hypervisor scheduler can effectively control the system-wide CPU consumption incurred by a domain with virtually no overhead.

The rest of this paper is organized as follows: in Section 2, we will provide the background on I/O virtualization and CPU management in Xen. The motivation of this work will

be presented in Section 3, with experiments showing the drawbacks of state-of-art, followed by the problem statement, challenges and solution in Section4. The design of our *VASE System* will be presented in Section 5, followed by evaluations in Section 6. Discussions and related work will be provided in in Section 7 and 8. We will conclude this paper in Section 9.

2 Background

2.1 I/O Virtualization

In virtualized environments, the *hypervisor* is responsible for managing and allocating hardware resources to VMs running on top of it. To provide guest VMs access to I/O devices like network and disk, three approaches are commonly available: 1) software-based virtualization approach where the hypervisor and guest VM cooperate to handle I/O requests; 2) full emulation approach; and 3) IOMMU-assisted pass-through approach. The software-based virtualization approach has gained popularity in practice since on the one hand, it has significant performance advantage compared to the full emulation approach; on the other hand, it also has better management flexibility with minimum additional overhead compared to hardware-assisted approach [5, 20].

In the Xen virtualized environment, VMs are known as *domains*. During booting, the first domain loaded by the Xen hypervisor is referred to as *Domain 0*, or *Dom0* for short, which has elevated privileges to manage resources and other guest domains. Those unprivileged guest domains are called *domain U*, or *DomU*. The Xen hypervisor itself does not include device drivers. Instead, it delegates hardware support to a special driver domain (usually Dom0) by exposing hardware access to that domain. Xen has implemented the *split driver* model for network and block device I/Os. Figure 2 illustrates the process of a DomU sending packets. During such I/O process, most of the CPU time is consumed by those components: physical device driver, *backend* and *frontend* of the split driver, TCP/IP stack, and event channel.

With Xen's split driver model, clearly the I/O requests initiated by or destined for one DomU will also consume CPU resources in Dom0. We denote this amount of CPU time consumed by Dom0 for serving the I/O workloads in DomUs as *the offloaded CPU time*. In the next section we will briefly summarize how CPU resources are managed and how software-based virtualized I/O design may potentially result in inaccurate resource management.

2.2 CPU Management in Xen

In the Xen virtualized environment, CPU resources are managed through Xen schedulers. The default *Credit scheduler* in Xen allocates CPU resources in terms of *credit*. Each domain has its own amount of credit and a domain with positive remaining credit will be prioritized to run its vCPUs on the physical CPUs (pCPUs). There are two parameters: *weight* and *cap* set for each domain that determine its allocation of

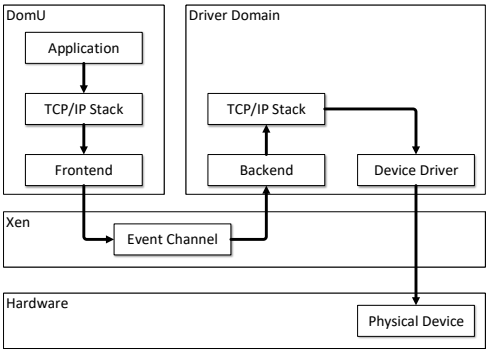


Figure 2. The path of sending a packet in Xen [10]. In the split driver model, a large portion of I/O processing happens in the driver domain, consuming a significant amount of CPU resource on behalf of the DomUs.

credit. The *weight* determines the allocation ratio between each domain when the system is oversubscribed, e.g., a domain with a weight of 512 may receive twice as much as credit of a domain with a weight of 256. The *cap* is used to limit the absolute amount of CPU time a domain may consume. For example, a domain with *cap* 250 may receive at most 2.5 pCPUs. The default *cap* value for domains is 0, which means its CPU usage is not capped, indicating a work-conserving mode. *Cap* is an important parameter for cloud providers to control resource allocation to domains. For example, Varadarajan et al. [34] reported that Amazon EC2 instances are capped. Based on the two parameters, the scheduler periodically allocates the credit to each domain. When a vCPU runs, it consumes the credit of its domain. As shown in previous section, the executions of I/O workload in DomUs require service from Dom0, which the hypervisor scheduler is totally unaware of. As a result, the DomU may effectively incur more CPU usage than allocated, causing performance degradations and variations [8, 19, 32].

3 Motivation

Suppose we have a Xen virtualized environment where only one DomU is running I/O workload with its cap set to 100. Intuitively, we would expect the *total system-wide* CPU utilization to be no more than 100% (of one CPU core). However, as suggested in previous sections, due to I/O offloading, the current Xen scheduler is unable to constrain the *real* CPU usage incurred by each domain. In this section, we will investigate how significant such excessive usage is and show the drawbacks of the estimation-based approaches [8, 19, 32].

For all the experiments in this paper, we use an HP ProLiant DL380 G6 server as our testbed, which is equipped with two Intel Xeon E5540 CPUs. To minimize the dynamics within the system, features including hyperthreading, turbo boost, and dynamic power management are all disabled. The second CPU socket is also left idle at all time to eliminate

Table 1. Workload Configurations using *sysbench* and *iperf3*

	Workload	Description	Parameters
sysbench	CPU	Primality test using trial division	*–test=cpu –num-threads=1*
	MEM	Allocate & randomly write to memory buffer	*–test=memory –num-threads=1*
	SEQ (Disk)	Sequentially read pre-allocated files	*–test=fileio –file-test-mode=seqrd –num-threads=1*
	RND (Disk)	Randomly read/write pre-allocated files	*–test=fileio –file-test-mode=rndrw –num-threads=1*
iperf3	TCP (Network)	Generate random TCP traffic to a remote host	*-l 128 -P 1 -b 96M*
	UDP (Network)	Generate random UDP traffic to a remote host	*-l 128 -P 1 -b 96M -u*
	MIX	Read data from disk and send it via network	*-l 128 -P 1 -b 96M -u -F ~/test_file.0*

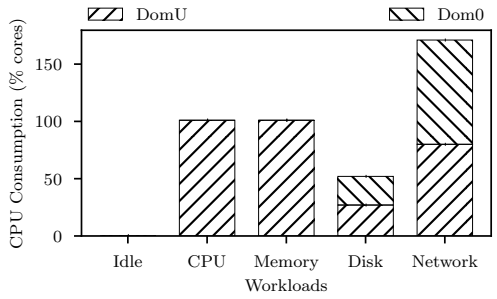


Figure 3. System-wide CPU usage with one DomU running various workloads. The total CPU usage reaches more than 170% (of a CPU core) in case of I/O intensive workloads, exceeding the amount of 100% that is allocated to the DomU. The exceeding part is "stolen" from other domains.

the Non-uniform memory access (NUMA) effect between sockets. Unless explicitly stated, Dom0 is configured to have eight vCPUs while each DomU is configured to have one vCPU. The CPU consumption of vCPUs is obtained using the Xen toolstack. Dom0 runs Ubuntu 16.04 and Xen 4.9.0 is used as the hypervisor. On each DomU, *iperf3* [33] and *sysbench* [23] are used to generate synthetic workloads listed in Table 1. We use these workloads here and also in Section 6. CPU utilization is measured by running the workload and collecting the active vCPU time over a ten-second period.

3.1 Offloaded CPU Time is Significant

First, we verify that the offloaded CPU time is non-negligible. To this end, we generate four types of workload in one DomU: CPU, MEM, SEQ, and UDP, as listed in Table 1. The result is shown in Figure 3, where the y-axis represents the total system CPU usage incurred by the workload in DomU. We can see from the figure that for Idle, CPU, and MEM workload in DomU, the total system CPU usage is within the allocated resource limit (100%). However, for I/O-intensive workloads – Network (UDP) and Disk (SEQ), the CPU usage in Dom0 becomes significant, especially for Network workload where Dom0's CPU usage surged to 90%, and total system CPU usage incurred by DomU's workload reaches more than 170%. Although it is well-expected that the paravirtualized I/O will incur some overhead in Dom0, it is not expected that such excessive CPU usage can be as much as 90% of the allocated amount. Hence, the implication from this experiment is that, such a significant offloaded CPU usage in Dom0 must be properly accounted for, or a large amount of CPU time may be "stolen" from Dom0, resulting in significant monetary loss and performance degradation for other domains, and extra energy consumption for physical host.

Since our experiment is conducted in a controlled environment where Dom0 does nothing but processing offloaded work from a single DomU, we can therefore use Dom0's total CPU usage as the offloaded CPU usage. However, obtaining the offloaded usage for a specific DomU is challenging in reality. The state-of-the-art [8, 19, 32] addresses this issue based on estimations and we will show its drawbacks next.

3.2 Estimation Approach is Inaccurate

To study the accuracy of the estimation approach in determining offloaded CPU consumption, we have replicated the profiling and estimation technique proposed in [8, 19, 32]. This approach is built on the assumption that the same workload *always* incurs the same amount of offloaded CPU consumption. So, by profiling the offloaded CPU consumption for a certain workload once, the offloaded CPU consumption incurred by this workload in the future can be estimated based on the amount of data transmitted. However, we claim this assumption is not true in a multi-tenant cloud environment, as activities from co-located DomUs pose significant interferences. The following experiments demonstrate the extent of estimation inaccuracy and prove our proposition.

To start with, we show the profiling, as a key step in the estimation-based approach, cannot generate stable result in the multi-tenant cloud environment. We profile the same TCP workload while running various combinations of DomUs in the same host. We repeat the profiling for 100 times and the result is shown in Figure 5, where the x-axis represents different DomUs configuration, e.g., N1C2 means one DomU running Network workload and the other two running CPU workloads, and the y-axis represents the profiling result for each combination. The result clearly shows that the profiling varies largely between different combinations, e.g., 4 times between N1C2 and N1C3.

Next, we demonstrate the estimation error by selecting one of the profiling results in the previous step to estimate

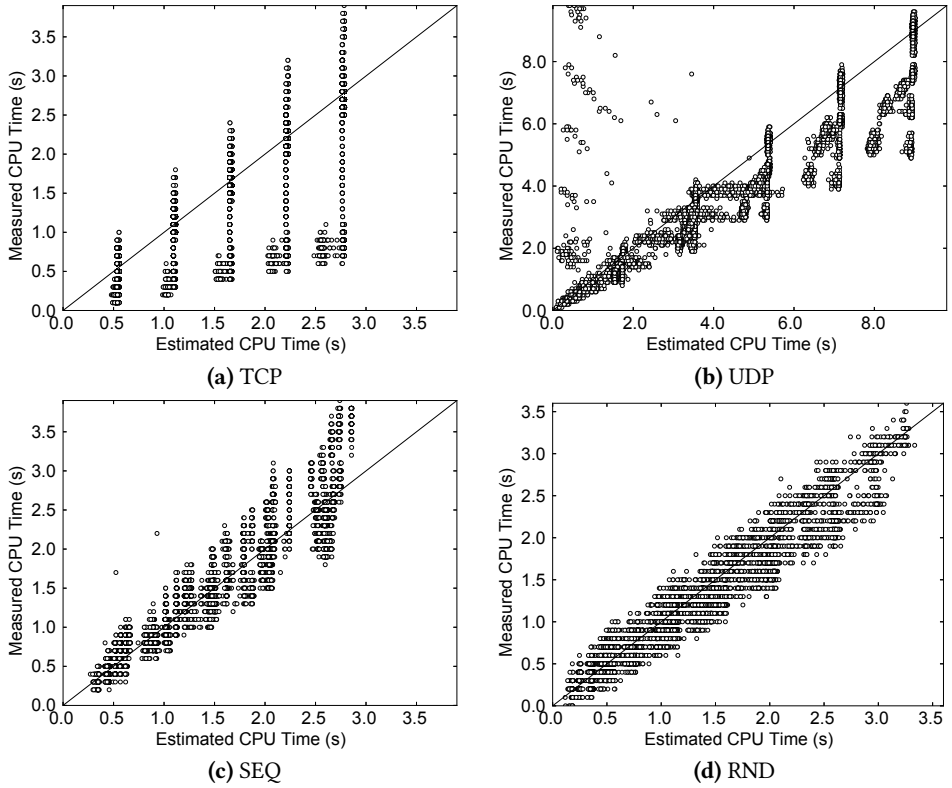


Figure 4. The true and estimated CPU time of the driver domain when running I/O intensive workloads. The diagonal line represents an accurate estimation. The estimation approach yields up to 79% errors for network and 50% for disk.

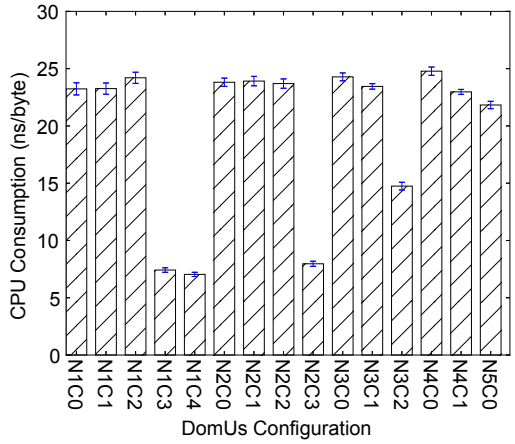


Figure 5. The profiling result of the estimation approaches, which proves it is false to assume the same workload always incurs the same amount of offloaded CPU consumption to the driver domain, especially with neighbors.

the offloaded CPU usage and compare it with the measured value. The experiments are conducted by varying the number of co-located DomUs, the DomUs configuration and its execution time. The result for TCP, UDP, SEQ and RND workload are shown in Figure 4a, 4b, 4c, and 4d, respectively. The x-axis represents the estimated CPU time and the y-axis represents the measured CPU time. The diagonal line indicates an accurate estimation. We can see the estimation errors are significant — up to 79% in case of TCP.

As we can conclude from the result above, the estimation-based approach cannot produce accurate accounting for offloaded CPU usage in multi-tenant settings and therefore cannot enforce resource allocation in the cloud. The dynamic neighbor interferences in multi-tenant systems are volatile and difficult to accurately predict, which motivates us to develop a direct and accurate approach to solve this issue.

4 Problem, Challenges and Our Solution

We have shown that, for the I/O-intensive workload, the offloaded CPU usage is significant and dynamically changing, while the estimation-based approaches cannot address the accounting issue. In general, an accurate CPU allocation relies on an accurate accounting of the offloaded CPU consumption, referred in this paper as *debt*. Hence, the first question we want to answer in this paper is that, *can we directly and accurately measure the debt (offloaded CPU time) and use such accounting information to enforce CPU resource allocation with minimum overhead?*

Accurately measuring each domain's debts is a challenging task. As shown in Figure 1, it is non-trivial to separate the offloaded I/O processing of each guest apart because their executions are all asynchronous and interleaved in the driver

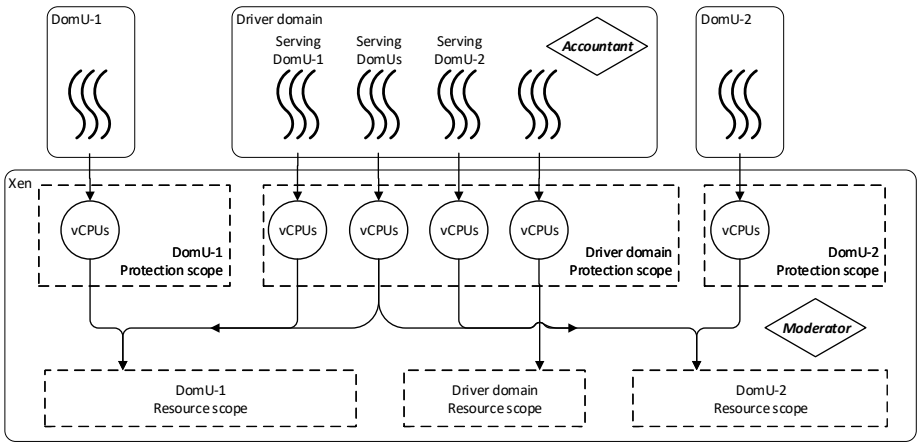


Figure 6. The overall design of *VASE System*. The resource scope of each domain is defined by the actual resource consumption of that domain as opposed to falsely defined by the protection scope. The existing vCPU abstraction is used as resource container to isolate and encapsulate the offloaded workload. *VASE System* enables such resource scope using two major components: 1) An *Accountant* component in the driver domain; and 2) A *Moderator* component in the Xen hypervisor.

domain. In addition, even though such processing could be traced and measured through extensive and costly kernel tracing, the driver-domain-reported duration is nonetheless inaccurate. This is because, in the middle of two timestamps, the underlying hypervisor may perform context switches that are transparent to the domains. Finally, the hypervisor, which is capable of accurately and thoroughly measuring the CPU runtime, is helpless in this case as the offloaded processing to be measured runs in the driver domain. In summary, the semantic gap between the hypervisor and domains and the asynchronous nature of the offloaded processing pose significant challenges for both the hypervisor and the domains to accurately measure the offloaded CPU usage.

Before diving deep into how to overcome those obstacles and measure all the debts, we wonder *what the fundamental cause behind the existence of the debt is*. In the example of Figure 1, a part of I/O processing is offloaded to the driver domain, since the I/O devices is in protection scope of the driver domain. Unfortunately, in the current design of the virtualization systems, the same protection scope is used as the resource scope of a domain. Any processing executed within the its protection domain is accounted as its CPU consumption. As a consequence, the debt is not accounted correctly to its source domain, but incorrectly to the driver domain. In general, debt exists whenever some processing is offloaded outside the protection scope of the source domain.

Hence, to fundamentally address the issue, we need to re-define the resource scope of a domain in the virtualization systems so that all the processing incurred by a guest domain will be contained and managed within its new resource scope. Now the next question is: *what the resource scope should be comprised of*. We note that physical CPU resource is consumed by vCPUs on behalf of domains, since vCPU is the abstraction of execution state of processing in domains. Thus, we seek to redefine the resource scope of a domain using the existing vCPU abstraction, and distinguish it from its protection scope. All the offloaded processing from different source domains is distinguished and pinned to dedicated vCPUs in the driver domain. Then we define the new resource scope of a domain by: (1) all the vCPUs in the domain, and (2) all dedicated vCPUs serving offloaded processing in the driver domain. With this new resource scope, the hypervisor can effectively limit the system-wide CPU usage of each domain. In the next section, we implement *VASE System* to demonstrate our solution.

5 *VASE System*

In this section, we present *VASE System* which consists of two components: the *Moderator* in the Xen hypervisor and the *Accountant* that cooperates in the driver domain. Without loss of generality, our approach is based on Xen 4.9.0 and Ubuntu 16.04. Recent Xen and Linux versions also feature the same mechanisms used in this paper.

The overall design of *VASE System* is shown in Figure 6. As aforementioned, the resource allocation problem is caused by the ill-defined resource scope of the domains in the current virtualization design. Our approach is therefore designed to create the correct resource scope in the hypervisor and let the hypervisor perform per-domain resource accounting and management based such resource scopes.

In a nutshell, the correct resource scope is established as follows: in the driver domain, the *Accountant* exploits the Linux kernel and device driver design to encapsulate and isolate the offloaded workload from each DomU into designated vCPUs. Subsequently, the *Moderator* in the Xen hypervisor is able to accurately learn the *exact* offloaded CPU usage of each DomU with negligible overhead, compared with

kernel tracing or estimation-based techniques used in previous approaches. With the offloaded usage from each DomU accurately obtained, the *Moderator* then enforces resource allocation by adjusting the credits in the Xen scheduler.

5.1 The *Accountant* Component

The *Accountant* runs in the driver domain and facilitates the *Moderator* for accurate runtime measurements. The *Accountant* is responsible for encapsulating and isolating the offloaded workload from each DomU into designated vCPUs in the driver domain. Before we explain how this is achieved, we first briefly introduce how Linux kernel and its device drivers handle I/O processing.

Device Driver Handling in Linux Device drivers generally follow the *top-half* + *bottom-half* scheme, which is designed to minimize the time spent in the interrupt handler and process longish tasks asynchronously. The top-half is a piece of concise code called the *Interrupt Service Routine* (ISR) which is triggered when the system receives hardware interrupts. It executes only the minimum necessary operations to schedule the corresponding bottom-half, and returns as soon as possible. The bottom-half performs the concrete I/O work and can be implemented in Linux with mechanisms including *softirq*, *tasklet*, and *workqueue*. In the meantime, helper threads spawned by device drivers may also be signaled to facilitate the bottom-half processing concurrently.

Based on such a design principle, we can see that all the I/O-related processing including the offloaded ones is handled by entities including ISRs, *softirq* handlers, *tasklets*, *kworkers* and other driver-specific kernel threads in Linux. In other words, the granularity of the I/O-related processing is at the thread/IRQ handler level. We will refer to these kernel threads and IRQ handlers as *workers* hereinafter. This indicates, instead of tracing through the entire I/O processing at the function level, we have the opportunity to distinguish the offloaded workload by isolating the few workers used by each DomU to separate vCPUs in the driver domain. Also, using vCPUs as the means of encapsulation provides additional advantages — now that the *Moderator* in the hypervisor can directly measure the usage of the offloaded workload by looking at the runtime of each vCPU, which means: 1) negligible overhead compared to explicit communication between the driver domain and the hypervisor; and 2) highest possible accuracy since the measurement is done in the hypervisor.

For the rest of Section 5.1, we demonstrate how the I/O-related processing workers in the driver domain can be identified and isolated to designated vCPUs according to the DomUs they are serving. As a proof of concept, we run two DomUs (DomU-1 and DomU-2), both are configured to have a network device and a block device. All information used below is stored in the driver domain and the Xen hypervisor, and is accessible from Dom0. Dom0 is also the driver domain in this example, and the workflow is the same when a dedicated driver domain is used.

5.1.1 Identify the Workers

With careful examinations of the design and code path of Linux kernel and Xen split-driver, we have implemented the *Accountant* to identify all offloaded I/O processing workers in our test environment. Figure 7 shows these IRQs and kernel threads for block and network backend device drivers. Each device type (network or block) has two components — backend driver and real device driver (per paravirtualized I/O design). Each driver component has workers including IRQ handlers and/or kernel threads to carry its workload (per Linux driver design). With these workers identified, the next step is to group them by their corresponding source DomU. This can be done by hooking into the Xen tool stack at domain creation. The dotted box in Figure 7 shows how the workers are grouped into three categories: 1) workers serving DomU-1 only; 2) workers serving DomU-2 only; and 3) workers serving both DomU-1 and DomU-2.

5.1.2 Isolate Workers to vCPUs

With all the related workers grouped by their source domains, the next step is to have the CPU time consumed collectively by those groups accurately measured in the hypervisor. As aforementioned, vCPU runtime can be utilized to overcome the semantic gap and establish an implicit communication between the driver domain and hypervisor. If we can have the driver domain bind the execution of the identified workers on designated vCPUs, accurate offloaded CPU consumption can be immediately obtained by the hypervisor. The *Accountant* achieves such a goal by manipulating scheduling and IRQ affinities in the driver domain OS.

The workers in the driver domain consist of kernel threads and IRQ handlers. For kernel threads, the *Accountant* modifies the CPU affinity settings in the scheduler to pin the kernel threads in the driver domain to designated vCPUs. For I/O-related IRQs, similarly, the *Accountant* sets their affinity to selected vCPU by changing interrupt handling settings in Linux. Note that here we only need to set the affinity of the hardware IRQs (hence the top-half), without touching any bottom-half mechanisms. The reason is that bottom-half scheduled by the top-half will always be executed on the *same* CPU where the task is originally scheduled [36]. Hence, once we have pinned the hardware IRQs, the rest of the processing will be executed on the same vCPU. Table 2 shows an example configuration for two DomUs with network and disk devices. In this example, all the network processing offloaded by DomU-1 can be measured by reading the runtime of vCPU-3 and 5. Similarly, we can also get the offloaded CPU consumption for DomU-2 disk processing by checking the runtime of vCPU-2 and 6.

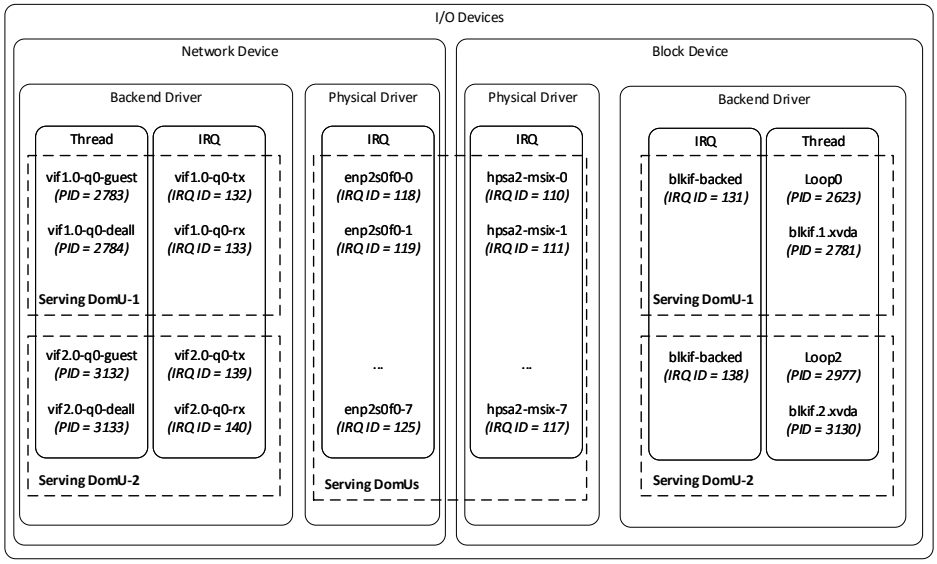


Figure 7. Identification of related workers in the driver domain for each domain. In this example, a network device and a block device are allocated to both DomU-1 and DomU-2. Thread "vif1.0-q0-guest", with PID = 2783, serves the virtual network device in DomU-1. IRQ "enp2s0f0-0", with IRQ ID = 118, serves the physical network device in the driver domain. Those workers can be grouped by the domains they serve: 1) serving only DomU-1; 2) serving only DomU-2; and 3) serving both.

5.1.3 Tweak Load Balancing

A potential issue with affinity is that the setting is not mandatory. The driver domain OS may, for load balancing or other purposes, migrate the pinned workers to another CPU through the scheduler or utilities like *irqbalance*. Hence, for a persistent pinning configuration, we need to prevent the load balancing facilities in the driver domain from dismantling our affinity settings. Meanwhile, we cannot simply disable load balancing facilities completely as we want them to continue balancing other irrelevant workloads in the driver domain. To this end, for schedulers, we utilize the *isolcpus* option in Linux kernel, which isolates given vCPUs from the driver domain scheduler so that the scheduler will not schedule any processes on the isolated vCPUs unless explicitly being asked by users. Similarly, for IRQs, we isolate vCPUs through the `IRQBALANCE_BANNED_CPUS` variable that *irqbalance* uses to decide which vCPUs receive interrupts.

5.2 The *Moderator* Component

So far, the *Accountant* has enabled the hypervisor to measure the offloaded CPU usage for I/O workloads. We call this offloaded CPU usage as the *debt* of DomUs. Here we present the *Moderator* in the hypervisor that enforces proper resource allocation. The *Moderator* is implemented inside the Xen hypervisor's Credit scheduler. It calculates and collects the debt from each DomU every time the scheduler executes.

At each scheduling tick, the *Moderator* calculates the debt of each DomU by checking the amount of credits burned by that DomU's designated vCPUs in the driver domain. Among those debts, some of them are *dedicated debt* which can be attributed to a specific DomU (i.e., CPU usage of its corresponding backend drivers), while the rest are *shared debt* among all DomUs who have work offloaded (i.e., CPU usage of the physical device drivers and TCP/IP stack). The dedicated portion can be directly accounted to its source DomU. While, for the shared portion, we divide the debt to each DomU in the proportion to their own dedicated debt accumulated during the past scheduling epoch. A scheduling *epoch* is the short time window between two consecutive scheduler ticks. For example, with the same setup in Table 2, for the past 30*ms*, vCPU-5 (DomU-1) and vCPU-7 (DomU-2) have burned 4 and 6 credits respectively, while vCPU-3 (shared) has burned 5 credits. The total debts for DomU-1 and DomU-2 are therefore (4+2)=6 and (6+3)=9 credits.

With the debt for each DomU accurately accounted, the *Moderator* then collects the debt for every scheduling epoch — it deducts all the debt accumulated so far from each DomU's credit when the credit is refilled in `csched_acct()`. After the debt is paid off, the remaining credit of each DomU will be allocated to its vCPUs as in the original Credit scheduler.

6 Evaluation

We have implemented the *VASE System* in Xen 4.9.0. The roadmap for evaluating our prototype is as follows. We first verify the workload encapsulation and pinning scheme in the *Accountant* component, and then we show the effectiveness of the *Moderator* component by comparing the CPU usage of each domain with and without our approach against the

Table 2. Processes and IRQs affinity setting in the driver domain. As shown in Figure 7, related processes and IRQs are identified to serve DomU-1, DomU-2 or DomUs. In *VASE System*, they are pinned to designated Dom0 vCPUs accordingly.

Source	Type	PID	IRQ ID	vCPU # (mask)
Dom0	all other	-	-	0-1 (0x03)
DomU-*	physical disk	-	110-117	2 (0x04)
DomU-*	physical network	-	118-125	3 (0x08)
DomU-1	backend disk	2623,2781	131	4 (0x10)
DomU-1	backend network	2783-2784	132-133	5 (0x20)
DomU-2	backend disk	2977,3130	138	6 (0x40)
DomU-2	backend network	3132-3133	139-140	7 (0x80)

Table 3. vCPU utilization when running different workloads in DomU-1 and DomU-2 after *VASE System* being implemented. The offloaded I/O processing is precisely encapsulated and correctly pinned to designated vCPUs as shown in Table 2. Each vCPU only consumes CPU time if and only if corresponding I/O workload runs in corresponding DomUs.

Case #	Workloads		Dom0 vCPU							
	DomU-1	DomU-2	0	1	2	3	4	5	6	7
1	Idle	Idle	-	-	-	-	-	-	-	-
2	CPU	MEM	-	-	-	-	-	-	-	-
3	UDP	Idle	-	-	-	12%	-	76%	-	-
4	Idle	UDP	-	-	-	9%	-	-	-	76%
5	SEQ	Idle	-	-	4%	-	21%	-	-	-
6	Idle	SEQ	-	-	4%	-	-	-	20%	-
7	UDP	UDP	-	-	-	10%	-	38%	-	48%
8	SEQ	SEQ	-	-	5%	-	13%	-	11%	-
9	UDP	SEQ	-	-	4%	12%	-	76%	17%	-
10	SEQ	UDP	-	-	4%	11%	18%	-	-	46%

given CPU caps. Finally we show that our approach introduces negligible overhead to both the scheduling process and the execution of the workload.

Here we use *Vase* to represent our solution and *Credit* for the default Xen settings — the I/O workloads in the driver domain are load-balanced across all available vCPUs whenever appropriate and the Credit scheduler is used. Previous solutions [8, 19, 32] have been evaluated in Section 3.2, and therefore will not be repeated for comparisons here. In our experiments, Dom0 is used as the driver domain. All the data points in the result were obtained through 100 repetitions and 95% confidence intervals are provided in all applicable figures, though most of them are too small to be visible.

6.1 Verify the Workload Encapsulation

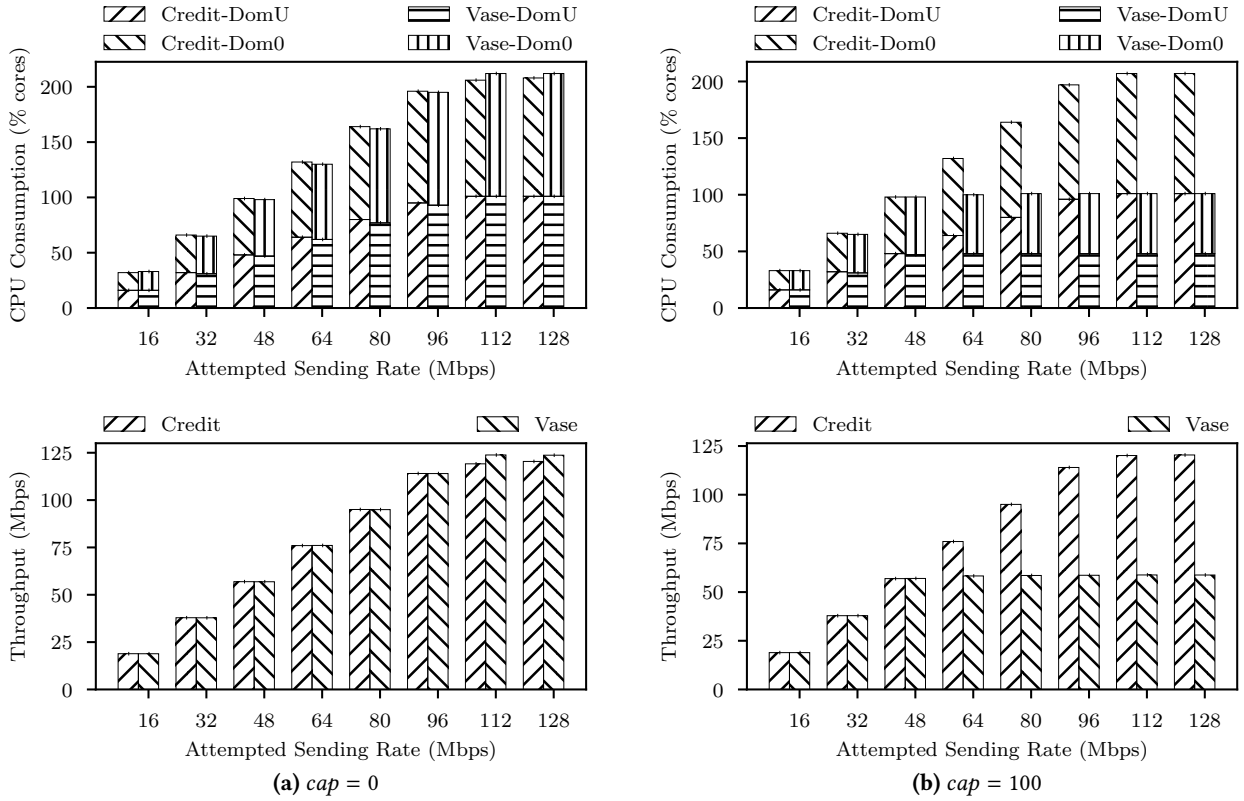
Our goal here is to verify that all the offloaded I/O processing in Dom0 has been thoroughly encapsulated and correctly pinned to designated vCPUs. In other words, with our settings, a designated vCPU in Dom0 should consume CPU time *if and only if* its corresponding DomU runs the corresponding type of I/O workload. To this end, we create two DomUs using the configuration shown in Table 2 and run various combinations of the workload listed in Table 1 on these two DomUs. The vCPU utilization is shown in Table 3, where "–" indicates such utilization is negligible.

As shown in Table 3, in cases of 3, 7, and 9 where DomU-1 performs network I/O, its corresponding vCPU-5 in Dom0 consumes significant CPU time, which concludes the *if* part. For the *only if* part, we can see whenever vCPU-5 consumes CPU time — again case 3, 7, and 9, DomU-1 is indeed performing network I/O. For other vCPUs and other type of workload, we can easily get the similar observations as well. Hence, we conclude all offloaded I/O processing capsules has been correctly and thoroughly identified and pinned to designated vCPUs in Dom0 as expected.

6.2 Accurate CPU Resource Allocation

Next, we evaluate the effectiveness of our *VASE System* to accurately enforce the capacity limits. In the experiment we let one DomU run MIX workload listed in Table 1 with various intensities and comparing the CPU consumption incurred with and without our approach against the cap value. The DomU is given 0 and 100 as its *cap* value in the scheduler and the result is shown in Figure 8a and 8b, respectively. The x-axis of each figure represents the intensity (sending rate) of the workload and the y-axis represents the CPU usage (upper figure) and network throughput (lower figure).

We first examine Figure 8a where work conserving mode is enabled ($cap = 0$). In both upper and lower figures, as the sending rate increases, both CPU time consumption and the



(a) *cap* = 0 **(b)** *cap* = 100

Figure 8. CPU usage and throughput of one DomU with different values for *cap*. When *VASE System* is implemented, the total CPU consumption of DomU is precisely limited by the given cap values. On the contrary, it can reach up to twice the allocated amount in the original setting. Thus, *VASE System* accurately enforce the resource consumption limit of DomUs.

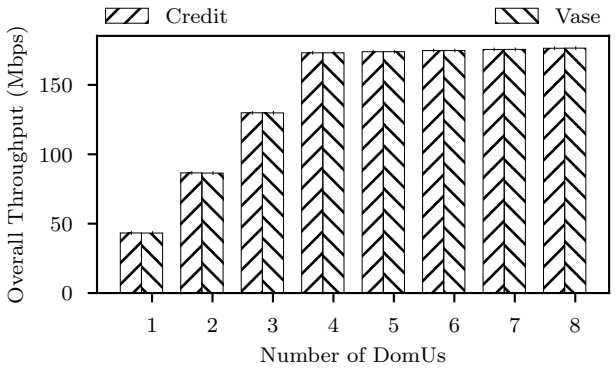


Figure 9. Overall throughput with various number of DomUs running CPU intensive workloads, which shows *VASE System* introduces negligible overhead to CPU throughput.

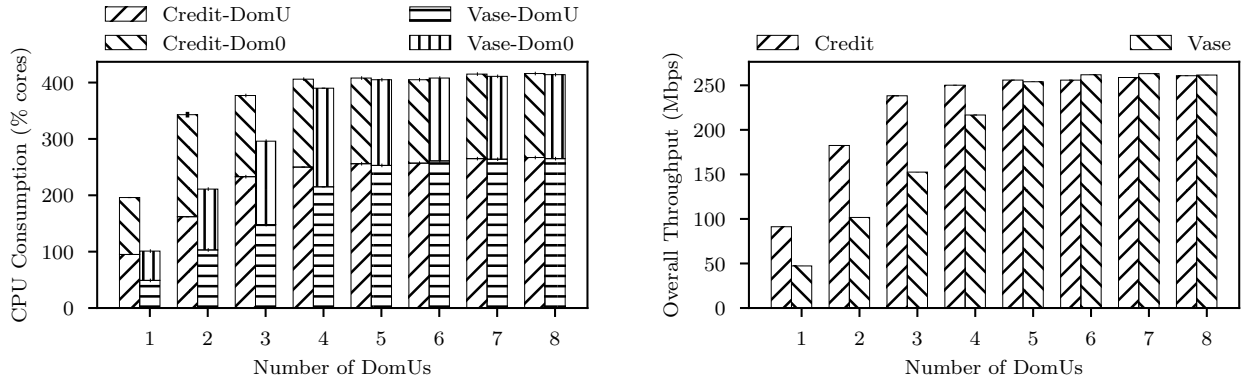
network throughput increase as expected and are not capped in both *Credit* and *Vase*.

Next in Figure 8b we set the *cap* to 100, indicating the DomU is expected to incur at most 100% system-wide CPU usage. We can see in the upper figure, as the sending rate increases, *Credit* fails to keep the total CPU usage under the *cap*, which in this case reaches 200%, significantly breaking the configured limit. In comparison, in *Vase*, the total CPU consumption including the offloaded portion is accurately constrained to 100%. A similar result can be observed in Figure 8b (lower), where the network throughput of the DomU can be limited to 60 Mbps in *Vase*, while *Credit* allows DomU to generate excessive traffic to stress the system and potentially impair the performance of neighbors. In the public cloud such as Amazon EC2, the instance is sold with preconfigured amount of CPU cores and its CPU usage should be limited accordingly. *VASE System* enables the cloud providers to achieve this exact purpose.

6.3 System Overhead

In this section we show our approach introduces negligible overhead to the scheduling process and the workload itself.

We first examine the scheduling overhead. With our approach, when there are n DomUs with one network and one disk device running in a host, at least $4+2\times n$ vCPUs need to be allocated to Dom0. As the number of vCPUs increases, the runtime for both hypervisor scheduler and Dom0 scheduler to iterate through all the vCPUs will also increase. Meanwhile, the extra routines in *VASE System* for calculating the debt will also incur overhead. To evaluate the scheduling overhead, we let DomU run CPU workload in Table 1 for 10 seconds, record the events/sec value reported by *sysbench*



(a) Overall CPU usage **(b)** Overall I/O throughput

Figure 10. Overall CPU usage and I/O throughput with various number of DomUs. *Vase* incurs negligible I/O overhead when CPU is saturated (DomUs > 4). When it is not, *Vase* accurately limits DomU's CPU usage, and hence their I/O throughput.

with different number of DomUs, and compare the result obtained for *Credit* and *Vase*. The max number of DomUs is set to twice the number of pCPUs used in this experiment. As we can see in Figure 9, the overall system CPU performance in both cases is either very close or statistically the same, indicating negligible overhead is introduced in the CPU scheduling process. Besides, extra memory space is required to keep the states of extra vCPUs, but it is orders of magnitude lower compared to the size of Xen and kernel structures and can be ignored.

Next, we examine *Vase's* performance impact to the workload. We create various numbers of concurrently active DomUs running MIX workload with each DomU's cap set to 100. The result is shown in Figure 10a and 10b. The x-axis represents the number of concurrent DomUs and the y-axis represents the total CPU usage in the Figure 10a and overall throughput in Figure 10b. In both figures, once the system become over-committed, the overall CPU usage and total achieved network throughput of *Vase* is on par with *Credit*. Hence, *Vase* introduces negligible overhead to both scheduling and the processing of the workload. Therefore, we can conclude that our approach is indeed light-weight. Beside, when the host is under-committed (DomUs <4), the cloud user may prefer *Credit* over *Vase* due to the seemingly higher overall throughput. However, this is not an intended/desired feature of the system, since the extra CPU and throughput comes at the cost of potentially performance variations and degradation of neighbor domains and extra energy consumptions of the host machine. Moreover, the cloud provider would prefer to consolidate the host by selling all available cores, preventing host being under-committed. Therefore, we believe *Vase* provides a better solution than *Credit* for CPU management in the cloud.

7 Discussion

Limitation Though our proposed approach is implemented in Xen-based virtualization systems with paravirtualized I/O devices (both PV and HVM [1] instances), the idea behind our approach is universal and can be extended to other platforms use software-based I/O virtualization. Admittedly, some technical aspects may be different and due to space limits we omit the detailed discussions here. However, in the high-level, many hypervisors share the same design principle. For example, the paravirtualized split driver model is also supported in type-2 hypervisors like KVM/QEMU [28] via virtio and VMware Workstation [2] via its guest tools. Besides paravirtualized I/O, our approach applies to other virtualized I/O implementations as well. For example, for QEMU-based fully emulated devices in Xen, the QEMU processes that handles I/O workload can be pinned to given vCPUs in a way similar to our approach.

Hardware-based I/O virtualization solutions [11] have been applied to some AWS instances recently. However, software-based I/O virtualization is still widely used in the cloud industry, e.g., all the Google Cloud instances and many AWS instances. In addition, unlike some other works [4, 19, 22], our solution applies to not only network I/O but also disk I/O, which also represents a huge number of hosts.

Scalability Our *VASE System* requires one designated vCPU in Dom0 for each device in each DomU. Naturally, the question to ask is that whether this approach will work as the number of devices and DomUs increases. In theory, Linux kernel v4.4 supports up to 8192 CPUs and Xen hypervisor allows assigning the Dom0 a user-defined number of vCPUs. Based on our experiments, Xen supports a maximum vCPUs of at least 255 to be allocated to Dom0. Considering a dual-socket Xeon E5 (up to 44 cores) server with two devices per DomU, even with a 200% over-commit ratio and all one-core small instances, the required 176 designated vCPUs is well below the vCPU limit. Also, given the fact that Xen can hot add and remove vCPUs, the number of designated vCPUs in Dom0 can be adjusted as DomUs are created/destroyed so that the scheduling overhead of maintaining many vCPUs can be minimized.

Load Balancing Another question is that, with this approach, whether we are limited to only one vCPU per device. In other words, is this approach elastic and flexible enough so that the number of vCPUs designated per device can be dynamically adjusted depending on the intensity of the workload. The answer is yes - with *Vase*, we can allocate more than one vCPU to carry out the work for a single device by changing the IRQ mask. That way the load is balanced across all the allocated vCPUs, improving the performance. In case of light workloads, similarly, several devices of the same type and same DomU can be consolidated to one vCPU.

8 Related Work

Accurate resource accounting and allocation is a long-standing research challenge in various contexts including both non-virtualized and virtualized environments [6, 7, 13, 18, 27, 35, 37]. In non-virtualized environments, when a user process issues intensive network I/Os, a large portion of CPU time is consumed asynchronously by OS kernel and not correctly accounted to that user process. LRP [12] and Resource Container [4] aimed to address this issue by accounting the network processing in the kernel to corresponding processes. The same case also applies to the container environment and Iron [22] was proposed to address the same issue for container-based multi-tenant environments. Ghanei et al. [17] highlighted the challenge in accounting asynchronous resource usage. They investigated the mobile computing scenario, where resources such as sensors and network may be consumed asynchronously by running processes. These three works are generally based on the kernel tracing techniques, which enable us to track and account kernel usage to the user process/container but also introduce significant overhead, reducing the overall performance.

Accurate resource accounting is challenging in the virtualized environment where multiple VMs share hardware and software resources, including I/O devices, intrusion detection systems (IDS), etc. For the case of the shared IDS between VMs, Resource Cage [24] was proposed to accurately bill each VM based on its usage of IDS service. Similarly, the drawback of this approach is also the high overhead, due to the use of sophisticated VM-introspection techniques to trace and account the cross-domain offloaded processing. By comparison, our solution is easy to implement and lightweight by exploiting the existing vCPU abstractions. As mentioned previously in this paper, software-based I/O virtualization is a major source of accounting inaccuracy. Cherkasova and Gardner [8] and Santos et al. [29] have demonstrated that when serving I/O requests in DomU, Dom0 consumes a non-negligible amount of CPU time on behalf of that domain. Gupta et al. [19] further investigated this issue by accounting the offloaded CPU consumption to that DomU in CPU management. They measured the unit CPU consumption per packet in Dom0 when serving network I/O in DomU, and estimated future CPU consumption on behalf of DomUs by the number of packets sent/received per DomU. With this estimation, they modified the SEDF scheduler to aggregate CPU consumption of domains in CPU allocation. Teabe et al. [32] tried to improve accuracy of the estimation by profiling I/O workloads with more features, such as packet sizes and virtualization configuration, for both disk and network I/O. They modified the Credit scheduler to charge DomUs for offloading. As we have shown previously, the drawback of such approaches is that, despite it improves estimation accuracy, they extract features solely from the workload while the actual offloaded CPU time may be affected by neighbor activities, compromising its overall estimation accuracy. In comparison, by direct measuring the usage of the offloaded workload as it happens, our solution generates much more accurate accounting of the offloaded CPU consumption.

When it comes to hardware-based virtualization techniques, such as SR-IOV [11] and SmartNIC [16], the offloaded I/O processing will happen in the specially designed hardware, which no longer consumes CPU usage in Dom0. The resource management can therefore be greatly simplified. However, it still requires extra investment for dedicated hardware in the host machines. On the contrary, our work solves same problem at the software level without specialized hardware and hence its additional cost.

9 Conclusion

Cloud computing relies on accurate resource allocation of domains (VMs) to better serve the needs for both cloud users and providers. In this paper, we have shown the current approaches fail to correctly account all the CPU usage incurred by domains due to I/O offloading. We claim the root cause is that the protection scope of a domain is incorrectly used as its resource scope in the resource management. To address this problem, we redefine the resource scope of a domain by using vCPU as a container, so that all the processing incurred by this domain is contained within its new resource scope. To demonstrate our solution, we have implemented *VASE System* that directly and accurately measures the offloaded CPU usage and uses it to strictly enforce the CPU usage limits in Xen. Our experiments have shown that our approach is light-weight and effective in constraining CPU usage with virtually no overhead.

Our future work includes extending the proposed *VASE System* to other virtualization platforms, e.g., KVM/QEMU and also the non-virtualized container environments.

Acknowledgments

We appreciate the constructive comments from the reviewers. This work is supported by NIST grant 70NANB18H272 and NSF grant CNS-1524462. This work was also supported in part by the Institute for Smart, Secure and Connected Systems at Case Western Reserve University through a grant provided by the Cleveland Foundation.

References

[1] 2015. PV on HVM. https://wiki.xen.org/wiki/PV_on_HVM. Accessed: 2018-09-09.

[2] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, Pratap Subrahmanyam, et al. 2006. VMI: An interface for paravirtualization. In *Proc. of the Linux Symposium*. Citeseer, 363–378.

[3] Anthony O Ayodele, Jia Rao, and Terrance E Boult. 2015. Performance measurement and interference profiling in multi-tenant clouds. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 941–949.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *OSDI*, Vol. 99. 45–58.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[6] Andy C Bavier, Mic Bowman, Brent N Chun, David E Culler, Scott Karlin, Steve Muir, Larry L Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. 2004. Operating Systems Support for Planetary-Scale Network Services.. In *NSDI*, Vol. 4. 19–19.

[7] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. 2001. Managing energy and server resources in hosting centers. *ACM SIGOPS operating systems review* 35, 5 (2001), 103–116.

[8] Ludmila Cherkasova and Rob Gardner. 2005. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor.. In *USENIX Annual Technical Conference, General Track*, Vol. 50.

[9] Ron C Chiang, Sundaresan Rajasekaran, Nan Zhang, and H Howie Huang. 2015. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for i/o resources. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (2015), 1732–1742.

[10] David Chisnall. 2008. *The definitive guide to the xen hypervisor*. Pearson Education.

[11] Yaozu Dong, Zhao Yu, and Greg Rose. 2008. SR-IOV Networking in Xen: Architecture, Design and Implementation.. In *Workshop on I/O Virtualization*, Vol. 2.

[12] Peter Druschel and Gaurav Banga. 1996. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *OSDI*, Vol. 96. 261–275.

[13] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. 2008. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *Proceedings of the 7th international conference on Information processing in sensor networks*. IEEE Computer Society, 283–294.

[14] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH computer architecture news*, Vol. 35. ACM, 13–23.

[15] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. 2012. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 20.

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[17] Farshad Ghanei, Pranav Tipnis, Kyle Marcus, Karthik Dantu, Steve Ko, and Lukasz Ziarek. 2016. OS-Based Resource Accounting for Asynchronous Resource Use in Mobile Systems. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 296–301.

[18] Ajay Gulati, Arif Merchant, and Peter J Varman. 2010. mClock: handling throughput variability for hypervisor IO scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 437–450.

[19] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. 2006. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., 342–362.

[20] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. 2013. Efficient and Scalable Paravirtual I/O System. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, San Jose, CA, 231–242.

[21] Stephen T Jones, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, et al. 2006. Antfarm: Tracking Processes in a Virtual Machine Environment.. In *USENIX Annual Technical Conference, General Track*. 1–14.

[22] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. 2018. Iron: Isolating Network-based CPU in Container Environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 313–328. https://www.usenix.org/conference/nsdi18/presentation/khalid

[23] Alexey Kopytov. 2004. SysBench: a system performance benchmark. *URL: http://sysbench. sourceforge. net* (2004).

[24] Kenichi Kourai, Sungho Arai, Kousuke Nakamura, Seigo Okazaki, and Shigeru Chiba. 2017. Resource Cages: A New Abstraction of the Hypervisor for Performance Isolation Considering IDS Offloading. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 170–177.

[25] Philipp Leitner and Jürgen Cito. 2016. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)* 16, 3 (2016), 15.

[26] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. 2013. Who is your neighbor: Net i/o performance interference in virtualized clouds. *IEEE Transactions on Services Computing* 6, 3 (2013), 314–329.

[27] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. 2011. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 321–334.

[28] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[29] Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Ian Pratt. 2008. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization.. In *USENIX Annual Technical Conference*. 29–42.

[30] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.

[31] Byung Chul Tak, Youngjin Kwon, and Bhuvan Urgaonkar. 2017. Resource Accounting of Shared IT Resources in Multi-Tenant Clouds. *IEEE Transactions on Services Computing* 10, 2 (2017), 302–315.

[32] Boris Teabe, Alain Tchana, and Daniel Hagimont. 2016. Billing the CPU time used by system components on behalf of VMs. In *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE, 307–315.

[33] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. 2005. Iperf: The TCP/UDP bandwidth measurement tool. *htt p://dast.*

nlanr. net/Projects (2005).

[34] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. 2012. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 281–292.

[35] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 14.

[36] Matthew Wilcox. 2003. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Linux. conf. au*.

[37] Heng Zeng, Carla S Ellis, Alvin R Lebeck, and Amin Vahdat. 2002. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGOPS operating systems review* 36, 5 (2002), 123–132.

[38] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2017. DoS Attacks on Your Memory in Cloud. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 253–265.