Energy Profiling and Performance Optimization for Network-related Transactions in Virtualized Cloud

by

Chi Xu

B.Sc., Xidian University, 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Chi Xu 2016
SIMON FRASER UNIVERSITY
Fall 2016

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name:	**Chi Xu**
Degree:	**Master of Science (Computing Science)**
Title:	***Energy Profiling and Performance Optimization for Network-related Transactions in Virtualized Cloud***
Examining Committee:	**Chair:** Jiannan Wang Assistant Professor

Jiangchuan Liu
Senior Supervisor
Professor _____

Joseph Peters
Supervisor
Professor _____

Qianping Gu
Internal Examiner
Professor _____

Date Defended: December 13, 2016

Abstract

Networking and machine virtualization play critical roles in the success of modern cloud computing. The energy consumption of physical machines has been carefully examined in the past, including the impact from network traffic. When it comes to *virtual machines* (VMs) in cloud data centers, it remains unexplored how the highly dynamic traffic affects the energy consumption in virtualized environments. In this thesis, we first present an empirical study on the interplay between energy consumption and network transactions in virtualized environments. Through the real-world measurement on both Xen- and KVM-based platforms, we show that these state-of-the-art designs bring significant overhead on virtualizing network devices and noticeably increase the demand of CPU resources when handling network traffic. Furthermore, the energy consumption varies significantly with *traffic allocation strategies* and *virtual CPU affinity conditions*, which was not seen in conventional physical machines. Next, we study the performance and energy efficiency issues when CPU intensive tasks and I/O intensive tasks are co-located inside a VM. A combined effect from device virtualization overhead and VM scheduling latency can cause severe interference in the presence of such hybrid workloads. To this end, we propose *Hylics*, a novel solution that enables an efficient data traverse path for both I/O and computation operations, and decouples the costly interference. Several important design issues are pinpointed and addressed during our implementation, including efficient intermediate data sharing, network service offloading, and QoS-aware memory usage management. Based on our real-world deployment in KVM, Hylics can improve computation and networking performance with a moderate amount of memory usage. Moreover, this design also sheds new light on optimizing the energy efficiency for virtualized systems.

Keywords: Cloud computing; virtual machine; networking; energy efficiency; performance

Dedication

To my family.

Acknowledgements

First and foremost I would like to express my deepest gratitude to my senior supervisor, Professor Jiangchuan Liu. He offered me invaluable guidance, encouragement, and support during my master's studies. The joy and enthusiasm he has for his research were contagious and motivational for me.

Secondly, I would like to express my sincere thanks to my defence committee: my supervisor Professor Joseph Peters, examiner Professor Qianping Gu, and defence chair Professor Jiannan Wang, for their support and helpful suggestions.

Sincere gratitude should be given to my dear friends and lab mates for their company during all my pursuits.

Last but not least, I am also grateful to my family who have always supported me through the difficult times. This accomplishment would not have been possible without you. Thank you so much.

Chi Xu
December 2016

Table of Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

Table of Contents vi

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Contributions . . . 3

1.2 Thesis Organization . . . 3

2 Background 5

2.1 Cloud Computing and Virtualization . . . 5

2.2 An Overview on Virtualization . . . 5

2.3 Network Subsystem Design in Virtualized Environments . . . 6

2.3.1 Case Study: Network Architecture in Xen . . . 6

2.3.2 Case Study: Network Architecture in KVM . . . 7

2.4 Related Works . . . 8

2.4.1 Network Performance Improvement . . . 8

2.4.2 Network Co-location Interference in Virtualized Systems . . . 9

2.4.3 Energy Consumption of Virtualized Cloud Systems . . . 9

3 The Interplay between Energy Consumption and Network Transactions 11

3.1 Methodology . . . 11

3.1.1 Measurement Platform . . . 11

3.1.2 Measurement Tools . . . 12

3.2 Energy Consumption from Network Traffic . . . 13

3.2.1 Extra Energy Overhead . . . 13
3.2.2 Impact from Traffic Allocation Strategy . . . 16
3.2.3 Explanations . . . 18
3.3 Energy Variation from Virtual CPU Affinity . . . 19
3.4 Further Discussion . . . 22

4 Performance and Energy Efficiency Issues of Hybrid Workloads 25
4.1 Motivation . . . 25
4.1.1 Measurement and Observations . . . 25
4.1.2 Existing Approaches and Opportunities . . . 29
4.2 Framework Design . . . 31
4.2.1 Overview . . . 32
4.2.2 Case Study . . . 33
4.3 System Implementation . . . 34
4.3.1 Sharing In-memory File System . . . 34
4.3.2 Offloading Network Operations . . . 35
4.4 Memory Usage Analysis and Enhancement . . . 36
4.4.1 Online Self-adaptive Control Scheme Design . . . 37
4.4.2 Parameter Selection . . . 40
4.5 Performance Evaluation . . . 41
4.5.1 Experiment Configuration . . . 41
4.5.2 Benchmark Performance . . . 43
4.5.3 Performance Gain in Real-world Applications . . . 44
4.6 Further Discussion . . . 48

5 Conclusions and Future Directions 50
5.1 Conclusions . . . 50
5.2 Future Directions . . . 50

Bibliography 52

List of Tables

Table 3.1 Xen CPU profiling 15
Table 3.2 KVM CPU profiling 15
Table 3.3 CPU profiling under different traffic combinations (Xen) 19
Table 3.4 CPU profiling under different traffic combinations (KVM) 19

Table 4.1 Summary of notations 38
Table 4.2 Aggregated response time error 47
Table 4.3 Perf profiling 48

List of Figures

Figure 2.1 Xen network architecture 7
Figure 2.2 KVM network architecture 8

Figure 3.1 Energy consumption of Xen sending scenario 13
Figure 3.2 Energy consumption of KVM sending scenario 14
Figure 3.3 CPU energy measurements with 5 active VMs 16
Figure 3.4 Server energy measurements with 5 active VMs 16
Figure 3.5 CPU energy measurements with 8 active VMs 17
Figure 3.6 Server energy measurements with 8 active VMs 17
Figure 3.7 Different CPU affinity conditions 20
Figure 3.8 CPU energy measurements with different virtual CPU affinity conditions (Xen) 21
Figure 3.9 Server energy measurements with different virtual CPU affinity conditions (Xen) 21
Figure 3.10 CPU energy measurements with different virtual CPU affinity conditions (KVM) 23
Figure 3.11 Server energy measurements with different virtual CPU affinity conditions (KVM) 23
Figure 3.12 Domain-*U* CPU utilization sum (Xen) 24
Figure 3.13 Domain-0 CPU utilization (Xen) 24

Figure 4.1 Network throughput when experiencing interference 26
Figure 4.2 CDF of network throughput 26
Figure 4.3 Comparison of transcoding performance 26
Figure 4.4 Comparison of file compression performance 26
Figure 4.5 Impact from scheduling policy 27
Figure 4.6 Comparison of energy consumption 27
Figure 4.7 KVM network I/O subsystem 28
Figure 4.8 KVM disk I/O subsystem 28
Figure 4.9 Native workflow for hybrid workloads 30
Figure 4.10 Hylics workflow for hybrid workloads 31
Figure 4.11 System architecture 32

Figure 4.12 Sysbench VM read performance 42
Figure 4.13 Sysbench VM write performance 42
Figure 4.14 dd VM write performance 43
Figure 4.15 Sysbench host read performance 43
Figure 4.16 Sysbench host write performance 44
Figure 4.17 Video trancoding performance 44
Figure 4.18 Varied number of vCPUs and threads 45
Figure 4.19 File compression performance 45
Figure 4.20 Streaming server performance 45
Figure 4.21 Web server performance 45
Figure 4.22 Response time of the synthetic workload #1 46
Figure 4.23 Response time of the synthetic workload #2 46
Figure 4.24 Energy consumption of streaming-only workload 47
Figure 4.25 Energy consumption of hybrid workloads 47

Chapter 1

Introduction

Cloud computing is emerging as a promising paradigm that enables on-demand and elastic access to computing infrastructures. The rapid adoption of the cloud computing has made modern data centers grow at a fast pace to support a wide spectrum of cloud-based systems. Unfortunately, the explosive expansion of large-scale data centers greatly aggravates the power consumption. This has unavoidably restricted the sustainable growth of cloud services and posed unprecedented pressure to the cloud service providers. According to the U.S. Environment Protection Agency (EPA) congressional report on data center infrastructures [1], in the year 2014 alone, data centers consumed about 70 billion kilowatt-hours (kWhs), accounting for 2% of the total U.S. electricity use.

The advances in modern inter-networking and machine virtualization technologies play critical roles in the success of cloud computing. Together they supply virtually unlimited resources from cloud data centers for massive geo-distributed clients, as if each being in its dedicated and isolated space. The energy consumption of physical machines has been carefully examined in the past, including the impact from network traffic [10, 13, 20, 21]. When it comes to *virtual machines* (VMs) in the cloud, the interplay between energy consumption and network traffic becomes much more complicated. In particular, the inter-VM traffic can reside in different physical machines with their respective *network interface cards* (NICs), or share the same physical machine [60]. When multiple VMs share one physical NIC (pNIC), their traffic can interfere with each other, causing extra overhead [42]. To make the matter worse, the virtual machines can also dynamically migrate across physical machines, thereby changing the traffic pattern [17, 71]. It remains largely unexplored if such dynamic traffic will eventually affect the energy consumption in virtualized environments.

In this thesis, we first take a comprehensive measurement study to understand the interplay between energy consumption and network traffic in representative virtualized environments. Conducted on real-world platforms with the synthetic workloads generated by commercial software, our study reveals a series of unique energy consumption characteristics of the network traffic in this context. We summarize our major observations as follows:

The state-of-the-art virtualization designs such as Xen and KVM noticeably increase the energy consumption when handling network-related transactions. This is mainly because the network packets will traverse through multiple layers in virtualized environments. Even when a physical machine is in an idle state, e.g., not involved in CPU intensive tasks, its VM's network transactions still incur non-trivial energy consumption. More interestingly, even with an identical number of active VMs and the same amount of network traffic, the energy consumption can vary significantly with different traffic allocation strategies and virtual CPU affinity conditions. However, network parameters such as bridging schemes cause negligible energy consumption variations on delivering network traffic in virtualized environments.

Next, we study the performance and energy efficiency issues when CPU intensive tasks and I/O intensive tasks are simultaneously allocated inside a VM. In the cloud context, a combined effect from device virtualization overhead and VM scheduling latency can cause severe *self-interference* in the presence of hybrid workloads[1]. Different from the well-discussed *cross-VM interference* [15, 39, 40, 43], self-interference happens within a VM when the I/O handling process is interfered or even starved by other processes inside the VM. This is very common when the co-located computation processes aggressively use the CPU resources. The impact of self-interference remains largely unexplored, and a solution is yet to be developed for common cloud services demanding both data processing and transmission.

To fully understand the self-interference, we first perform a comprehensive measurement study in the KVM environment. Motivated by the measurement results and an in-depth analysis, we present *Hylics*, a modified virtualization architecture that resolves the self-interference for hybrid workloads. The insight of the Hylics design is to shorten the data traverse path for both data processing and transmission. Meanwhile, it also decouples I/O and computation operations for cloud VMs. Several important design issues are addressed during our implementation, including efficient intermediate data sharing, network service offloading, and QoS-aware memory usage management. Based on our deployment in the KVM environment, Hylics can significantly improve the computation and networking performance. Moreover, the design also optimizes the overall energy efficiency of virtualized systems.

[1]The workloads containing both CPU intensive tasks and I/O intensive tasks are referred to as *hybrid workloads* in the rest part of the thesis.

1.1 Contributions

The contributions in this thesis are summarized as follows:

- We take a comprehensive measurement study to understand the interplay between energy consumption and network traffic in representative virtualized environments.
- We explore the possible relationship between energy consumption and various parameters in the cloud context, including traffic allocation strategy, virtual CPU affinity, and network bridging configuration.
- We further examine the impact of self-interference with real-world cloud-based workloads and provide an in-depth analysis to pinpoint the root cause.
- We propose and implement Hylics, an enhanced virtualization framework to resolve the self-interference and address several key design issues.
- We prove that Hylics can jointly boost the network and computation performance for typical cloud applications. Consequently, the energy efficiency of the underlying server is improved as well.

1.2 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2 ,we first introduce the background of cloud computing and virtualization technologies (Section 2.1). We then investigate the state-of-the-art virtualization technologies (Section 2.2), and use such typical environments as Xen and KVM to explain how these virtualized systems are implemented to handle network traffic in Section 2.3. We then present related research works in Section 2.4.

In Chapter 3, we present a measurement study on real-world virtualized systems to profile their energy consumption. We first introduce the measurement configurations and experiment design in Section 3.1. Based on the measurement results, Section 3.2 reveals the relationship between energy consumption and different network traffic allocation strategies in virtualized environments. A follow-up investigation in Section 3.3 further indicates that other factors, such as *virtual CPU affinity*, can also affect virtual machine's energy consumption when handling network-related transactions. In Section 3.4, we provide a further discussion to validate our observations.

Chapter 4 systematically investigates the performance and energy efficiency issues of hybrid workloads in cloud environments. In particular, Section 4.1 introduces the background and motivations. In Section 4.2, we present the framework design of the Hylics architecture. Section 4.3 describes the implementation details. Section 4.4 introduces the analysis and enhancement of Hylics memory usage. In Section 4.5, we show the experimental results

of the Hylics prototype. In Section 4.6, we present the profiling statistics of the Hylics architecture.

In Chapter 5, we conclude the thesis and discuss the future research directions.

Chapter 2

Background

2.1 Cloud Computing and Virtualization

Cloud computing is a type of Internet-based computing that offers on-demand computer processing resources. It provides a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources (e.g., computer networks, servers, storage, applications and services), which can be rapidly provisioned and released with minimal management effort. Cloud computing and storage solutions provide users with various capabilities to store and process their data in third-party data centers that may be located far from the user.

The main enabling technology for cloud computing is virtualization. Virtualization software separates a physical server into virtual system resources, which can be easily used and managed to perform computing tasks. With virtualization solution essentially creating a scalable system of multiple independent computing devices, idle computing resources can be allocated and used more efficiently. Virtualization provides the agility required to speed up IT services, and reduces operational cost by increasing infrastructure utilization.

2.2 An Overview on Virtualization

State-of-the-art virtualization solutions can be broadly classified into three categories, *Paravirtualization (PVM)*, *Hardware-assisted Virtualization (HVM)*, and *Container Virtualization* [16] [53].

PVM and HVM share the common feature that they generally introduce a *hypervisor* between multiple VMs and the underlying hardware to provide abstractions of physical resources. This design allows VMs to share devices and ensures security as well as performance isolation. To name a few mature products in market, Xen is one representative of PVM, and KVM is one widely-known project of HVM. The difference is that PVM does not require virtualization extensions from the host CPU architecture [9]; it however requires

a specialized kernel that is ported to run natively above a PVM hypervisor. In the PVM design, the guests are aware of the existence of hypervisor and can run efficiently without emulation or virtual emulated hardware [61]. HVM, instead, allows guest operating systems to run on a hypervisor without modifications [6] [45]. An HVM hypervisor requires CPU virtualization extensions from the host CPU architecture (e.g., Intel VT, AMD-V). The hypervisor is then capable of trapping and virtualizing the execution of sensitive, non-virtualizable instructions.

Container Virtualization [59], also known as operating system virtualization, is a light-weight virtualization solution which creates multiple secure containers hosting different applications. The containers provide isolation in between different applications and guarantee the scalability and performance. Typical examples of container virtualization include OpenVZ [14] and Linux-VServer [18]. It is known that container-based solutions incur minimal interference between two containers [47], and the extra energy overhead is almost negligible [56]. They, however, lack flexibility in terms of OS selection. The users are usually limited to run a single operating system with multiple containers, e.g., users cannot run Linux and Windows together on the same physical machine.

Virtualization technologies are the building foundations of modern cloud computing. Amazon AWS[1] and Rackspace Cloud[2], two major cloud platforms, are both based on customized Xen hypervisors; KVM hypervisors have been used in Eucalyptus Cloud Service[3] and the Ubuntu Enterprise Cloud[4], another two representative cloud platforms. Therefore, we focus on Xen and KVM in this thesis to understand their energy consumption and performance issues.

2.3 Network Subsystem Design in Virtualized Environments

As a next step, we take Xen and KVM as two examples to illustrate how these platforms handle network traffic.

2.3.1 Case Study: Network Architecture in Xen

A closer look into the network architecture in Xen is given in Fig. 2.1, which lists the key steps involved in delivering packets to a Xen-based VM. In this figure, Domain-0 is the initial domain started by the Xen hypervisor up on boot. It runs the Xen management toolstack, and has special privileges. Domain-U refers to a set of unprivileged VMs, being provided to tenants. In Domain-U, the *netfront module* is designed to manage the network traffic from/to Domain-0. This module and its counterpart, *netback module* (in Domain-0),

[1] `https://aws.amazon.com/`
[2] `https://www.rackspace.com/cloud`
[3] `http://www8.hp.com/us/en/cloud/helion-eucalyptus.html`
[4] `http://www.ubuntu.org.cn/cloud`

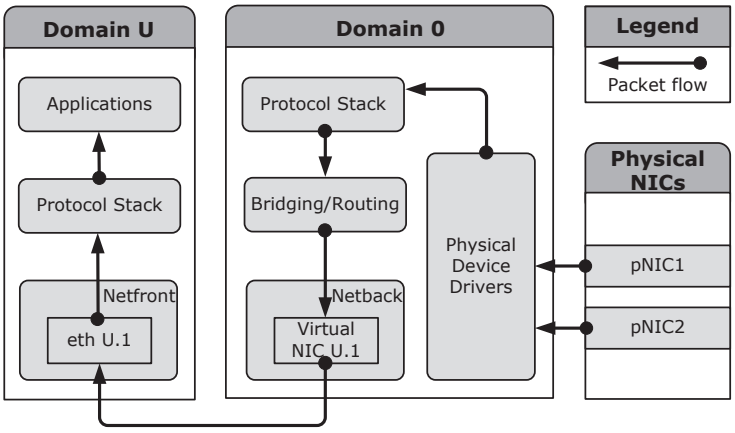


Figure 2.1: Xen network architecture

are a pair of inter-linked drivers, bridging the network communications between different domains. For example, upon the recipient of a packet in Xen, the physical NIC will deliver the packet to the physical device driver in Domain-0 [54]. Once the packet arrives at the bridge through the protocol stack, it will be transferred through the netback module to the netfront module. In particular, the netback module allocates resources to process the packet and notifies the netfront module in Domain-U. Finally, the netfront module receives the packet and passes it to the guest's network layer.

2.3.2 Case Study: Network Architecture in KVM

A similar network architecture is adopted in the KVM environment, despite that, the Linux kernel takes most of the responsibilities of Domain-0 in Xen. The KVM hypervisor runs in the kernel space and provides the core virtualization infrastructure. To reveal more details, Fig. 2.2 presents a state-of-the-art network architecture of KVM, where the *vhost* driver [3] provides in-kernel virtual I/O device emulation. This design puts virtual I/O emulation codes into the Linux kernel, which enables the device emulation function to directly call into kernel subsystems instead of performing system calls from the user space. Note that the in-kernel vhost module does not emulate a complete virtual network I/O function. Instead, it restricts itself to implement virtual network queue operations only. QEMU [16] is a generic and open source machine emulator, and is used in the KVM architecture to help perform virtual I/O feature negotiation. This means a vhost driver is not a self-contained virtual I/O device implementation, but depends on QEMU process to handle the control plane of network traffic. The data plane of network traffic is done in the kernel space. In particular, a vhost working thread running inside the kernel space waits for the virtual queue dumping and handles buffers that have been placed in the virtual queue.

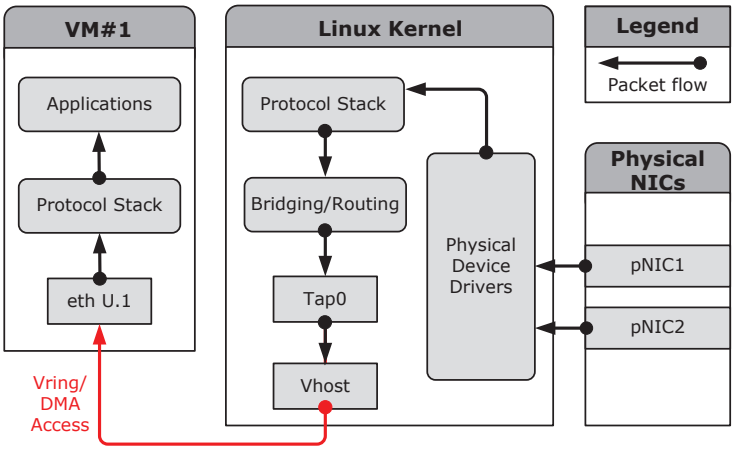


Figure 2.2: KVM network architecture

2.4 Related Works

From the design of the networking architecture in typical virtualized environments, it is clear that network packets have to traverse through multiple layers, and thus the extra overhead is inevitable. Although much effort has been put on reducing latency and enhancing network throughput in virtualized environments, the trade-off between network performance and energy efficiency remains an open problem to answer. To our best knowledge, there is no perfect solution to achieve the best of the both.

2.4.1 Network Performance Improvement

Improving network I/O performance in virtualized environments has long been a challenge. In recent years, we have seen solutions to reduce the network device emulation overhead. For example, Gamage *et al.* proposed to create shortcuts (known as vPipe) without involving guest VMs [25] to improve the network performance. Their solutions focused on the static data transfer in virtualized environments. Protocol offloading has also been suggested to enhance TCP-based applications, e.g., in vPro [24]. SR-IOV enabled NICs [19] can eliminate device virtualization overhead when handling network transactions by completely bypassing the hypervisor. The scheduling latency for network I/O intensive VMs is also critical in virtualized environments, since VM scheduling can bring delays of tens of milliseconds to the network I/O processing. There have been works on avoiding such delays by prioritizing certain interrupts [28, 37], and using soft real-time methods to shorten delays [51, 63].

Recent studies [65] [55] also suggest reducing CPU time slice for latency sensitive VM. Such approaches enable VMs to get scheduled more often so as to improve the I/O throughput. To take a step further, vTurbo [64] offloads VMs' I/O processing to a dedicated core

with extremely small time slices. In summary, these studies focus dominantly on modifying the hypervisor scheduler to hide the virtualization reality by reducing the delays as much as possible. However, this type of approaches brings increased VM context switches and more complicated CPU resource allocation.

2.4.2 Network Co-location Interference in Virtualized Systems

The rising popularity of cloud-based system deployment has also attracted an increasing number of studies to investigate the VM co-location interference. Oh *et al.* [46] pinpointed the existence of the co-location interference among cloud VMs, which brings negative effects on the computation and I/O performance. Zhu *et al.* [73] further quantified the co-location interference of two VMs by using the correlation of their resource utilization. To avoid such co-location interference, approaches such as static resource isolation are first suggested by research communities. As for the network subsystem, the study from Shieh *et al.* [57] proposed concrete isolation techniques such as the provision of multiple network adapters and the static bandwidth allocation for VM.

As a matter of fact, the interference discussed in previous papers comes from the multi-tenancy nature in virtualized cloud environments. The extent of such interference heavily depends on the combination of workloads running on co-located VMs [48]. Later on, effective profiling and prediction tools have been developed [8, 72]. There have been recent efforts towards identifying the impact and pattern of the interference among such co-located workloads, as well as developing efficient workload handling strategies [43, 44, 49, 70]. It is noted that these methods are designed for cloud providers, which require detailed runtime information and a global view on the resource management. The techniques include smart scheduling, live migration, and resource containment. Service reconfiguration is also suggested to mitigate interference from cloud consumers' point of view [39, 40].

2.4.3 Energy Consumption of Virtualized Cloud Systems

There have also been a significant number of studies on the energy consumption of virtualized cloud systems. Mastelic *et al.* [41] provided a comprehensive analysis on the underlying infrastructure supporting cloud computing regarding energy efficiency. The survey covers energy efficiency in server domain, supporting management system domain, and appliance domain. Jin *et al.* [33] presented an empirical study on how virtualization techniques have influence on server power consumption. Huang *et al.* [31] further evaluated the power consumption caused by VM migration. The authors showed the effectiveness of adopting smart consolidation strategies to achieve less power overhead. In [36], Kansal *et al.* presented a solution for VM power metering, named *Joulemeter*. The authors also proposed different models to infer the power consumption from resource usage indicators at runtime. To address the power consumption issues, Kusic *et al.* [38] implemented and validated a

dynamic resource provisioning framework for virtualized server environments. A recently published paper from Shea *et al.* [56] showed that energy consumption caused by network transactions can be reduced by using adaptive packet buffering.

Chapter 3

The Interplay between Energy Consumption and Network Transactions

3.1 Methodology

To reveal the interplay between network traffic and energy consumption in Xen and KVM, it is required to monitor the power usage of the physical data centers. Unfortunately, we cannot perform such experiments on public cloud platforms because the cloud providers do not reveal such system-level information to the general public. However, both Xen and KVM are based on open-source implementations and have publicly available documents. Therefore, we configured such virtualization software on our local cloud testbed, and the detailed measurement setup is presented as follows.

3.1.1 Measurement Platform

Our testbed includes a typical midrange server equipped with an Intel's core i5 2400 3.09GHz quad core CPU, 8GB 1333MHz DDR3 RAM. We have two Broadcom network interface cards attached to the PCI-E bus, each of which has a maximum throughput of 1000 Mbps. The reason why we choose Intel's i5 as the CPU is that the Intel's x86 architecture is dominating the CPU market for a long period of time. Most major cloud computing providers, including Amazon, base their virtual machine implementation on the x86 architecture. The core i5 also well reflects Intel's effort toward energy-efficient CPU design [22]. Since we focus on the relationship between network traffic and power consumption, we have configured a second machine to work as the other end of the network traffic. These two machines are directly connected by a 1000 Mbps Linksys SD2005 SOHO switch.

We configured Xen version 4.3 as the hypervisor on our testbed machine. We set the number of accessible virtual CPUs to be two and the amount of RAM to be 2048MB for each targeted virtual machine. Also, We compiled KVM version 1.2.0 from the official Debian source repository and deployed on the testbed machine. Similarly, each virtual machine was given full access to both of the two virtual processor cores as well as 2048MB of the total memory. The disk interface was configured as a flat file on the physical host's file system.

3.1.2 Measurement Tools

Energy efficiency has been an important consideration in the new generation of CPU design, and Intel has introduced the Running Average Power Limit (RAPL) hardware counters [2] in the Sandy Bridge line of processors. These accurate and versatile hardware counters allow users to extract the record of their CPU power consumption. Therefore, we measured the power consumption using RAPL counters when we ran all the network-related experiments.

Unfortunately, the RAPL counters only record the power consumption of CPU, and thus we need other measurement tools to evaluate the overall power consumption of the physical machine. In general, there are modules other than CPU that consume a considerable amount of power, namely, cooling fans, hard drives, GPUs, etc. To measure the power spent on these parts, we wired a digital multi-meter (Mastech MAS-345) into the AC input power line of our machine. We obtained the power readings by collecting samples every second throughout our experiment.

To examine the interplay between network traffic and power consumption, we used both benchmark tool `Iperf` and real-world application `Apache HTTP server` to generate network traffic. Iperf is a widely used configurable network benchmark, which allows users to generate network traffic and then gauge the performance of the network flows. The performance metrics include throughput, round trip time (RTT) and jitter. Also, we can use Iperf to tune the traffic load on different VMs. We further set up multiple Apache web servers on these VMs and used Apache benchmark suite to generate the downloading traffic. To provide further resource usage information, we captured the virtual CPU utilization in Xen using `xentop`, which is a standard resource monitoring tool integrated into the Xen distribution. In KVM, we used `perf`, a Linux hardware performance analysis tool to collect system level statistics, such as CPU cycles and context switching information, which can reveal more details on how much effort for CPU to deliver the network traffic. Both of our CPU benchmarks are set to use the lowest Linux scheduler priority by using the `nice` command. This command ensures that the network task is given priority to run on the virtual CPU. To avoid randomness in our data, we ran each experiment 100 times and calculated the average and their standard deviation. The standard deviation is shown in final results as error bars in our figures.

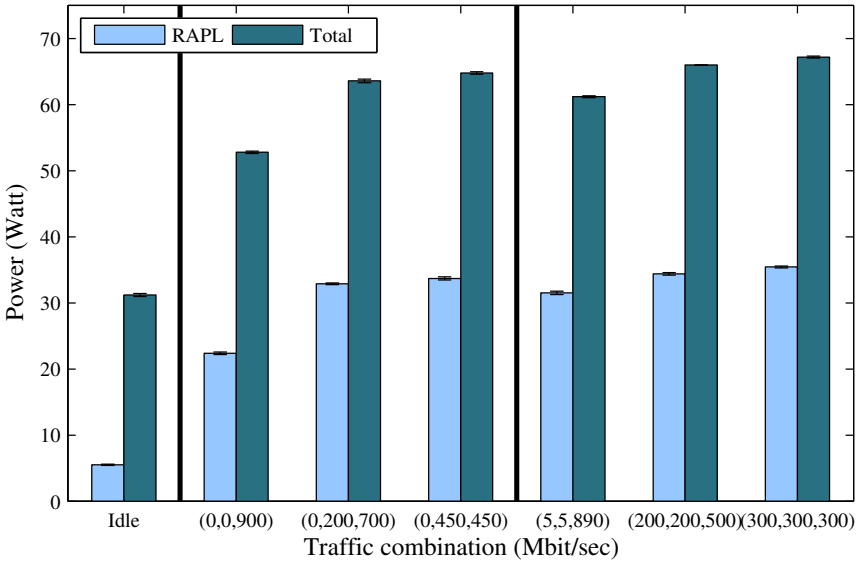


Figure 3.1: Energy consumption of Xen sending scenario

3.2 Energy Consumption from Network Traffic

In this section, we will clarify the extra energy overhead in virtualized environments when handling network transactions. We will further explore the possible relationship between energy consumption and different traffic allocation strategies.

3.2.1 Extra Energy Overhead

We first set up three VMs on the physical machine in both Xen and KVM environments. To quantify the extra energy overhead caused by network traffic, we fixed the cap of total traffic amount on this physical machine to be 900 Mbps, which is corresponding to the maximum transmission capability of a single NIC. The fixed cap ensures that the energy variation observed is not due to the change of the total traffic amount, e.g., generating 800 Mbps traffic surely consumes more power than generating 100 Mbps traffic. After that, by tuning the specific traffic amount on each VM, we used both RAPL and AC power meter to collect the energy consumption readings. In the first experiment, we define the traffic combination as a *3-tuple* across these three VMs. For example, $(100, 100, 700)$ denotes that the three co-located VMs are sending or receiving at 100 Mbps, 100 Mbps, 700 Mbps, respectively. Value 0 simply means there is no traffic assigned to this VM.

Observation 1. State-of-the-art virtualization designs such as Xen and KVM noticeably increase the energy consumption when handling network transactions.

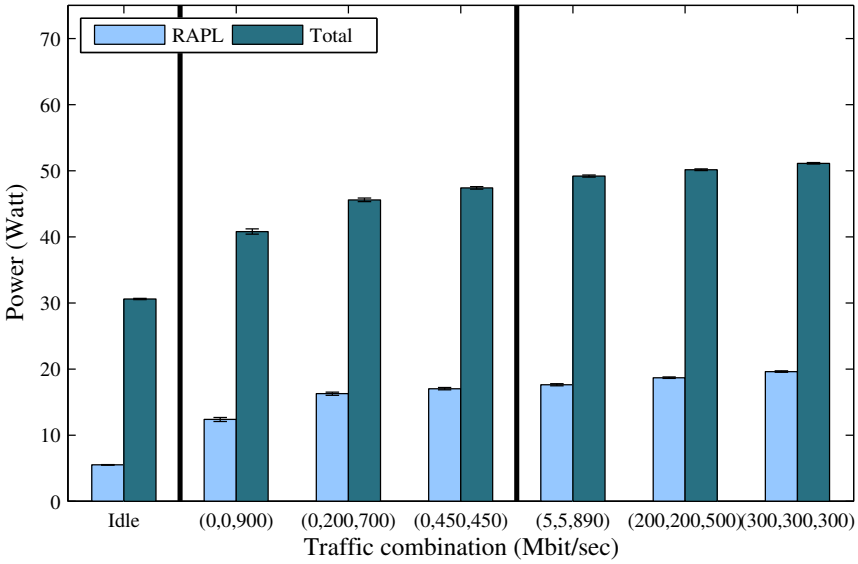


Figure 3.2: Energy consumption of KVM sending scenario

As we can see from Fig. 3.1 and Fig. 3.2, when the physical machine is in a totally idle state (with no CPU- or network-related tasks), the RAPL reading is 5.50W in the Xen environment and 5.14W in the KVM environment. Meanwhile, the power meter captures that the effective current is 0.261A and 0.252A, respectively in the two environments. The effective power is then calculated to be 31.3W and 30.24W, respectively. When we allocated 900 Mbps traffic on the physical machine, we can observe a significant increase (17.3W in Xen and 7.2W in KVM) on the CPU energy consumption. Meanwhile, by comparing the difference between the RAPL readings and the total energy consumption, we find that the energy consumption of other components, such as RAM, disk, NIC, and cooling fans, is relatively stable with negligible variations.

The main reason behind the extra overhead is that the design of hypervisor introduces extra layers between the physical devices and the destination applications. The detailed CPU utilization information is shown in Table 3.1. To handle the network traffic, the virtualized system generates excessive interrupt requests. At the same time, the hypervisor also needs to schedule Domain-0 and Domain-U to run and share the physical cores. All these operations greatly increase the CPU utilization in both Domain-0 and Domain-U, consuming non-trivial energy. We can observe that the CPU utilization is 55.2% in Domain-0 and 51.3% in Domain-U when dealing with 900 Mbps network traffic. We show the corresponding CPU cycles and context switching information in Table 3.2 when performing the same experiments in the KVM environment. Similar conclusions can also be made.

Observation 2. The total number of active VMs affects the energy consumption in virtualized environments.

Table 3.1: Xen CPU profiling

VM #1	**VM #2**	**VM #3**	**Domain-0 CPU utilization**	**Domain-U CPU utilization sum**
0 Mbps	0 Mbps	900 Mbps	$(55.2 \pm 1.1)\%$	$(51.3 \pm 1.3)\%$
0 Mbps	200 Mbps	700 Mbps	$(81.1 \pm 1.3)\%$	$(73.3 \pm 0.9)\%$
300 Mbps	300 Mbps	300 Mbps	$(95.4 \pm 1.2)\%$	$(94.7 \pm 2.2)\%$

Table 3.2: KVM CPU profiling

VM #1	**VM #2**	**VM #3**	**CPU cycles per sec**	**Context switches per sec**
0 Mbps	0 Mbps	900 Mbps	$(1.46 \pm 0.28) \times 10^9$	$(57.80 \pm 1.22) \times 10^3$
0 Mbps	200 Mbps	700 Mbps	$(2.48 \pm 0.25) \times 10^9$	$(60.71 \pm 1.89) \times 10^3$
300 Mbps	300 Mbps	300 Mbps	$(3.25 \pm 0.30) \times 10^9$	$(103.73 \pm 3.51) \times 10^3$

In both Fig. 3.1 and Fig. 3.2, we can observe that the power consumption has spikes from case $(0, 0, 900)$ (with one VM actively involved with network transactions) to case $(0, 200, 700)$ (with two active VMs). Similar spikes also exist when we compare case $(0, 200, 700)$ (with two active VMs) and case $(300, 300, 300)$ (with three active VMs). For example, in the Xen environment, the RAPL readings in these three cases are 22.4W, 32.9W and 35.4W, respectively. The effective power of the physical server is 52.9W, 63.8W, and 67.3W, respectively in these three cases. The same observation can also be made in the KVM environment.

This result is relatively intuitive because an extra active VM introduces increased network interrupt requests, which incurs even more context switches. Particularly, as with an extra virtual machine involved in dealing network traffic, the hypervisor has to maintain an additional set of network stack to deliver network packets; meanwhile, more network interrupt requests incur increased switching in between multiple VMs, and the VM scheduling is becoming more complicated and unpredictable. As can be concluded from Table 3.2, in the KVM environment, an extra active VM will approximately consume 10^9 more CPU cycles in one second. It also largely increases the CPU utilization in both Domain-0 and Domain-U in the Xen environment, which is shown in Table 3.1. This will unavoidably lead to higher energy consumption.

It is also worth noting that the energy consumption can still be quite different with an identical number of active VMs in Fig. 3.1 and Fig. 3.2. In detail, the energy consumption varies by $11.3\% - 12.6\%$ when there are three active VMs in the system, e.g., by comparing case $(5, 5, 850)$, case $(200, 200, 500)$ and case $(300, 300, 300)$. It is thus interesting to see whether different *traffic allocation strategies* also affect the energy consumption of the underlying server.

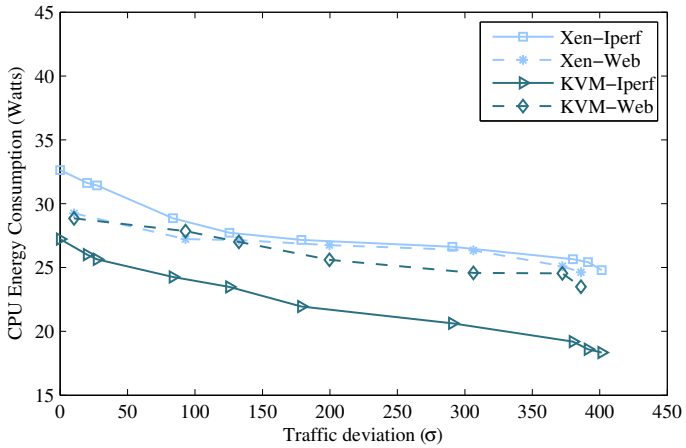


Figure 3.3: CPU energy measurements with 5 active VMs

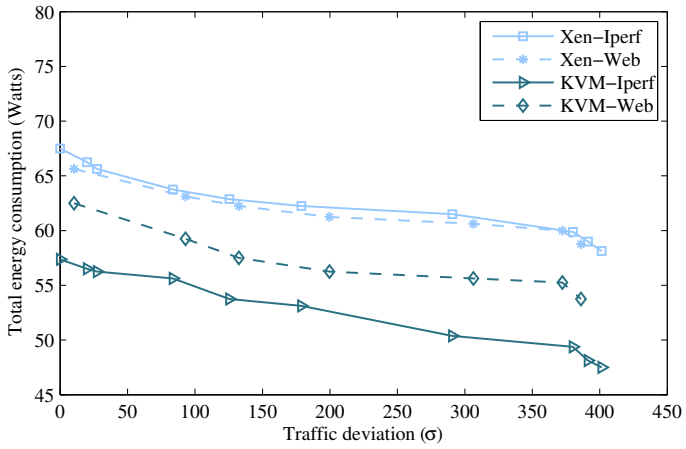


Figure 3.4: Server energy measurements with 5 active VMs

3.2.2 Impact from Traffic Allocation Strategy

To further understand the relationship between energy consumption and different traffic allocation strategies, we enlarged the scale of our experiments and enabled 5 and 8 VMs actively involved with network transactions. The traffic combination is therefore a 5-tuple or 8-tuple under the new configurations. For the sake of clarity, we define *traffic deviation* σ as an indicator to capture the skewness of the traffic distribution across different VMs.

Traffic deviation σ is defined as follows: $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(T(i) - \overline{T})^2}$, where n is the number of the co-located VMs involved in network transactions, $T(i)$ denotes the traffic load on the ith VM. $\overline{T}$ refers to the average traffic load. The traffic deviation σ can serve as an inverse indicator on the interference in between the traffic on VMs. Note that when we achieve a perfect balanced traffic across the VMs, the value of σ will be 0. On the other

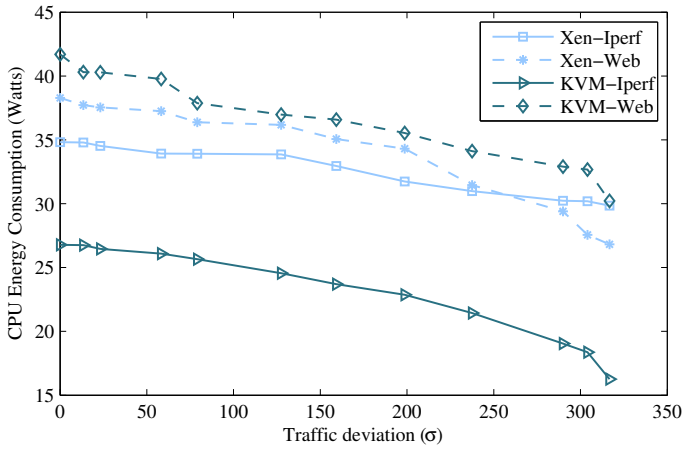


Figure 3.5: CPU energy measurements with 8 active VMs

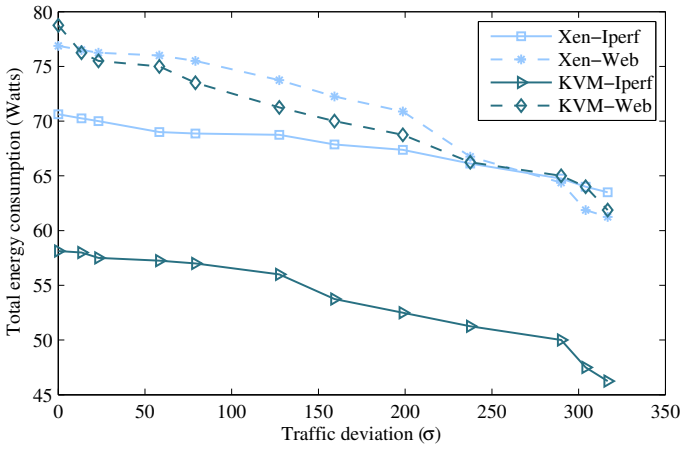


Figure 3.6: Server energy measurements with 8 active VMs

hand, the increasing skewness will lead to a larger σ. For example, σ can reach up to 400 when the traffic allocation strategy is $(0.5, 0.5, 0.5, 0.5, 898)$.

Observation 3. Even with an identical number of active VMs and same total amount of traffic on the physical machine, the energy consumption can vary significantly with different traffic allocation strategies.

With the results presented in Fig. 3.3 and Fig. 3.4, we investigate the correlation in between the traffic deviation σ and the energy consumption. We can see that all the curves are monotonically decreasing. This indicates that the load-balanced traffic ($\sigma = 0$) will unfortunately introduce very high energy consumption. Meanwhile, by carefully adjusting the skewness of the traffic allocation, we can achieve up to 31.6% energy savings on the physical machine. This is a significant improvement in the context of data center environment.

It is also worth noting that in Fig. 3.3 and Fig. 3.4 the energy consumption is decreasing very fast when σ is between 0 to 100. The slope will then decrease when σ goes larger than 100. Based on this observation, it will be easy to achieve reasonable energy savings by slightly adjusting the traffic allocation strategies (with a slightly increased σ) in real-world cloud systems. To further verify our conclusion, we also applied the traffic generated by Apache web server to validate our observations. In this context, We used Apache benchmark suite to actively tuning the number of connections and the traffic load on each VM. The results are also presented in Fig. 3.3 and 3.4, which remain consistent with the previous Iperf experiments. We also present the experiment results with 8 active VMs in Fig. 3.5 and Fig. 3.6, which show similar results as those in the smaller-scale experiments and hence is the observation: increasing skewness of the traffic leads to energy savings.

3.2.3 Explanations

Based on the above observations, it is thus reasonable to believe that the observed energy consumption variations in these experiments are due to the cross-VM interference. The designs of typical PVM and HVM platforms introduce virtual switches to bridging and routing the network traffic [54] [5]. A virtual switch, which is implemented by software, has the same functionality as a physical switch. On one side of the virtual switch are ports that connect to virtual NICs, and on the other side are connections to physical NICs. In the default design of Xen and KVM, the virtual switch is implemented with a Linux bridge [54]. For example in KVM, when a VM sends out network traffic, the packets inside the VM are first handled by the VM's kernel process, then a software interrupt will be sent to the hypervisor process. The hypervisor process is scheduled to copy the packet from the VM's memory space into the host's memory space. The hypervisor process then notifies the host kernel process to collect the packet and eventually the virtual switch gathers all packets from the virtual NIC's backend and sends to the physical NICs. In a nutshell, the fully balanced traffic makes the hypervisor to serve VMs more frequently. Once receiving the software interrupt, the hypervisor process is scheduled to run on one specific CPU core to handle the traffic, which meanwhile causes such interference that the original VM running on this core is blocked and re-scheduled. Such a great number of context switches put extra load on the CPU scheduling.

As an additional reference, Table 3.3 presents the traffic combination information and CPU utilization in Xen. Table 3.4 shows the similar profiling results captured in KVM. In the Xen environment, we can see that the increased CPU utilization of Domain-U sum is approximately 25%, while the major increase, a growth of 42% can be witnessed in Domain-0. As we have discussed, Domain-0 controls the physical NIC drivers and the virtual switches, as well as handles the I/O operations. Therefore, the overhead generated in Domain-0 greatly contributes to the total CPU utilization increase and unavoidably leads

Table 3.3: CPU profiling under different traffic combinations (Xen)

VM traffic combinations (5 VMs)	**Traffic deviation**	**Domain-0 CPU utilization**	**Domain-U CPU utilization sum**
(180, 180, 180, 180, 180) Mbps	0	$(83 \pm 1.1)\%$	$(100 \pm 1.5)\%$
(150, 150, 200, 200, 200) Mbps	27.39	$(80 \pm 0.5)\%)$	$(95 \pm 1.3)\%$
(100, 100, 200, 200, 300) Mbps	83.67	$(78.6 \pm 0.8)\%$	$(94 \pm 1.2)\%$
(50, 50, 200, 300, 300) Mbps	125.50	$(70.6 \pm 0.5)\%$	$(90 \pm 0.7)\%$
(50, 50, 50, 50, 700) Mbps	290.69	$(57 \pm 0.3)\%$	$(83 \pm 0.5)\%$
(10, 10, 10, 10, 860) Mbps	380.13	$(48.9 \pm 1.1)\%$	$(80 \pm 0.8)\%$
(0.5, 0.5, 0.5, 0.5, 898) Mbps	401.37	$(41 \pm 1.3)\%$	$(75 \pm 1.0)\%$

Table 3.4: CPU profiling under different traffic combinations (KVM)

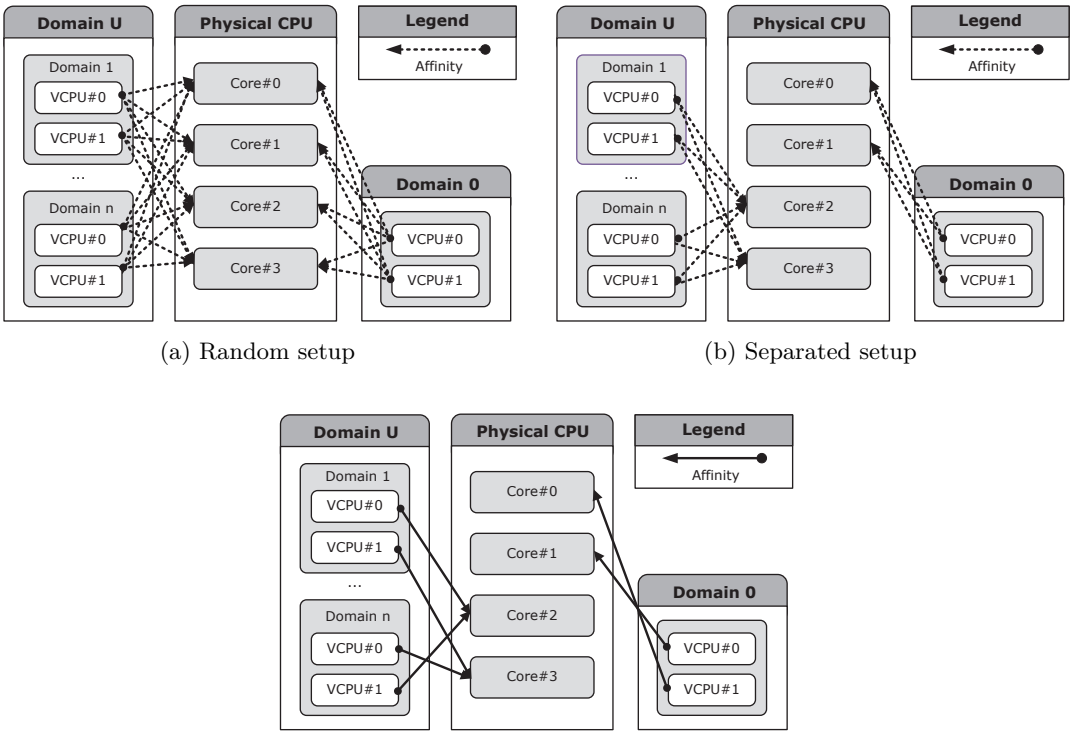
VM traffic combinations (5 VMs)	**Traffic deviation**	**CPU cycles per sec**	**Context switches per sec**
(180, 180, 180, 180, 180) Mbps	0	$(2.71 \pm 0.43) \times 10^9$	$(221.01 \pm 2.28) \times 10^3$
(150, 150, 200, 200, 200) Mbps	27.39	$(2.53 \pm 0.22) \times 10^9$	$(203.19 \pm 1.32 \times 10^3$
(100, 100, 200, 200, 300) Mbps	83.67	$(2.38 \pm 0.27) \times 10^9$	$(186.7 \pm 1.84) \times 10^3$
(50, 50, 200, 300, 300) Mbps	125.50	$(2.18 \pm 0.12) \times 10^9$	$(154.73 \pm 2.23) \times 10^3$
(50, 50, 50, 50, 700) Mbps	290.69	$(1.74 \pm 0.25) \times 10^9$	$(124.28 \pm 1.59) \times 10^3$
(10, 10, 10, 10, 860) Mbps	380.13	$(1.38 \pm 0.22) \times 10^9$	$(91.44 \pm 1.57) \times 10^3$
(0.5, 0.5, 0.5, 0.5, 898) Mbps	401.37	$(1.29 \pm 0.23) \times 10^9$	$(81.91 \pm 2.32) \times 10^3$

to higher energy consumption. Similarly in KVM, an extra of 1.42×10^9 cycles are consumed by the cross-VM interference. The context switching overhead is almost tripled.

3.3 Energy Variation from Virtual CPU Affinity

To further validate our conjecture of the cross-VM interference, we examined another factor in this section that could possibly lead to the energy variations when handling network traffic, that is, *virtual CPU affinity*.

It is known that virtual CPU affinity, or vCPU pinning enables the binding of a VM to a CPU core or a set of CPU cores, so that the VM will be scheduled to execute only on the designated CPU core or CPU cores rather than an arbitrary one. Each VM waiting to be scheduled in the scheduling queue has a tag indicating its kin CPU. At the time of resource reallocation, each VM is more likely to be allocated to its kin CPU core. Scheduling one VM to execute on one specific CPU core can result in an efficient CPU usage by reducing performance-degrading situations such as cache misses. Therefore, virtual CPU affinity can also have an impact on the energy consumption variations. As a concrete example, in Fig. 3.7, we provide three common virtual CPU affinity conditions in a machine with 4-core



(a) Random setup

(b) Separated setup

(c) Pinned setup

Figure 3.7: Different CPU affinity conditions

CPU in the Xen environment. Fig. 3.7 (a) is the default setup adopted in Xen, that is, each virtual CPU can run on any physical CPU cores. Therefore, we define it as *random setup*. This setup has an advantage that it achieves higher resource utilization, while the problem is that this setting can cause a bottleneck that Domain-0 might occasionally have to compete with Domain-*U* in terms of CPU scheduling. According to a previous study [68], Fig. 3.7 (b) is a setup resembling what Amazon EC2 small instance uses. Domain-0 can run on two CPU cores exclusively, and other user domains will share the remaining two CPU cores. We define it as *separated setup* in this work. Unlike the Amazon EC2 configurations, we did not set a hard cap on the virtual CPU running time. In fact, researchers observe that setting such cap will certainly have impairments on networking performance [60] [55]. To avoid this issue, we only consider pinning the virtual CPU to physical CPU core. Note that *separated setup* simply resolves the contention in between Domain-0 and Domain-*U*, however, the CPU utilization could be lower than the random setup configuration. Fig. 3.7 (c) is a case that each virtual CPU is pinned on one specific core. We therefore define it as *pinned setup*. This configuration also guarantees that no contention exists between Domain-0 and Domain-*U*.

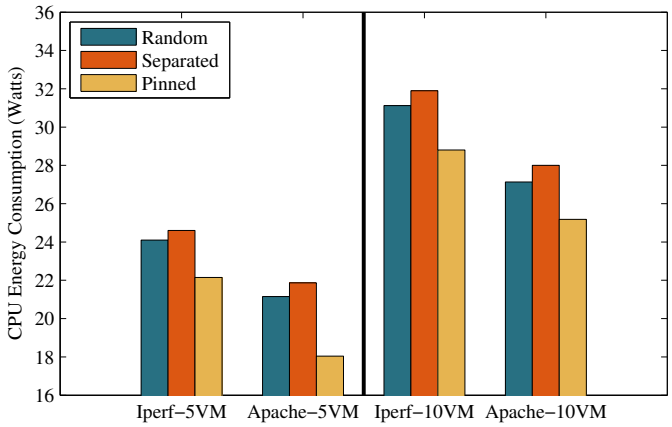


Figure 3.8: CPU energy measurements with different virtual CPU affinity conditions (Xen)

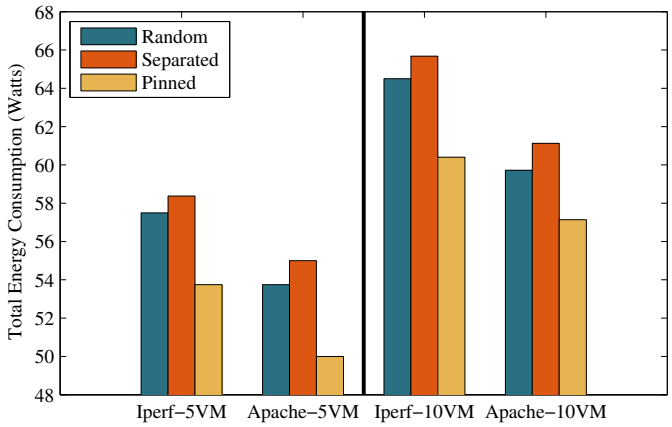


Figure 3.9: Server energy measurements with different virtual CPU affinity conditions (Xen)

To carry on our experiments, we used the built-in command `vcpu-set` in Xen to adjust the virtual CPU affinity. As for the KVM environment, since KVM does not utilize the concept of Domain-0, we adjusted the CPU affinity of each VM using the `taskset` tools. The hypervisor-related processes in the KVM environment are allowed to be scheduled on any available cores. Considering the capacity of the testbed machine, we set each VM to have one virtual CPU and performed the following experiments with 5 and 8 VMs on the testbed machine.

Observation 4. The energy consumption varies with different virtual CPU affinity settings. Pinning virtual CPU can reduce the energy overhead. However, if not handled properly, the pinning strategy can also lead to unbalanced core utilization, causing energy hot spots.

In Fig. 3.8 and Fig. 3.9, we present the measurement results in the Xen environment. The experiment details are described as follows: In the first test scenario, we let 5 co-located VMs send at the same rate (180 Mbps) with the Iperf command. In the second scenario, we let the Apache web server running inside the VMs fulfill the file downloading requests. The traffic sending rate on each VM is also 180 Mbps.

While performing the experiments, we adjusted the underlying virtual CPU affinity to be exactly the same as the three cases shown in Fig. 3.7. We also carried out the experiments on 10 co-located VMs, with each one sending network traffic at 90 Mbps. By comparing the energy consumption of pinned setup with others in Fig. 3.8 and Fig. 3.9, we can see that the CPU energy consumption can be reduced from 21.87W to 18.04W (shown in the second bar set in Fig. 3.8), that is, a 17.5% energy reduction. The pinned setup can always achieve lower energy overhead since each virtual CPU is limited to run on one specific physical core, which essentially reduces cache miss ratio and inter-core scheduling cost.

We also have an interesting observation that the separated setup adopted by Amazon [68] increases the energy usage under these scenarios. The key reason is that, two cores (core#0 and core#1) are taken over by Domain-0 in this setup, which leaves the other two cores (core#2 and core#3) shared by the five VMs. In this case, the network traffic consumes a large number of CPU cycles, and the VMs are frequently scheduled in and out. This in turn makes core#2 and core#3 become energy hot spots. Although there are enough time slices left on core#0 and core#1, those VMs have to compete with each other to run on core#2 and core#3.

We performed similar experiments in the KVM environment with the similar random, separated and pinned setups. In the KVM environment, random, separated, and pinned setups imply that each virtual CPU can be scheduled on 4, 2, and 1 physical cores, respectively. The experiment results are presented in Fig. 3.10 and Fig. 3.11. In this case, since each VM only has one virtual CPU, and hypervisor related processes can be scheduled on any available cores, we can observe that the energy consumption is monotonically reducing with decreased freedom on the virtual CPU scheduling.

3.4 Further Discussion

Besides the traffic allocation strategies and the vCPU affinity conditions, we also examined other factors that have the potential to affect the energy consumption. In this section, we provide further discussions to validate the observations.

It is known that the underlying servers in cloud data centers are often embedded with multiple NICs. For instance, a typical mid-range IBM server is attached with one or two physical NICs, while high-end servers could have even more NICs, e.g., IBM's BladeCenter E series[1] can have fourteen NICs equipped. Meanwhile, the VMs that are co-located in these

[1] `http://www.redbooks.ibm.com/redbooks/pdfs/sg247523.pdf`

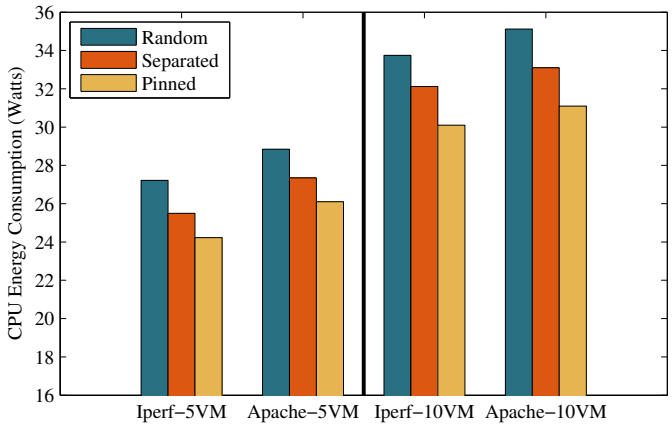


Figure 3.10: CPU energy measurements with different virtual CPU affinity conditions (KVM)

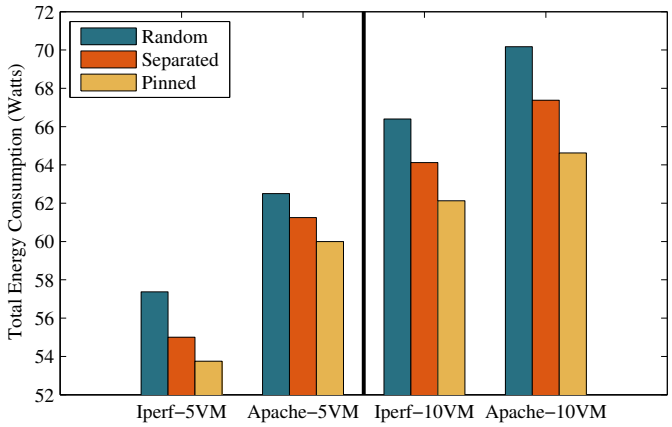


Figure 3.11: Server energy measurements with different virtual CPU affinity conditions (KVM)

data centers can also support multiple virtual NIC configurations. For example, Amazon's Virtual Private Cloud service allows the cloud users to dynamically attach/detach multiple virtual NICs on their VMs. The co-located VMs can share one virtual bridge or be attached to different virtual bridges. It is also possible to bind two (or more) physical NICs together to achieve link aggregation. Such bonding interface is widely applied to enable fail-over guarantee as well as load balancing for the cloud platforms [29].

While different network bridging schemes are seemingly complicating the internal path of traffic, our measurements indicate that it can only cause negligible energy consumption variations. Fig. 3.12 and Fig. 3.13 compare the CPU utilization when two co-located VMs are configured using different network bridging schemes. In this experiment, the VMs

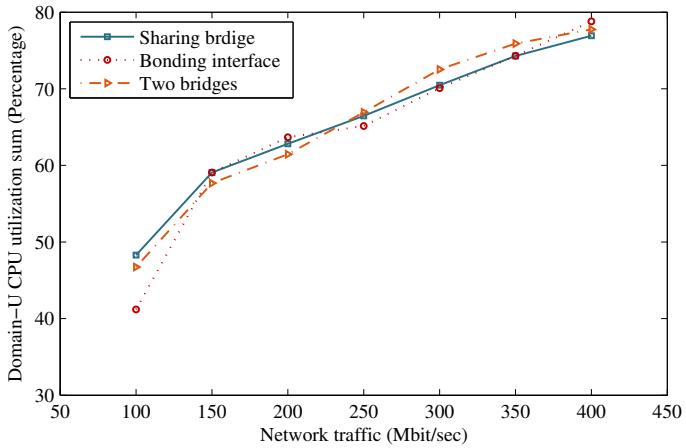


Figure 3.12: Domain-U CPU utilization sum (Xen)

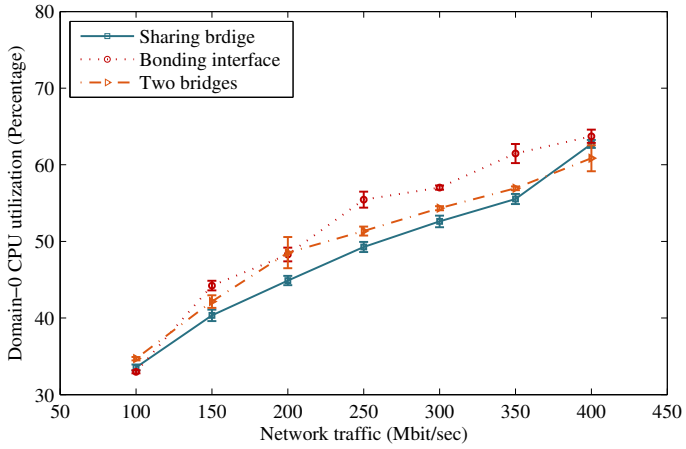


Figure 3.13: Domain-0 CPU utilization (Xen)

generated an exactly same amount of traffic[2]. As we can see in Fig. 3.12, for Domain-U, different network configurations hardly have influence on the CPU utilization, since the variation range is under 3%. In Fig. 3.13, however, we found that the bonding interface case increases the CPU utilization by up to 5% when compared with the sharing bridge case. We also noticed that when using two virtual bridges, the CPU utilization also has a slight increase. To summarize, the different bridging schemes can only introduce a 8% difference on the CPU utilization, which only leads to a 0.4W difference on the energy consumption.

[2]Based on our measurements in Section 3.2.2, this configuration can maximize the traffic interference.

Chapter 4

Performance and Energy Efficiency Issues of Hybrid Workloads

4.1 Motivation

Chapter 3 mainly focuses on the energy consumption variations when co-located VMs are all handling network transactions. We reveal that the energy variations come from the cross-VM interference. In this chapter, we further examine the performance and energy consumption variations brought by the self-interference inside a single VM.

Previous research has confirmed the existence of self-interference, and quantified the degradation of network performance with benchmark tools [55]. Different from the previously discussed cross-VM interference, self-interference happens within one specific VM. When the co-located computation processes aggressively use the CPU resources, the I/O handling process could be interfered or even starved inside the VM.

To further understand the self-interference from real-world applications, we have conducted measurements on two sets of typical hybrid workloads in cloud environments: 1) video transcoding and streaming, and 2) file compression and delivery. To provide a systematic investigation, the measurement results include network and computation performance, as well as energy efficiency metrics. Despite the great effort towards highly modularized and layered design for cloud applications, such hybrid workloads involving both network transmission and real-time computation still widely exist in cloud environments, which can hardly be decoupled from the cloud application itself.

4.1.1 Measurement and Observations

The first experiment is conducted on a VM which serves as a video streaming server and also a transcoder with real-world applications `LIVE555` and `FFmpeg`. The VM is allocated with 2 vCPU, 4GB RAM, running in the KVM environment and fulfilling on-demand transcoding and streaming requests from 200 clients. We first present our experiment results on

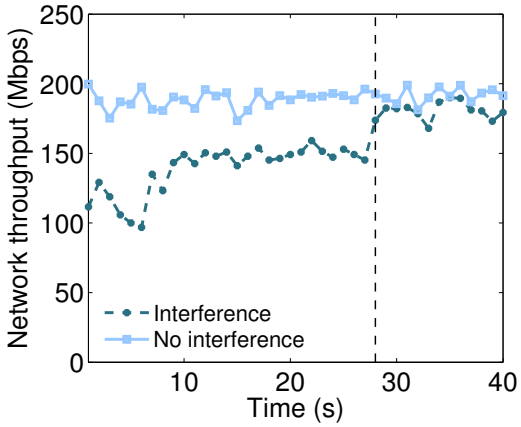


Figure 4.1: Network throughput when experiencing interference

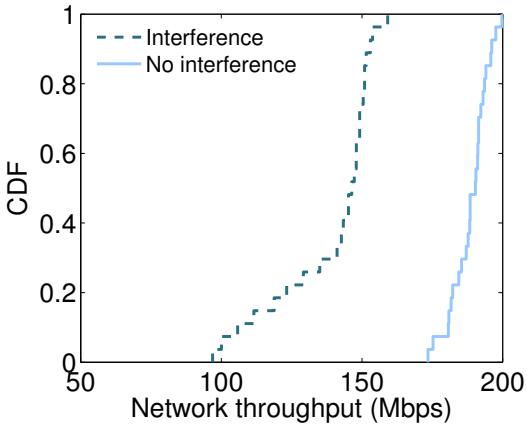


Figure 4.2: CDF of network throughput

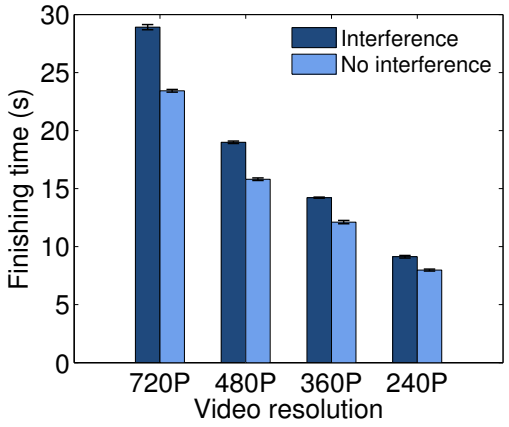


Figure 4.3: Comparison of transcoding performance

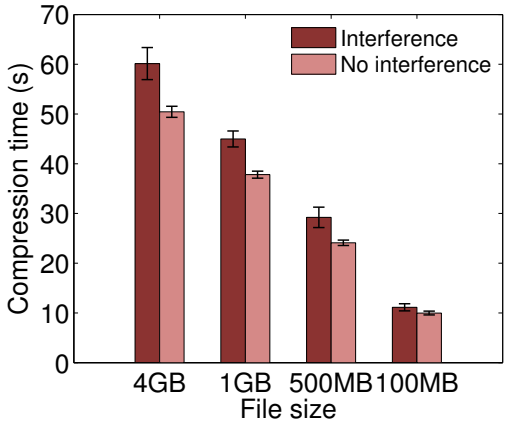


Figure 4.4: Comparison of file compression performance

networking performance in Fig. 4.1. As the experiment began, the throughput of this VM is relatively stable around 190 Mbps when it is dedicated to handling the streaming traffic. Unfortunately, when we started to add concurrent transcoding tasks to the VM, the throughput becomes as low as 115 Mbps, not to mention the high variations. Before the transcoding task finished at 28s, the clients experienced nearly 40% throughput degradation. We also plot the CDF of the network throughput in Fig. 4.2 to present the performance loss. On the other hand, for the concurrent transcoding tasks, the processing is also delayed. Our experiments show that the transcoding task completion time increased by $10\% - 18\%$ for different resolution levels, which is shown in Fig. 4.3.

Our measurement results of the second set of hybrid workloads are presented in Fig. 4.4 and Fig. 4.5, which is captured while simultaneously running file compression and delivery

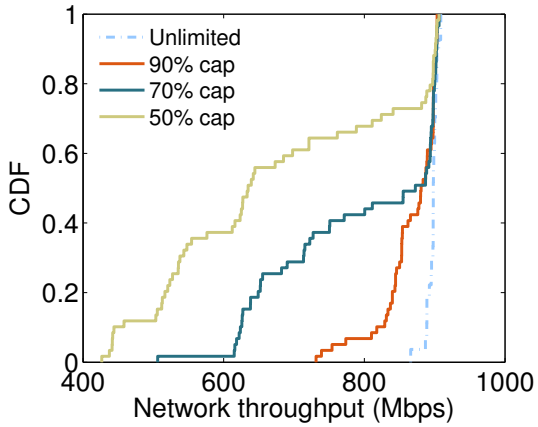


Figure 4.5: Impact from scheduling policy

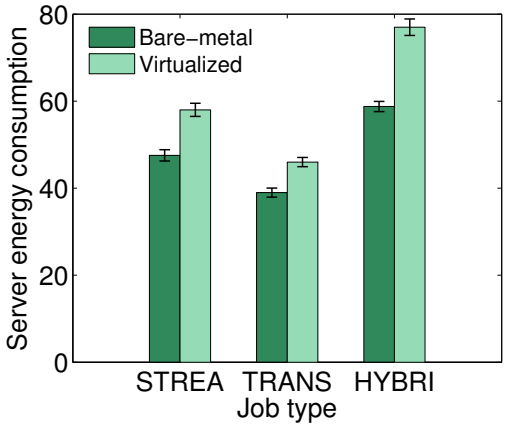


Figure 4.6: Comparison of energy consumption

tasks with `pigz` and `Lighttpd web server`. In this case, the VM serves as a file entrepot for compressing and transferring data blocks. To take a step further, we maintained a hard limit on the CPU usage of the VM. Such a non-work conserving scheduling policy[1] has often been adopted in real-world commercial clouds to achieve better isolation between VMs. The experiments show similar results as the previous set, with increased task completion time by $5\% - 19\%$. Even worse, the CDF of the VM's network throughput in Fig. 4.5 shows that, as we set a lower cap on the CPU usage of the VM, the average network throughput significantly decreases, together with a perpetual network instability.

To summarize, our measurement results indicate that adopting such a scheduling policy further deteriorates the network performance with more severe self-interference. As a matter of fact, many cloud applications may involve even more complex hybrid workflows. It is reasonable to believe that there is prevalent existence of the self-interference in the cloud context. The root cause of such self-interference is a combined effect of the I/O subsystem design (including network I/O and disk I/O) and the scheduling policy in virtualized environments. This significantly deteriorates the overall performance when the hybrid workloads involve complicated I/O and computation operations. We take the KVM environment as an example and briefly introduce its I/O architecture.

A closer look into KVM's network architecture is given in Fig. 4.7, which lists the key steps involved in delivering network packets to a VM. A state-of-the-art solution, vhost-net [3], is a high-performance virtio-net emulation that takes advantage of advanced zero-copy and interrupt handling features. Despite the reduced data copies, we can see that the network packets still have to traverse through multiple protection layers before eventually reaching the end application. Furthermore, when hybrid workloads are introduced on the

[1]The scheduler may leave the physical CPU idle despite that there is possibly one or more VMs are ready to be scheduled.

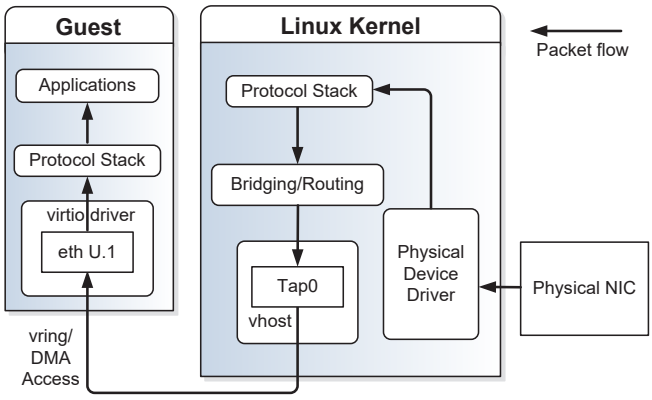


Figure 4.7: KVM network I/O subsystem

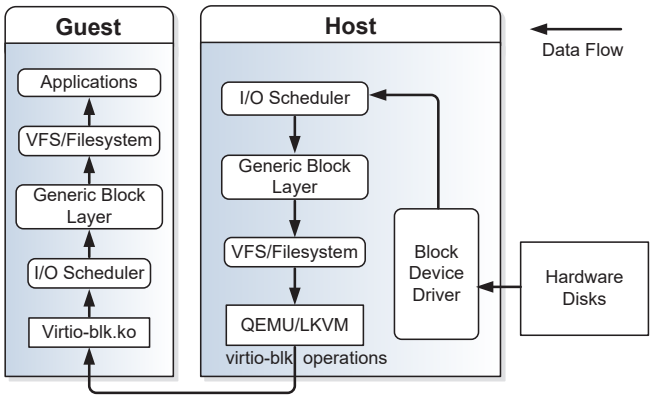


Figure 4.8: KVM disk I/O subsystem

VM, the computation tasks may consume the entire CPU quotas required for handling the network traffic, and vice versa. If a VM expends the entire CPU time slice, it will probably go to the end of the scheduler's queue and have to wait a considerable amount of time before reaching the front again. If the system administrator uses a non-work conserving scheduling policy on the VM, the VM will not be scheduled again until the next scheduler accounting period, even if there are available CPU resources on the physical system.

Apart from the network I/O operations, the hybrid workloads also involve a large number of disk I/O operations. For example, in the aforementioned transcoding tasks, it also involves reading the raw files from the virtual disk space. In Fig. 4.8, we present the disk I/O workflow in the KVM environment. In the native KVM design, a disk read operation involves these following steps: an application inside the guest VM uses a generic virtual file system (VFS) interface to issue disk read requests. The guest VM first fills in request descriptors, then writes to virtio-blk virtqueue and notifies the registers. Afterwards, the QEMU process issues I/O requests on behalf of the guest VM, then the QEMU process fills in

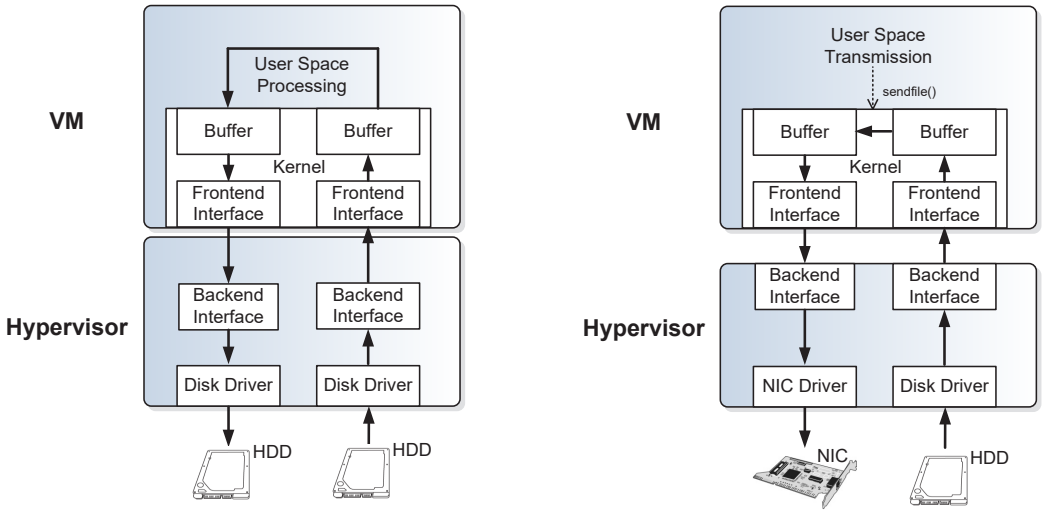
request footer and injects completion interrupts. The guest VM then receives the interrupts and executes the I/O handler. Eventually, the application reads data from the kernel buffer. Similarly as the network I/O subsystem, the disk I/O data also need to traverse through multiple protection boundaries, which also incurs a great amount of virtualization overhead.

Meanwhile, the ceaseless state changing of a VM (e.g., from the running state to the block state), together with the context switching between computation and I/O handling inside the VM, can further increase the energy consumption of the underlying server. As a concrete example, Fig. 4.6 shows that, when we run the stand-alone streaming task or the stand-alone transcoding task inside the VM, the energy consumption of the physical server increase by 22% and 18%, respectively. However, when we run the hybrid workloads inside the VM, the energy consumption increases by 31%. The results are all compared with the bare-metal case. Such a noticeable increase calls for a revisit on the current virtualization architecture design.

4.1.2 Existing Approaches and Opportunities

Recent works on improving I/O performance in virtualized environments can be broadly classified into four categories: 1) reducing virtualized device overhead [26, 30, 32]; 2) optimizing network I/O path [24, 25]; 3) providing middleware support at the hypervisor layer [27, 66]; and 4) customizing the scheduling policy for I/O intensive VMs [55, 64, 65].

SR-IOV enabled NICs [19] can handle network traffic without the involvement of hypervisor. Unfortunately, such devices do not allow hypervisors to inspect and control VMs' I/O activities [11], e.g., performing security and QoS regulations. Another well-known issue is that a VM with passthrough devices is not compatible with live migration. Using exit-less interrupt delivery mechanisms [26,30] can effectively reduce the interrupt handling overhead of virtual devices. In particular, I/O events are passed to a VM without exiting the hypervisor. However, such methods cannot eliminate the self-interference between the computation and I/O processing inside the VM. The negative effect of VM consolidation also dominates the I/O performance in virtualized systems. To mitigate such problems, vPro [24] proposes to offload TCP protocol functionality, including congestion control and acknowledgement to the hypervisor. The solution only benefits *small* TCP flows and does not work for UDP protocol. vPipe [25] discusses the opportunity of enabling piped I/O at the hypervisor layer for static data transfer. vRead [66] and VAMOS [27] represents the effort on providing I/O middleware support at the hypervisor level. These works target on specific workloads in cloud environments, e.g., Hadoop workloads and MySQL database workloads. The scheduling latency for network I/O intensive VM is also critical in virtualized environments, since VM scheduling can bring noticeable delays to the network I/O processing. Recent studies [55, 65] suggest reducing CPU time slice for I/O intensive VM. Such approaches enable VMs to get scheduled more often so as to improve the I/O throughput. vTurbo [64] takes a step forward to offload VMs' I/O processing to a dedicated core



(a) Processing (b) Transmission

Figure 4.9: Native workflow for hybrid workloads

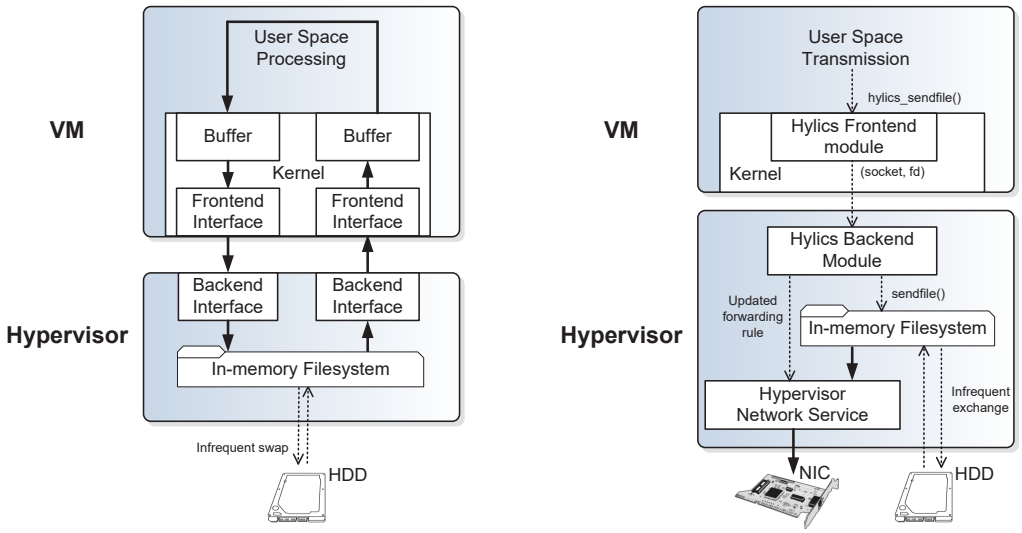
with extremely small time slices. In summary, these works focus on modifying the hypervisor scheduler to reduce the scheduling delays. However, this type of approaches brings increased VM state changes, as well as complicated CPU resource allocation strategies.

We can see that the previous approaches mainly focus on *I/O stack* optimization, without evaluations on the overall performance of hybrid workloads with concurrent CPU and I/O operations. In this work, we target at jointly optimizing I/O and computation performance via enabling cloud applications to better cooperate with the virtualization layer. Our solution rests on the facts that,

(1) The network I/O data transfer is essentially expensive in between VM and the hypervisor layer. As mentioned before, the complex network encapsulation and configuration in virtualized environments is one of the key bottlenecks to be revisited. What has been largely missed in mainstream virtualization technologies is the opportunities for raising the level of provisioned interface from physical devices to higher-level services. In this thesis, we discuss the opportunities to provide guest VM with the entire networking service at the hypervisor level.

(2) Hypervisor-level network I/O operations can achieve nearly bare-metal performance as well as high energy efficiency. In a general virtualized system, since the hypervisor has privileges to use its native device driver to access I/O devices directly. It is therefore reasonable to take advantage of hypervisor-level network I/O operations to shorten the network packet traverse path which originally begins or ends inside a VM.

(3) For the computation part in hybrid workloads, instead of performing read and write operations on the file system hosted in virtual disk, it will be generally more efficient to



(a) Processing (b) Transmission

Figure 4.10: Hylics workflow for hybrid workloads

operate on the file system hosted in the memory space. Therein, instead of providing a virtual disk device to the VM, we also seek to provide a lightweight in-memory file system interface for the hybrid workloads.

Based on these observations, we propose Hylics, an enhanced virtualization framework for hybrid workloads in cloud environments.

4.2 Framework Design

In this section, we present the complete design of the Hylics framework. To mitigate the self-interference inside VM and jointly optimize I/O and computation performance, the principle is to decouple network I/O operations and computation operations, as well as shorten the data traverse path for both of the two parts. In particular, Hylics provides an in-memory file system for storing application data and shifts VM's network operations to the hypervisor layer. Such network operation offloading borrows the idea of separating the control plane and data plane of network transmission. In the Hylics architecture, data transfer only takes place at the hypervisor layer.

To provide an intuition of the Hylics design, in Fig. 4.9 and Fig. 4.10, we present a comparison between the Hylics abstraction and the native virtualization abstraction. For the data processing part, we can see that, in the current virtualization architecture design, a pair of frontend and backend virtual disk interface is designed to transfer data in between a VM and its hypervisor, which incurs a great amount of virtualization overhead. The data first needs to be loaded from the hardware disk drive to the hypervisor memory space. Such

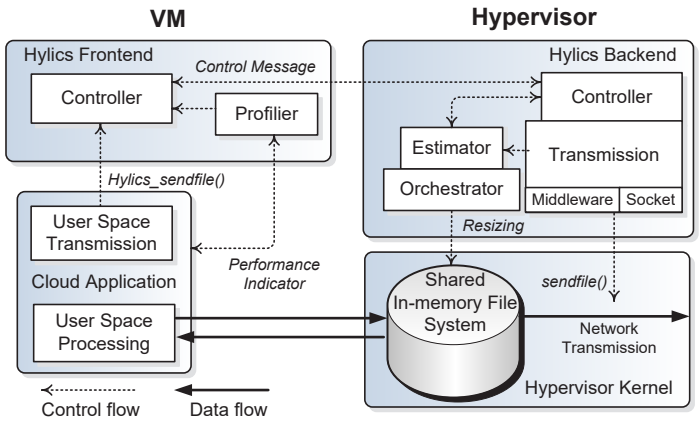


Figure 4.11: System architecture

overhead is eliminated with the in-memory file system provision in Hylics. For the network data transmission part, instead of transferring network buffered data in and out, the Hylics design directly enables VM to send data from the in-memory file system, with the network service provisioned at the hypervisor layer.

4.2.1 Overview

As shown in Fig. 4.11, this design consists of the following components:

Shared in-memory file system: The shared in-memory file system is a key component in the Hylics architecture, which stores the intermediate data for cloud applications. In our design, it is allocated in the virtual memory maintained by the host kernel, which can dynamically grow and shrink to accommodate the files it contains. Note that the maximum size limits of the space can be adjusted on-the-fly; this feature enables us to provide an SLO-aware memory usage control scheme. We will further present the detailed shared in-memory file system design in Section 4.3.1.

Cloud applications: Cloud applications that involve data processing and network transmission can utilize the in-memory file system provided by the Hylics architecture. This is enabled by using the VFS interface provided by the Linux kernel. The purpose of a VFS interface is to allow applications to access different types of concrete file systems in a uniform way. Note that the processing logic of cloud applications remains unchanged in this design, which is beneficial for the application developers. As for the network transmission part, cloud applications are provided with a modified version of system calls to issue network transmission request. In this fashion, the actual data transfer is originated from the in-memory file system with the offloaded network service at the hypervisor level.

Hylics frontend module: Hylics frontend module first provides programming interfaces for dealing with the network transmissions of user applications. To cope with different

cloud applications, we propose two different schemes for offloading network transmission to the hypervisor layer, either by offloading network middleware modules, or by offloading valid socket copies. The Hylics frontend module communicates with the cloud applications and serves as an agent to retrieve the targeted file descriptor, the destination information, and the socket level information. It will then pass the information to the backend module to perform the actual data transfer. Meanwhile, it also closely measures the performance metric of cloud applications to provide raw profiling results for further performance optimization.

Hylics backend module: Hylics backend module is responsible for VM's network transmission, and runs in the user space of the hypervisor layer. With the middleware or socket operations offloaded to the hypervisor layer, the backend module starts threads to send the data from the in-memory file system. Both of the two schemes utilize hypervisor-layer network stack to transmit the application data. To ensure security, the module runs inside a separated zone in the host space. We will further elaborate the design in Section 4.3.2.

System profiler and estimator: The profiler in the Hylics architecture is a collection of monitoring tools to get the application performance indicators in a real-time fashion, e.g., task response time, file system usage information and detailed logs. We collect such data for the estimator to analyze and identify the system model. In particular, we leverage a widely-used queueing model predictor to achieve a fast system model approximation. The identified system model is further used for optimizing the memory usage. The detailed estimator design is discussed in Section 4.4.

Hylics controller and orchestrator: We design the controller to coordinate the computation progress and the network transmission. Meanwhile, it also controls the file system size allocated for each VM. The controller receives the feedbacks from the system profiler and estimator. It also accepts users' SLO indicator as an input. Based on the information, it makes optimization decisions to enhance the resource usage. In consideration of computation complexity, we design a simple yet effective online feedback control loop. The design is further explained in Section 4.4. The orchestrator implements and executes the controller's optimization decisions, i.e., the optimized in-memory file system space assignment of each VM. The decisions are executed by the in-memory file system resizing mechanism with standard kernel interfaces.

4.2.2 Case Study

We use the hybrid workloads "transcoding and streaming" as a concrete example to illustrate the entire workflow in the Hylics architecture. The raw video segment which originally stores in the virtual disk space is now lifted and stores in the shared in-memory file system. The transcoding task fetches and processes the video file from the shared in-memory file system. This is enabled with a paravirtualized file system interface via offloaded Linux VFS API. Due to the general abstraction of the Linux VFS API, transcoding task still can

use the same system call on handling these file I/O operations. The post-processed video segment is also stored in the shared file system for the on-demand transmission. To fulfill the concurrent streaming request, the VM can use the network middleware module at the hypervisor layer for video streaming. The sending result is then passed back to the VM via a guest-host communication channel. Such communications only involve control message exchange between the VM and its hypervisor, causing minimum overhead as compared to transferring the actual video data. The VM also has another option to offload socket level operations to the hypervisor layer. In this scenario, the VM first needs to set up connections with valid sockets and then sends out video files with modified programming interfaces provided by the Hylics architecture. While running the hybrid workloads, the profiler module continuously collects the runtime information and keeps track of the performance indicators. In this case, we use transcoding response time and sending throughput as the default indicators. The Hylics controller gathers such information from both the guest VM and its hypervisor and then the estimator calculates the optimized memory space required by the VM. Afterwards, the orchestration is performed to adjust the size of the in-memory file system.

4.3 System Implementation

4.3.1 Sharing In-memory File System

We delve into the implementation details of the shared in-memory file system in this section. Typically, a file system is used to define how file data is stored and retrieved, which includes two types of information—the data blocks residing on the file system, and the control information used to maintain the state of the file system. An in-memory file system uses resources and structures of memory subsystem, and supports UNIX file semantics meanwhile is fully compatible with other common file systems. Hosting a file system inside the memory space provides better performance for file reading and writing. This feature is utilized by our Hylics design so that the general file access of cloud applications causes a memory-to-memory copy of data, no I/O requests for file control updates are generated. Meanwhile, since file system attributes are stored once in memory, no additional I/O requests are needed for file system maintenance. In the Hylics architecture, rather than directly using dedicated physical memory, we choose to utilize the operating system page cache maintained by the kernel for storing file data. Such an implementation generally provides increased read and write performance with no adverse effects on the system compatibility. This is because we can take advantage of the native resource management policies in the Linux kernel. Another important aspect is the space management of the in-memory file system, instead of allocating a fixed amount of memory for exclusive use, using system page cache space enables a dynamic resizing mechanism depending on use, which allows the Hylics architecture to adjust its memory usage on-the-fly.

To meet all the design criteria, we employed tmpfs file system [58] during the implementation. When we initialize the Hylics architecture, a VFS structure is allocated, initialized and added to the kernel's list of mounted file systems. A tmpfs-specific mount routine is then called, which allocates and initializes a tmpfs mount structure, and then allocates the root directory for the file system. It allows us to use anonymous memory in the page cache to store and maintain file data. Since the kernel does not differentiate tmpfs file data from other page cache uses, the stored file blocks can be written to swap space. This could happen when the system is in an urgent need of memory usage. Control information is maintained in physical memory allocated from kernel heap. The file data is then accessed through only one level of indirection provided by the virtual memory system.

Besides the in-memory file system provision, another critical issue is to efficiently share the file system in between a VM and its hypervisor. In our design, we enable a paravirtualized file system driver based on the virtio framework [52]. This interface presents some unique advantages over the traditional virtual block device. By paravirtualizing a file system interface, we further avoid a layer of indirection in converting VM's file system operations into block device operations and then again into host file system operations. A paravirtualized interface provides a preconnected and isolated channel between a VM and the hypervisor, which incurs none of the overhead of arbitrary and unnecessary encapsulation when going over a network stack incurs.

Our implementation is to leverage a lightweight distributed file system protocol directly on top of a paravirtualized transport [34]. We then provide a virtualization-specific transport interface for it through the virtio framework. The shared file system space is then ported as a local file system on the guest VM. To ensure system security, we also enable standard Linux access control mechanisms (e.g., SELinux, chroot, seccomp) to limit QEMU process to accessing specific resources. In this context, each QEMU process is restricted to only access the part of the shared file system space that is relevant to the VM it runs.

4.3.2 Offloading Network Operations

To complete the Hylics design, another critical issue is to offload VM's network operations to the hypervisor. One possible solution is directly using the network stack at the hypervisor layer with offloaded network middleware modules. Such modules run at the hypervisor level and cooperate with the hypervisor resources. The execution results, interpreted as control messages, can be sent back to the VM via an inter-domain communication channel, e.g., virtio-vsock [4]. Such offloading is favorable for those workloads whose functional logic is easy to further decouple. They can use a client side in the VM and a server side running at the hypervisor level. The network I/O middleware modules offloaded at the hypervisor level can effectively reduce the number of VM-hypervisor context changes, as well as the device emulation overhead. However, running an additional module at the hypervisor level may potentially raise concerns about the complexity and security risks. We emphasize

that, with careful design, the modification is minimal. The network middleware module can be loaded as a user space process at the hypervisor layer, and runs in an isolated environment. To neutralize security vulnerabilities, the hypervisor can further restrict the privilege of the network middleware module. For instance, in the KVM environment, to defend against compromised running components, a QEMU process can use a plugin-isolation mechanism [23, 69].

Another possible solution is offloading TCP/UDP socket level operations. Therein, we also design interfaces for Hylics to pass the entire TCP/UDP processing functionality by establishing a socket copy in the hypervisor layer. Particularly, we first provide a variant of the system call `send()` or `sendfile()` for user applications. An application can either use `hylics_send()` or `hylics_sendfile()` for network transmission. The parameter includes socket descriptor (which is obtained by accepting clients' connection), a file descriptor (by locating the targeted file), file offsets and byte count. After getting parameters from cloud applications, the Hylics frontend module in the VM first converts this socket descriptor to a socket structure with necessary information for establishing the socket copy in the hypervisor layer. Such information includes the source and destination IP addresses, source and destination ports, etc. Meanwhile, the Hylics frontend also interprets the file descriptor as a file path inside the shared in-memory file system. As soon as these parameters are passed to the Hylics backend module, the backend module sets up the socket copy. The backend module then begins to fetch data from the in-memory file system and performs the actual data transfer. When the transfer is done, the number of bytes sent is returned to the cloud application. After the transmission begins, the original socket inside the VM is disabled and the traffic is redirected to the hypervisor layer by explicitly setting the packet forwarding policy.

4.4 Memory Usage Analysis and Enhancement

Hylics leverages the memory resource at the hypervisor layer to improve the overall performance of hybrid workloads. Therein, the space management of the shared in-memory file system is undoubtedly a key design issue. Intuitively, the static management or fixed in-memory file system size for each VM can provide isolation and fairness in between VMs. However, it may also result in either resource waste or VM performance degradation. Furthermore, the rigid management of memory usage also suffers from the inflexibility in the presence of memory intensive tasks. We argue that system administrators should be able to switch between static and dynamic strategies. Although almost all modern hypervisors implement memory overcommitment mechanisms such as ballooning, page sharing, and swapping; they lack policies to coordinate these mechanisms in order to minimize performance degradation for cloud applications in the VM. We then introduce an online approach to assign and adjust the utilized memory space among different VMs. By online, we mean

that the controller design achieves system identification and makes adjustment decision by processing pairs of input-output measurements sequentially, as opposed to offline identification methods[2]. In the cloud context, such online processing is important since the task size and arrival rate are highly dynamic and hence is the efficiency of the Hylics memory space usage. As a next step, we propose an *online self-adaptive control scheme* to meet users' SLO while keeping moderate memory usage.

4.4.1 Online Self-adaptive Control Scheme Design

To provide a robust control scheme, we combine queueing theory modeling and adaptive control together in this work. The reason why we need such a design is twofold. First, we learn from a large cloud provider-Google's trace analysis [50] that typical job inter-arrival time exhibits an exponential distribution. Meanwhile, queueing model is also widely applied in the cloud context to provide simplification on the system that has a bottleneck stage [12, 67]. Second, the adaptive feedback loop can build the residual error model and enhance the controller performance. It can reduce inaccuracies in the queueing model and handle the sudden change of hybrid workloads in a dynamic fashion. The combination of the queueing model predictor and the adaptive feedback control provides a better performance regulation under a wide range of workload conditions.

Our abstraction for each Hylics-enabled VM is an M/G/1 processor sharing queue (M/G/1/PS). With the network-related tasks offloaded to the hypervisor layer, we assume that the VM is now exclusively working on the computation tasks with the assistance of the shared in-memory file system. To begin the self-adaptive control scheme design, we first introduce notations in the queueing model. We denote by $R(x)$ the average response time of a computation task whose service time is x. According to the queueing model definition, the service time is an i.i.d random variable in an M/G/1/PS system, denoted by X, of which probability distribution function is $P(X)$, with an average $E[X]$. The load and arrival rate of the queue is denoted by ρ and λ, respectively. The average response time for all tasks on the VM, is then calculated by:

$$R = \int_0^\infty R(t)dP(t) = \frac{E[X]}{1-\rho} \tag{4.1}$$

Let the size of the in-memory file system space allocated for the VM be m. In this work, we assume that the average service time of a task is a variable subject to the file system size m. Then we have:

$$R(t) = \frac{E[X|m]}{1-\lambda(t)E[X|m]} \tag{4.2}$$

The goal of the adaptive control is to adjust the average response time of tasks $R(t)$ as close to the reference delay τ as possible. The reference delay τ simply indicates users' SLO.

[2]Offline identification methods process the data trace acquired over a certain time horizon at once.

Table 4.1: Summary of notations

Notations	Definitions
$R(x)$	average response time of a task
$P(x)$	PDF of service time
$E[X]$	average service time
ρ	the load of a queue
λ	the arrival rate of a queue
m	current allocated in-memory file system space
τ	reference delay
k	sequential number of a measurement pair
A, B, a, b, q	general control model parameters
n_A, n_B, d	control model orders
$y(k)$	k^{th} control output
$u(k)$	k^{th} control input
$\theta(k)$	k^{th} parameter vector
$\phi(k)$	k^{th} input-output pair
$F(k)$	k^{th} adaption gain matrix
I	identity matrix

Suppose the queueing model is accurate, then from Equation (4.2), we can then directly set the file system size m to get the ideal average service time, so that we can further get the steady response time to be exactly τ.

As the next step, we present the controller design of the adaptive feedback loop. In this context, the purpose of the adaptive control loop is to correct the "residual errors" of the response time ($\Delta\tau$) by dynamically tuning the adjustment of Hylics file system space (Δm). Considering the performance of the system, we apply direct adaptive control for its efficiency and simplicity. We then consider the system as a discrete time model with adjustable parameters estimated by a recursive least squares (RLS) estimator [7], which is an online version of the well-known least-square estimator. Such online parameter estimation can provide us with real-time feedback. The parameters are updated during each control interval to minimize the queueing model error. Then the control law is calculated by setting the adjustment of Hylics space (Δm) to diminish the residual errors ($\Delta\tau$).

To be more specifically, in order to construct the control law, the adaptive controller first needs to estimate the residual error model for the system whose parameters can be used in the controller. In the following discussion, we describe the scheme with a general model:

$$A(q^{-1})y(k) = q^{-d}B(q^{-1})u(k) \tag{4.3}$$

where:

$$\begin{aligned} A(q^{-1}) &= 1 + a_1 q^{-1} + ... + a_{n_A} q^{-n_A}, \\ B(q^{-1}) &= b_0 + b_1 q^{-1} + ... + b_{n_B} q^{-n_B}, b_0 \neq 0, \end{aligned} \tag{4.4}$$

Algorithm 1 Hylics file system size control algorithm

```
Initialize adaption gain matrix F = f_0 I and parameter vector θ
while control interval (ci) expires do
    Acquire average response time R(t) from profiler
    if ΔR ≥ R_thresh then
        Update λ(t) and queueing model
        Update m̂ by solving Equation (4.2)
        m ← m̂
    end if
    Update adaption gain matrix F by Equation (4.8)
    Update parameter vector θ by Equation (4.9)
    Acquire Δm by solving Equation (4.10)
    if Δm ≥ m_thresh then
        m ← m + Δm
        ci ← min{ci/2, ci_maxthresh}
    else
        ci ← max{ci * 2, ci_minthresh}
    end if
end while
```

and $y(k)$ is the control output, $u(k)$ is the control input (command). In the Hylics context, $y(k)$ corresponds to response time residual error $\Delta\tau(k)$. and $u(k)$ corresponds to memory space adjustment $\Delta m(k)$. Due to the digital implementation of the feedback controller, the effect of control command determined on time interval k can only be measured in interval $k+1$, then we set the delay order as $d = 1$. The model parameter of Equation (4.4) are estimated with the RLS estimator. Let

$$\theta(k) = [\theta_1(k), ..., \theta_{n_A}(k), \theta_{n_A+1}(k), \theta_{n_A+2}(k), ..., \theta_{n_A+n_B+1}(k)]^T, \tag{4.5}$$

and

$$\phi(k) = [y(k), y(k-1), ..., y(k-n_A+1), u(k), u(k-1), ..., u(k-n_B)]^T, \tag{4.6}$$

The target function of the RLS estimator is defined as

$$\min_{\hat{\theta}(k)} J(k) = \sum_{i=1}^{k} [y(i) - \hat{\theta}^T(k)\phi(i-1)]^2 \tag{4.7}$$

The term $\hat{\theta}^T(k)\phi(i-1)$ in Equation (4.7) corresponds to $\hat{y}[i|\hat{\theta}(k)]$. This is the prediction of the output at instant i ($i \leq k$) based on the parameter estimate at instant k, which is obtained by using k measurements. The RLS algorithm works by calculating the adaption gain matrix F and updating the model parameter θ:

$$F(k-1) = F(k-2) - [1 + \phi(k-1)^T F(k-2)\phi(k-1)]^{-1} F(k-2)\phi(k-1)\phi(k-1)^T F(k-2), \tag{4.8}$$

$$\theta(k) = \theta(k-1) + F(k-1)\phi(k-1)[y(k) - \phi(k-1)^T\theta(k-1)], \tag{4.9}$$

then the control law for the space adjustment is given by solving:

$$\phi(k)^T\theta(k) = y^*(k+1), \tag{4.10}$$

where $y^*(k+1)$ is the residual response time error at instant $k+1$. In our case, we are considering to let the "residual errors" to diminish, then we need to set $y^*(k+1) = 0$. The above algorithm begins with initial condition $F(-1) = f_0 I$ and $f_0 > 0$. As a summary, we list the key steps of the Hylics memory space control algorithm in Algorithm 1. In the algorithm, the control loop is executed whenever the control interval (ci) expires. The control interval is also adaptively set to avoid oscillations, which is shown in line 14 and line 16 in the algorithm.

4.4.2 Parameter Selection

The average service time $E[X|m]$ given specific m for different workloads used in the queueing model predictor is measured offline. To this end, we first initiated a lightweight workload on the VM. We then varied the utilized in-memory file system size and measured the response time. The measured average response time approximates the average service time since there is no queueing delay during the experiment. We conducted the measurement test multiple times (≥ 100) and then averaged the measured time to get $E[X|m]$. As the next step, we need to determine the model orders n_A and n_B to complete the adaptive controller design. The model orders n_A and n_B are also measured in an offline fashion. We used the following method to determine these parameters: we first disabled the adaptive feedback controller and collected offline data $\Delta\tau$ by using only the queueing model predictor and a white noise input of Δm. Under different combinations of n_A and n_B, we used a direct model identification method — a default least square estimator to get the corresponding model parameters, then we tested which θ acquired from these combinations has a good fitting performance. This means using the θ, a new data group of collected data pairs (Δm, $\Delta\tau$) produces high r^2 value [7]. r^2 denotes the percentage of variations that can be explained by the model. In the Hylics system design, we found the parameter choice $n_A = 1$ and $n_B = 0$ has the best fitting result.

4.5 Performance Evaluation

In this section, we conducted experiments to understand the performance of the proposed design. The criteria include file system read and write performance, computation performance, networking performance, and energy efficiency. First, we introduce the testbed configuration, together with the hardware and software environments.

4.5.1 Experiment Configuration

Our experiment applied a typical mid-range server equipped with an Intel's core $i5$ 2400 3.09GHz quad core CPU and 8GB 1333MHz DDR3 RAM. To understand the networking performance, we deployed another server with similar hardware configurations. These two servers are installed with Intel $i350t$ network interface cards, and the interfaces are connected by a Netgear GS724Tv3 switch with 1000 Mbps Ethernet link.

The operating system running on the host machine and the guest VM is Debian 3.18.9. We then configured KVM 1 : 1.1.2 on this testbed machine. The qemu-kvm version is 2.0.0. Based on the typical VM configurations in public clouds, we set the default number of accessible vCPU to be 3 for the guest VM. The VM is then equipped with 4GB RAM. We used `ifstat` to measure the network throughput with the probing interval set as 1 second. To collect the detailed system information, we captured the virtual CPU utilization using `top`, which is a standard resource monitoring tool integrated into the Linux distribution. Furthermore, we used the Linux hardware performance analysis tool `perf` to collect such system level statistics as CPU cycles and context switching information. Our CPU benchmarks are set with the lowest Linux scheduler priority by using the `NICE` command. By doing this, the CPU benchmark process does not take CPU cycles needed for the cloud applications. To avoid randomness in our data, we ran each experiment 100 times and calculated the average and standard deviation. We used a Linux library function `gettimeofday()` to calculate the running time of computation tasks. The granularity is one microsecond. In terms of energy consumption, the CPU power consumption is captured by RAPL counters in Intel's Sandy Bridge processors. Moreover, to measure the energy consumption of other system components, we wired a digital multi-meter (Mastech MAS-345) into the AC input power line of our machine. We collected samples every second from the power meter throughout our experiments.

To comprehensively understand the proposed Hylics architecture, we also selected representative benchmark tools and real-world applications, which are listed as follows:

- `dd`: Benchmark tool dd is used to test the speed of sequential file write.
- `sysbench`: Benchmark tool sysbench is used to test the speed of sequential and random file I/O.

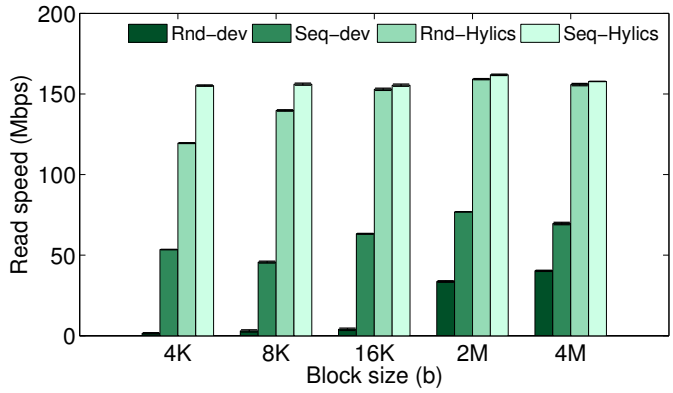


Figure 4.12: Sysbench VM read performance

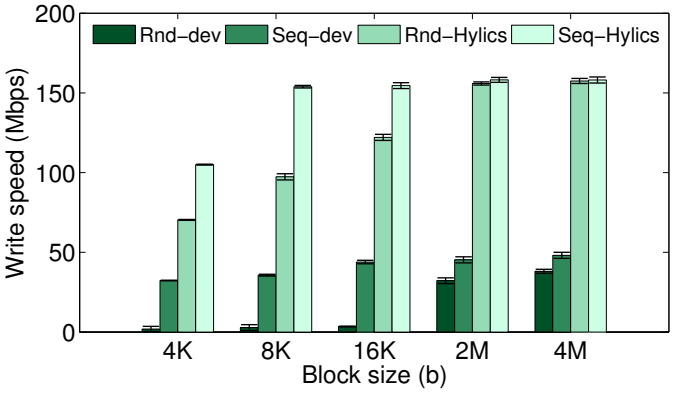


Figure 4.13: Sysbench VM write performance

- `LIVE555`: The LIVE555 libraries are designed for multimedia streaming, suitable for various cloud-based streaming applications.
- `FFmpeg`: FFmpeg is a common multimedia framework able to decode, encode, transcode, mux, demux multimedia files.
- `pigz`: pigz is a compression utility that exploits multiple processors and multiple cores to the hilt when compressing file data.
- `Lighttpd`: Lighttpd is a fast and flexible web server implementation with a low memory footprint, which is optimized for high-performance environments.

In particular, the transcoding task referred in the following context is a transsizing task. Transsizing consists of operations on changing the picture size of video, which is commonly seen in cloud environments for streaming to different devices. Our video source is multiple 1080P (1920×1080) video segments with 24FPS frame rate. The length of one video segment is 30 seconds. They are encoded with the widely used H.264 encoder. In our experiments, we performed transsizing task to convert the video segment into different resolutions. The

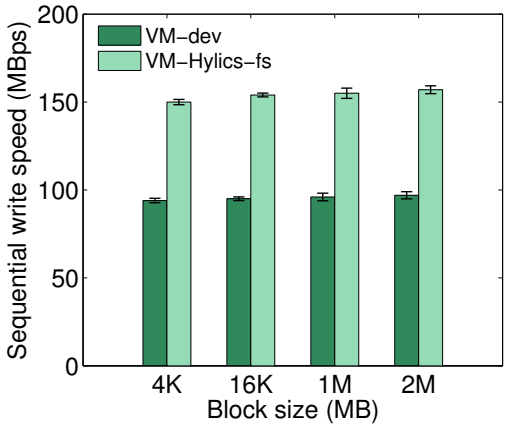


Figure 4.14: dd VM write performance

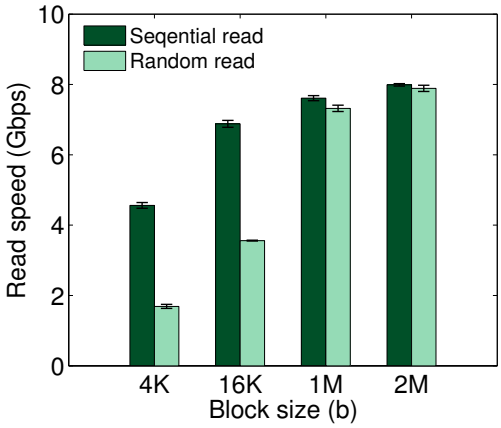


Figure 4.15: Sysbench host read performance

file compression task is performed by pigz with the default compression ratio, and we set the number of compression threads to be exactly the same number of the allocated vCPUs.

4.5.2 Benchmark Performance

As the first step, we used benchmarking tool `sysbench` and `dd` to understand the file I/O performance of the shared in-memory file system. Fig. 4.12 compares the read throughput when the VM is reading from the virtual block device and the Hylics in-memory file system. In this experiment, we disabled the operating system block cache and varied the block size to thoroughly understand the overall performance. As shown by our results, Hylics file system performs better in both sequential read and random read scenarios. This is mainly because the read operations are all executed in memory space. Fig. 4.13 presents the sequential write and random write throughput. Similarly as the read benchmark tests, the Hylics file system outperforms the native block device and has a more stable throughput. During the above experiments, we can find that the in-memory file system provision especially favors for the random read and write operations of small data blocks. Fig. 4.14 shows the comparison when the VM is writing null characters with the benchmarking tool `dd`. To ensure the fairness, we also disabled the writing cache and required physically writing the output data before the finish. This is performed by setting the `dd` parameter "conv=fsync". We found that, the in-memory file system achieves better write throughput than the default virtual disk device in all experiments, typically with a 31%-42% throughput increase. Meanwhile, we are also curious about the read and write performance of the in-memory file system at the hypervisor level. We therefore performed the same sysbench tests at the hypervisor layer. The results are listed in Fig. 4.15 and Fig. 4.16. As a matter of fact, the read and write throughput at the hypervisor level can both achieve gigabit-level performance.

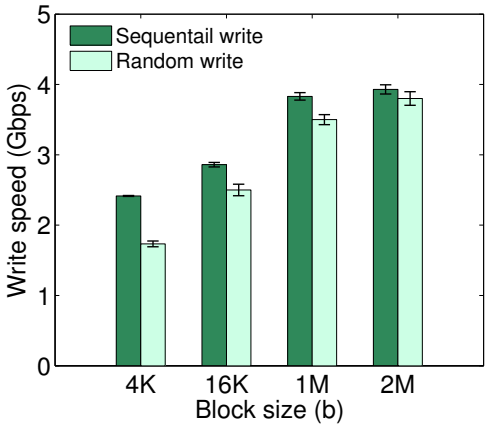


Figure 4.16: Sysbench host write performance

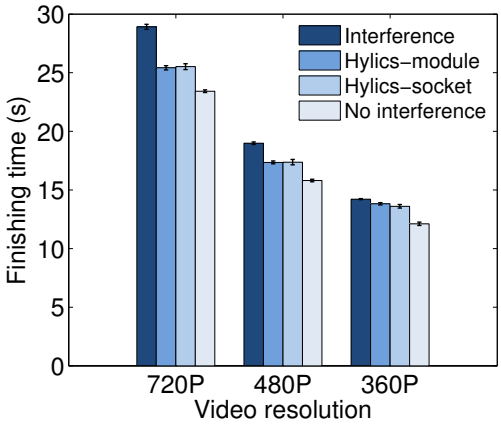


Figure 4.17: Video trancoding performance

4.5.3 Performance Gain in Real-world Applications

We further demonstrate the performance gain brought by the Hylics design in real-world applications. To understand the computation performance when enabling the Hylics architecture, we measured the transcoding task completion time with different transcoding output settings. The results are shown in Fig. 4.17. As we have discussed in Section 4.1, when the transcoding tasks and the streaming tasks simultaneously run inside the VM (labelled as "Interference" in the figure), we can observe the longest task completion time. Meanwhile, the completion time of stand-alone transcoding tasks in the VM is labelled as "No Interference", which is used as a baseline scenario. We measured the computation performance when offloading network I/O modules and socket level operations to the hypervisor layer. The results are labelled in the figure as "Hylics-module" and "Hylics-socket", respectively. In all the experiments, we can observe that Hylics shortens the transcoding task completion time by resolving the self-interference.

As a next step, we varied the number of vCPUs assigned to the VM and the total number of threads for the transcoding tasks to investigate the impact from different levels of computation power. The results presented in Fig. 4.18 indicate that Hylics also achieves nearly ideal performance. Note that in the (2 vCPUs, 2 threads) case, since the transcoding task runs on both of the two available vCPUs, such a configuration causes more severe self-interference inside the VM, as well as the worst computation performance. Hylics design, however, has a negligible increase on the task completion time. In these experiments, Hylics exhibits a 7%-27% computation performance enhancement when compared with the "Interference" case. To extend our experiment to other general applications, in the second experiment, file compression tasks and file delivery tasks were co-located on the VM. We varied the size of the file target to examine the performance with different workload stress.

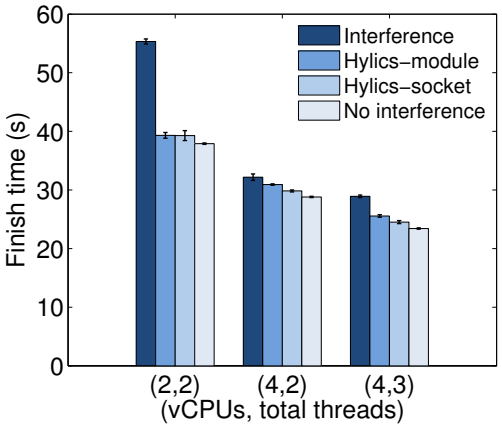


Figure 4.18: Varied number of vCPUs and threads

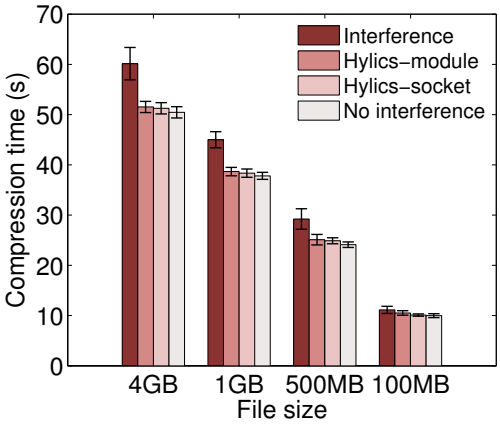


Figure 4.19: File compression performance

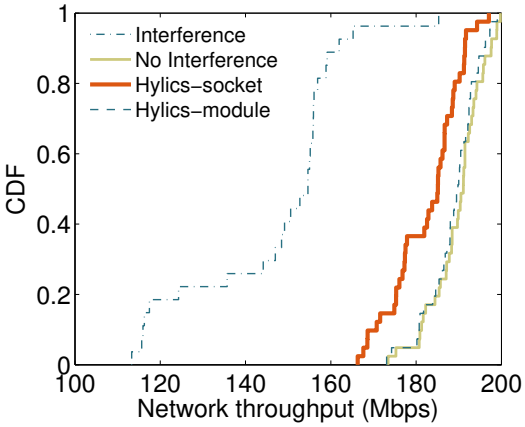


Figure 4.20: Streaming server performance

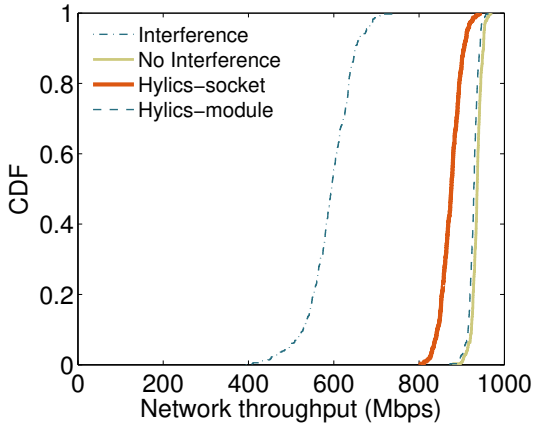


Figure 4.21: Web server performance

The results are shown in Fig. 4.19. Similarly as in the first experiment, Hylics design achieves almost the same computation performance as the "No Interference" case.

As for the networking performance, in Fig. 4.20 we present the CDF of the streaming throughput. In this figure, we also use "No interference" to label the stand-alone streaming performance inside the VM. The "Interference" curve describes the streaming throughput when we added concurrent transcoding tasks. By comparing the results between "Interference" and "No Interference", we can observe a 40% network throughput degradation. In this figure, "Hylics-module" and "Hylics-socket" label the streaming performance when the actual network data transfer is offloaded to the hypervisor level. To summarize, when we applied the Hylics architecture, the average throughput reached up to 190 Mbps, which is close to the "No interference" case. It is noted that the performance of offloading socket

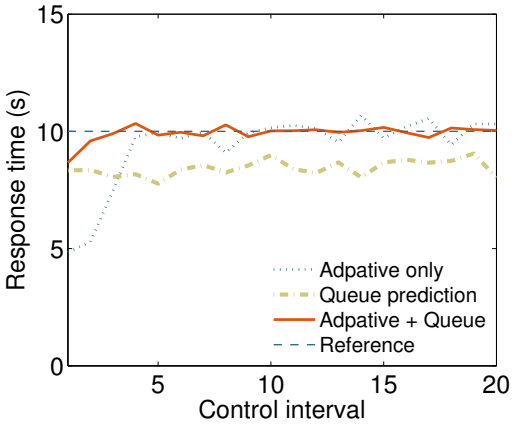


Figure 4.22: Response time of the synthetic workload #1

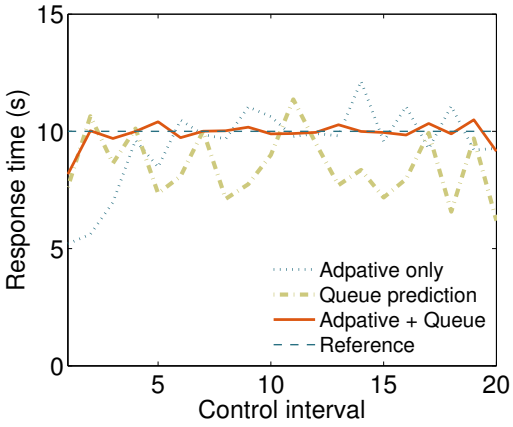


Figure 4.23: Response time of the synthetic workload #2

level operations to the hypervisor level is slightly worse than directly offloading network I/O modules. This is because the socket level offloading needs to return the sending result to the VM more frequently. In general, Hylics resolves the self-interference and achieves a 62-68% network throughput enhancement. We also tested the networking performance with file compression and delivery workloads, which is shown in Fig. 4.21. The comparison also shows the performance enhancement achieved by the Hylics design.

To demonstrate the effectiveness of the memory control scheme design, we applied the proposed file system space control algorithm when running the transcoding and streaming experiments. To maintain a reasonable stress on the VM, we selected multiple 720P videos as the input. We used two sets of workloads in our tests. The first set, workload #1 is a simple workload which has exponentially distributed inter-arrival time with an average of 8 seconds. The second set, workload #2 is a more complicated workload which periodically changes the average inter-arrival time from 8 seconds to 4 seconds. We consider the following running scenarios: If one video segment has not been requested in the last 1 minute, it will be evicted from the tmpfs to the standard file system inside the virtual disk space. We further make the assumption that if the tmpfs is temporarily full, then all the arrived transcoding requests will be executed on the file system hosted in the virtual disk space. Based on our real-world measurements in the target VM, when the transcoding task is executed on the in-memory file system, the average task completion time is 3.32 seconds, if the task is executed on the default file system inside the virtual block device, the average execution time is 4.02 seconds. We then used an offline measurement method to identify the correlation between the file system size m and the average service time $E[X|m]$, which can generally be fitted by an extended inversely proportional function.

To make a fair comparison, we used three different memory control schemes, namely, "adaptive control only", "queueing model prediction only" and the combination of the

Table 4.2: Aggregated response time error

Synthetic Workloads	Adaptive only	Queue	Adaptive+Queue
Workload #1	68.13s	35.39s	18.75s
Workload #2	22.1s	33.5s	5.56s

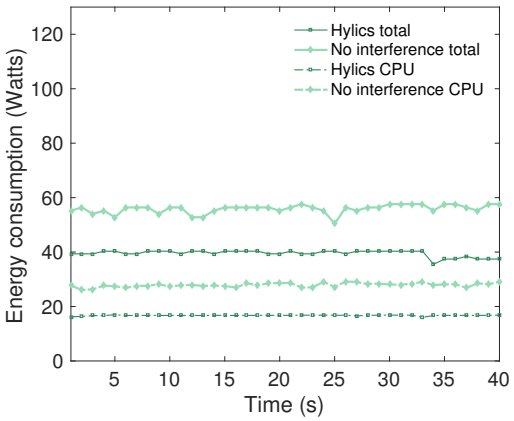


Figure 4.24: Energy consumption of streaming-only workload

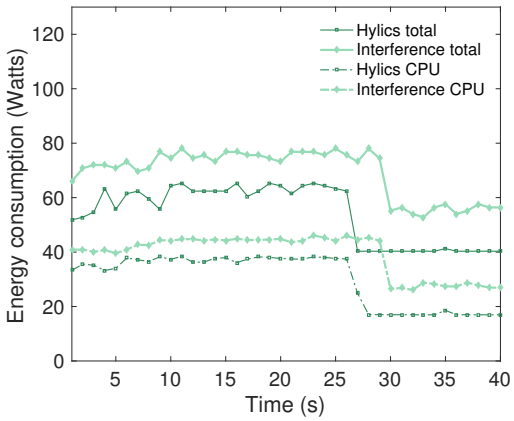


Figure 4.25: Energy consumption of hybrid workloads

both. The test lasted for one hour and we set the referenced response time τ to be 10 seconds because it is greater than the length of the video segment. In addition, we set the parameter *ci_minthresh* to be 30 seconds, and set *ci_maxthresh* to be 4 minutes. The average response time of the transcoding tasks is shown in Fig. 4.22 and Fig. 4.23. In these two figures, we can see that: (1) At the beginning stage, the adaptive control only method needs to gather enough measurement inputs to identify the system performance, which leads to a slow convergence speed. (2) The queueing model provides an approximation of the real system performance. There still exists gaps between the queueing model prediction and the real system performance. However, when combined with the adaptive control schemes, the queueing model helps to set a better starting point to identify the real system performance and hence provides a better performance as well as a fast convergence. We further present the aggregated response time errors of workload #1 and workload #2 in Table 4.2. The results show that combining the queueing model prediction and the self-adaptive control can achieve the least aggregated response time error. With the online memory usage control scheme, Hylics can operate with moderate memory usage under dynamic workload stress. The maximum amount of memory used for the file system when handling workload #1 and workload #2 is 245MB and 388MB, respectively.

We also pinpoint the energy savings achieved by the Hylics design. We closely measured the energy consumption while running the above experiments. Fig. 4.24 shows the results

Table 4.3: Perf profiling

Perf statistics	VM stream	Hylics-module	Hylics-socket
Context switches per second	$(762 \pm 12) \times 10^3$	$1,979 \pm 153$	$2,423 \pm 208$
Stalled cycles frontend	$(67.54 \pm 1.26)\%$ idle	$(70.18 \pm 1.25)\%$ idle	$(69.68 \pm 1.02)\%$ idle
Stalled cycles backend	$(51.93 \pm 0.58)\%$ idle	$(36.31 \pm 0.52)\%$ idle	$(39.52 \pm 0.77)\%$ idle
Instructions per cycle	0.61 ± 0.05	0.97 ± 0.02	0.91 ± 0.03
Stalled-cycles per instruction	1.10 ± 0.05	0.72 ± 0.03	0.79 ± 0.03

when we only introduced the streaming traffic, with no transcoding tasks. In this experiment, Hylics is running with the offloaded network I/O module. The energy consumption is therefore stable throughout the comparison. The results indicate that, when the network I/O module sends at the same rate, the energy consumption of Hylics architecture is much lower. As further shown in Fig. 4.25, after we initialized the transcoding task, the energy consumption of the Hylics system is still better than the "Interference" case. Note that in this case, the Hylics system had the same sending rate and also had a better processing performance. This can be observed in the figure that the transcoding task completion time is 26s for the "Hylics" case, better than 29s for the "Interference" case. In summary, Hylics can provide up to 32% improvement in terms of total energy efficiency when handling the hybrid workloads.

4.6 Further Discussion

To reveal more details, we conducted an in-depth investigation to pinpoint where exactly the gain is from. We collected low-level profiling benchmarks by `perf` when running the stand-alone streaming tasks. We tested three cases: streaming in VM, using network middleware offloading, and using socket offloading. The results are presented in Table 4.3. The first benchmark *context-switches per second* refers to the operation when the scheduler determines to run another process or when an interrupt triggers a routine's execution (such as handling the networking buffer). The average context switches per second of these three cases are 762,991, 1,979, and 2,423, respectively, which demonstrates a significant improvement brought by Hylics design. An explanation on this is that when handling high-volume streaming traffic inside the VM, the streaming process and the traffic handling process keep interrupting with each other. Consider if we add the co-located CPU intensive operations and disk I/O operations, the result can get much worse. As a comparison, Hylics design

offloads the actual network data transfer to the hypervisor level, and initiates the data transfer at memory space. As a consequence, the frequent interrupts no longer exist and as is the self-interference. We can see that plenty of context switches are saved by the Hylics design. The second and third benchmark presented are *stalled cycles at the frontend/backend stage*. A CPU cycle is stalled when the instruction pipeline does not advance during this cycle. In particular, the "frontend stages" are a group of stages during which the instructions are fetched and decoded. During the "backend stages", the instructions are then accordingly executed. From our measurement results, the number of stalled cycles in the frontend has only 2-3% difference, the number of stalled cycles in the backend, however, has 13-15% difference. This shows that the streaming process running inside the VM keeps waiting to be scheduled to send out the data. Yet it is often interrupted by the traffic handling process for sending the network buffer. The comparison on *instructions per cycle* and *stalled cycles per instruction* also shows that it is more efficient to shift the I/O intensive operations to the hypervisor layer.

Chapter 5

Conclusions and Future Directions

5.1 Conclusions

In this thesis, we systematically examine the power consumption brought by network traffic in virtualized cloud environment. We conducted experiments to measure the CPU power usage as well as the power consumption of the server. We revealed a series of detailed factors in virtualized environments that can have an impact on energy consumption. The results showed that such state-of-the-art virtualization designs noticeably increase the demand of CPU resources when handling network traffic. Besides on the total amount of the network traffic, the traffic allocation strategies and virtual CPU affinity conditions on the VMs also play key roles in deciding the power consumption. We analyzed the root cause and confirmed that the measurement results can be extended to different network configurations.

Next, we closely examined the self-interference from real-world applications in virtualized environments. To jointly optimize performance and energy efficiency for hybrid cloud workloads, we designed and developed *Hylics*, a novel solution that leverages the hypervisor-level in-memory file system sharing. We implemented a prototype of Hylics in KVM and evaluated the overall performance through real-world workloads, which indicates that such a design can largely improve network throughput and accelerate computation tasks in the presence of the self-interference. The energy efficiency of the underlying server is also enhanced.

5.2 Future Directions

At the time of writing this thesis, we have witnessed concerted effort towards optimizing the networking performance and energy efficiency in virtualized environments. We will discuss some of the future directions in this section.

Hardware manufacturers have noticed the high overhead for handling network traffic in virtualized environments, and have created devices that greatly alleviate the virtual NIC

overhead. Two popular solutions are Virtual Machine Device Queues (VMDQ) and Single Root I/O Virtualization (SR-IOV) [54]. Both of them work by duplicating components of the NIC to give a VM more direct access to the hardware. Note that in the current VMDQ architecture, the software switch and the hypervisor process are still involved in copying packets in between the host's memory space and the VM's memory space. Therefore, the interference observed in this thesis still exists, and there remains much room for further improvement. For SR-IOV, the hypervisor only handles interrupts generated by SR-IOV enabled NICs, which will greatly alleviate the extra overhead on CPUs, thereby mitigating the impact that we have observed. Although such sophisticated hardware is still expensive and with scalability and security issues, with advanced hardware technologies in the future, hardware-assisted solutions are promising to achieve better energy efficiency for cloud virtualization.

On the other hand, we can have solutions developed from the perspective of network traffic itself. For instance, based on the detailed log information on the cloud VM, cloud providers could reshape the traffic combination on each physical machine and increase the traffic deviation. A concrete example is to delay certain low-priority traffic (e.g., data synchronization traffic). Although this requires cross-layer design for cloud virtualization, the idea of "application-aware" is already one of the development trends for many enterprise cloud service providers as IBM [62]. Note we may not need a perfect algorithm to optimally skew the traffic load. Our measurement indicates that a slight increase on traffic deviation can already achieve reasonable energy savings.

There have been other pioneer studies on mitigating the overhead of network transactions in virtualized clouds by offloading work from the VM space to the host space. An example is vSnoop [35], which offloads the TCP acknowledgement function from the VM to the hypervisor. vPro [24] further offloads the TCP congestion control function to the hypervisor inside the host space. Despite the increased TCP throughput, such offloading approaches migrate only the control plane of the TCP protocol, with the data plane remaining inside the VM. As such, the data transmission still involves complicated cooperation between a VM, its hypervisor and the host kernel. Meanwhile, the modifications to the existing TCP protocol further limit the flexibility to adapt to different application scenarios. There is still much to explore to effectively offload the data plane of network packet processing to the host space.

Based on our observations on virtual CPU affinity, cloud service providers could also adaptively reconsider virtual CPU affinity to obtain a better tradeoff between energy efficiency and resource utilization of the underlying hardware. Although pinning VM to exclusively run on certain physical core decreases the utilization of the infrastructure, under certain circumstances, it is a feasible solution to reduce the energy overhead by 10-20%. This problem is not trivial and should be carefully investigated through classic resource provisioning and VM migration approaches.

Bibliography

[1] Electric power annual 2014. `http://www.eia.gov/electricity/annual/`. Accessed: 2016-09-30.

[2] Running average power limit. `https://lwn.net/Articles/545745/`. Accessed: 2016-08-30.

[3] vhost driver. `http://www.linux-kvm.org/page/UsingVhost`. Accessed: 2016-10-10.

[4] virtio-vsock. `http://wiki.qemu.org/Features/VirtioVsock`. Accessed: 2016-10-30.

[5] Vmware infrastructure architecture overview. `http://www.vmware.com/pdf/vi_architecture_wp.pdf`. Accessed: 2016-08-30.

[6] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[7] K J. Åström and B. Wittenmark. *Adaptive Control*. Courier Corporation, 2013.

[8] A O. Ayodele, J. Rao, and T E. Boult. Performance measurement and interference profiling in multi-tenant clouds. In *Proc. of IEEE CLOUD*, 2015.

[9] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[10] R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–76, 2004.

[11] J. P. Billaud and A. Gulati. hclock: Hierarchical qos for packet scheduling in a hypervisor. In *Proc. of ACM EuroSys*, 2013.

[12] D. Bruneo, A. Lhoas, F. Longo, and A. Puliafito. Modeling and evaluation of energy policies in green clouds. *IEEE TPDS*, 26(11):3052–3065, 2015.

[13] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS Operating Systems Review*, 35(5):103–116, 2001.

[14] J. Che, C. Shi, Y. Yu, and W. Lin. A synthetical performance evaluation of openvz, xen and kvm. In *Proc. of IEEE Asia-Pacific Services Computing Conference (APSCC)*, 2010.

[15] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt. Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds. In *Proc. of IEEE MASCOTS*, 2015.

[16] S N T. Chiueh and S. Brook. A survey on virtualization technologies. *RPE Report*, 2005.

[17] C. Clark, K. Fraser, S. Hand, J G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of USENIX NSDI*, 2005.

[18] B. des Ligneris. Virtualization of linux based computers: the linux-vserver project. In *Proc. of IEEE International Symposium on High Performance Computing Systems and Applications (HPCS)*, 2005.

[19] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with sr-iov. *Elsevier Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[20] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proc. of International Workshop on Power-Aware Computer Systems*, 2002.

[21] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, 2003.

[22] H. Esmaeilzadeh, T. Cao, Y. Xi, S M. Blackburn, and K S. McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. *ACM SIGARCH Computer Architecture News*, 39(1):319–332, 2011.

[23] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proc. of USENIX ATC*, 2008.

[24] S. Gamage, D. Xu R. Kompella and, and A. Kangarlou. Protocol responsibility offloading to improve tcp throughput in virtualized environments. *ACM Transactions on Computer Systems (TOCS)*, 31:1–34, 2013.

[25] S. Gamage, C. Xu, R. R. Kompella, and D. Xu. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proc. of ACM SoCC*, 2014.

[26] A. Gordon, N. Amit, N. Har'El, et al. Eli: Bare-metal performance for i/o virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[27] A. Gordon, M. Ben-Yehuda, D. Filimonov, and M. Dahan. Vamos: Virtualization aware middleware. In *Proc. of USENIX WIOV*, 2011.

[28] S. Govindan, A R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proc. of ACM VEE*, 2007.

[29] F. Hao, T V. Lakshman, S. Mukherjee, and H. Song. Secure cloud computing with a virtualized network infrastructure. In *Proc. of USENIX HotCloud*, 2010.

[30] N. Har'El, A. Gordon, A. Landau, et al. Efficient and scalable paravirtual i/o system. In *Proc. of USENIX ATC*, 2013.

[31] Q. Huang, F. Gao, R. Wang, and Z. Qi. Power consumption of virtual machine live migration in clouds. In *Proc. of IEEE CMC*, 2011.

[32] J. Hwang, KK. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[33] Y. Jin, Y. Wen, and Q. Chen. Energy efficiency and server virtualization in data centers: An empirical investigation. In *Proc. of IEEE INFOCOM WKSHPS*, 2012.

[34] V. Jujjuri, E V. Hensbergen, A. Liguori, and B. Pulavarty. Virtfs-a virtualization aware file system pass-through. In *Proc. of Ottawa Linux Symposium (OLS)*, 2010.

[35] A. Kangarlou, S. Gamage, R. Kompella, and D. Xu. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In *Proc. ACM/IEEE Supercomputing Conference (SC)*, 2010.

[36] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A A. Bhattacharya. Virtual machine power metering and provisioning. In *Proc. of ACM SoCC*, 2010.

[37] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for smp vms. *ACM SIGPLAN Notices*, 48(4):369–380, 2013.

[38] D. Kusic, J.O. Kephart, J.E. Hanson, K. Nagarajan, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *Proc. of IEEE ICAC*, 2008.

[39] A K. Maji, S. Mitra, and S. Bagchi. Ice: An integrated configuration engine for interference mitigation in cloud services. In *Proc. of IEEE ICAC*, 2015.

[40] A K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proc. of ACM Middleware*, 2014.

[41] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J M. Pierson, and A V. Vasilakos. Cloud computing: Survey on energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):33, 2015.

[42] Y. Mei, L. Liu, X. Pu, and S. Sivathanu. Performance measurements and analysis of network i/o applications in virtualized cloud. In *Proc. of IEEE CLOUD*, 2010.

[43] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proc. of ACM EuroSys*, 2010.

[44] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of USENIX ATC*, 2013.

[45] J F. Ogden. Hardware support for efficient virtualization. *University of California, San Diego, Tech. Rep*, 2006.

[46] F. Oh, H S. Kim, H. Eom, and H Y. Yeom. Enabling consolidation and scaling down to provide power management for cloud computing. In *Proc. of USENIX HotCloud*, 2011.

[47] P. Padala, X. Zhu, Z. Wang, S. Singhal, K G. Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.

[48] 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*, 6(3):314–329, 2013.

[49] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov. Stay-away, protecting sensitive applications from performance interference. In *Proc. of ACM Middleware*, 2014.

[50] C. Reiss, A. Tumanov, G R. Ganger, R H. Katz, and M A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, 2012.

[51] R. Rivas, A. Arefin, and K. Nahrstedt. Janus: a cross-layer soft real-time architecture for virtualization. In *Procs. of ACM HPDC*, 2010.

[52] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[53] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Proc. of IEEE International Conference on Computer and Network Technology (ICCNT)*, 2010.

[54] R. Shea and J. Liu. Network interface virtualization: challenges and solutions. *IEEE Network*, 26(5):28–34, 2012.

[55] R. Shea, F. Wang, H. Wang, and J. Liu. A deep investigation into network performance in virtual machine based cloud environments. In *Proc. of IEEE INFOCOM*, 2014.

[56] R. Shea, H. Wang, and J. Liu. Power consumption of virtual machines with network transactions: Measurement and improvements. In *Proc. of IEEE INFOCOM*, 2014.

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

[58] P. Snyder. tmpfs: a virtual memory file system. In *Proc. of EUUG Conference*, 1990.

[59] S. Soltesz, H. Pötzl, M E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *ACM SIGOPS Operating Systems Review*, 41(3):275–287, 2007.

[60] G. Wang and T E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proc. of IEEE INFOCOM*, 2010.

[61] A. Whitaker, M. Shaw, and S D. Gribble. Scale and performance in the denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.

[62] D. Williams, S. Zheng, X. Zhang, and H. Jamjoom. Tidewatch: Fingerprinting the cyclicality of big data workloads. In *Proc. of IEEE INFOCOM*, 2014.

[63] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: towards real-time hypervisor scheduling in xen. In *Proc. of IEEE EMSOFT*, 2011.

[64] C. Xu, S. Gamage, H. Lu, et al. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proc. of USENIX ATC*, 2013.

[65] C. Xu, S. Gamage, P. N. Rao, et al. vslicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proc. of ACM HPDC*, 2012.

[66] C. Xu, B. Saltaformaggio, S. Gamage, R R. Kompella, and D. Xu. vread: Efficient data access for hadoop in virtualized clouds. In *Proc. of ACM Middleware*, 2015.

[67] F. Xu, F. Liu, H. Jin, and A V. Vasilakos. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31, 2014.

[68] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proc. of USENIX NSDI*, 2013.

[69] B. Yee, D. Sehr, G. Dardyk, et al. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of IEEE Symposium on Security and Privacy*, 2009.

[70] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proc. of ACM EuroSys*, 2013.

[71] K. Zheng, X. Wang, L. Li, and X. Wang. Joint power optimization of data center network and servers with correlation analysis. In *Proc. of IEEE INFOCOM*, 2014.

[72] Q. Zhu and T. Tung. A performance interference model for managing consolidated workloads in qos-aware clouds. In *Proc. of IEEE CLOUD*, 2012.

[73] Q. Zhu, J. Zhu, and G. Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proc. of IEEE SC*, 2010.