FORTIFYING MULTI-TENANT CLOUD ENVIRONMENTS
VIA IMPROVED CPU MANAGEMENT

by

Li Liu

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____	Dr. Songqing Chen, Dissertation Director
_____	Dr. Fei Li, Committee Member
_____	Dr. Yue Cheng, Committee Member
_____	Dr. Brian L. Mark, Committee Member
_____	Dr. David S. Rosenblum, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Fall Semester 2020
George Mason University
Fairfax, VA

Fortifying Multi-tenant Cloud Environments via Improved CPU Management

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Li Liu
Master of Engineering
Peking University, 2013
Bachelor of Management
Nanjing University, 2009

Director: Dr. Songqing Chen, Professor
Department of Computer Science

Fall Semester 2020
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my parents.

Acknowledgments

This dissertation would not have been possible without the support of my advisor, Dr. Songqing Chen. I am deeply grateful to him for his continuous support and patience during my Ph.D. studies. He led me to research and trained me to become a researcher. Throughout my graduate studies, he consistently provided me with guidance, support, and encouragement. He is also a role model whose enthusiasm, dedication to research, and commitment to high standards I can only hope to emulate.

I also want to thank the rest of my dissertation committee: Dr. Fei Li, Dr. Yue Cheng, and Dr. Brian L. Mark, for their time, effort, and invaluable advice. They played a critical role in shaping my dissertation.

I would give my sincere gratitude to my lab mates and friends at George Mason University: Haoliang Wang, An Wang, Mengbai Xiao, Zili Zha, to name a few. Their help and support during these years are indispensable.

Finally, I would like to thank my parents for their full support of my studies.

Table of Contents

List of Tables

List of Figures

Abstract

FORTIFYING MULTI-TENANT CLOUD ENVIRONMENTS VIA IMPROVED CPU MANAGEMENT

Li Liu, PhD

George Mason University, 2020

Dissertation Director: Dr. Songqing Chen

Cloud computing technology has significantly changed the computing diagram. Ever since Amazon Web Service (AWS) began offering IT infrastructure services in 2006, cloud computing technology keeps maturing, developing, and offering more benefits. A cloud computing environment allows multiple cloud tenants to share the same physical or virtual host, thus called a multi-tenant environment. A cloud tenant can be an individual user or a group of users sharing cloud resources. A multi-tenant cloud environment is supposed to be secure and fair. However, due to the inevitable resource sharing among multiple tenants, one tenant's behavior may impact the other tenants running simultaneously. It often leads to performance anomalies and security concerns.

In this dissertation, I investigate these security and performance problems in the multi-tenant cloud environment and propose novel solutions to address these issues. Given that the root causes of these issues are due to resource sharing (i.e., CPU) among multiple tenants, our solutions are always centered around CPU management, as discussed below. First, I demonstrate a virtual machine (VM) is vulnerable to side-channel attacks when a malicious neighboring VM runs on the same host. To secure the host against such attacks, I design new schemes via a different CPU scheduling strategy for VMs. It can effectively

defeat such side-channel attacks with a negligible performance overhead.

Second, cloud computing often employs a pay-as-you-go pricing model, which relies on precise resource accounting to allocate the configured amount of resources to VMs. However, I reveal that the existing hypervisors often fail to do an accurate resource accounting, leading to allocate the incorrect amount of CPU resources to VMs. To address this issue, I propose to redefine the resource scope of VMs using its virtual CPU (vCPU). Through experiments, I show that the CPU consumption can now be correctly accounted for and managed.

Third, the cloud increasingly adopts container technology these days, and the cloud sees more and more containerized applications deployed. This trend requires even stronger isolation between applications. However, I show that containerized applications' expected performance isolation is not yet achieved. The primary reason is that the containers are inadequately managed due to complex interactions between various scheduling mechanisms in the current OS scheduler design. I propose to augment the underlying host's scheduling mechanism to support the container orchestration system to address this problem. Extensive evaluations show that our new scheme can bring significant improvement in resource management and performance.

Chapter 1: Introduction

Cloud computing has significantly changed the computing arena in the past decade. Driven by the significant benefits of scalability, elasticity, and automatic control of large-scale operations, both companies and individuals have widely adopted the cloud. The cloud computing market is still rapidly growing. Gartner predicts the cloud infrastructure as a service (IaaS) market revenue to increase to 81.5 billion by 2022, up from 41.4 billion in 2019 [2]. According to Amazon Web Services (AWS), by moving IT infrastructure to the cloud, its user gains an average of 31% cost saving while achieving 62% management efficiency improvement [3]. Underpinning such transitions is the common belief that today the cloud infrastructure can promise security and performance that are important for more services to migrate to the cloud [4].

A cloud tenant can be an individual user or a group of users, such as an organization that shares cloud resources. The cloud service enables a tenant to lease computing resources on-demand. It offers a pay-as-you-go [5] approach for payment. The cloud infrastructure relies on resource sharing among multiple tenants to achieve high resource utilization and economies of scale. Most cloud services, almost everything except dedicated hosting services, are based on multi-tenant hosting or shared hosting, allowing multiple tenants to share the same host. For a host machine in the cloud, either a physical machine or a virtual machine, virtualization technologies provides a logical abstraction for multiple tenants to share resources properly. Ideally, each tenant's data is isolated from other neighboring tenants sharing the same host. So is their computing performance. However, in reality, this is often not the case yet.

1.1 Problems and challenges in the multi-tenant environment

A cloud user often leases computing resources in the form of a virtual machine (VM). In a multi-tenant host, different VMs are regulated by the hypervisors, or virtual machine managers, such as Xen and KVM. The underlying virtualization technologies provide an isolated protection mechanism for the states and executions of each VM. However, such isolation is not well-conceived in the current multi-tenant environment. It introduces challenges in such a sharing environment, as I discuss below. For example, sophisticated attacks could exploit the underlying shared resources to extract sensitive information from neighboring VMs, resulting in security and privacy breaches.

Cache-based side-channel attacks, among others, represent the primary and most threatening concerns of cloud security in previous studies [6]. Many solutions are proposed. One category of solutions [7, 8] is to harden the targeted operations being attacked so that the victim is no longer vulnerable to attacks in the insecure cloud environment. However, hardened programs may still be vulnerable to other side-channel attacks [9]. Compared to hardening vulnerable programs individually, a more general solution is to secure the cloud environment. For example, dedicated host services offered by the cloud provider [10] physically isolates a user's VMs from all other VMs, thus preventing those VMs from being attacked. An alternative is to dedicate a portion of CPU caches to each VM [11, 12]. While effectively shutting down the cache side-channel at the cost of resource sharing, these solutions are not favorable for the cloud paradigm considering the performance. There are solutions employing the moving target defense philosophy by frequently migrating VMs [13] to other hosts. Nevertheless, live migration will introduce unaffordable overhead. Also, Liu et al. [14] reported that it was still possible to complete the attacks in minutes.

Another challenge in the multi-tenant host is resource accounting. Among various types of computing resources, the CPU is the most important one. Its accurate allocation and management directly affect the operations and the revenue of the cloud providers. According

to [15], each physical CPU core sells for a maximum potential annual revenue of $900. However, as previous works [16,17] have observed, and I will also demonstrate in Chapter 3, a guest domain (or VM) can consume up to 70% more of its allocated CPU time. It prevents the cloud providers from selling that overused CPU to other clients and resulting in a substantial monetary loss. The resource overuse issue may also potentially degrade the performance of neighbor domains [18–25] and increase the energy consumption of the host machines [26,27].

Solving the resource overuse problem relies on an accurate accounting of the offloaded processing, which is a challenging task, especially in a virtualized environment. The semantic gap between the hypervisor and domains and the offloaded processing's asynchronous nature pose significant challenges to measure the offloaded CPU usage. Previous works attempted to overcome such a semantic gap using VM-introspection techniques [28–30] at the cost of complicated kernel tracing and heavy runtime overhead, limiting its usage in modern cloud systems. Others [31,32] attempted to circumvent these challenges by estimating the offloaded CPU time instead of accurately measuring. Such estimation is based on the assumption that the offloaded CPU usage for the same workload is always the same. However, as I will show in Chapter 3, this is not necessarily true. Therefore the estimation-based approach also fails to produce accurate accounting results.

The increasingly popular microservices bring a third challenge to cloud computing. The microservice-based architecture decomposes the traditional monolithic applications into multiple loosely-coupled, single-purpose service modules. The transformation into microservices improves service scalability and reliability. It has been widely adopted into the modern software DevOps workflow. Microservices are typically deployed as containers on top of the multi-tenant cloud infrastructure. They are managed by container orchestration systems such as Google Kubernetes Engine [33] and Amazon Elastic Kubernetes Service [34]. Cloud tenants lease a certain amount of computing resources for the containers based on their estimations of workload demand and performance expectations.

However, it has been observed that the performance of containers running in the cloud

can vary significantly and is difficult to predict [35–42]. For example, it is reported in [36] that container co-location can lead to more than 66× tail latency increasing. Such significant performance variations make it difficult for developers to estimate the container's performance accurately and how much resources one should request. Therefore, this will cause severe performance bottlenecks such as stragglers for batch workloads [43] and violation of end-to-end QoS guarantee for latency-sensitive, interactive applications [44].

1.2 Dissertation contributions

This dissertation proposes three novel schemes to solve these security and performance problems in the multi-tenant cloud environment. As cloud computing's underlying key is the sharing of computing resources among multiple tenants, our research and solutions mainly focus on proper CPU scheduling and management.

First, to cost-effectively secure the cloud against side-channel attacks without sacrificing resource sharing, I investigate the factors that can impact such attacks' success. Our study reveals that the root cause of such attacks is the constant sharing patterns of hardware resources between VMs. Based on these findings, I quantify the negative impacts that a VM can have on another VM on the same machine using the *vulnerable probability*. Then I propose lightweight and generic scheduler-based defense mechanisms called *Shuffler Schedulers*, limiting the vulnerable probability of all VMs. The key is that distributing CPU time to vCPUs with equal probability would reduce the system's overall vulnerable probability. The analyses and experimental results show that the Shuffler Schedulers can effectively reduce information leakage, mitigating cross-VM side-channel attacks, with little performance penalty while preserving high resource utilization.

Second, in a multi-tenant environment, some VM may consume significantly more resources than allocated. In this study, I find that this problem's root cause 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. In contrast, a domain's

resource scope should contain all the resource consumption incurred by this domain. For instance, in the I/O offloading, these two scopes are not aligned. Such mismatch in the current design prevents the hypervisor from correctly allocating resources to each domain.

To tackle the problem in the virtualized multi-tenant cloud environment, I set to re-align the CPU resource scope of a domain with its actually-incurred CPU usage to enforce accurate resource allocation for all guest domains. Specifically, I 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 is contained in that source domain's resource scope. Therefore, the resulting resource scope of a domain contains all the incurred CPU consumption and can be used by the hypervisor to manage the CPU resource per domain accurately.

Third, I demonstrate the performance variation in the containerized cloud environment. I find that this results from the complex interactions between various scheduling mechanisms and considerations in the current OS scheduler design (e.g., CFS in Linux). To make it worse, the increasing number of CPU cores and the co-located containers in modern host systems exacerbate the problems. I systematically analyze the performance degradation in a multi-tenant environment. The key findings are: *(1) User-requested CPU resources at the container level (via cgroups) are not honored by the host OS. (2) The root cause lies in the mismatch of the container' design goals and Linux's default one-size-fits-all CPU scheduler CFS.*

Motivated by these findings, I propose to augment the scheduling mechanism in the host OS for the container orchestration system. It bridges the gap between one's expectation when they request the resource and the actual utilization and performance they can get out of such reservations. I implement this approach called rKubernetes or, *rKube* for short, on top of Kubernetes. It correctly enforces the requested amount of CPUs on host systems by isolating the container workload to designated physical cores.

1.3 Dissertation organization

The rest of this dissertation is organized as follows: in Chapter 2, I study the side-channel attacks in the multi-tenant hosts and propose the mitigation scheme based on CPU scheduling. Then I examine the CPU resource accounting in Chapter 3 and present the improvement for the scheduler. Chapter 4 discusses our investigation of performance isolation and augments to the scheduler in the containerized environment. I conclude this dissertation in Chapter 5.

Chapter 2: Shuffler: mitigate cross-VM side-channel attacks via hypervisor scheduling

The work presented in this chapter has been published in [45].

2.1 Introduction

Cloud has become an extremely successful paradigm for conveniently storing, accessing, processing, and sharing information. One of the building blocks of the cloud computing economy is its resource sharing empowered by virtualization techniques. Virtualization provides a logical abstraction for multiple VMs to share the same hardware resources. The hypervisor regulates VM isolation and resource sharing.

However, such sharing among VMs may cause potential vulnerabilities. For example, sophisticated attacks could exploit the underlying shared resources to extract sensitive information from neighboring VMs, resulting in security and privacy breaches. Studies showed that normal cloud users could achieve co-location with little cost [46–48] in the public clouds. As a result, side-channel attacks are demonstrated to be a real threat to cloud tenants [46, 49].

Cache-based side-channel attacks, among others, represent the primary and most threatening concerns of cloud security in previous studies [6]. Many solutions are proposed. One category of solutions [7, 8] is to harden the targeted operations being attacked so that the victim is no longer vulnerable to attacks in the insecure cloud environment. However, hardened programs may still be vulnerable to other side-channel attacks [9]. Compared to hardening vulnerable programs individually, a more general solution is to secure the cloud environment. For example, dedicated host service provided by the cloud provider [10] physically isolates a user's VMs from all other VMs, thus preventing those VMs being attacked.

An alternative is to dedicate a portion of CPU caches to each VM [11, 12]. Both solutions close the cache side channel at the cost of resource sharing, which is not favorable for the cloud paradigm. There are solutions employing the moving target defense philosophy by frequently migrating VMs [13] to other hosts. However, Liu et al. [14] reported completing the attacks in minutes. Moreover, live migration will introduce unaffordable overhead.

Therefore, to mitigate this continuous threat, it is imperative to have a solution that can (1) effectively mitigate cache-based side-channel attacks without sacrificing resource sharing, and (2) incur as little overhead as possible without significant performance or monetary cost. These objectives become even more challenging to achieve, given that one cannot tell in advance which VM(s) is (are) the attacker(s) in a cloud environment. That is, I should assume that any VM could be an attacker.

In this chapter, I set to find such a solution. For this purpose, the critical question is, "what makes the victim vulnerable to side-channel attacks?" I reveal that it is the runtime sharing patterns that enable the attacker to spy on the victim via shared resources. Furthermore, I quantify the time such patterns last by the vulnerable probability. By reproducing the Prime+Probe attack [14,49], I confirm that the vulnerable probability limits the attack results. Therefore, the attacks could be mitigated by reducing the vulnerable probability.

Motivated by my previous work [50], I find that distributing CPU time to candidate vCPUs with equal probability would effectively reduce the overall vulnerable probability. Thus, I propose my shuffling scheduling scheme based on a random virtual CPU (vCPU) selection mechanism. The *Local Shuffler (LS)* scheduler and the *Global Shuffler (GS)* scheduler are designed and implemented. My experimental results show that Shuffler schedulers can significantly reduce the vulnerable probability without sacrificing performance. When repeating the side-channel attack on a 4096 bits RSA key, the Shuffler schedulers reduce the (key bits) recovery rate from 100% to below 72%. Note that this is for the worst-case scenario, which favors the attacker as much as possible. Furthermore, I show that this recovery rate reduction can effectively mitigate such attacks.

Compared to other solutions, my scheme has several advantages: (1) it preserves high resource utilization, (2) it is lightweight in terms of overhead, (3) the implementation requires only minor revisions to the current hypervisor scheduler, thus making it easy to deploy, and (4) it is effective not only to cache-based side-channel attacks, but also to attacks exploiting other shared resources in runtime, such as DRAM [51] and processor interconnect [52].

The remainder of this chapter is organized as follows: Section 2.2 introduces related background information. Section 2.3 describes my motivation examples along with the identification of the vulnerable probability. Section 2.4 discusses the detailed design and implementation of my defense mechanism. Section 2.5 demonstrates the evaluation results of the Shuffler schedulers. Some closely related issues are further discussed in Section 2.6. Related work is summarized in Section 2.7. Finally, Section 2.8 concludes this chapter.

2.2 Background

2.2.1 Cross-VM side-channel attacks

In cross-VM side-channel attacks, the attacker VM resides in the same physical host as the victim VM, and spies the victim VM's memory accesses by frequently interleaving with the victim VMs on the shared resources. In such a spying process, the victim VM's memory accesses are only exposed to the attacker during runtime.

As exposed by Zhang *et al.* [53], an attacker could spy on the victim's memory accesses by frequently preempting the victim. However, this is no longer possible in recent versions of Xen [54]. Cross-core shared resources such as CPU last level cache (LLC) [14, 49, 55], processor interconnect [52], DRAM [51], etc., can still be utilized to launch cross-VM side-channel attacks in the cloud. In such attacks, the victim's memory accesses are only exposed to the attacker when the attacker VM and the victim VM run concurrently on different cores. In Section 2.3.1, I will introduce the vulnerable probability to quantify such a vulnerability of a victim during this process, and demonstrate that this value limits the side-channel leakage.

2.2.2 Hypervisor scheduling mechanisms

As each memory access usually takes tens of nanoseconds, a fine-grained view of how the attacker VM and the victim VM run in time is needed to analyze the spying process of cache-based side-channel attacks. To understand the vCPU scheduling trace made by the hypervisor scheduler, I take the hypervisor schedulers of Xen as an example.

Virtualization enables multiple (guest) VMs to run on the same host. The hypervisor mediates the requests for shared CPU resources by multiple VMs through built-in schedulers. To facilitate the scheduling of CPU, the concept of vCPU is introduced, which refers to the virtual processor of a VM. Each VM can have one or more vCPUs. Upon this, the hypervisor schedules physical CPUs (cores) for vCPUs using different schemes. Additionally, Xen provides a flexible scheduler interface, via which customized scheduling algorithms can be implemented.

For recent versions of the Xen hypervisor, there are four different schedulers, namely the Credit scheduler, the Credit2 scheduler, the Real-Time Deferrable Server scheduler (RTDS), and the ARINC653 scheduler. By default, the Credit scheduler [56] is utilized. It is a general-purpose scheduler that aims to provide a proportional fair share of CPU resources to different VMs. I built my prototype based on this scheduler.

2.3 Motivation

As introduced in Section 2.2.1, in cross-VM side-channel attacks, the victim is vulnerable to attacks when the victim and the attacker run concurrently on different cores. Such a runtime pattern enables the attacker to effectively spy on the victim via shared resources. To precisely capture this vulnerability, I define a new measurement metric called the *vulnerable probability*, which is the normalized time during which the victim runs concurrently with the attacker using the victim's accumulative running time as a measure of scale.

To verify the impact of the vulnerable probability on the attack results, I reproduce cross-VM side-channel attacks and demonstrate through experiments my key observations

along with their insights. These insights further motivate us to study how the victim can avoid being attacked, and to design my solution in the next section.

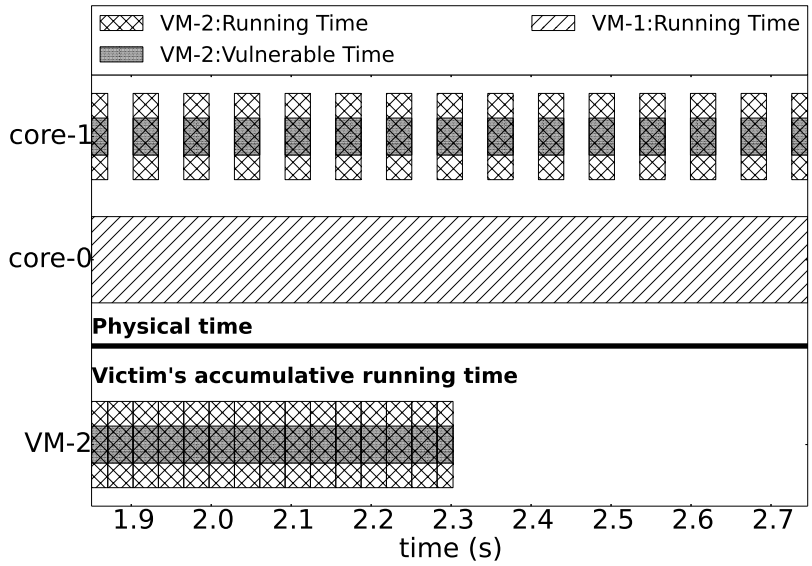
2.3.1 Vulnerable probabilities in attacks

Prime+Probe via LLC has been thoroughly studied [6] and demonstrated in the public cloud [46, 49]. This attack is used in my discussion, but my discussion is effective for other attacks such as Flush+Reload [55] as well.

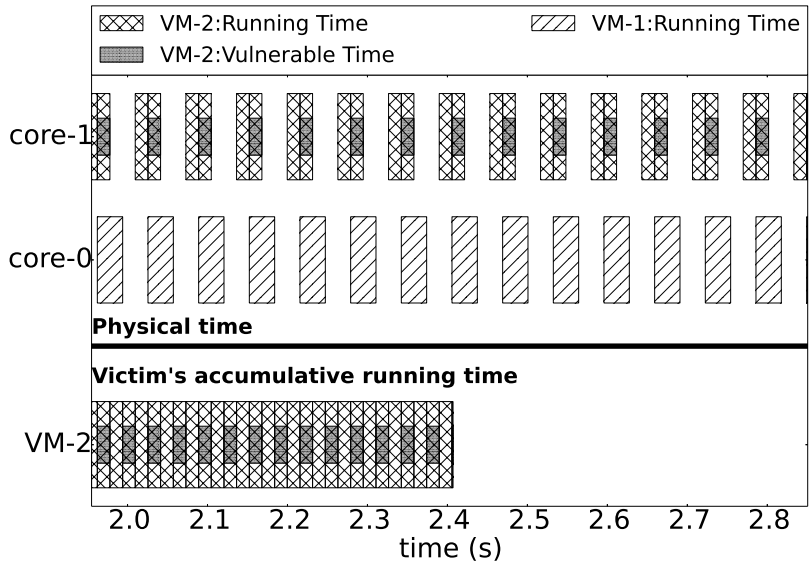
I use HP ProLiant DL380 G6 equipped with two Xeon E5540 CPUs. All 4 cores in each CPU package share the same LLC. Xen hypervisor version 4.6.0 with the Credit scheduler is used to manage VMs running on the host. The victim VM repeats signing a file with 4096 bits RSA key using GnuPG-1.4.13. The attacker VM spies the square-and-multiply implementation of modular exponentiation used by GnuPG-1.4.13 via shared LLC as described in [14, 49]. I also run additional background VMs (2-10) during the attacks. To reduce the noise introduced by Dom0, I pin Dom0 to one CPU package and all DomUs (guest VMs, including the attacker and the victim) to the other CPU package. Each DomU is configured to have one vCPU.

In each attack, the attacker can collect a portion of the secret key bits as a partial key. I define this rate as the *(key bits) recovery rate*, which is used to quantify the attack results. To verify the impact of the vulnerable probabilities on the attack results, I the repeat attack $30, 60, 90, ...240$ times when running 5-12 VMs, including the attacker VM and the victim VM, respectively.

Figure 2.1 demonstrates how the attacker (VM-1) and the victim (VM-2) were scheduled to run in two independent attacks. In the first attack (Figure 2.1a), 100% of the victim's crypto execution on core-1 is spied on by the attacker running on core-0, while in the other attack (Figure 2.1b), only 49% of the victim's crypto execution is exposed to the attacker. Meanwhile, the attacker collects 99.5% and 48.5% key bits separately in these two attacks. The vulnerable probabilities (100% and 49%) and the recovery rates (99.5% and 48.5%) are closely correlated.



(a) An attack case in which the attacker collects 99.5% key bits. 100% of the victim's crypto execution is under the attacker's spy.



(b) An attack case in which the attacker collects 48.5% key bits. 49% of the victim's crypto execution is under the attacker's spy.

Figure 2.1: Vulnerable probabilities in the two attacks

Intuitively, the vulnerable probabilities determine the attack results. To verify this hypothesis, I compare the vulnerable probabilities and the recovery rates of all 1080 attacks, as shown in Figure 2.2. I calculate the Pearson correlation coefficient (PCC) that quantifies

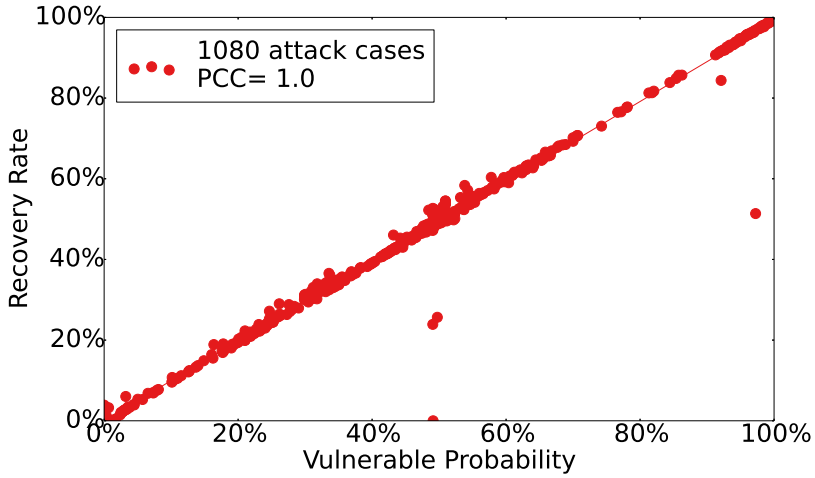


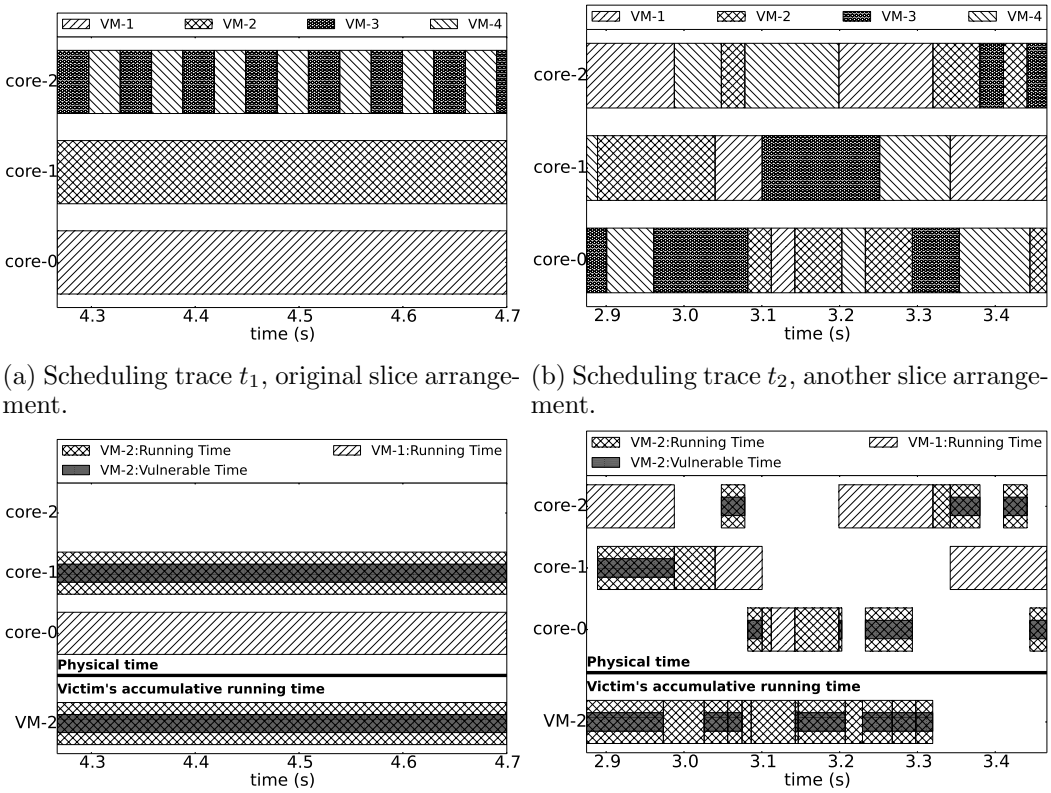
Figure 2.2: Recovery rates versus vulnerable probabilities in 1080 attacks

dependencies and correlations. The PCC value is 1.0, suggesting that the attack result is largely determined by the vulnerable probability. Therefore, if I manage to reduce the vulnerable probabilities, I could reduce the side-channel leakage.

2.3.2 Effects of hypervisor scheduler

Based on the previous discussions, I aim to minimize the vulnerable probability. Clearly, this value is determined by how VMs are scheduled to run. There are many factors that affect the vCPU scheduling process, including the vCPUs/cores ratio, the time slice length, the time slice arrangement, etc. In the previous work [50], I revealed that slice arrangement affected side-channel leakage more significantly than other factors. Below I demonstrate the effects of the slice arrangement on the vulnerable probability.

The scheduling trace t_1 in Figure 2.3a shows the original slice arrangement in an attack collected in Section 2.3.1, while t_2 in Figure 2.3b shows another slice arrangement in the same attack. Compared to t_1, t_2 assigns the vCPUs more evenly to all the available cores. Correspondingly, how the victim (VM-2) is spied on by the attacker (VM-1) is shown in Figure 2.3c and Figure 2.3d. The vulnerable probability is reduced from 100% to 68%. Thus, the proper arrangement of time slices can effectively reduce the vulnerable probability.



(a) Scheduling trace t_1, original slice arrangement.

(b) Scheduling trace t_2, another slice arrangement.

(c) VM-2's vulnerable probability is 100%, when spied on by VM-1 in t_1.

(d) VM-2's vulnerable probability is 68%, when spied on by VM-1 in t_2.

Figure 2.3: Vulnerable probability of the same attack with different slice arrangements. In both scheduling traces, the victim (VM-2) completes the same crypto operation.

Motivated by this result, I move on to design and implement a scheduler-based mechanism that can significantly reduce the vulnerable probability.

2.4 Solution

As shown in the last section, the slice arrangement largely determines the vulnerable probability. In this section, I will revisit the design of the hypervisor scheduler, particularly focusing on the slice arrangement in the vCPU scheduling process. From the scheduler's perspective, I begin by scoping the attacker's goals and capabilities.

2.4.1 Threat model

I assume that in each attack, the goal of an attacker is to extract as much information as possible. Under this condition, I consider a threat model meeting the common requirements in published the attacks [14, 49, 55], with the following characteristics:

- Co-location: the attacker VM runs in the same physical host with the victim VM.
- Unknown attacker: from the perspective of the hypervisor scheduler, attacker is unknown in advance, meaning that any VM could be the attacker.
- Single attacker: I assume the state-of-the-art setting in which each attacker vCPU spies the victim individually. I discuss colluding attacks involving multiple vCPUs in Section 2.6.1. Below I use *attacker*, *attacker vCPU* and *attacker VM* interchangeably.
- CPU overcommitment: I assume that there are more vCPUs than cores in the host. I discuss this assumption in Section 2.6.2.
- Effective spy during runtime: I assume that the attacker can effectively spy on the victim's memory access via shared resources during runtime, and that the third party VMs introduce minimum noise to the side channels.
- Persistent attacks: I assume the attacker can repeat the same attack for a reasonable large number of times. For example, Liu et al. [14] repeated the same attack for more than $20,000$ times to recover the secret key.

2.4.2 Problem statement

I aim to reduce the vulnerable probability by scheduling. In a given attack with scheduling trace t, the victim (vCPU or VM) v and the attacker a, the vulnerable probability is represented as $P(t, v, a)$, which is defined by the time of the victim runs concurrently with the attacker on the scale of the victim's accumulative running time. For example, in Figure 2.3, $P(t_1, VM_2, VM_1) = 100\%$ and $P(t_2, VM_2, VM_1) = 68\%$.

Since it is not possible to pinpoint the specific attacker, the goal for the hypervisor is to mitigate the overall vulnerability of the system, which is bound by the most vulnerable vCPU pairs quantified as:

$$P(t) = \max_{\forall v,a}\{P(t,v,a)\} \tag{2.1}$$

Therefore, an effective defense mechanism can be mathematically captured by the solution to $\min_t \mathrm{P}(t)$. To this end, I conduct a mathematical analysis on the problem and propose a scheduler-based scheme to achieve the optimization accordingly.

2.4.3 Problem analysis and solution

In the discussion below, I assume that there are m available cores and n ($n > m$) active vCPUs in a host. For a given scheduling trace t and victim vCPU v, there are $m-1$ vCPUs run concurrently whenever v runs. Thus, its vulnerable probabilities against to all potential attackers are subject to:

$$\sum_{\forall a} P(t,v,a) = m-1$$

Then the largest vulnerable probability an attacker can obtain can be calculated by Eq. 2.2. Here I assume the worst-case that any of the $n-1$ vCPUs may be the attacker.

$$\max_{\forall a}\{P(t,v,a)\} \geq \frac{\sum_{\forall a} P(t,v,a)}{n-1} = \frac{m-1}{n-1} \tag{2.2}$$

The intuitive interpretation of Eq. 2.2 suggests that the balanced allocation of the CPU time would guarantee the minimal vulnerable probability for a specific victim vCPU. Take the scheduling traces t_1 and t_2 in Figure 2.3 as an example.

$$\text{subject to} \quad \sum_{\forall a} P(t_1, VM_2, a) = \sum_{\forall a} P(t_2, VM_2, a) = 2$$

$$\max_{\forall a} P(t_1, VM_2, a) = \max_{\forall a \in \{VM_1, VM_3, VM_4\}} P(t_1, VM_2, a)$$

$$= \max\{1, 0.52, 0.48\} = 1$$

$$\max_{\forall a} P(t_2, VM_2, a) = \max_{\forall a \in \{VM_1, VM_3, VM_4\}} P(t_2, VM_2, a)$$

$$= \max\{0.68, 0.60, 0.72\} = 0.72$$

From the results, we can infer that the victim in t_2 has a smaller worst-case vulnerable probability. This advantage stems from a more balanced distribution of CPU time. Combining Eq. 2.1 and Eq. 2.2 we can obtain that:

$$P(t) = \max_{\forall a,v}\{P(t, v, a)\} = \max_{\forall v}\{\max_{\forall a}\{P(t, v, a)\}\} \geq \frac{m-1}{n-1}$$

Thus,

$$\min_{t} \mathrm{P}(t) \geq \frac{m-1}{n-1} \tag{2.3}$$

Based on the above discussions, distributing CPU time to vCPUs with equal probability would reduce the overall vulnerable probability of the system. Thus, I propose to select all candidate vCPUs with equal probability when making scheduling decisions. Later I will demonstrate that such a scheme effectively reduces the overall vulnerable probability to the near optimal value through experiments.

2.4.4 Implementation of the shuffler schedulers

To demonstrate the effectiveness of my solution, I use Xens Credit scheduler as an example, and the scheme could be applied to other schedulers as well. Revisiting the example shown in Figure 2.3, we can observe that the vulnerable probability could reach as high as 100% due to the constant runtime patterns of vCPUs, which originate from two scheduling schemes.

- The tendency of vCPUs to be scheduled to the same core. In the credit scheduler, each core maintains a local run queue (runq) of the active vCPUs. Each time the scheduling routine is triggered, the vCPU currently running on this core is returned to its runq, and the next vCPU to run is selected from this runq.

- VCPU's scheduling in round-robin order. Each runq is managed in a round-robin fashion in the runq. The returned vCPU is appended to the end of the runq of the same priority and the next vCPU to run is selected from the head of the runq.

Intuitively, we can design a deterministic scheduler, which records the vulnerable probability of all vCPU pairs, and each time greedily selects a vCPU that minimizes $P(t)$ in Section 2.4.3. However, the scheduling decision of a deterministic scheduler is predictable. Zhang *et al.* [53] abused the open source Xen hypervisor to trick the Credit scheduler to behave in the attacker's favor. Gullasch *et al.* [57] utilized similar feature in the Linux process scheduler to launch attacks as well.

Thus, I propose to integrate the uniform and random selections in the design of the Xen's Credit scheduler. Following this scheme, I implement the *Local Shuffler (LS)* scheduler. I minimize the modifications to make the implementation lightweight. Specifically, I have the following changes:

1. Runq selection: during the scheduling, a runq is uniformly and randomly selected from all available runqs.

2. VCPU selection: within the selected runq, the next vCPU to run is uniformly and randomly selected from all candidate vCPUs with the highest priority, and the current

running vCPU is returned to the same runq.

In the LS scheduler, a runq is still maintained for each core. In addition, I further propose and implement the *Global Shuffler (GS)* scheduler, where only one global runq is maintained for all cores. In the GC scheduler, the scheduling scheme is modified as following:

1. Runq selection: during the scheduling, all candidate vCPUs are kept in a globally shared runq.

2. VCPU selection: the same with that of the LS scheduler within the selected runq.

I limit the above changes to relevant functions in the source code of the Credit scheduler (*sched_credit.c*). In the following section, I will evaluate and compare the schedulers LS and GS with Credit from multiple perspectives.

2.5 Performance evaluation

In this section, I evaluate the effectiveness of my proposed schedulers from different perspectives. I use the same hardware and configurations as used in Section 2.3.1. The hypervisor utilized for my evaluations is the Xen hypervisor version 4.6.0.

2.5.1 Vulnerable probability

To evaluate the vulnerable probabilities when different schedulers are used, I repeated the experiment 30 times when running 9 VMs and collected 2160 ($30 \cdot \binom{9}{2}$) potential attacker and victim pairs for each scheduler. Figure 2.4 shows the cumulative distribution function (CDF) of the vulnerable probabilities for all possible attacks.

In this figure, the x-axis represents the vulnerable probabilities while the y-axis shows the CDF values. The distributions of the vulnerable probabilities for each scheduler are distinguished by different markers. “Credit”, “LS” and “GS” represents the default Credit

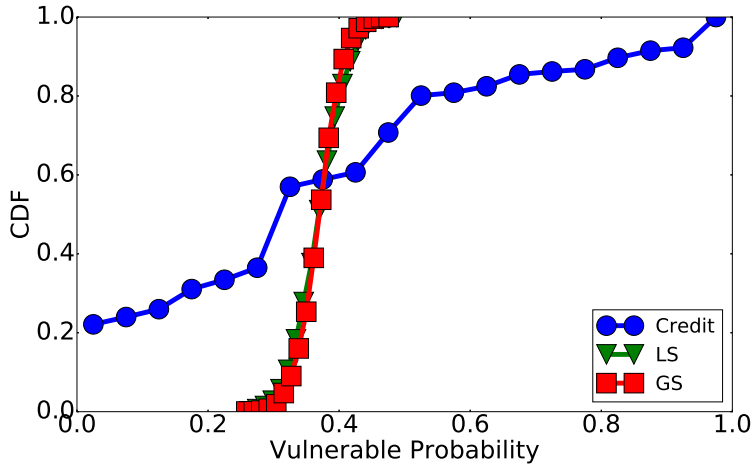


Figure 2.4: Vulnerable probabilities distributions

scheduler, the Local Shuffler scheduler and the Global Shuffler scheduler separately. Furthermore, the maximum vulnerable probability illustrates the worst-case scenario when the potential attackers could obtain the most information from the victim.

In this figure, we can clearly observe that Credit leads to the most widely distributed values, with the worst-case value being 100%. This suggests that an attacker could obtain an almost complete data set of the victim's memory accesses in persistent attacks. In contrast, LS and GS can limit this worst-case value to 49%. Since the attacker can repeatedly launch the same attack in my threat model, the worst-case value indicates the effectiveness of the attacks. I use it in the following evaluations. Another observation is that the vulnerable probabilities of LS and GS is more evenly distributed within a smaller range of 25% - 49% than that of Credit.

There are many factors that may affect the scheduling trace in attacks, including the running time, the vCPUs/cores ratio, the workloads, and the time slice length. Next, I compare the worst-case vulnerable probabilities using different schedulers under various settings (among 2160 possible attacks if not otherwise specified).

I first varied the running time from 1 second to 16 seconds and the results are shown in Figure 2.5. In this figure, the x-axis represents the running time and the y-axis represents

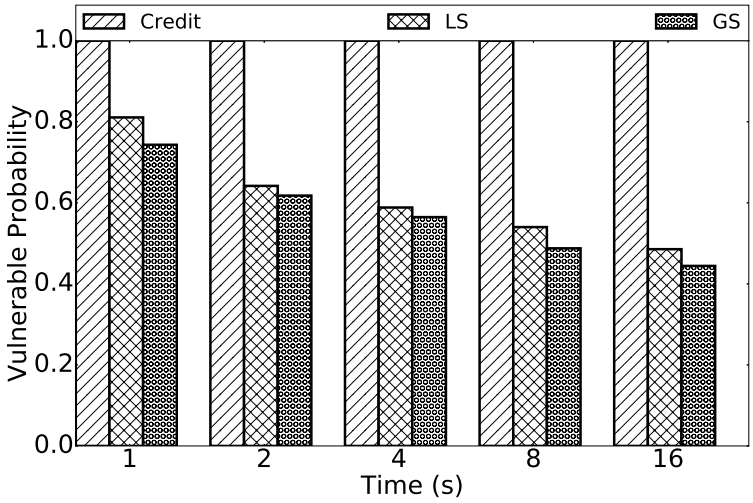


Figure 2.5: The worst-case vulnerable probabilities with different running time

the worst-case vulnerable probabilities. Furthermore, each group of data represents the results of the three schedulers.

We can observe from the figure that for Credit, the worst-case vulnerable probabilities can always reach 100% despite the variation of the running time. While for other schedulers, the worst-case vulnerable probabilities can be effectively reduced by $20\% - 50\%$. In addition, the worst-case vulnerable probability continuously decrease as the running time increases for my proposed schedulers. This suggests that the effectiveness of my schemes is more remarkable for long-term executions due to the more even distribution of time slices to vCPUs.

I further evaluate the worst-case vulnerable probabilities with different vCPUs/cores ratios as shown in Figure 2.6. By increasing the number of VMs, I changed the number of vCPUs from 5 to 12 running on 4 cores as represented on the x-axis in the figure. From this figure, we can see that my solution can also achieve $20\% - 50\%$ reductions. Furthermore, I also observe that the result approximates the optimal value calculated by Equation 2.3.

For different workloads, using sysbench [58], I generated CPU intensive workloads, memory intensive workloads and I/O intensive workloads to evaluate different schedulers. The configuration is shown in Table 2.1. The results are displayed in Figure 2.7. With the

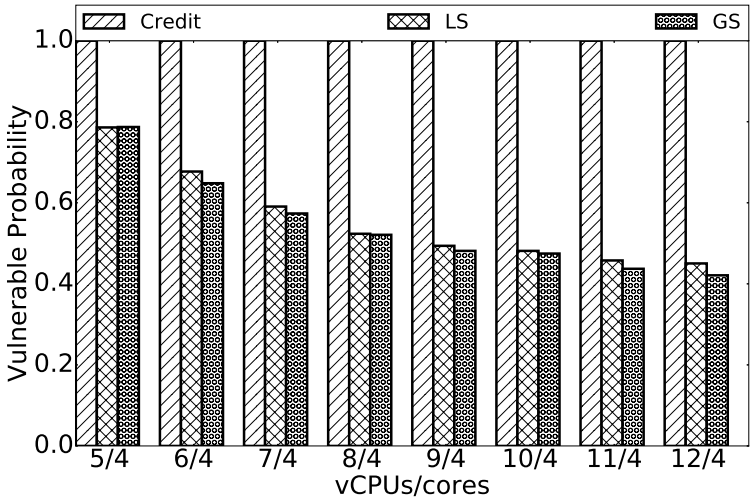


Figure 2.6: The worst-case vulnerable probabilities with different vCPUs/cores

Table 2.1: Sysbench workloads

Workloads	Description	Parameters
CPU intensive	verify prime numbers by doing standard division of the number starting from 1	--test=cpu --max-time=10
Memory intensive	allocate a memory buffer and then write from it randomly	--test=memory --max-time=10
I/O intensive	randomly read/write previously created files	--test=fileio --max-time=10 --file-test-mode=rndrw

Shuffler schedulers, the worst-case vulnerable probabilities can be reduced to below 60%.

Finally, I changed the time slice length configured with parameter *tslice_ms* (default 30) in Xen, from 1 ms to 16 ms for all schedulers. The result is shown in Figure 2.8. My schedulers can effectively reduce the worst-case vulnerability by more than 50%.

To sum up, for the worst-case vulnerable probability in various settings:

1. It can reach almost 100% in most settings when the default Credit scheduler is used.
2. My proposed Shuffler schedulers can reduce it to below 80%.
3. The GS scheduler is slightly more effective than the LS scheduler, suggesting a global queue implementation is preferable for evenly distributing CPU time.

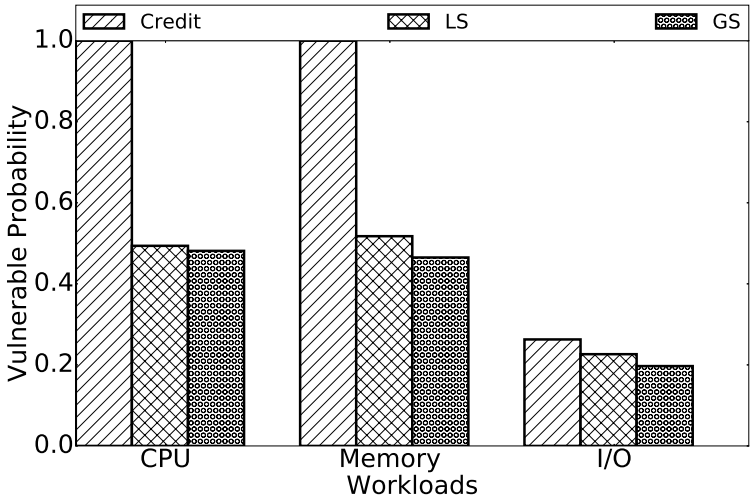


Figure 2.7: The worst-case vulnerable probabilities with different workloads

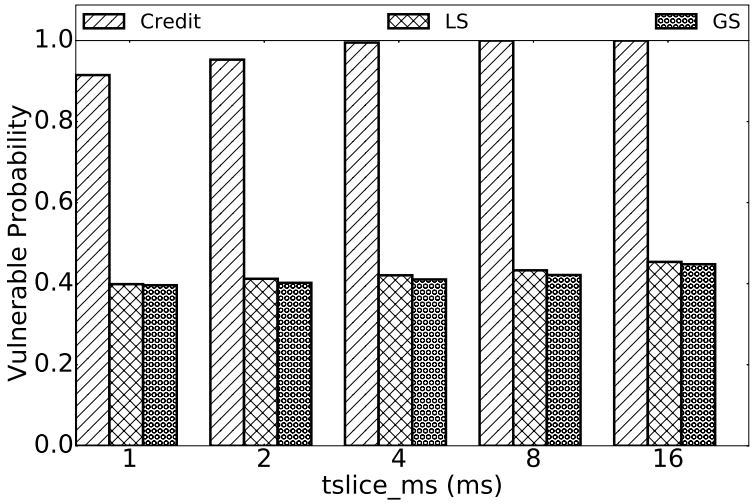


Figure 2.8: The worst-case vulnerable probabilities with different time slice lengths

2.5.2 Recovery rate

In this section, I reproduce the same Prime+Probe attack used in Section 2.3.1, and evaluate how the Shuffler schedulers reduce the (key bits) recovery rate compared to the Credit scheduler.

Figure 2.9 shows the CDF of the recovery rates for repeated attacks. From the figure,

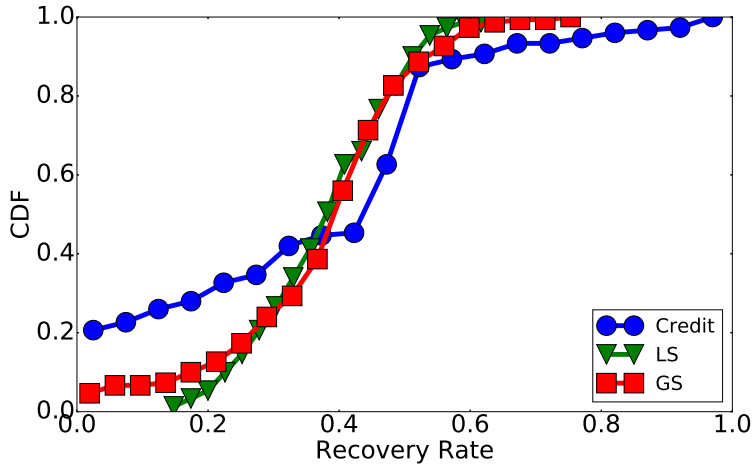


Figure 2.9: Recovery rates distributions

we can clearly observe that Credit leads to the most widely distributed values, with the worst-case recovery rate being 99%. In contrast, LS and GS limit the worst-case recovery rate to 66% and 78%, respectively. The worst-case recovery rate indicates the effectiveness of persistent attacks. I use it in the following evaluations.

I further evaluate the worst-case recovery rates with different vCPUs/cores ratios as shown in Figure 2.10. I also changed the number of vCPUs from 5 to 12 running on 4 cores as represented on the x-axis in the figure, and present the highest recovery rates on the y-axis. The number of attacks repeated for different vCPUs number is $30, 60, 90, ..., 240$, respectively.

From the figure, we can see that my solution can limit the recovery rates to below 85%. In my experiment, all background VMs run CPU intensive workloads that introduce little noise to the attacks. In the cloud environment, the existence of error bits will further reduce the worst-case recovery rates.

Furthermore, the worst-case recovery rates can be further reduced to the near optimized value shown in Eq. 2.3. This can be confirmed by repeating the same attack while setting smaller tslice_ms value, as shown in Figure 2.11. I configured vCPUs/cores to 5/4, and changed the tslice_ms from 1 ms to 16 ms for all schedulers. The worst-case recovery rates

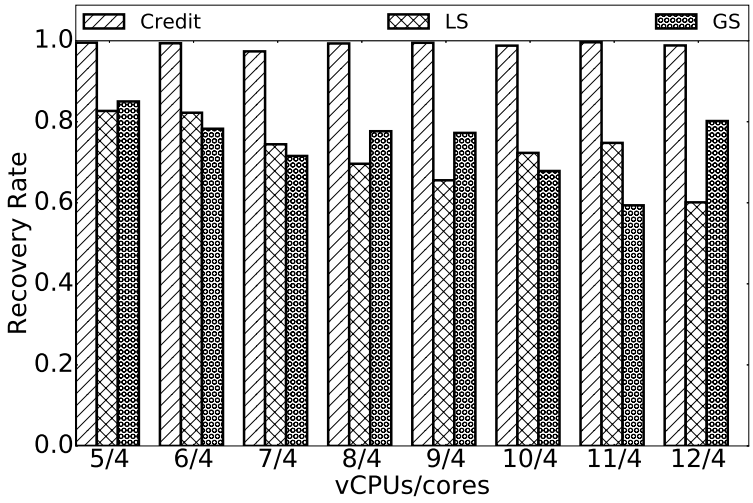


Figure 2.10: The worst-case recovery rates with different vCPUs/cores

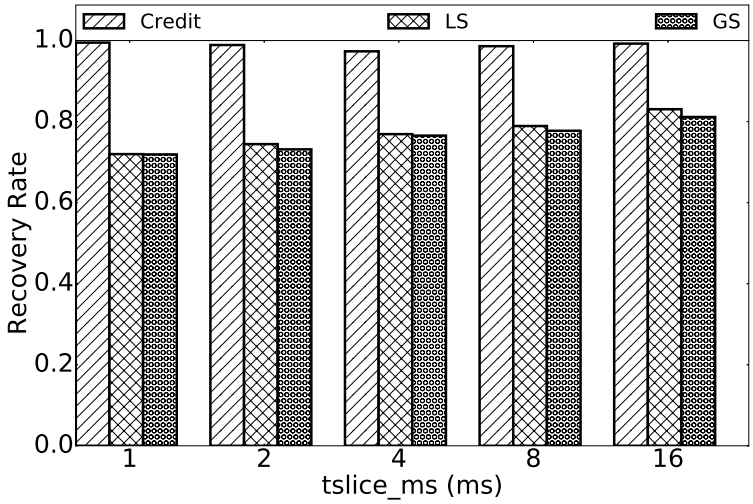
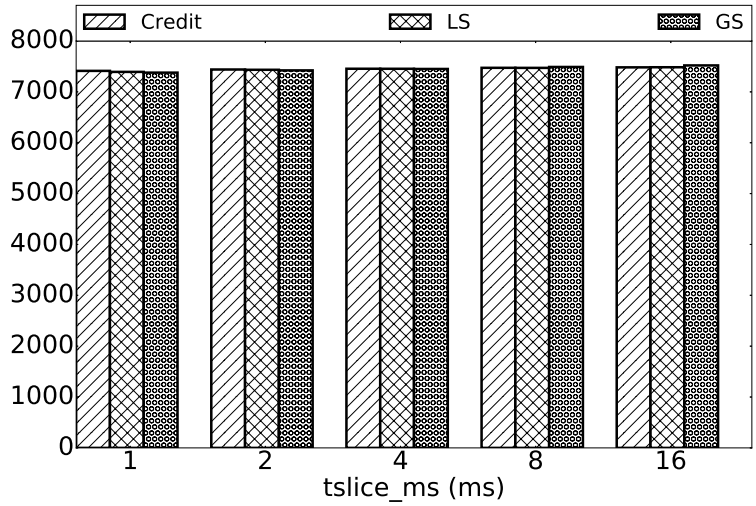
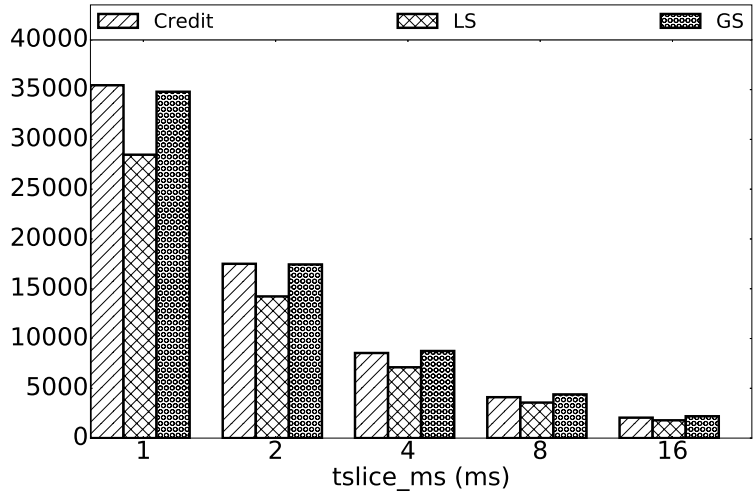


Figure 2.11: The worst-case recovery rates with different time slice lengths

are reduced to below 72% for the Shuffler schedulers, from 99% for the Credit scheduler, when tslice_ms is set to 1 ms. The worst-case recovery rate of 72% makes reconstructing the full key infeasible, as I will discuss in Section 2.6.3.



(a) Overall number of events executed in all VMs.



(b) Overall number of vCPU context switches.

Figure 2.12: Overhead of different schedulers

2.5.3 Scheduling overhead

Besides security, I also evaluate the performance overhead of the Shuffler schedulers. The overhead incurred by the schedulers mainly comes from the CPU time consumed for scheduling operations and performance penalty due to extra context switches. I will evaluate them separately.

For the CPU time consumption, I measure the system-wide performance when executing CPU intensive workloads in all VMs. The performance of each VM is reported by sysbench [58]. During a given period, the more CPU time consumed by the scheduling operations, the less number of events can be executed in the VMs. For the performance penalty, I count the total number of vCPU context switches during the same period. I use these two metrics together to profile the overhead for each scheduler, as shown in Figure 2.12. Intuitively, the smaller the time slice length, the higher frequency the scheduler is triggered, thus the higher overhead. So I used different tslice_ms values in this experiment as shown in the x-axis.

In Figure 2.12a, the y-axis represents the system-wide performance during 10 seconds. The higher the better. Each bar shows the average value among 150 repetitions. We can observe that decreasing the tslice_ms values always imposes extra overhead, which can reach up to 2%. However, the extra overhead introduced by the Shuffler schedulers using the same tslice_ms values is less than 0.5% compared to the Credit scheduler.

In Figure 2.12b, the y-axis represents the total number of vCPU context switches during the same period. The lower the better. As expected, the number of vCPU context switches decreased by half as I doubled the tslice_ms value. Furthermore, I also find that GS caused similar number of vCPU context switches compared to Credit, while LS reduced this number by $13\% - 20\%$. It is because the current running vCPU has higher probability to continue running when the LS scheduler is used.

To sum up, the Shuffler schedulers introduce negligible overhead compared to the Credit scheduler, since they neither consumed more CPU time in scheduling operations, nor generated more vCPU context switches.

2.6 Discussion

2.6.1 Colluding attacks

In my threat model, I assumed the state-of-the-art attacking scenarios in which a single attacker vCPU was used [14, 49, 55]. I argue that colluding attacker vCPUs could hardly gain more advantages over single attacker vCPU, based on two observations: (1) To avoid the mutual pollution of the monitored cache sets, no overlapping executions of the attacker vCPUs are desired, since they spy the shared resources by "writing". This requires the cooperation of the hypervisor, which is not possible without compromising the hypervisor. (2) To stitch the collected information together, the attacker vCPUs need to synchronize at overly high frequency, which is not readily available. Thus, the capability of colluding attacks is limited by my proposed solution as well.

2.6.2 CPU overcommitment

In my thread model, I assumed CPU is overcommitted in the physical hosts. CPU overcommitment is commonly used to consolidate the VMs to save power consumption and to improve resource utilization [59–61]. For example, the default vCPUs/cores ratio is 16 in OpenStack [62]. Though for public clouds, the vCPUs/cores ratio is not disclosed by the cloud providers, there are some clues to overcommitment. For example, for the latest general purpose T2 instances in Amazon EC2, the vCPUs/cores ratio may be 5 (12 credits/hour), 10 or 20, etc.[63]

If CPU is not overcommitted, the idle cores can be used to inject noise to the side channels to mitigate attacks. Zhang et al. [64] had discussions in this direction. Such a mechanism can be a good complement to my solution.

2.6.3 Key reconstruction

The evaluation shows that when my proposed Shuffler schedulers were used, the attacker could only collect less than 72% key bits in a single attack. As a result, the side-channel

attacks fail since it is infeasible to guess the missing 28% key bits.

One may wonder if advanced crypto key analysis techniques enable the attacker to reconstruct the full key string from such partial keys. For example, when there was bits corruption in the partial keys, Heninger and Shacham [65] managed to reconstruct the full key string by utilizing the redundant information in the key string. However, such key reconstruction methods cannot directly apply to my case, since the partial keys have missing bits at random, and key bits positions are incorrect.

Alternatively, the attacker could accumulate collected information across multiple partial keys to reconstruct the complete key. Similar problem was discussed in the context of deletion channel [66–68]. In this model, a transmitter sends a bit and the receiver either receives the bit (with probability $1-P$) or does not receive anything without being notified (with probability P). Similarly, I state the key reconstruction problem when the Shuffler schedulers are used:

For a victim's key bits sequence $X = x_1x_2...x_l$ of length l. In each attack, the attacker can collect a subsequence, Y_1, Y_2, ..., Y_n, where each Y_i is obtained independently by deleting each of X's element with probability P. Under these conditions, what is the number of subsequences (n) needed to reconstruct X with high probability?

I model the attacker's data collection in this way, since the hypervisor scheduling is transparent to the attacker, and the Shuffler schedulers promised vulnerable probabilities to be smaller than 72%. Then bits deletion is inevitable with $P \geq 28\%$.

The state-of-the-art result of this problem to the best of my knowledge is [66]:

$$\begin{cases} n = O(l \cdot poly(logl)), & P \leq (1/\sqrt{l}) \\ n = exp(\sqrt{l} \cdot poly(logl)), & \text{any } P \end{cases}$$

Considering $P = 28\%$ and $l = 4096$, only the second result applies to my key reconstruction problem. In this case, the number of repeated attacks required to reconstruct the full key increases exponentially with $\sqrt{l} = \sqrt{4096} = 64$, making the attacking time unaffordable

when my proposed Shuffler schedulers are used.

To sum up, my scheduling-based scheme effectively mitigates cross-VM side-channel attacks.

2.7 Related work

Side-channel attacks have attracted a lot of attention during the past and many schemes have been proposed to mitigate such attacks. For many attacks, visible timing difference for different hardware events is required. Askarov et al. [69] provided a timing mitigator to bound the information leaked through the timing channel. To eliminate the timing channel, Stefan et al. [70] proposed an instruction-based scheduling, Vattikonda et al. [71] suggested to remove the fine grained timer in Xen. However, obfuscating timing information negatively affects benign cloud tenants as well, and the attacker may obtain precise timing information using other methods [72]. Zhang et al. [64] introduced bystander VMs running configurable workloads to inject noise to covert channels, which could be a good complement to my solution.

Crypto operations are often the target of side-channel attacks. Gueron [7] proposed a new modular exponentiation implementation to secure RSA against side-channel attacks. Raccoon [8] was also proposed to harden programs against side-channel attacks by obfuscating the program at the source code level. However, the modified programs may still be vulnerable to new side-channel attacks [9].

Compared to harden individual program, a more general solution is to secure the cloud environment against side-channel attacks. Dedicated host service provided by the cloud providers [10] can be used to physically isolate VMs from all other VMs, thus preventing them being attacked. However, dedicated hosts come at the cost of higher price for the cloud user and lower resource utilization for the cloud provider. Alternatively, certain degree of VM isolation could be achieved by carefully placing and frequently migrating VMs in its life-cycle [13, 47, 48]. Based on prior studies [13, 73] and confirmed by my measurements,

it took about 1.47 seconds to live migrate an instance with 2048MB RAM and 7GB hard drive via 1Gbps network. Furthermore, the latency of such migrations could be translated into monetary cost of the providers ranging from $1 to $100.

Once the attacker and the victim run their cloud tenants in the same host, different shared resources may be exploited to launch side-channel attacks. Various prior efforts focused on mitigating known attacks via different shared resources, such as networks-on-chip [74], memory controller [75], memory pages [76], CPU caches [11,12,77,78], etc. Specifically, to mitigate cache side-channel attacks, Godfrey and Zulkernine [77] proposed to flush caches during context switches, Wang and Lee [78] suggested a new cache design, Liu et al. [11] proposed to partition the LLC for each cloud tenant using Intels Cache Allocation Technology, Kim et al. [12] designed a memory page coloring scheme to prevent usage patterns of sensitive data being leaked. These defenses are often effective to a specific group of attacks. However, they usually introduced significant overhead, reduced resource utilization, and even required far-reaching changes to the hardware.

A finer-grained isolation via scheduling is more economically desirable for both the cloud user and the cloud provider. Hu [79] discussed the impact of scheduling policy on hardware timing covert-channel and proposed a lattice scheduler for process scheduling. In virtualized environment, the impact of various scheduling factors, including load balancing, weight, cap, time slice and (context-switch) rate limiting, on covert-channel attacks were studied [80]. Varadarajan et al. [54] found that the attacker needed to measure the cache state frequently in side-channel attacks, and that the efficacy of such attacks can be dramatically reduced by enabling the minimum runtime guarantee feature in Xen. They both targeted side-channel attacks via core-shared resources such as L1 cache. In comparison, my previous work [50] studied how different factors affected the more advanced side-channel attacks via cross-core shared resources such as LLC. In this work, I revealed that the root cause of the side-channel leakage is the runtime resources sharing patterns between cloud tenants. I had an in-depth discussion of it, defined the vulnerable probability to quantify it, and proposed a scheduling-based mechanism to reduce it. In addition, the extra overhead is less than 0.5%

in my evaluation.

2.8 Summary

The multi-tenancy in the cloud infrastructure enables side-channel attacks to be launched by co-locating VMs. In this chapter, I revisit the cache-based side-channel attacks where the attackers exploit the shared hardware resources such as CPU cache. Unfortunately, existing solutions either fail to provide sufficient protections at economic costs or limit their scope to specific attacks. In this chapter, I propose a lightweight and generic solution to eliminate a wide range of cross-VM and possibly unknown attacks. My thorough analyses have demonstrated that the efficacy of such attacks could be dramatically reduced by distributing CPU resource as evenly as possible to all candidate vCPUs. Accordingly, I have designed and implemented the Shuffler schedulers by incorporating this strategy and randomization into Xen's Credit scheduler. The evaluation results show that the Shuffler schedulers significantly reduce the vulnerable probabilities of all VMs, thus mitigating attacks without sacrificing the original resource sharing or performance.

Chapter 3: vCPU as a container: towards accurate CPU allocation for VMs

The work presented in this chapter has been published in [81].

3.1 Introduction

The adoption of cloud computing has become increasingly popular among various Internet services. Cloud computing enables the flexible provisioning and sharing of computing resources 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 guest domains' resource usage. Consequently, some guest domains may consume significantly more resources than allocated, or *resource overuse*, as I refer to in this chapter.

Among various types of resources, the CPU is the most important one. Its accurate allocation and management directly affect the operations and the revenue of the cloud providers like Amazon and Google. According to [15], each physical CPU core sells for a maximum potential annual revenue of $900. However, as previous works [16, 17] have observed, and I will further demonstrate in Section 3.3.1, a guest domain can consume up to 70% more of its allocated CPU time, preventing cloud providers from selling those 70% overused CPUs to other clients and resulting in a noteworthy monetary loss. The resource overuse issue may also potentially degrade the performance of neighbor domains [18–25] and increase the energy consumption of the host machines [26, 27].

Similar concerns over the resource overuse issue have been raised previously in non-virtualized environments for processes [82] and containers [83]. In this work, I investigate the problem in the virtualized environment, which both imposes unique challenges compared

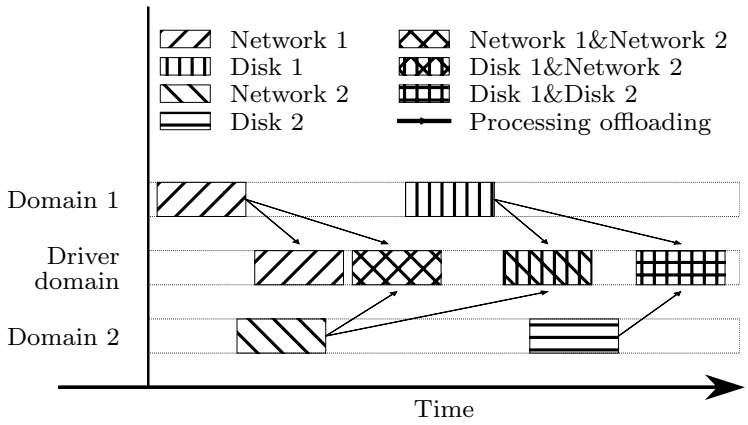


Figure 3.1: Illustration of software-based I/O virtualization: much of I/O processing is offloaded to the driver domain, which is not accounted to its source domain (domain-1/domain-2). Such offloading processing: 1) is asynchronous with processing in the source domain; and 2) interleaves with each other in the driver domain.

to previous works and provides new opportunities to enable a more accurate and lightweight solution than existing ones in non-virtualized environments.

In the virtualized environment, one major contributing factor to the resource overuse problem *offloading* in software-based I/O virtualizations. An illustrative example is shown in Figure 3.1. With the software-based I/O virtualization, I/O devices are managed by the driver domain (or the hypervisor). 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, the guest domains may effectively consume more CPU resources by burdening the driver domain. The case also applies to the shared intrusion detection system [28] 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 the virtualized environment. The

semantic gap between the hypervisor and domains and the offloaded processing's asynchronous nature pose significant challenges to measure the offloaded CPU usage accurately. Previous works attempted to overcome such a semantic gap using VM-introspection techniques [28–30], at the cost of complicated kernel tracing and heavy runtime overhead, limiting its usage in modern cloud systems. Others [31, 32] attempted to circumvent these challenges by estimating the offloaded CPU time instead of accurately measuring. Such estimation is based on the assumption that offloaded CPU usage for the same workload is always the same. As I will show in Section 3.3.2, this is not necessarily true. Therefore, the estimation-based approach also fails to produce accurate accounting results.

In this chapter, I claim that this problem's root cause 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 a coincidence in the current design prevents the hypervisor from correctly allocating resources to each domain.

In this work, I aim to tackle the problem 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, I 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 comprises 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 manage the CPU resource per domain accurately.

To demonstrate my proposed approach, I implement *VASE System*, a novel and lightweight solution built on top of the Xen hypervisor. The evaluations in various settings

show that my approach effectively manages the system-wide CPU consumption incurred by the guest domains with virtually no overhead.

To summarize, the contribution of this chapter is three-fold:

- I distinguish the resource scope from the protection scope in virtualized systems and redefine the resource scope using existing vCPU abstraction, enabling accurate resource management per domain.
- My 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, my approach eliminates explicit communications between the hypervisor and domains, 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 chapter is organized as follows: In Section 3.2, I will provide the background on I/O virtualization and CPU management in Xen. This work's motivation will be presented in Section 3.3, with experiments showing the drawbacks of state-of-art, followed by the problem statement, challenges, and solution in Section3.4. The design of my *VASE System* will be presented in Section 3.5, followed by evaluations in Section 3.6. Discussions and related work will be provided in Section 3.7 and 3.8. I will conclude this chapter in Section 3.9.

3.2 Background

3.2.1 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

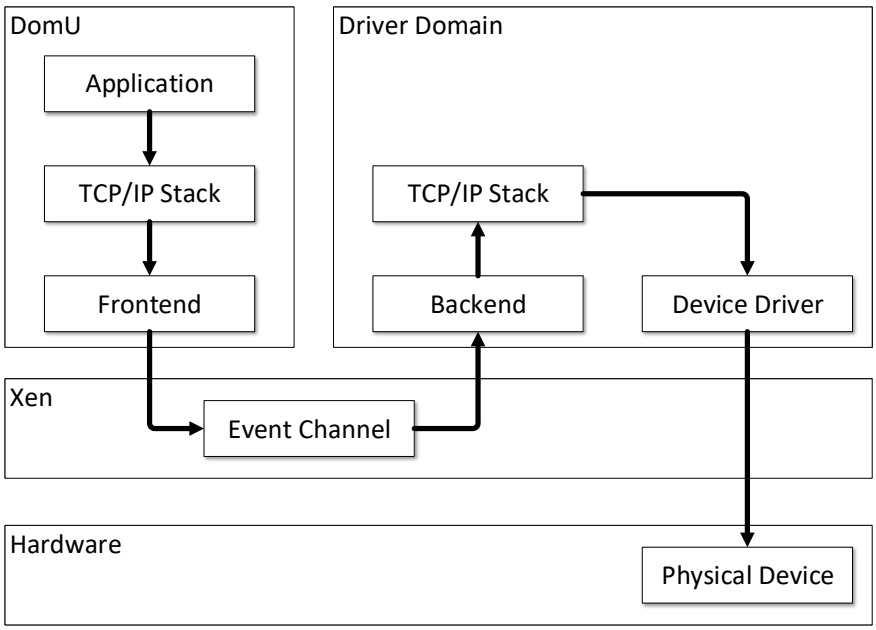


Figure 3.2: The path of sending a packet in Xen [1]. In the split driver model, a large portion of I/O processing happens in the driver domain, consuming a significant amount of CPU resource on behalf of the DomUs.

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 [84, 85].

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 3.2 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. I 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 I will briefly summarize how CPU resources are managed and how software-based virtualized I/O design may potentially result in inaccurate resource management.

3.2.2 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 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. [19] 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

Table 3.1: 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*

incur more CPU usage than allocated, causing performance degradations and variations [16, 31, 32].

3.3 Motivation

Suppose I have a Xen virtualized environment where only one DomU is running I/O workload with its cap set to 100. Intuitively, I 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, I will investigate how significant such excessive usage is and show the drawbacks of the estimation-based approaches [16, 31, 32].

For all the experiments in this chapter, I use an HP ProLiant DL380 G6 server as my 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 times to eliminate 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

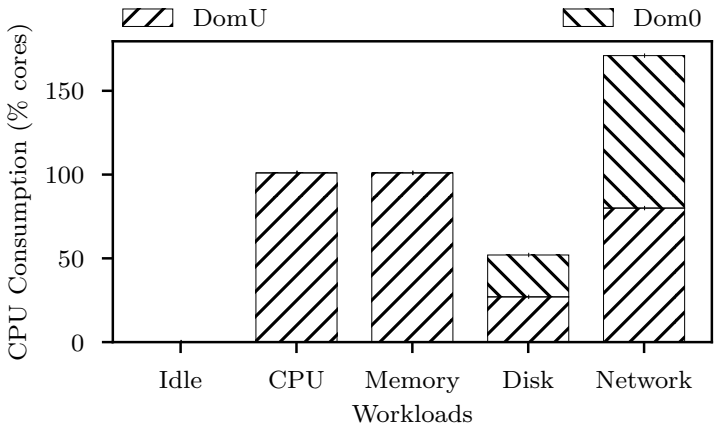


Figure 3.3: System-wide CPU usage with one DomU running various workloads. The total CPU usage reaches more than 170% (of a CPU core) in case of I/O intensive workloads, exceeding the amount of 100% that is allocated to the DomU. The exceeding part is "stolen" from other domains.

16.04 and Xen 4.9.0 is used as the hypervisor. On each DomU, *iperf3* [86] and *sysbench* [87] are used to generate synthetic workloads listed in Table 3.1. I use these workloads here and also in Section 3.6. CPU utilization is measured by running the workload and collecting the active vCPU time over a ten-second period.

3.3.1 Offloaded CPU time is significant

First, I verify that the offloaded CPU time is non-negligible. To this end, I generate four types of workload in one DomU: CPU, MEM, SEQ, and UDP, as listed in Table 3.1. The result is shown in Figure 3.3, 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 the physical host.

Since my 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 [16, 31, 32] addresses this issue based on estimations and I will show its drawbacks next.

3.3.2 The estimation approach is inaccurate

To study the accuracy of the estimation approach in determining offloaded CPU consumption, I have replicated the profiling and estimation technique proposed in [16, 31, 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, I 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 my proposition.

To start with, I show that profiling, as a key step in the estimation-based approach, cannot generate stable results in the multi-tenant cloud environment. I profile the same TCP workload while running various combinations of DomUs in the same host. I repeat the profiling 100 times and the result is shown in Figure 3.4, 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

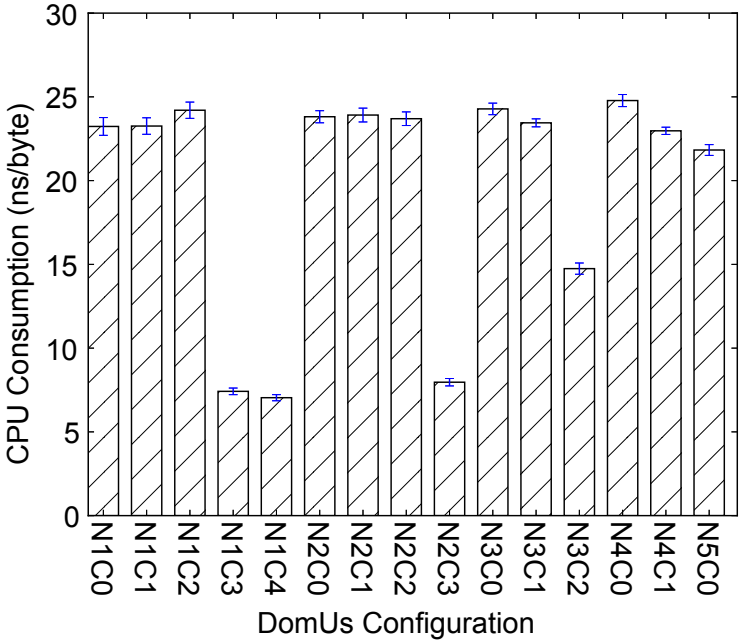
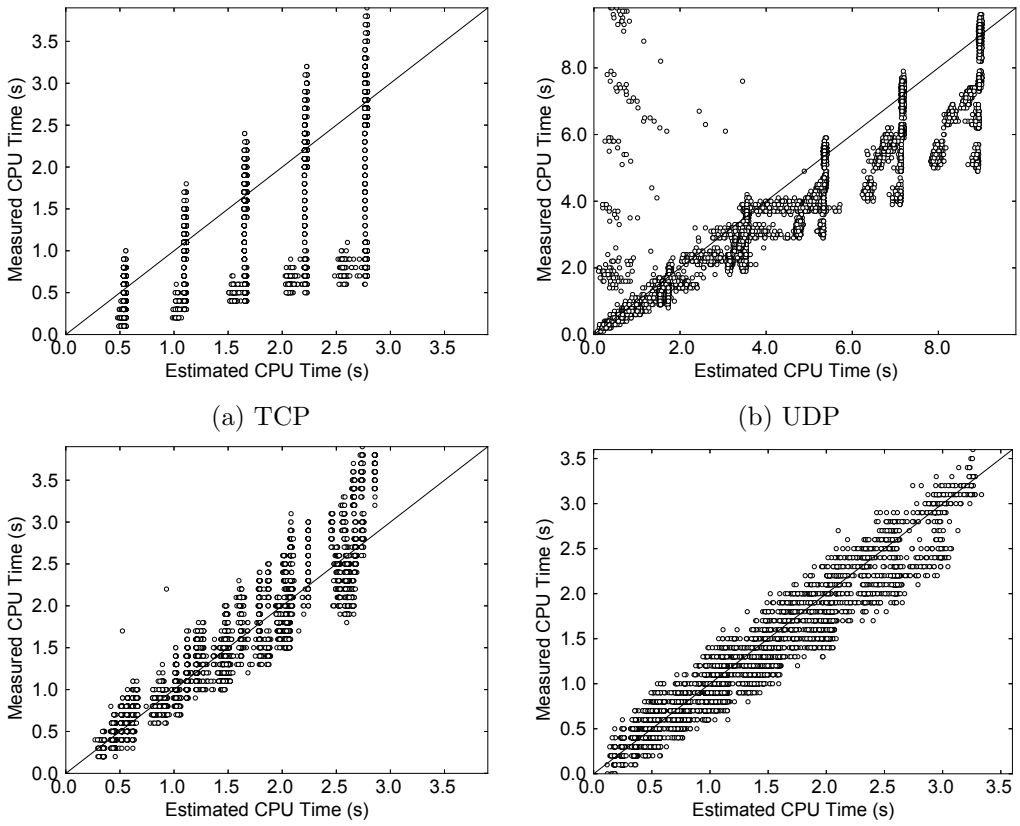


Figure 3.4: The profiling result of the estimation approaches, which proves it is false to assume the same workload always incurs the same amount of offloaded CPU consumption to the driver domain, especially with neighbors.

different combinations, e.g., 4 times between N1C2 and N1C3.

Next, I demonstrate the estimation error by selecting one of the profiling results in the previous step to estimate 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 3.5a, 3.5b, 3.5c, and 3.5d, 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 the 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



(a) TCP (b) UDP

(c) SEQ (d) RND

Figure 3.5: 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 the network and 50% for the disk.

and accurate approach to solve this issue.

3.4 Problem, challenges and solution

I 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 chapter as *debt*. Hence, the first question we want to answer in this chapter is that, *can I 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 3.1, 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 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, I wonder *what the fundamental cause behind the existence of the debt is*. In the example of Figure 3.1, 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, I 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*. I 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, I 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 I 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, I implement *VASE System* to demonstrate my solution.

3.5 VASE system

In this section, I 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, my 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 chapter.

The overall design of *VASE System* is shown in Figure 3.6. As aforementioned, the resource allocation problem is caused by the ill-defined resource scope of the domains in the current virtualization design. My 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.

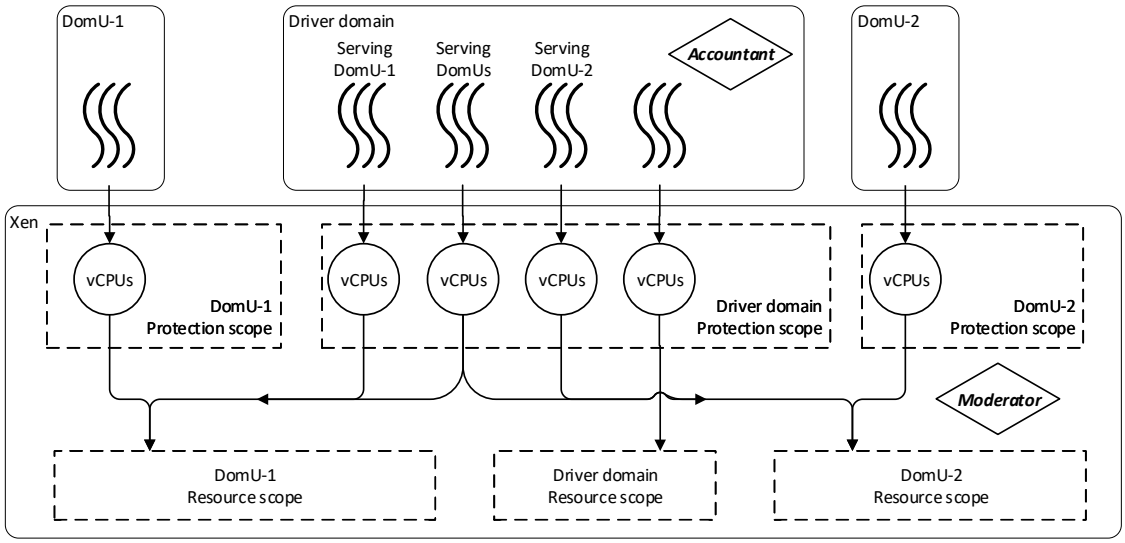


Figure 3.6: The overall design of *VASE System*. The resource scope of each domain is defined by the actual resource consumption of that domain as opposed to falsely defined by the protection scope. The existing vCPU abstraction is used as resource container to isolate and encapsulate the offloaded workload. *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.

3.5.1 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 I explain how this is achieved, I 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. I 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, I 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, I 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, I 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.

Identify the Workers

With careful examinations of the design and code path of Linux kernel and Xen split-driver, I have implemented the *Accountant* to identify all offloaded I/O processing workers in my test environment. Figure 3.7 shows these IRQs and kernel threads for block and network backend device drivers. Each device type (network or block) has two components

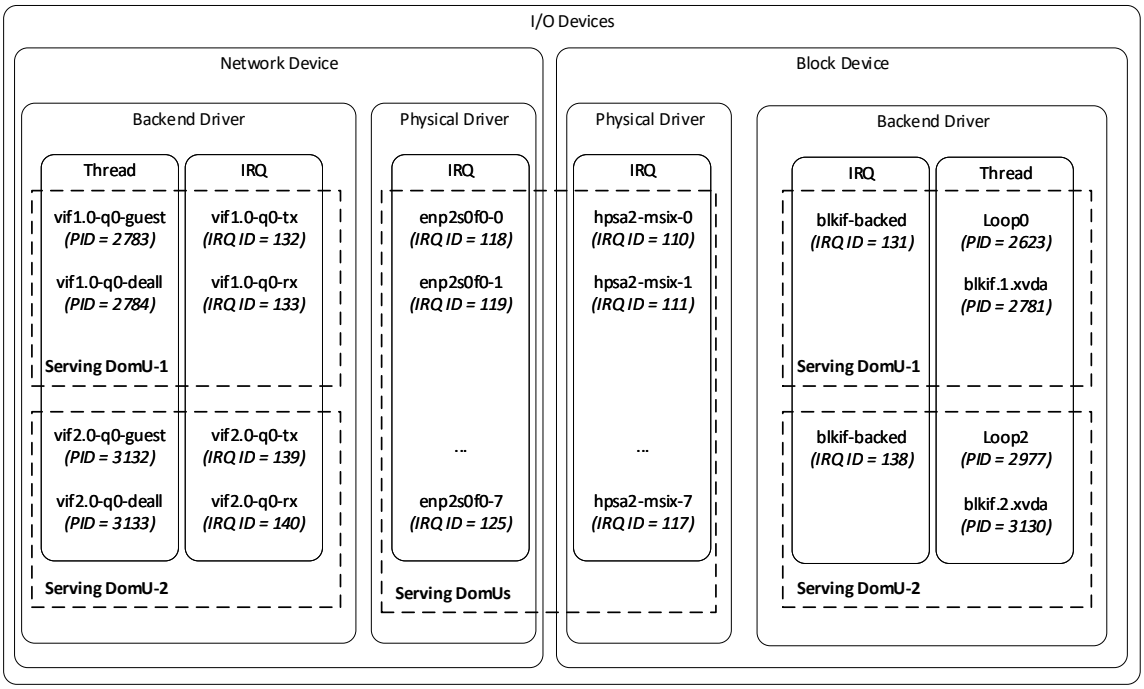


Figure 3.7: Identification of related workers in the driver domain for each domain. In this example, a network device and a block device are allocated to both DomU-1 and DomU-2. Thread "vif1.0-q0-guest", with PID = 2783, serves the virtual network device in DomU-1. IRQ "enp2s0f0-0", with IRQ ID = 118, 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.

— 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 3.7 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.

Table 3.2: Processes and IRQs affinity setting in the driver domain. As shown in Figure 3.7, 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)

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 I 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 I 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 [88]. Hence, once I have pinned the hardware IRQs, the rest of the processing will be executed on the same vCPU. Table 3.2 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, I can also get the offloaded CPU consumption for DomU-2 disk processing by checking the runtime of vCPU-2 and 6.

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, I need to prevent the load balancing facilities in the driver domain from dismantling my affinity settings. Meanwhile, I cannot simply disable load balancing facilities completely as I want them to continue balancing other irrelevant workloads in the driver domain. To this end, for schedulers, I 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.

3.5.2 The *Moderator* component

So far, the *Accountant* has enabled the hypervisor to measure the offloaded CPU usage for I/O workloads. I call this offloaded CPU usage as the *debt* of DomUs. Here I 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, I 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 3.2, for the past $30ms$, 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.

3.6 Evaluation

I have implemented the *VASE System* in Xen 4.9.0. The roadmap for evaluating my prototype is as follows. I 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 my approach against the given CPU caps. Finally I show that my approach introduces negligible overhead to both the scheduling process and the execution of the workload.

Here I use *Vase* to represent my 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 [16, 31, 32] have been evaluated in Section 3.3.2, and therefore will not be repeated for comparisons here. In my 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

Table 3.3: 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 3.2. Each vCPU only consumes CPU time if and only if corresponding I/O workload runs in corresponding DomUs.

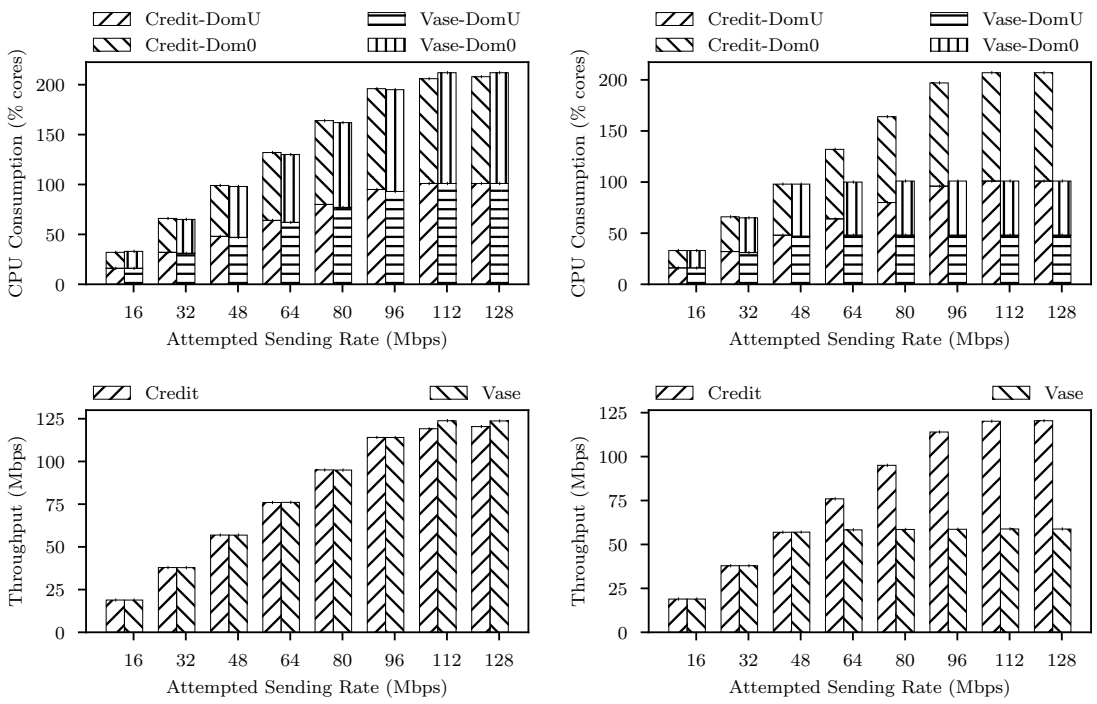
Case #	Workloads		Dom0 vCPU							
	DomU-1	DomU-2	0	1	2	3	4	5	6	7
1	Idle	Idle	-	-	-	-	-	-	-	-
2	CPU	MEM	-	-	-	-	-	-	-	-
3	UDP	Idle	-	-	-	12%	-	76%	-	-
4	Idle	UDP	-	-	-	9%	-	-	-	76%
5	SEQ	Idle	-	-	4%	-	21%	-	-	-
6	Idle	SEQ	-	-	4%	-	-	-	20%	-
7	UDP	UDP	-	-	-	10%	-	38%	-	48%
8	SEQ	SEQ	-	-	5%	-	13%	-	11%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

figures, though most of them are too small to be visible.

3.6.1 Verifying the workload encapsulation

My 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 my 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, I create two DomUs using the configuration shown in Table 3.2 and run various combinations of the workload listed in Table 3.1 on these two DomUs. The vCPU utilization is shown in Table 3.3, where "–" indicates such utilization is negligible.

As shown in Table 3.3, 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, I conclude all



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

Figure 3.8: 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.

offloaded I/O processing capsules has been correctly and thoroughly identified and pinned to designated vCPUs in Dom0 as expected.

3.6.2 Accurate CPU resource allocation

Next, I evaluate the effectiveness of my *VASE System* to accurately enforce the capacity limits. In the experiment I let one DomU run MIX workload listed in Table 3.1 with various intensities and comparing the CPU consumption incurred with and without my 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 3.8a and 3.8b, 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).

I first examine Figure 3.8a 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 network throughput increase as expected and are not capped in both *Credit* and *Vase*.

Next in Figure 3.8b I 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 3.8b (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 pre-configured amount of CPU cores and its CPU usage should be limited accordingly. *VASE System* enables the cloud providers to achieve this exact purpose.

3.6.3 System overhead

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

I first examine the scheduling overhead. With my 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, I let DomU run CPU workload in Table 3.1 for 10 seconds, record the events/sec value reported by *sysbench* 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 3.9, the

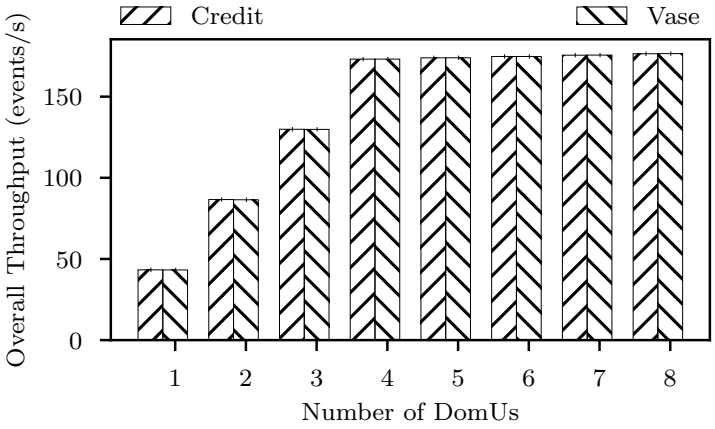
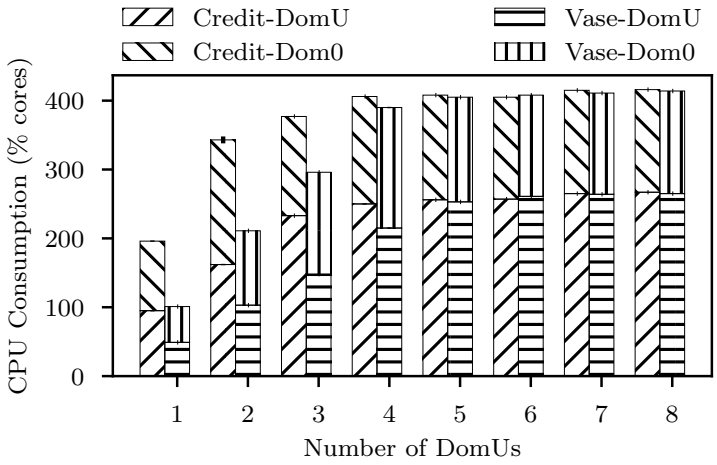


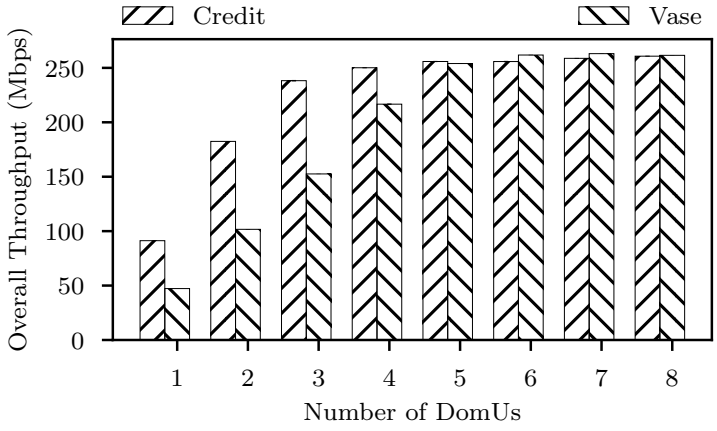
Figure 3.9: Overall throughput with various number of DomUs running CPU intensive workloads, which shows *VASE System* introduces negligible overhead to CPU throughput.

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.

I create various numbers of concurrently active DomUs running MIX workload with each DomU's cap set to 100. The result is shown in Figure 3.10. The x-axis represents the number of concurrent DomUs and the y-axis represents the total CPU usage in the Figure 3.10a and overall throughput in Figure 3.10b. 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 my approach is indeed lightweight. 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.



(a) Overall CPU usage



(b) Overall I/O throughput

Figure 3.10: 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.

Therefore, I believe *Vase* provides a better solution than *Credit* for CPU management in the cloud.

3.7 Discussion

Limitation Though my proposed approach is implemented in Xen-based virtualization systems with paravirtualized I/O devices (both PV and HVM [89] instances), the idea

behind my 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 I 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 [90] via virtio and VMware Workstation [91] via its guest tools. Besides paravirtualized I/O, my 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 my approach.

Hardware-based I/O virtualization solutions [92] 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 [31, 82, 83], my solution applies to not only network I/O but also disk I/O, which also represents a huge number of hosts.

Scalability My *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 my 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 I 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*, I 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.

3.8 Related Work

Accurate resource accounting and allocation is a long-standing research challenge in various contexts including both non-virtualized and virtualized environments [26, 93–98]. 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 [99] and Resource Container [82] 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 [83] was proposed to address the same issue for container-based multi-tenant environments. Ghanei et al. [100] 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 [28] 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, my solution is easy to implement and lightweight by exploiting the existing

vCPU abstractions. As mentioned previously in this chapter, software-based I/O virtualization is a major source of accounting inaccuracy. Cherkasova and Gardner [16] and Santos et al. [17] 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. [31] 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 I 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, my solution generates much more accurate accounting of the offloaded CPU consumption.

When it comes to hardware-based virtualization techniques, such as SR-IOV [92] and SmartNIC [15], 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, my work solves same problem at the software level without specialized hardware and hence its additional cost.

3.9 Summary

Cloud computing relies on accurate resource allocation of domains (VMs) to better serve the needs for both cloud users and providers. In this chapter, I have shown the current

approaches fail to correctly account all the CPU usage incurred by domains due to I/O offloading. I 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, I 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 my solution, I 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. My experiments have shown that my approach is lightweight and effective in constraining CPU usage with virtually no overhead.

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

Chapter 4: Bridging the gap between promise and reality: performance isolation in container-based multi-tenant clouds

4.1 Introduction

The microservice-based architecture decomposes the traditional monolithic applications into multiple loosely-coupled, single-purpose service modules. The transformation into microservices improves service scalability and reliability. It has been widely adopted into the modern software DevOps workflow. Microservices are typically deployed as containers on top of the multi-tenant cloud infrastructure. They are managed by container orchestration systems such as Google Kubernetes Engine [33] and Amazon Elastic Kubernetes Service [34]. Developers request a certain amount of resources, such as CPU and memory, for the containers based on their estimations of workload demand and performance expectations. However, it has been observed that the performance of containers running in the cloud can vary significantly and is difficult to predict [35–42]. For example, It is reported in [36] that container co-location can lead to more than 66× tail latency increasing. Such significant performance variations cause severe performance bottlenecks such as stragglers for batch workloads [43] and violation of end-to-end QoS guarantee for latency-sensitive, interactive applications [44].

It is well-known that such performance variations often come from *noisy neighbors* in a multi-tenant cloud environment. However, the specific sources and mechanisms for the impact of such a magnitude remain unclear. Contentions at the hardware level, such as CPU cache, bus, I/O devices, are commonly believed causes. However, in this chapter, I argue that, for the container environment, the primary source of the variation lies in the operating system (OS) and specifically its scheduling mechanism.

My investigation started with a series of experiments to reproduce the performance variations of containers running in a multi-tenant environment. Significant performance degradation was found despite different levels of hardware contentions. More importantly, I also discovered that the container's actual CPU usage is significantly less than its reserved value, despite there was enough workload to saturate all its requested CPUs. Such counter-intuitive observation motivated us to continue investigating how the containers are scheduled on the CPU cores. Through measurement and analysis, I discovered that the CPU *request*, as I know it in the current container orchestration systems, is, in fact, a false promise due to neighbor interference. The impression that a container can always use all the CPU resources it requested is invalid. It cannot be faithfully enforced in today's multi-tenant container clouds, even without over-commitment.

The reason behind this is the complex interactions between various scheduling mechanisms and considerations in the current OS scheduler design (e.g., CFS in Linux). To make it worse, the increasing number of CPU cores and the co-located containers in modern host systems exacerbate the problems. I systematically analyze the performance degradation in a multi-tenant host and summarize my key findings as follows:

- *User-requested CPU resources at the container level (via cgroups) are not honored by the host OS.*
- *The root cause lies in the mismatch of the container' design goals and Linux's default one-size-fits-all CPU scheduler CFS.*

Motivated by these findings, in this chapter, I propose to augment the scheduling mechanism in the container orchestration system and its underlying OS. It bridges the gap between one's expectation when they request the resource and the actual utilization and performance they can get out of it. I implement my approach called rKubernetes, or *rKube* for short, on top of Kubernetes. It correctly enforces the requested amount of CPUs on host systems by isolating the container workload to designated physical cores. I evaluate the proposed approach by first comparing the CPU utilization and performance of the

applications with and without *rKube*. The results show that the application performance under *rKube* is consistent regardless of neighboring containers' different nature. Compared to vanilla Kubernetes, the application performance could be improved by a speedup ranging from 1.2x to 5.6x. I also conduct several case studies to demonstrate the practical value of *rKube*: 1) developers trying to deploy their microservices with a) vertical and b) horizontal scaling, and 2) cloud providers trying to guarantee better QoS of their container services with redundant resources.

While the details are provided in this chapter, some highlights of my contributions include:

- I quantitatively verify and confirm the existence of significant performance variations (e.g., 5x slowdown) in a container-based multi-tenant environment and show the current system cannot always deliver on resource promises.
- I investigate the root cause of the performance variations in containers and reveal that the underlying scheduling mechanism often does not comply with the user's resource specifications.
- I augment the existing scheduling mechanism in Kubernetes to bridge the gap between the resource promise and the user experience. Evaluation results show that compared to the Kubernetes, *rKube* can deliver a 2.1x-5.6x speedup for batch applications, and 1.2x-1.7x throughput increase and 12.9x and 13.7x tail latency reduction for interactive applications. Overall, *rKube* can effectively level off the container performance variations and make it more predictable.

The rest of this chapter is organized as follows. Section 4.2 presents the background information on the container orchestration systems and their scheduling approaches within the cluster. Section 4.3 shows some representative experimental results demonstrating the significant performance variations/degradation observed, which motivates me to investigate the underlying reasons in Section 4.4. To deal with the problem, I augment the scheduling mechanism and present it in Section 4.5, followed by the evaluation in Section 4.6. I discuss

related work in Section 4.7 and conclude this chapter in Section 4.8.

4.2 Background

4.2.1 Container orchestration systems

Modern container orchestration systems, such as Google's Borg [101], Kubernetes [102], and Docker Swarm [103], automate the deployment, management, and scaling of containers. When deploying a containerized application, users typically submit a specification file to the orchestration system. It will take care of the administration tasks, including application placement (on which hosts), resource allocation, and resource scaling.

Take Kubernetes as an example. The basic deployment unit of a Kubernetes application is called a pod [104]. A pod can host one or multiple containers. A user can configure the resource requirements, such as CPU and memory, for a pod's containers by using Kubernetes' specification file. When the job is submitted to Kubernetes, it will launch the application's pod(s) on one (or multiple) hosts it manages.

4.2.2 CPU specification in Kubernetes

In Kubernetes, the CPU resource of each container inside of a pod is configurable. The CPU resource is measured in *CPU units*, equivalent to one vCPU in a VM or one hyperthread in a bare-metal host. Furthermore, the CPU resource can be represented as CPU request (`spec.containers[].resources.requests.cpu`) and CPU limit (`spec.containers[].resources.limitss.cpu`). Suppose the host has some idle CPU resources available. In that case, the container is allowed to consume more CPUs than its specified CPU request, but not more than its CPU limit if specified. Kubernetes uses CPU request to decide which node to place the container and reserves the requested amount of CPUs for that container to use [105].

4.2.3 Container runtime

When a pod's container starts on a host, the CPU request and CPU limit are passed to the container runtime (e.g., Docker) and will be enforced by the host OS. The container runtime is responsible for local container management, including setting up the control groups (cgroups), namespace isolation, starting the containerized processes. In a multi-tenant environment, where multiple containers are co-located in the same host, they share and compete for the same set of resources of the host.

Linux uses cgroups for container resource allocation and isolation. Cgroups limit the resource consumption of the containers and control how different containers share resources. Specifically, the CPU request and limit for a container are enforced by the cgroup values of *cpu.shares*, *cpu.cfs_period_us* and *cpu.cfs_quota_us*.

While Kubernetes makes sure the host has enough resources for all the containers deployed in that host, different containers are managed indifferently in the same host. As I will demonstrate in Section 4.3, containers often receive less-than-requested CPU even when there is no CPU over-commitment in the host.

4.3 Motivation: container performance variations in multi-tenant environments

This section demonstrates that significant performance variation exists in a multi-tenant containerized environment. In a nutshell, my motivational study shows that: 1) the significant performance degradation is due to neighbor activities, 2) hardware contention may not be the main contributing factor of the degradation, and 3) a surprisingly low CPU utilization clear symptom of the performance degradation. The motivational study will then be used as a baseline to investigate the root causes (Section 4.4).

Table 4.1: Summary of containerized target and neighbor applications. R denotes the value of CPU request (and CPU limit) for the target applications. Since I allocate 22 CPU cores to the container applications on the host, the CPU request of the neighboring application is $22 - R$ cores. "-" means the neighboring application's CPU limit is not set (i.e., burstable).

Target Application	Threads	CPU Request (R)	CPU Limit	Description
bodytrack	6	6	6	Tracks a human body through space
fluidanimate	8	8	8	Physical simulation of a fluid
ocean_cp	8	8	8	Computes the cholesky factorization of a sparse matrix
streamcluster	12	12	12	Oonline clustering of an input stream
volrend	4	4	4	Computes the cholesky factorization of a sparse matrix
Memcached	4	4	4	High-performance, distributed memory object caching system
Neighboring Application				
Capped & CPU-Intensive (C-C)	32	22-R	22-R	CPU-intensive stress-ng w/ CPU limit set to CPU request (capped)
Burstable & CPU-Intensive (B-C)	32	22-R	-	CPU-intensive stress-ng w/o CPU limit set (burstable)
Capped & Memory-Intensive (C-M)	32	22-R	22-R	Memory-intensive stress-ng w/ CPU limit set to CPU request (capped)
Burstable & Memory-Intensive (B-M)	32	22-R	-	Memory-intensive stress-ng w/o CPU limit set (burstable)

4.3.1 Experiment setup and methodology

Environment

I use Dell PowerEdge R420 as the hosting machines for all my experiments, equipped with 2 x Intel Xeon E5-2420 CPUs (6C12T) and 24GB RAM. The host machine is running Debian 9.12 with Linux kernel 4.9.0. Docker 18.09.7 is used as the container runtime at each host machine, and Kubernetes 1.17.3 is used as the container orchestration system. I reserve 2 CPUs to Kubernetes system and OS system services on each host and leave 22 allocatable (i.e., *available*) CPUs for hosting containers from users. For simplicity, I configure one container per pod. Hence, for the rest of this chapter, I will use container and pod interchangeably.

Once Kubernetes receives a container, it will find the best Kubernetes node, i.e., one of the hosts, to deploy and launch that container. A container is likely to be co-located with other containers that share the same host. When container co-location happens, I consider this host to provide a `multi-tenant` environment. Otherwise, I consider it to provide a

`single-tenant` environment. In a production environment, the hosts are typically multi-tenant.

Workload

Batch and interactive applications are two major types of data center applications running in a production containerized environment [106].

Batch application I use PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark suite [107] and SPLASH-2 (Stanford ParalleL Applications for SHared memory) [108] as batch application benchmarks. I use the application completion time and CPU utilization as the major evaluation metrics for batch applications.

Interactive application I use `Memcached` [109] as an example of interactive applications. `Memcached` is a high-performance, distributed, in-memory key-value (KV) cache system. I use YCSB (Yahoo Cloud Serving Benchmark) [110] to evaluate the `Memcached` deployment performance in terms of *throughput* and *latency*. The YCSB workload contains 1 million unique KV records. I choose an update-heavy workload that contains 50% reads and 50% writes (the read-heavy workloads show a similar trend and thus are omitted due to space limitations). I only report the tail latency for read operations as the latency difference between update and reading operations is negligible.

Neighboring application I use `stress-ng` [111] as the neighboring application that generates different types of workloads (Table 4.1), including CPU-intensive and memory-intensive workloads. Both workloads will be run with 32 worker threads.

Capped and burstable neighbors Kubernetes allows developers to set two parameters to specify the container's CPU consumption: CPU request (*requests.cpu*) and CPU limit (*limits.cpu*). The CPU request determines a lower bound for the expected CPU resource while the CPU limit determines an upper bound. By not setting a limit or setting a limit larger than the CPU request, the container can have bursts of activities when some CPU resource becomes available [112]. I call such a container a `burstable` container. Otherwise, when the CPU limit is the same as the CPU request, I call it a `capped` container.

I use the publicly available Docker images [113–115] to run the applications mentioned above. Table 4.1 summarizes the batch and interactive applications that I use.

Methodology

When Kubernetes deploys containers to hosts, multiple containers may share the same host, which I call a `multi-tenant` host. Otherwise, it is a `single-tenant` host.

In my experiments, I have the *target* container and the *neighbor* container. A target container is a container that I measure the performance metrics. For my multi-tenant scenario, I deploy both the target container and neighbor containers to the same host and measure the target container's performance with the neighbor containers' interference.

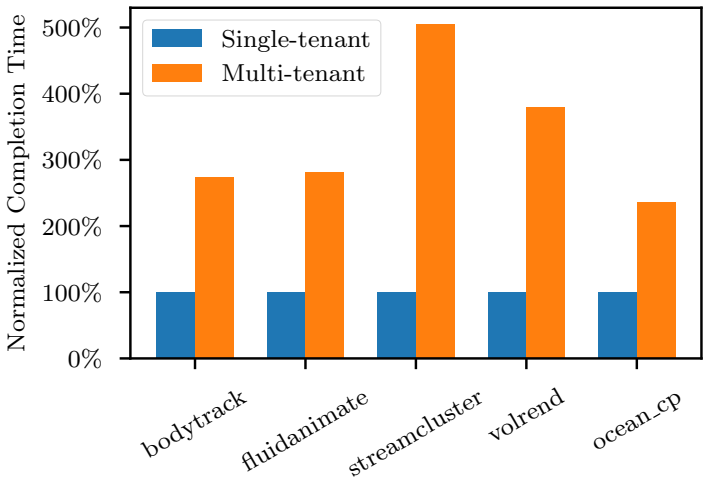
The target containers run either batch or interactive applications, with their CPU request and limit value specified in Table 4.1. In my setup, 22 CPUs are available on the host machine for kubernetes to allocate. Hence, for the neighbor containers, I assign their CPU request value to be the remaining CPUs available in the host, i.e., $22 - CPUs$ requested by the target container. Depending on whether the neighbor is burstable or capped, the CPU limit will be set to default (empty) or the same requested value.

In my experiments, I vary the number of threads for each application so that they can take a similar amount of time (around 60 seconds) to complete.

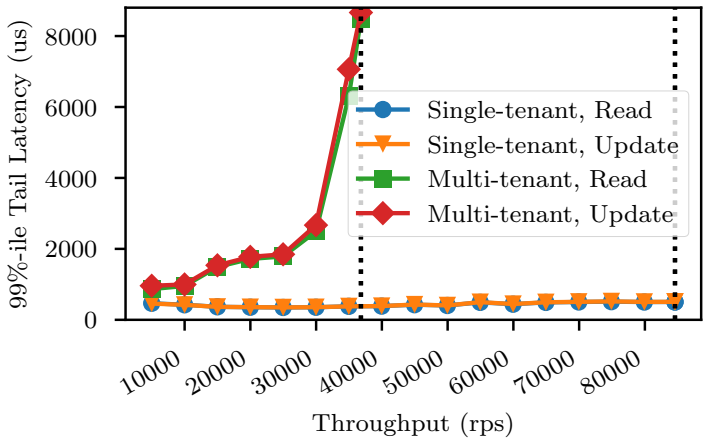
For batch applications, each application runs ten times, both in the single-tenant and multi-tenant hosts. The variance of completion time in repeated runs is minimal and, thus, is omitted. For `Memcached`, I gradually increase the throughput (requests per seconds or rps) by 1000 until `Memcached` is fully saturated; each `Memcached` test runs for 300 seconds.

4.3.2 Neighbors pose a significant impact

In this subsection, I aim to demonstrate that running in a multi-tenant environment can result in significant performance degradation compared to running in a single-tenant environment. This can be observed for both batch and interactive workloads, even with the same amount of CPU resources are requested.



(a) Batch applications.



(b) Memcached.

Figure 4.1: Comparison of the container performance under single-tenancy and multi-tenancy.

Figure 4.1a shows the normalized completion time (normalized against the completion time measured in a single-tenant environment) of the five batch applications. As shown, the performance degradation is severe when the applications are running in the multi-tenant host. For example, `bodytrack` and `streamcluster` take nearly 3x and 5x amount of time to complete in a multi-tenant host.

Similar performance degradation is observed for the interactive application `Memcached`.

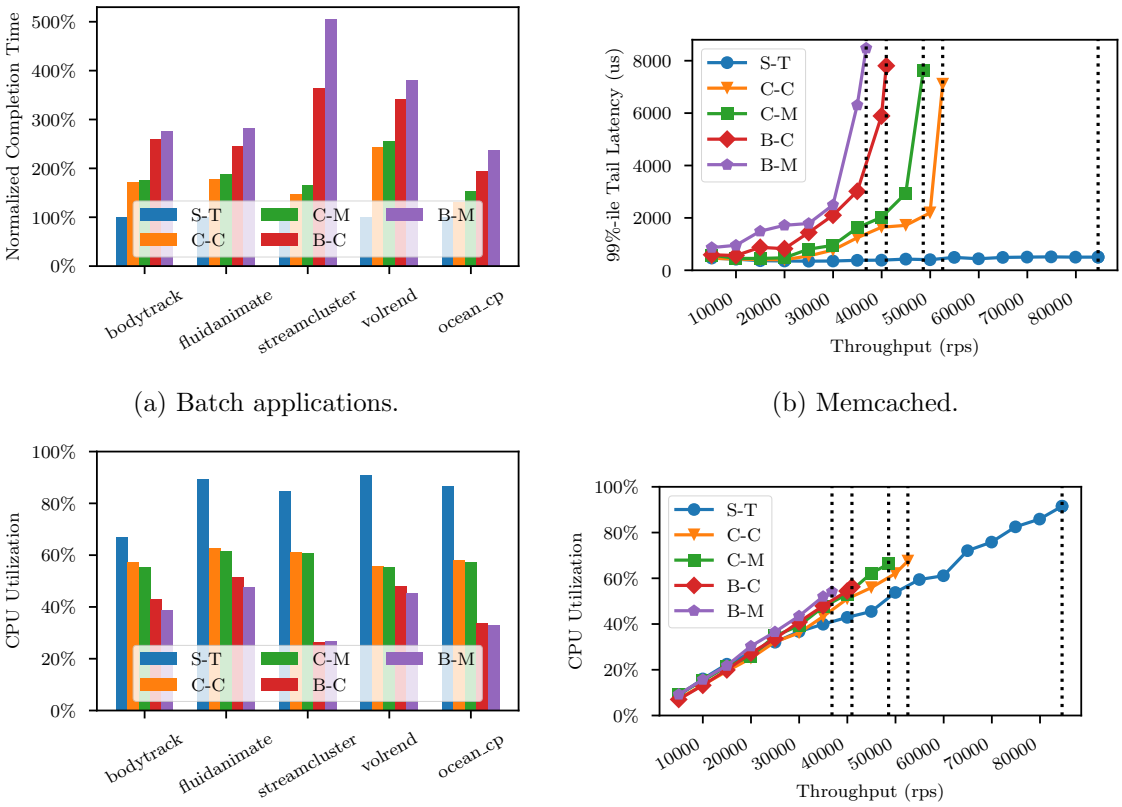
As shown in Figure 4.1b, the 99^{th}-percentile (99%-ile in short) latency for the update operations is much higher when `Memcached` is running in a multi-tenant host compared to in a single-tenant host. The same trend holds for the throughput metric: 1) the maximum throughput drops from 85k (single-tenancy) to 37k rps (multi-tenancy), as depicted using the vertical dotted lines, and 2) under the same throughput, e.g., 37k rps, the 99%-ile tail latency increases from 383 to 8479 μs, a slowdown of more than 22x!

The performance analysis shows that significant performance variations exist for containerized applications in a multi-tenant environment. The performance degradation can be as high as 22x, making it extremely difficult to predict the application performance and accurately estimate the applications' resource requirements.

4.3.3 Hardware contention is not the sole cause

The observation motivates me to further investigate the root cause of the performance degradation. One might wonder if hardware contention is the culprit. To verify that, I quantify the performance degradation of the target applications by varying the neighboring application behaviors. To this end, I experiment with four different configurations of the neighboring stress-ng, namely, capped & CPU-intensive (C-C), capped & memory-intensive (C-M), burstable & CPU-intensive (B-C), and burstable & memory-intensive (B-M). Here I intentionally create contention on two shared hardware resources – CPU and memory. A burstable (with no CPU limit set) stress-ng poses pressure on the CPU resources. In contrast, a memory-intensive stress-ng poses pressure on the shared memory subsystem of the host.

As shown in Figure 4.2a and 4.2b, B-M causes the most significant performance degradation for the target applications compared to the performance achieved under single-tenancy or S-T. It suggests that hardware contention might be the cause of the significantly dropped performance. However, I also observe notable performance degradation for C-C. In this case, the neighbor is least aggressive in competing for hardware since the neighboring application is CPU-intensive, and the neighboring container's CPU usage is capped. I infer that the



(a) Batch applications.

(b) Memcached.

(c) Batch applications.

(d) Memcached.

Figure 4.2: Performance and CPU utilization of the target container when running alone: S-T (Single-Tenant) and with different neighboring apps as described in Table 4.1

performance degradation is due to factors other than the hardware contention from the experiments.

4.3.4 CPU capping and low CPU utilization

Next, I examine the impact of CPU capping on the performance of the target applications. Recall that a burstable neighbor can use more CPU resources than specified in the CPU request but less than specified in the CPU limit, while a capped neighbor has its CPU limit capped as its CPU request. One may expect that a capped container would not be able to "steal" any CPU resources from the target application. However, as shown in Figure 4.2a

and 4.2b, this is not the case – by switching from a burstable neighbor to a capped one, surprisingly, the performance degradation is not eliminated, but just mildly mitigated. The evidence is, C-C, though imposing a minimum level of hardware contention as described earlier, still observes up to 243% performance reduction.

To further investigate such a counter-intuitive phenomenon, I measure the corresponding CPU utilization of my target applications. The CPU utilization is calculated by the percentage of CPU time the application consumes over the experiment period: $CPU\_utilization = CPU\_consumption/requests.cpu$. As shown in the Figure 4.2c and 4.2d, for both batch and interactive applications, the CPU utilization decreases as the neighbor's contention level and burstiness increase. This is consistent with the performance degradation of the target application shown in Figure 4.2a and 4.2b, indicating the resulted performance degradation is highly correlated with the insufficient CPU consumed by the target application.

What remains unclear is why the container is getting insufficient CPU, especially for the C-C case, given that: 1. no over-commitment on the host system - each container can theoretically get its requested CPU, 2. the container CPU usage is capped - no containers can use more than it requested, 3. minimal hardware contention, and 4. there is clearly enough workload left to be done as indicated by the S-T case. With all these conditions, I conclude that the orchestration system, together with the underlying host OS, fails to satisfy the container's CPU request, which should have been able to use all its requested CPUs.

This puzzling fact motivates us to continue investigating how the requested CPU resources of a container are managed jointly by the orchestration system and the underlying OS.

4.4 The journey of a CPU request and its fulfillment

In this section, I analyze how a CPU request is fulfilled in a multi-tenant containerized environment. I describe the journey of a CPU request, which is issued from a user, to the

orchestration system, the host OS, and its scheduler, and finally to the physical CPU cores. I then discuss Linux's CPU scheduling mechanism and possible scheduling examples where the CPU request is not correctly fulfilled.

4.4.1 From orchestration system to host kernel

The journey starts when a user deploys containers to a data center managed by a container orchestration system (Kubernetes, in my case). The user specifies the value of $c_i.requests.cpu$ for the container c_i. When the system deploys the container, at least a requested amount (c_i) of CPU should be reserved for it [105]. Upon receiving the CPU request from the user, Kubernetes performs the following two tasks in order to fulfill the requested CPU resources:

- Kubernetes chooses one (assuming the application is deployed on a single host) host node n_k in the data center where there are enough CPU resources available. In other words, the following constraint needs to hold when the new container is added to the host node:

$$\sum_{c_i \in C_k} c_i.requests.cpu \leq n_k.allocatable, \tag{4.1}$$

  where C_k is the set of containers in the hosting node n_k and $n_k.allocatable$ is the total amount of CPUs that is allocatable to users' containers in that host.

- Kubernetes translates the CPU request $c_i.requests.cpu$ into a value that can be understood and enforced by the host OS. Control groups (cgroups) [116] in Linux is a widely used OS kernel feature that implements a host's containerized environment. Cgroups uses an integer value $g_i.cpu.shares$ to indicate the amount of CPU resources that a group g_i can consume. Kubernetes then maps each container c_i to its associated control group g_i in the host and the conversion between $c_i.requests.cpu$ and $cpu.shares$ is done as follows:

$$g_i.cpu.shares = c_i.requests.cpu \times 1024.$$

Note that the value of *cpu.shares* used by cgroups is a relative weight for allocating CPUs [117], as opposed to an absolute amount of CPU cores that a group can use. Such discrepancy may cause the host node to break its resource promises if it becomes over-committed. Therefore, to keep its promise to the user application, Kubernetes imposes Eq. 4.1 to each host so that intuitively such a case would never happen [105, 118]. Despite this effort, a co-located container still cannot fully utilize all its requested (allocated) CPU resources as shown in Section 4.3.4. To this end, I continue my analysis of how the value of *cpu.shares* affects the scheduling decisions of the host OS.

4.4.2 From OS scheduler to physical CPUs

The OS scheduler determines how the CPU resources are allocated to the processes. Linux uses Completely Fair Scheduler (CFS) by default, which implements weighted fair queueing (WFQ). CFS optimizes the overall CPU utilization while keeping the process fairness in check, whereas cgroups is used to provide resource isolation for containerized applications at the OS level. *The misaligned design goals of Linux CFS and cgroups, as will be described in Section 4.4.3, cause the failure of CPU request fulfillment.* Next I introduce several key aspects that affect the scheduling decisions for the containerized applications.

CFS on a single-core system is reasonably straightforward. However, when it comes to a multi-core system, the scheduling decision making becomes a much more complicated optimization process. In a multi-core system, each physical CPU core has its individual *runqueue*. The processes will first be assigned to a runqueue and then be selected to run on that CPU. Ideally, when a user requests X CPUs for the container, all the processes spawned inside that container should be scheduled exclusively onto the runqueues of those X CPUs. However, due to CFS's load balancing activities, it may not be the case.

Load balancing between runqueues is key to the overall system performance and CPU utilization. CFS periodically balances the processes of each runqueue based on multiple factors and metrics. One of these metrics is the *load* of a runqueue, which is derived from the weight (share) and nature of its tasks (i.e., processes and threads) [119]. The

nature of a task can be derived from its historical CPU utilization, using a load tracking scheme called per-entity load tracking (PELT) [120]. For example, a lower-weight CPU bound task may contribute more load to the runqueue than a higher-weight, I/O-bound task, due to different historical CPU utilization. Hence, the processes in a container with a large amount of requested CPUs (and therefore a high CPU share) are not guaranteed to occupy a runqueue exclusively. To make things even worse, there are multiple other factors including cache locality, the Non-Uniform Memory Access (NUMA) topology, CPU affinity, CPU bandwidth control, and CPU capacity in heterogeneous platform, which are not related to CPU shares but affects how processes are allocated to the runqueues.

Once the processes are assigned to their runqueues, for each CPU at every scheduling tick, the scheduler will pick the next process on that runqueue to run on the associated CPU core. As the workload is increasingly consolidated, it is common to have thousands of concurrent processes on a single host. The chance that the processes of one container share the same runqueue with other containers is very high. During a "scheduling period" all processes in the runqueue would have a chance to run; the amount of CPU time is proportionally divided into "time slices" for all the processes in the runqueue based on their weights. For each process, the time slice is a time interval it is expected to run on a CPU core, or its CPU time share during this scheduling period. So a task weight only indicates a *relative* CPU share, not an absolute CPU share.

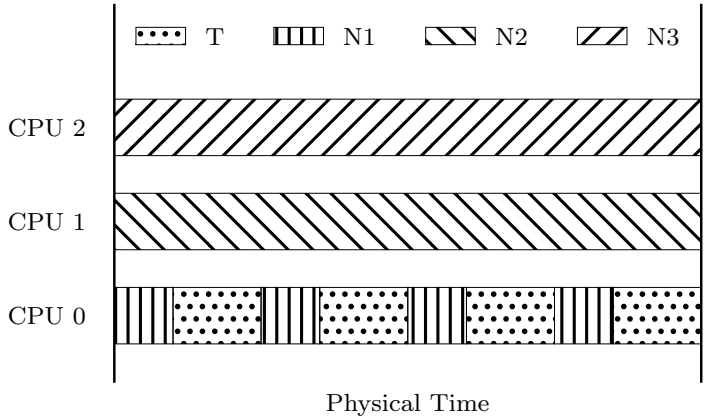
There is also a design consideration to keep a "minimum granularity" for each time slice, which determines the minimum amount of time that a process needs to run before it can be preempted. It is set to prevent the slices from becoming too short, calling the scheduler too frequently, and increasing the overhead. The minimum granularity is determined by several scheduler parameters, including *sched_min_granularity_ns*, *sched_latency_ns* and *sched_wakeup_granularity_ns*, which is to balance the interactivity and the overhead. However, as I will show in the following section, the interaction between them and the fact that processes from multiple containers shares the same runqueue can lead to insufficient CPU consumption. Thus, performance degradation was observed previously.

4.4.3 CPU scheduling under the microscope

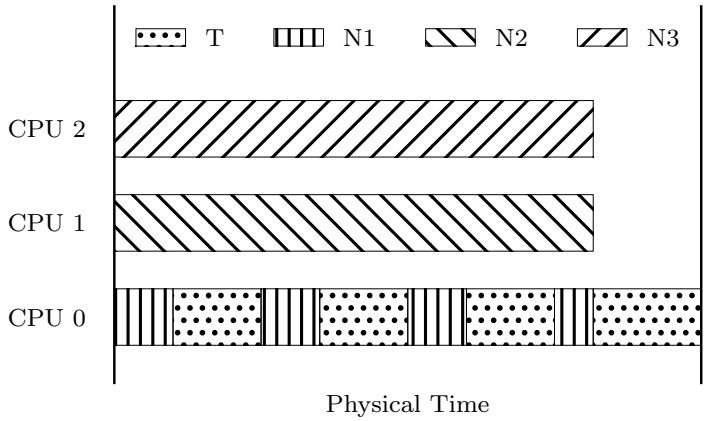
I examine the activities of each CPU during one scheduling period in two scenarios, which leads us to the root cause of the low CPU utilization problem. Here two containers T and N are running on a host with three CPUs. Container T has only one thread, while container N has three threads: $N1 - N3$. The requested CPUs for T and N are 1.0 and 2.0, respectively. Because of the nature of the applications, load balancing and other reasons mentioned in the previous section, T and $N1$ are assigned to the same runqueue of CPU0. $N2$ and $N3$ are assigned to CPU1 and CPU2, respectively.

Let us first look at the case when T is a batch application. When the neighbors are set to be burstable, as shown in Figure 4.3a, CPU1 and CPU2 are fully consumed by the neighboring threads $N2$ and $N3$. The container T only gets 0.6 of CPU0, as its CPU weight from its cgroup is 1024, and the CPU weight of $N1$ is 683. Even if I cap the neighbors, as shown in Figure 4.3b, the situation does not get better for T. When $N1 - N3$ consume all its CPU quota (i.e. 2.0), all these three threads will be suspended during the current scheduling period, and CPU0-CPU2 will become available. If T were to utilize all the remaining time of CPU1 and CPU2, it would be able to consume the time of one full CPU it has requested. However, since T only has one thread, it can only run on one CPU at one time and, therefore, cannot take advantage of those *phantom* available CPUs. Hence, no matter whether the neighbors are capped or burstable, the target container will always receive significantly lower actual CPU time than its request amount, despite no over-commitment in the system.

Now let us examine the case when T is an interactive application. Unlike the batch application scenario, an interactive application may wake up and only do a relatively small amount of work before going back to sleep. For example, Memcached receives a request, processes it, replies to the client, and then goes back to sleep. However, it needs to react quickly, i.e., wake up as soon as a user request arrives. When the neighbors are burstable, as shown in Figure 4.4a, T cannot wake up in time because of the enforcement of the minimum granularity. It limits frequent preemption and context switch, results in an increased request



(a) The neighbors are burstable.

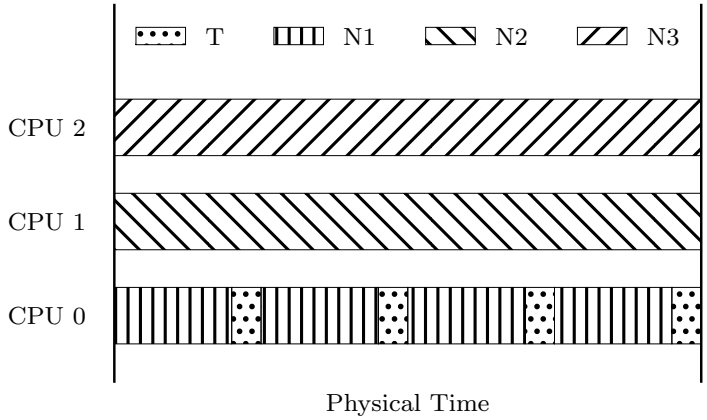


(b) The neighbors are capped.

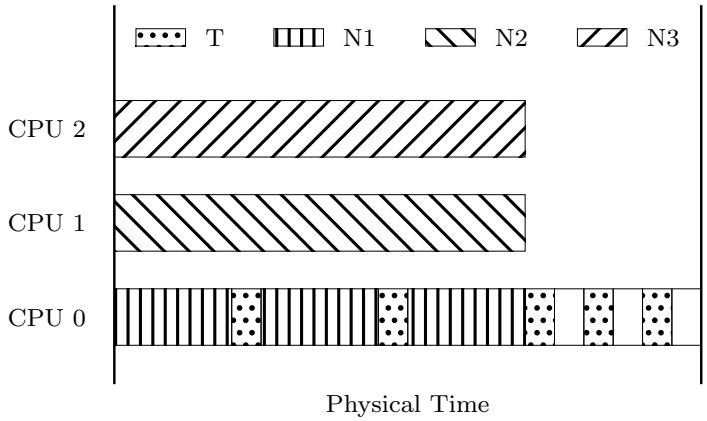
Figure 4.3: Illustration of a scheduling period where the target container T is running batch applications.

processing latency. Similar to the batch application scenario, capping the neighbors does not help. As shown in Figure 4.4b, only when the neighboring application's threads exhaust all their CPU quota T can finally wake up more frequently than before. Nevertheless, T still cannot take advantage of the same phantom CPU time available on CPU1 and CPU2, simply because T has only a single thread.

Such performance impact directly results of the interactions between different design considerations in the scheduler and the cgroup components. It exists not only in containers



(a) The neighbors are burstable.



(b) The neighbors are capped.

Figure 4.4: Illustration of a scheduling period where the target container T is running interactive applications.

but also in virtually all systems. However, it becomes a huge performance concern in the containerized environments because modern host systems are often equipped with many CPU cores and are hosting hundreds of containers [121]. This results in much more phantom CPU time that should have been fully utilized (as requested by the applications), leading to significant performance degradation.

4.4.4 Key finding

To summarize, the performance degradation of the target containerized application is caused by multiple correlated factors: (1) Linux's CFS scheduler tries to balance the CPU load of an aggressive neighboring application on as many cores as possible, which results in multiple applications sharing the same runqueue(s); (2) to honor the CPU shares as requested (via cgroups), CFS time shares the CPU cores by scheduling tasks of different applications from the shared runqueue, thus leading to performance degradation of the target application.

My key finding of this chapter is that the user' request, requests.cpu set via Kubernetes, or cpu.shares and task weights calculated through Linux cgroups, is not sufficient to provide stringent CPU guarantee with promised CPU share to the co-located containers in the same host. This is due to the mismatch of the design goals of containers and the underlying CFS scheduler.

Such a mismatch strongly demands a new mechanism to fulfill the promise. This leads to the design of a new resource configuration scheme, called *rKube*, presented in the next section.

4.5 Design and implementation of *rKube*

Based on my findings in the previous section, I propose to design and implement *rKube*, to enable the containers to deliver the promised resource and performance.

4.5.1 Rationale of *rKube*

Based on my analysis, the current Kubernetes and kernel scheduler design cannot keep its resource availability promise to the containers. There are several directions to solve this problem. Here I discuss several potential design choices regarding their effectiveness in addressing the problem and complexity and practicality for implementation.

Task Priority. The CFS scheduler implements priorities by assigning weights to tasks based on their static priorities. Any task in the higher-priority queue will be scheduled

before the ones in the lower-priority queue. Therefore, the availability of CPU to a task is dependent on its priorities. Intuitively, increasing the task weight or priority will allow the target application to obtain more timeslices of CPU in every scheduling period. This, however, cannot prevent other tasks from being assigned to the same runqueue, which inevitably consumes a portion of CPU. Furthermore, this does not help I/O bound containers to get CPU more frequently.

Preemption Frequency. Intuitively, better task interleaving could be achieved by increasing the preemption frequency of all tasks. However, this would benefit specific tasks at the expense of others and may result in an unacceptable system-wide overhead of context switches, as preemption frequency is a major consideration for the scheduling overhead. For example, an I/O intensive task could benefit more than CPU intensive workloads as it may have more opportunities to be scheduled.

Based on the above discussions, changing the scheduler's configuration parameters could not effectively mitigate the issue. Even worse is that conflicts may be introduced when fiddling with the existing parameters. I argue that scalability considerations have made more impact than the actual user demand in designing the existing schedulers, thus leading to the issue I described in Section 4.4.4. Therefore, I propose *rKube*, which is my effort to augment the Kubernetes by introducing the currently missing parameters that allow users to express their demand to the kubelet. It relies on CPU reservation (instead of dynamic weight adjustment) to implement resource allocation and scheduling.

Specially, CPU reservations could be achieved by setting CPU affinity for individual containers. For this purpose, Linux provides a control feature, called cgroups, that could be utilized by *rKube* via setting *cpuset.cpus*. While *cpu.shares* provides relative CPU shares among groups, *cpuset.cpus* limit CPU usages in absolute values that are independent of the CPU speed and the scheduling of neighboring containers. In my design, *rKube* first selects a set of CPUs to be reserved for the target container based on their *requests.cpu*; then, it sets *cpuset.cpus* for all the containers to guarantee that the runqueues of the selected CPUs are exclusive for the target container. In this way, no other neighboring containers

will be scheduled on these CPUs. Eventually, the system and the other containers will not be affected by the scheduling overhead of the reserved CPUs, the cost of which will be paid by the users who are in demand.

4.5.2 Implementation of *rKube*

I build my *rKube* prototype based on Kubernetes v1.17.3. To enable a user to require specific scheduling for a container explicitly, I add a new field named "policy" to the Kubernetes pod template. It refers to if the CPU request is `strict` or `standard`. Based on this field's value, a container can have its *requests.cpu* fulfilled by *cpuset.cpus* instead of the original *cpu.shares* if the request is `strict`, i.e., using *rKube*. Otherwise, it falls back to the original implementation. Thus, this option provides backward compatibility.

To implement *rKube*, I modify Kubernetes components `kubectl`, `apiserver`, `kube-scheduler` and `kubelet` to make sure the new field is correctly passed to `kubelet` in the host. In the host, once a container has strict CPU request, the configured number of CPUs is selected. As I observed in Figure 4.2, contention on the hardware resources also impacts the target application. To reduce the interference with its neighboring applications in the shared memory hierarchy, the CPU topology is taken into consideration in the CPU selection. *rKube* aims to separate them from shared CPUs used by other containers. For example, when *requests.cpu* is larger than the CPU number in a socket and all CPUs in that socket are not reserved yet, then all these CPUs will be selected. They will be added to *cpuset.cpus* of the target container, and will be removed from *cpuset.cpus* of all other containers in the host.

There are other design choices in building *rKube*. Originally, in the host, `kube-reserved` and `system-reserved` can be used to request CPU for Kubernetes and OS system daemons, such as the kubelet, container runtime, and sshd [118]. Similar to user containers, these system services may also be starved by other user containers. To prevent it from becoming the performance bottleneck, *rKube* also applies the same CPU reservation for the CPU request of the system services.

Furthermore, since containers may fail unexpectedly, like other Kubernetes components, my subsystem that manages cgroup should also be resilient to container crashes. For this purpose, *rKube* does not rely on Docker to update `cgroup` values, but instead, it maintains host status and writes to cgroup filesystem directly. The subsystem also checks for any inconsistency on all pod life cycle events. Hooking pod life cycle events rather than container life cycle events makes the CPU reservation simple, stable, and complete.

Note that my implementation of *rKube* did not change the underlying host's default scheduling, i.e., CFS. I chose not to directly modify CFS to make it fit for containerized environments because 1) I strive to make the implementation transparent to the underlying OS, and 2) CFS is widely used for other purposes, and modification of that would impede the adoption of *rKube*. For similar considerations, *rKube* keeps the default standard approach for requesting resources, but with the new option added, so that it is backward compatible.

4.6 Evaluation

My evaluation is conducted using the same setup as described in Section 4.3.1. The performance of *rKube* is evaluated from two perspectives: 1) its effectiveness in reducing the neighbor interference and keeping the CPU resource promise, and 2) how does a good resource promise translate into the direct benefits for developers and cloud service providers, i.e., its practical value. For effectiveness, *rKube* is compared with the default scheduling strategy, referred to as `standard` in the corresponding figures. For the practical value, I study three common use cases in cloud performance tuning and evaluate the benefits of using *rKube* against the typical best practices in production [122], including "resource scaling" (Section 4.6.2 and 4.6.3) and "resource under-commitment" (Section 4.6.4).

4.6.1 Effectiveness of *rKube*

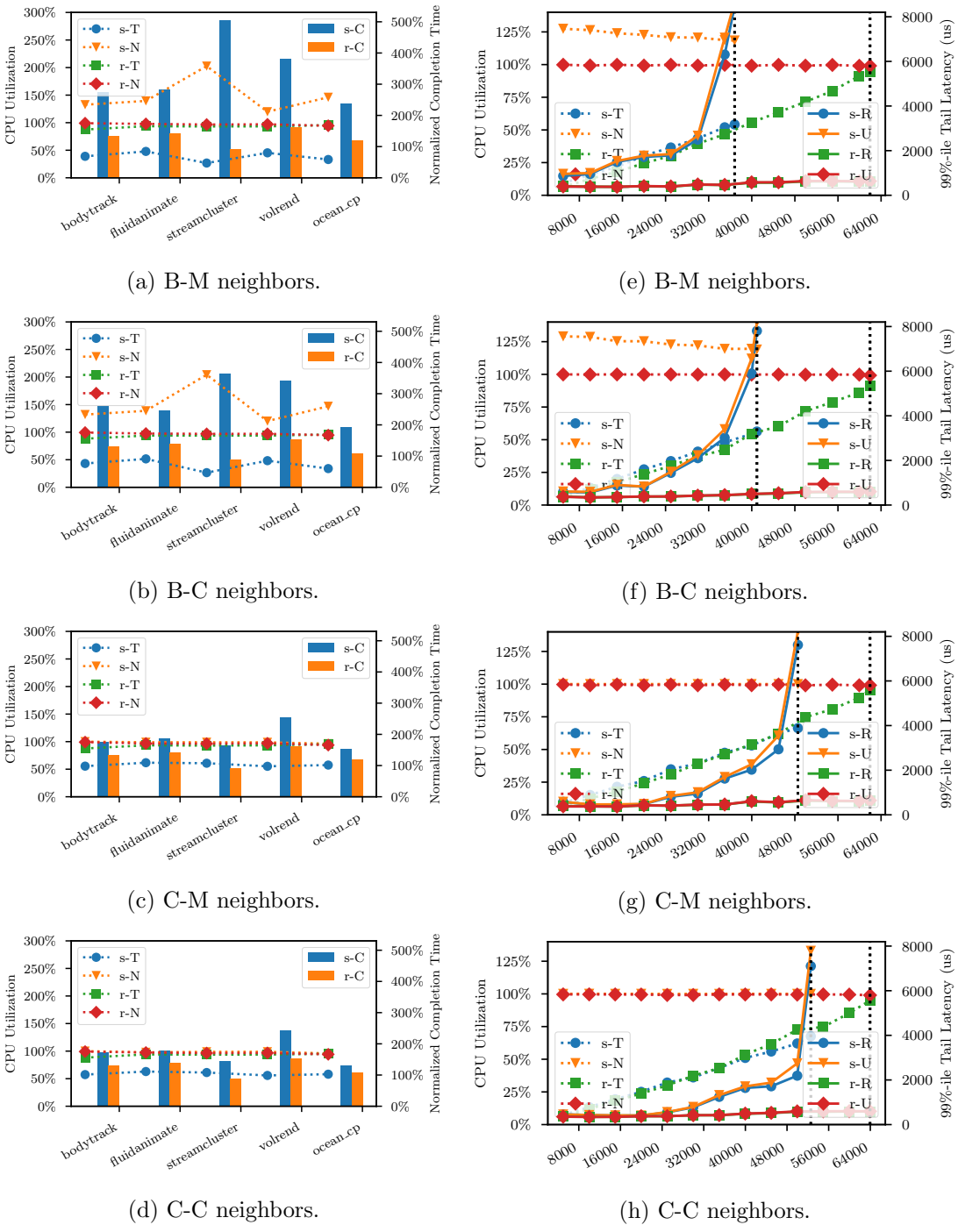
First, I study whether the containerized applications' performance can be improved with *rKube* and how much it can help if any. For this purpose, I compare the CPU consumption

and performance of different containerized applications in the multi-tenant environment when different neighboring containers are used, under the standard Kubernetes or *rKube*.

Figures 4.5a to 4.5d show the results of batch applications, while Figures 4.5e to 4.5h show that of the interactive application, `Memcached`. Each sub-figure has two y-axes. The left y-axis shows the CPU utilization. The right y-axis shows normalized completion time (by its corresponding completion time in the single-tenant environment) for batch applications and the 99%-ile tail latency (ms) for `Memcached`. In all the figures, the dotted lines represent the CPU utilization. The results show that *rKube* effectively increases the CPU utilization and performance of the target applications, regardless of what type of settings a neighboring container uses. For example, for `streamcluster` when its neighbor runs a memory-intensive workload and does not have *limits.cpu* set, as shown in Figure 4.5a, *rKube* increases its CPU utilization from 27% to 93%; accordingly, its completion time is reduced from 482.0 seconds to 86.4 seconds, a speedup by more than 5x. For the batch applications, the speedup of task completion time ranges from 2.1x to 5.6x depending on the different neighboring workloads. For `Memcached`, as shown in Figure 4.5e, *rKube* increases its CPU utilization from 55% to 95%, while its maximum serving throughput (denoted by the vertical dotted line in the Figures 4.5e to 4.5h) increases from 37k to 62k rps. The corresponding 99%-ile tail latency for read operations is reduced by over 13x, from 8479 to 621 microseconds. We observe that the improvement of `Memcached` happens across all the situations, with a throughput boost ranging from 1.2x to 1.7x and a reduction of tail latency between 12.9x and 13.7x.

4.6.2 Vertical scaling vs. *rKube*

My evaluation so far demonstrates that, with correctly enforced CPU request, *rKube* can effectively improve the application's CPU utilization and performance in the multi-tenant containerized environment. In current practices, when a user deploys an application to the cloud, and the application's performance is not satisfying, the user often has to invest more resources to improve its performance. As most cloud providers charge users based on the



(a) B-M neighbors.

(e) B-M neighbors.

(b) B-C neighbors.

(f) B-C neighbors.

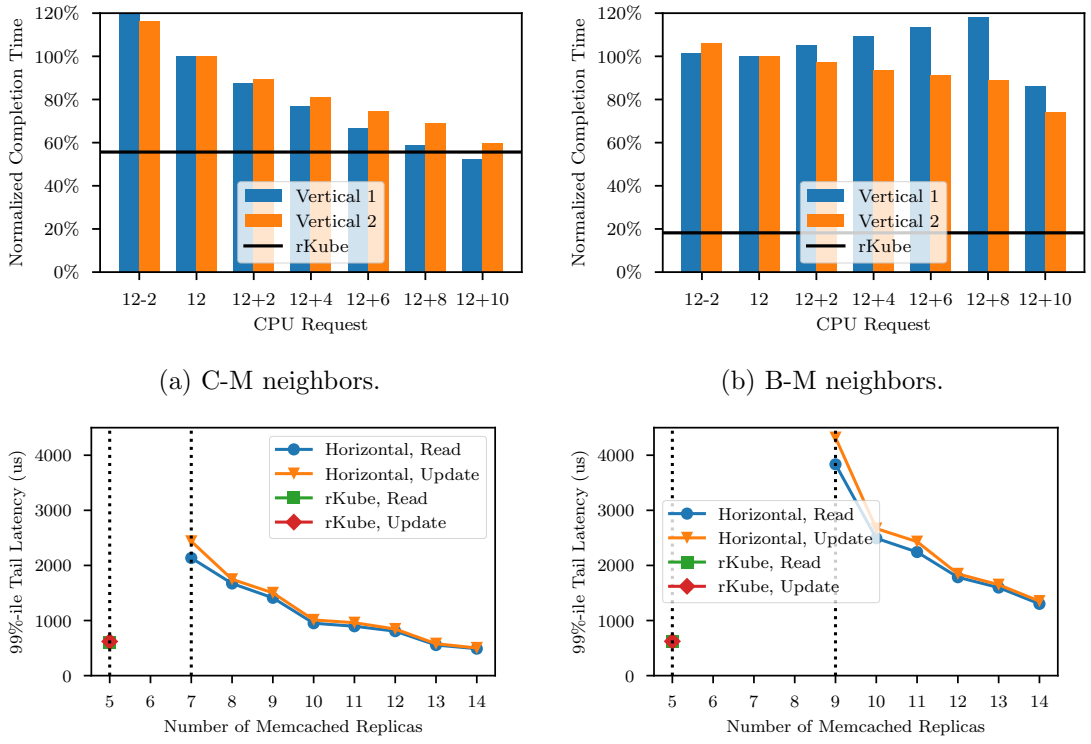
(c) C-M neighbors.

(g) C-M neighbors.

(d) C-C neighbors.

(h) C-C neighbors.

Figure 4.5: Performance of batch applications and Memcached with different neighboring apps defined in Table 4.1. The *s-* and *r-* prefix refers to standard and *rKube*. *-T* and *-N* suffix refers to CPU utilization of the target and neighbor. *-R*, *-U*, *-C* suffix refers to Read and Update latency, and Completion time, respectively.



(a) C-M neighbors.

(b) B-M neighbors.

(c) C-M neighbors.

(d) B-M neighbors.

Figure 4.6: Comparison of performance improvement when using vertical scaling, horizontal scaling, and *rKube*.

resources requested, an accurate estimate of resource demand for the application is not only critical for predicting its performance (e.g., when the task can complete), but can also help reduce the user's cost. I next study how *rKube* can effectively avoid unnecessary resource wastage with improved cost-effectiveness.

Upon performance slowdown, a common practice is to vertically scale (or scale up) the application by adding more CPU resources from the same host. Figure 4.6a and Figure 4.6b show the effect of vertically scaling `streamcluster` vs. when *rKube* is applied alone, respectively. Two strategies are available when performing vertical scaling: 1. the number of threads equals to the CPU request, thus increasing along with the CPU request; 2. the number of threads remains fixed while the CPU request increase. I denote these two strategies as `Vertical 1` and `Vertical 2` in the figure.

By default, the number of threads and the CPU request are both set to 12 for `streamcluster`. I then vary the CPU request between 10 and 22, as shown by the figure's x-axis. In this case, strategy "Vertical 2" keeps the number of threads 12. Figure 4.6a and Figure 4.6b show the normalized completion time, which is calculated as the ratio of the completion time of various scaling strategies to the baseline. Thus, the smaller the ratio is, the less the completion time it takes.

The results show that by requesting more CPUs for `streamcluster`, the application performance improves slowly with both scaling strategies. As a comparison, *rKube* requires no additional CPU resources but reduces the application's completion time by 40%, as shown in Figure 4.6a (when the neighbor application is capped and memory-intensive), and by 80%, as shown in Figure 4.6b (when the neighbor application is burstable and memory-intensive). Furthermore, vertical scaling causes further performance drop with a burstable neighboring application since the CPU contention increases as more threads are spawned by vertical scaling. This trend can be observed from the results of "Vertical 1" in Figure 4.6b.

The results indicate that *rKube* is more effective than vertical scaling strategies when guaranteeing a predictable application performance. Solely increasing resources does not prevent the target applications and neighbor threads from sharing the same CPU runqueue, thus "stealing" CPU resources from the target.

4.6.3 Horizontal scaling vs. *rKube*

For interactive applications, "horizontal scaling" or "scaling out" is often used when the applications hit a performance bottleneck. Unlike vertical scaling, horizontal scaling typically requires provisioning and committing additional infrastructure capacity to achieve the desired performance. For example, to enable the `Memcached` cluster to serve at higher throughput, the admin can increase the number of `Memcached` replicas[1] and distribute the load across all the replicas.

Figure 4.6c and Figure 4.6d show the effect of horizontally scaling `Memcached` vs. when

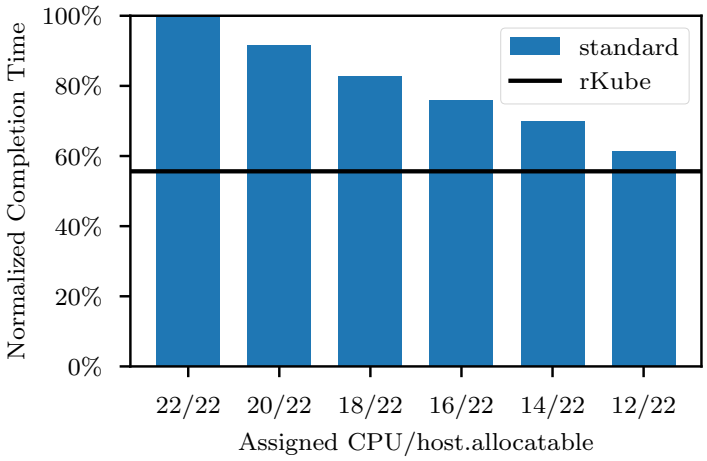
[1] The Kubernetes community uses the term *replicas* to describe the same pod instances associated with a single application.

rKube is applied alone. For both tests, the deployed `Memcached` observes a target throughput of at least 300k rps. The vertical dotted lines denote the minimum number of replicas needed to achieve the target throughput. The minimum number of replicas is 7 and 9 in Figure 4.6c and Figure 4.6d, respectively, due to the different settings of neighboring containers. We observe that by increasing the number of replicas, the tail latency decreases gradually. On the other hand, with the help of *rKube*, the minimum number of replicas is 5 for both types of neighbors, with a tail latency of 606 and 613 ms. Finally, by comparing the results in Figure 4.6c and Figure 4.6d, I find that *rKube* is incredibly helpful when having burstable neighbors than when having capped neighbors. Comparatively, capped neighbors will be throttled after their CPU consumption meets their demand, while burstable neighbors can be more aggressive in competing resources.

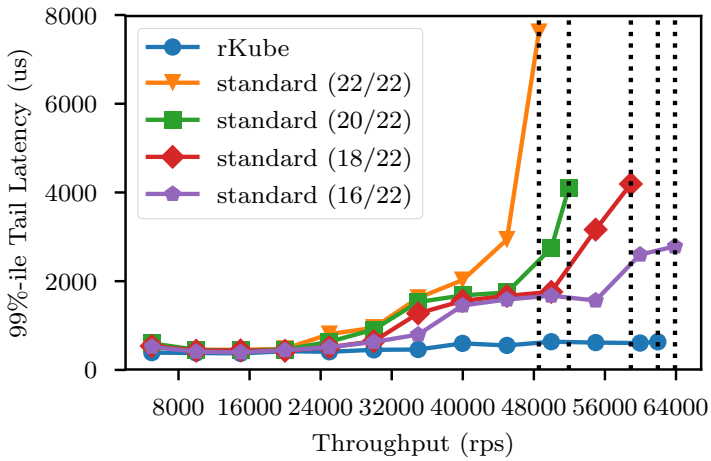
Overall, the results show that while scaling out can improve Memcached's throughput, the latency is also increased due to neighbor containers. *rKube* can mitigate such effects, delivering better service quality to users.

4.6.4 Resource under-commitment vs. *rKube*

By default, Kubernetes assigns all the allocatable CPUs of the host to containers. When a host node in a Kubernetes cluster runs out of resources, the kubelet (the primary "node agent" that runs on each host node) will be triggered to reclaim resources by evicting pods until the resource usage is under a pre-defined threshold again. Therefore, an admin or the cloud provider may intentionally leave some resources unassigned (i.e., *under-commitment*) to guarantee that there is enough resource headroom when the load of the (higher-priority) applications spikes. I call such a strategy "resource under-commitment". I study the effectiveness of resource under-commitment and compare it with that of *rKube*. I gradually reduce the number of CPU request of the neighbor containers (so that more allocatable CPUs become available and can be utilized if needed) to simulate the over-supplied resource scenario. Figure 4.7 shows the performance improvement of `streamcluster`, when I reduce the number of CPUs assigned to the neighboring containers. The neighboring container is



(a) `streamcluster`, completion time.



(b) `Memcached`, read latency.

Figure 4.7: Comparison of the performance improvement for applications by utilizing undercommitment and by *rKube*.

capped and runs memory-intensive workloads in this experiment.

In Figure 4.7a, the x-axis represents the number of assigned CPUs (to both the target and neighboring applications) vs. total allocatable CPUs. Throughout, the target application `streamcluster` always requests 12 CPUs, as listed in Table 4.1. For example, 20/22 means 20 are assigned to the applications (where 12 is assigned to `streamcluster` and 8 is to the neighbor), and 2 are allocatable but not assigned. For *rKube*, the number of assigned

CPUs for the target application is fixed at 12. By default, all CPUs (22/22) are assigned to the target and neighboring containers. The performance of the target application in the default setting is used as my baseline for comparisons. As shown by the y-axis (normalized completion time), the target application's performance improves when the neighboring container requests fewer CPUs.

rKube is more effective and efficient; *rKube* correctly enforces the CPU shares and provides a performance guarantee for the target application. For example, the completion time is reduced by 40% when I decrease the number of allocated CPUs from 22/22 to 12/22. In this case, *rKube* outperforms the under-commitment strategy even when there are 10 additional CPUs made unassigned (and could be used by the target application).

Figure 4.7b compares under-commitment and *rKube* for `Memcached`. As shown, the throughput improves (while the 99%-ile tail latency decreases) as more CPUs become available (unassigned). I also observe similar trends for the latencies of `Memcached` update operations (omitted due to space limitations). Overall, when serving at the same throughput level, *rKube* significantly outperforms the under-commitment strategies in terms of the 99%-ile tail latency. Furthermore, with the help of *rKube*, `Memcached` achieves higher maximum throughput (the vertical dotted line in the figure), which is comparable with that of under-commitment. Again, unlike under-commitment, which sacrifices the overall CPU utilization (the CPU utilization drops from 22/22 to 16/22), *rKube* is able to maintain a high CPU utilization of 22/22.

With *rKube*, a higher level of performance improvement is achieved. More importantly, *rKube* maintains the highest level of CPU utilization without needing to sacrifice the resource share of the neighboring application. This is desirable for modern data centers that have long been suffering a notoriously low overall resource utilization [123, 124].

4.7 Related work

The Kubernetes and Docker community has long been suffering from the performance issues related to Linux scheduling and OS virtualization [35–39]. In the meantime, the Linux community has been actively investigating new cases where Linux fails to provide proper CPU isolation support for containerized applications [125–129]. However, some of these performance issues still persist in the latest Kubernetes and Docker release, stressing that improvement for the containerized environment is imperatively needed.

Researchers have identified drawbacks of the CPU schedulers. For example, Lozi et al. [130] reported that the Linux scheduler sometimes fails to make good use of the CPU resources and causes the performance degradation for some applications. Kim et al. [131] found that Complete Fair Scheduler (CFS) with distributed runqueues fails to achieve a global fare share scheduling, and proposed a global virtual time fair scheduling as a solution. Bouron et al. [132] analyzed the impact of two OS schedulers: ULE (the default FreeBSD scheduler), and CFS (the default Linux scheduler), on applications performance, and concluded there is no overall winner for complex use cases. My work focuses on the context of containerized environments, where fulfilling the promise of CPU resource requests is crucial to users.

Considerable prior works have examined performance isolation in multi-tenant clouds. CPI^2 [133] used cycle-per-instruction data to identify and throttle misbehaved tasks in the shared hosts. Bubble-Up [134] identifies contention on the shared resource as a major obstacle for high-priority, latency-sensitive tasks to share hosts with other tasks, and provides a characterization method that predicts the performance degradation due to the contention on shared resources in the memory subsystem. Studies showed that small latency variation per microservice would result in significant (tail) latency increases, and severely impact the end-user experience [40, 135]. *rKube* also targets the multi-tenant container cloud environment but is among the first to provide a solution to address this issue. Though contention

on shared hardware is commonly known as a challenge for performance isolation, I demonstrate that, even without CPU over-commitment, performance isolation is non-trivial due to the Linux scheduler design.

Some other studies have explored container performance isolation. Iron [41] improves the network CPU isolation by accounting the CPU time spent on the network stack on behalf of the co-located containers. Gao et al. [42] argued that Linux cgroups fails to achieve consistent and fair resource accounting, and showed that the resource consumption is not correctly charged to the specified cgroup configurations. In my work, I show that, in addition to those issues, the mismatched interaction between cgroups and Linux's default CFS scheduler is the main cause of severe performance interference.

4.8 Summary

Containerized cloud environments are becoming more and more popular in the production environment. While offering plenty of advantages, it has also been found that the application performance suffers from significant variations, making it difficult for resource requesting (for users) and resource planning (for cloud providers). In this chapter, I have quantitatively evaluated the performance variations of batch and interactive applications, and showed that the application could suffer a slowdown of 5x. These results motivated us to reason the underlying causes, which leads to the scheduling mechanism used in the underlying host. To address this issue, I have designed and implemented *rKube* by merely augmenting the existing Kubernetes with an additional option, making it backward compatible. My evaluation results show that *rKube* can effectively deliver the performance corresponding to a user's request and outperform the common best practices for scaling up in the production environments for improving the application's performance.

Chapter 5: Conclusion and future work

5.1 Conclusion

Cloud computing relies on multi-tenant sharing to achieve high resource utilization and economies of scale. As many previous studies have demonstrated, resource sharing among multiple tenants could lead to severe security and performance issues.

In this dissertation, I have first revisited the side-channel attacks in the multi-tenant cloud environment. I found that existing solutions either fail to provide sufficient protection at economical costs or limit their scope to specific attacks. Accordingly, I have proposed a lightweight and generic solution to eliminate a wide range of cross-VM and possibly unknown attacks. Our analyses have demonstrated that such attacks' efficacy could be dramatically reduced by distributing CPU resources as evenly as possible to all candidate vCPUs. Therefore, I have designed and implemented the *Shuffler schedulers* by incorporating this strategy and randomization into Xen's Credit scheduler. The evaluation results show that the Shuffler schedulers significantly reduce the vulnerable probabilities of all VMs, thus mitigating attacks without sacrificing the original resource sharing or performance.

Second, I have examined resource accounting in the multi-tenant cloud environment. Cloud computing relies on accurate resource allocation of VMs to better serve both cloud users and providers' needs. I have shown that the current approaches cannot correctly account for all the CPU usage incurred by domains due to I/O offloading through measurement and analysis. 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 issue, I have redefined the resource scope by using vCPU as a container. All the processing incurred by this domain is contained within its new resource scope. I have implemented *VASE System* that directly and accurately measures the offloaded CPU usage and uses it to enforce the CPU

usage limits in Xen strictly. The experiments have shown that our approach is lightweight and effective in constraining CPU usage with virtually no overhead.

Third, today containerized cloud environments are becoming more and more popular in the production environment. While offering plenty of advantages, the application performance suffers from significant variations, making it difficult for resource requesting (for users) and resource planning (for cloud providers). I have quantitatively evaluated the performance variations of batch and interactive applications and showed that the application could suffer a slowdown of 5x. These results motivated us to reason the underlying causes, which leads to the scheduling mechanism used in the underlying host. I have designed and implemented *rKube* by merely augmenting the existing Kubernetes with an additional option, making it backward compatible. Evaluation results show that my solution can effectively deliver the performance corresponding to a user's request. It outperforms the best practices for scaling up in the production environments for improving the application's performance.

5.2 Future work

In the most recent years, the container-based microservice architecture is rapidly changing the software DevOps workflow. However, containers' performance and security issues running in multi-tenant clouds have become a significant roadblock towards its wide adoption. The existing OS scheduler design, such as one-size-fits-all CFS in Linux, fails to accommodate the users' needs in the current container orchestration systems. CPU scheduling should be augmented to fulfill users' security and performance requirements in the multi-tenant host.

To secure the multi-tenant cloud, cross-container side-channel attacks worth further investigation. CPU scheduling in the multi-tenant host may be exploited to secure containers against attacks. Compared to cross-VM side-channel attacks studied in this dissertation, it is more challenging since 10x or even 100x more containers are deployed to a host.

To isolate tenants in the containerized environment is more challenging than in the virtualized environment due to co-located containers sharing more resources, such as the OS kernel and container runtime. This dissertation reveals the resource accounting and performance issues when I/O virtualization is concerned. Similar issues exist in the containerized environment and should also be studied. Furthermore, more use cases in which the Linux scheduler fails the user's expectation in the containerized environment need to be explored. By examining how existing scheduling mechanisms and considerations in the current scheduler design result in undesirable situations, new scheduling designs for the multi-tenant host are desirable.

Bibliography

[1] D. Chisnall, *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[2] Gartner, "Magic quadrant for cloud infrastructure as a service, worldwide," https://www.gartner.com/doc/reprints?id=1-1CMAPXNO&ct=190709&st=sb, (Accessed on 05/31/2020).

[3] AWS, "Cloud migration - amazon web services," https://aws.amazon.com/cloud-migration/, (Accessed on 05/31/2020).

[4] ——, "Global infrastructure," https://aws.amazon.com/about-aws/global-infrastructure/, (Accessed on 05/31/2020).

[5] "Pricing," https://aws.amazon.com/pricing/, (Accessed on 05/31/2020).

[6] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, pp. 1–27, 2016.

[7] S. Gueron, "Efficient software implementations of modular exponentiation," *Journal of Cryptographic Engineering*, vol. 2, no. 1, pp. 31–43, 2012.

[8] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Security*, 2015, pp. 431–446.

[9] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[10] "Amazon ec2 dedicated hosts," https://aws.amazon.com/ec2/dedicated-hosts/, 2018, accessed: 2018-02-19.

[11] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[12] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 11–11.

[13] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1595–1606.

[14] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, San Jose, CA, US, 2015.

[15] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: Smartnics in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[16] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor." in *USENIX Annual Technical Conference, General Track*, vol. 50, 2005.

[17] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization." in *USENIX Annual Technical Conference*, 2008, pp. 29–42.

[18] A. O. Ayodele, J. Rao, and T. E. Boult, "Performance measurement and interference profiling in multi-tenant clouds," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 941–949.

[19] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 281–292.

[20] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net i/o performance interference in virtualized clouds," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 314–329, 2013.

[21] R. C. Chiang, S. Rajasekaran, N. Zhang, and H. H. Huang, "Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for i/o resources," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1732–1742, 2015.

[22] T. Zhang, Y. Zhang, and R. B. Lee, "Dos attacks on your memory in cloud," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 253–265.

[23] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[24] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: exploiting performance heterogeneity in public clouds," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 20.

[25] P. Leitner and J. Cito, "Patterns in the chaosa study of performance variation and predictability in public iaas clouds," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, p. 15, 2016.

[26] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "Ecosystem: Managing energy as a first class operating system resource," *ACM SIGOPS operating systems review*, vol. 36, no. 5, pp. 123–132, 2002.

[27] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH computer architecture news*, vol. 35, no. 2. ACM, 2007, pp. 13–23.

[28] K. Kourai, S. Arai, K. Nakamura, S. Okazaki, and S. Chiba, "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, 2017, pp. 170–177.

[29] B. C. Tak, Y. Kwon, and B. Urgaonkar, "Resource accounting of shared it resources in multi-tenant clouds," *IEEE Transactions on Services Computing*, vol. 10, no. 2, pp. 302–315, 2017.

[30] S. T. Jones, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau *et al.*, "Antfarm: Tracking processes in a virtual machine environment." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 1–14.

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

[32] B. Teabe, A. Tchana, and D. Hagimont, "Billing the cpu time used by system components on behalf of vms," in *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 307–315.

[33] "Kubernetes - google kubernetes engine (gke) — google cloud," https://cloud.google.com/kubernetes-engine, (Accessed on 05/29/2020).

[34] "Amazon eks - managed kubernetes service," https://aws.amazon.com/eks/, (Accessed on 05/29/2020).

[35] "Cpu scheduler imbalance with cgroups — josef baciks blog," https://josefbacik.github.io/kernel/scheduler/cgroup/2017/07/24/scheduler-imbalance.html, (Accessed on 09/15/2020).

[36] "Cpu throttling - unthrottled: Fixing cpu limits in the cloud," https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/, (Accessed on 09/15/2020).

[37] "Cfs quotas can lead to unnecessary throttling issue #67577 kubernetes/kubernetes," https://github.com/kubernetes/kubernetes/issues/67577, (Accessed on 09/15/2020).

[38] "How to optimize i/o intensive containers on kubernetes - neuvector," https://neuvector.com/container-security/optimize-i-o-intensive-containers/, (Accessed on 09/15/2020).

[39] "Cpu considerations for java applications running in docker and kubernetes — by christopher batey — medium," https://link.medium.com/H3WcqAfND9, (Accessed on 09/15/2020).

[40] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[41] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based {CPU} in container environments," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 313–328.

[42] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, "Houdini's escape: Breaking the resource rein of linux control groups," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1073–1086.

[43] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004.

[44] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[45] L. Liu, A. Wang, W. Zang, M. Yu, M. Xiao, and S. Chen, "Shuffler: Mitigate cross-vm side-channel attacks via hypervisor scheduling," in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li, and S. Zhu, Eds. Cham: Springer International Publishing, 2018, pp. 491–511.

[46] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security.* ACM, 2009, pp. 199–212.

[47] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud," in *24th USENIX Security Symposium (USENIX Security 15).* Washington, D.C.: USENIX Association, Aug. 2015, pp. 929–944.

[48] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift, "A placement vulnerability study in multi-tenant public clouds." in *USENIX Security*, 2015, pp. 913–928.

[49] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud," IACR Cryptology ePrint Archive, Tech. Rep., 2015.

[50] L. Liu, A. Wang, W. Zang, M. Yu, and S. Chen, "Empirical evaluation of the hypervisor scheduling on side channel attacks," in *Communications (ICC), 2018 IEEE International Conference on*. IEEE, 2018.

[51] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)(Austin, TX, 2016), USENIX Association*, 2016, pp. 565–581.

[52] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 353–364.

[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 305–316.

[54] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 687–702.

[55] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732.

[56] "Credit scheduler," http://wiki.xen.org/wiki/Credit_Scheduler, 2017, accessed: 2018-02-19.

[57] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505.

[58] A. Kopytov, "Sysbench: a system performance benchmark," *URL: http://sysbench. sourceforge. net*, 2004.

[59] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.

[60] R. Ghosh and V. K. Naik, "Biting off safely more than you can chew: Predictive analytics for resource over-commit in iaas cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 25–32.

[61] S. D. Lowe, "Best practices for oversubscription of cpu, memory and storage in vsphere virtual environments," *Technical Whitepaper, Dell*, 2013.

[62] "Overcommitting cpu and ram," https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html, 2018, accessed: 2018-02-19.

[63] "Amazon ec2 instance types," https://aws.amazon.com/ec2/instance-types/, 2018, accessed: 2018-02-19.

[64] R. Zhang, X. Su, J. Wang, C. Wang, W. Liu, and R. W. Lau, "On mitigating the risk of cross-vm covert channels in a public cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2327–2339, 2015.

[65] N. Heninger and H. Shacham, "Reconstructing rsa private keys from random key bits," in *Advances in Cryptology - CRYPTO 2009*, S. Halevi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–17.

[66] T. Holenstein, M. Mitzenmacher, R. Panigrahy, and U. Wieder, "Trace reconstruction with constant deletion probability and related results," in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2008, pp. 389–398.

[67] M. Mitzenmacher *et al.*, "A survey of results for deletion channels and related synchronization channels," *Probability Surveys*, vol. 6, pp. 1–33, 2009.

[68] A. McGregor, E. Price, and S. Vorotnikova, "Trace reconstruction revisited," in *European Symposium on Algorithms*. Springer, 2014, pp. 689–700.

[69] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 297–307.

[70] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières, "Eliminating cache-based timing attacks with instruction-based scheduling," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 718–735.

[71] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 41–46.

[72] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[73] H. Wang, F. Li, and S. Chen, "Towards cost-effective moving target defense against ddos and covert channel attacks," in *Proceedings of the 2016 ACM Workshop on Moving Target Defense*. ACM, 2016, pp. 15–25.

[74] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*. IEEE, 2012, pp. 142–151.

[75] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 225–236.

[76] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 871–882.

[77] M. Godfrey and M. Zulkernine, "A server-side solution to cache-based side-channel attacks in the cloud," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 163–170.

[78] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 494–505.

[79] W.-M. Hu, "Lattice scheduling and covert channels," in *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*. IEEE, 1992, pp. 52–61.

[80] T. Vateva-Gurova, N. Suri, and A. Mendelson, "The impact of hypervisor scheduling on compromising virtualized environments," in *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1910–1917.

[81] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen, "Vcpu as a container: Towards accurate cpu allocation for vms," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 193206. [Online]. Available: https://doi.org/10.1145/3313808.3313814

[82] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *OSDI*, vol. 99, 1999, pp. 45–58.

[83] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based CPU in container environments," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 313–328. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/khalid

[84] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 231–242.

[85] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[86] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The tcp/udp bandwidth measurement tool," *htt p://dast. nlanr. net/Projects*, 2005.

[87] A. Kopytov, "Github - akopytov/sysbench: Scriptable database and system performance benchmark," https://github.com/akopytov/sysbench, (Accessed on 10/25/2020).

[88] M. Wilcox, "Ill do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Linux. conf. au*, 2003.

[89] "Pv on hvm," https://wiki.xen.org/wiki/PV_on_HVM, 2015, accessed: 2018-09-09.

[90] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[91] Z. Amsden, D. Arai, D. Hecht, A. Holler, P. Subrahmanyam *et al.*, "Vmi: An interface for paravirtualization," in *Proc. of the Linux Symposium*. Citeseer, 2006, pp. 363–378.

[92] Y. Dong, Z. Yu, and G. Rose, "Sr-iov networking in xen: Architecture, design and implementation." in *Workshop on I/O Virtualization*, vol. 2, 2008.

[93] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "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, 2008, pp. 283–294.

[94] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 437–450.

[95] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 321–334.

[96] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: enabling high-level slos on shared storage systems," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 14.

[97] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," *ACM SIGOPS operating systems review*, vol. 35, no. 5, pp. 103–116, 2001.

[98] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating systems support for planetary-scale network services." in *NSDI*, vol. 4, 2004, pp. 19–19.

[99] P. Druschel and G. Banga, "Lazy receiver processing (lrp): A network subsystem architecture for server systems," in *OSDI*, vol. 96, 1996, pp. 261–275.

[100] F. Ghanei, P. Tipnis, K. Marcus, K. Dantu, S. Ko, and L. Ziarek, "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, 2016, pp. 296–301.

[101] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[102] "Production-grade container orchestration - kubernetes," https://kubernetes.io/, (Accessed on 05/29/2020).

[103] "Swarm mode overview — docker documentation," https://docs.docker.com/engine/swarm/, (Accessed on 05/29/2020).

[104] "Pod overview - kubernetes," https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, (Accessed on 05/29/2020).

[105] "Managing resources for containers — kubernetes," https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, (Accessed on 08/27/2020).

[106] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[107] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[108] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.

[109] "memcached - a distributed memory object caching system," https://memcached.org/, (Accessed on 06/03/2020).

[110] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC 10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[111] C. I. King, "Stress-ng," 2020, accessed: 05/20/2020.

[112] "Assign cpu resources to containers and pods — kubernetes," https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#motivation-for-cpu-requests-and-limits, (Accessed on 09/16/2020).

[113] "spirals/parsec-3.0 - docker hub," https://hub.docker.com/r/spirals/parsec-3.0, (Accessed on 06/05/2020).

[114] "bitnami/memcached - docker hub," https://hub.docker.com/r/bitnami/memcached, (Accessed on 09/16/2020).

[115] "alexeiled/stress-ng - docker hub," https://hub.docker.com/r/alexeiled/stress-ng, (Accessed on 09/16/2020).

[116] "Control groups the linux kernel documentation," https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html, (Accessed on 09/17/2020).

[117] "Cfs scheduler the linux kernel documentation," https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, (Accessed on 09/06/2020).

[118] "Reserve compute resources for system daemons — kubernetes," https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/, (Accessed on 08/27/2020).

[119] "Load tracking in the scheduler [lwn.net]," https://lwn.net/Articles/639543/, (Accessed on 09/06/2020).

[120] "Per-entity load tracking [lwn.net]," https://lwn.net/Articles/531853/, (Accessed on 09/06/2020).

[121] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 347360. [Online]. Available: https://doi.org/10.1145/3267809.3267830

[122] "Horizontal scaling - aws well-architected framework," https://wa.aws.amazon.com/wat.concept.horizontal-scaling.en.html, (Accessed on 09/15/2020).

[123] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2391229.2391236

[124] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387517

[125] "Bkk19-tr06 - deep dive in the scheduler - youtube," https://www.youtube.com/watch?v=1xhK0cH2Dkg&ab_channel=LinaroOrg, (Accessed on 09/15/2020).

[126] "San19-220 deep dive in the scheduler - youtube," https://www.youtube.com/watch?v=_re97U8Vlzc&ab_channel=LinaroOrg, (Accessed on 09/15/2020).

[127] "Rework cfs load balance - youtube," https://www.youtube.com/watch?v=cfv63BMnIug&ab_channel=RetisLab, (Accessed on 09/15/2020).

[128] "[ospm-summit-17] parameterizing cfs load balancing: nr_running/util/load - youtube," https://www.youtube.com/watch?v=JyA5MpVpAAM&ab_channel=RetisLab, (Accessed on 09/15/2020).

[129] "[ospm-summit-17] tracepoints for pelt - youtube," https://www.youtube.com/watch?v=tyoFqxviXOY&ab_channel=RetisLab, (Accessed on 09/15/2020).

[130] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[131] C. Kim, S. Choi, and J. Huh, "Gvts: Global virtual time fair scheduling to support strict fairness on many cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 79–92, 2018.

[132] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena, "The battle of the schedulers: Freebsd {ULE} vs. linux {CFS}," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 85–96.

[133] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 379–391.

[134] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 248–259.

[135] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 149161. [Online]. Available: https://doi.org/10.1145/3267809.3267823

Curriculum Vitae

Li Liu received a Bachelor of Management degree from Nanjing University in 2009, and a Master of Engineering degree from Peking University in 2013. He started his Ph.D. studies in the Computer Science Department in the Volgenau School of Engineering at George Mason University in 2013.