Virtualization Empowered Resource Management and Content Distribution in the Cloud

by

Silvery Di Fu

B.Sc., Simon Fraser University, 2016
B.Eng., Zhejiang University, 2016

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

in the
School of Computing Sciences
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Summer 2017

Approval

Name:	**Silvery Di Fu**
Degree:	**Master of Science (Computing Science)**
Title:	***Virtualization Empowered Resource Management and Content Distribution in the Cloud***
Examining Committee:	**Chair:** Greg Baker Senior Lecturer

Jiangchuan Liu
Senior Supervisor
Professor _____

Ryan Shea
Supervisor
University Research Associate _____

Qianping Gu
Internal Examiner
Professor _____

Date Defended: 1 June 2017

Abstract

Virtualization is the cornerstone technology of cloud computing. Advancements in virtualization enable researchers to tackle key challenges in today's cloud. The first part of this thesis delves into the emerging container virtualization and how leveraging containers we address resource management and pricing challenges in the cloud. We try calling for an end to the constant battle between public cloud providers and users over the pricing options of cloud instances: the users generally have to pay for the entire billing cycle even on fractional usage. Ideally, idle cloud instances with residual billing cycle should be resalable by their users. Such trading demands efficient resource consolidation and multiplexing, because the revenue and use cases are confined by the transient nature of the instances. This thesis presents HARV, a novel cloud service that facilitates the management and trade of cloud instances. The platform relies on hybrid virtualization, an infrastructure layout integrating both the hypervisor-based virtual machines and lightweight containers, incorporating a truthful online auction mechanism for instance trading and resource allocation. Our design achieves efficient resource consolidation with no need for provider-level support, and we have deployed a prototype of HARV on the Amazon EC2 public cloud. Our evaluations reveal that applications experience negligible performance overhead when hosted on HARV; trace-driven simulations further show that HARV can achieve substantial cost savings.

The second part of the thesis explores the emerging Network Function Virtualization (NFV). Virtualization and cloud computing constitute a major driving force for Internet innovations. In today's Internet, multimedia content traffic accounts for the largest share of all traffic. Downstream towards the consumers, multimedia traffic often traverse through middleboxes, undergoing additional data processing imposed by content distributors. With NFV, middleboxes are embedded in general-purpose, off-the-shelf servers, allowing content distributors to conveniently borrow existing cloud technologies to process traffic. Despite these benefits, we find NFV incurs an undue amount of energy consumption when carrying out high packet forwarding performance. We identify the energy inefficiency issue in the NFV dataplane which can be exacerbated if not handle properly. We outline a power management framework that exploits CPU frequency scaling to save energy.

Keywords: Cloud Computing; Virtualization; Multimedia Content Distribution

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background

Cloud computing has been rapidly evolving in the past a few years. Emerging virtualization technology such as container and Network Function Virtualization (NFV) are reshaping the way people *manage* the cloud; whereas performance and cost remain the primary concerns when people *deploy* cloud services or applications. A representative cloud service nowadays is video streaming, whose traffic accounts for more than 80% of today's Internet traffic; video content providers such as Netflix migrate the majority of their infrastructure to the public cloud while *paying the bill* to the cloud providers. In the meantime, downstream towards the consumers, multimedia/video traffic often traverse through middleboxes, undergoing additional data processing imposed by content distributors. The advent of NFV brings unprecedented flexibility to implement and deploy these middleboxes. On the other hand, the software-based packet processing approach adopted by NFV may not be as energy-efficient as the traditional dedicated hardware-based approach, particularly when carrying out high packet forwarding performance.

1.1 Contributions of this thesis

With an emphasis on real system measurement and implementation, my thesis explores, evaluates, and stretches the capabilities of virtualization in the following aspects: resource manageability, pricing, application performance, and energy-efficiency, all in the context of cloud computing. As described in what follows:

In **Chapter 1**, we provide a background on the two emerging virtualization technologies covered in this thesis, namely the container virtualization and Network Function Virtualization. The technical survey on container and container empowered cloud computing was published in IEEE Internet Computing, March 2016 [26].

In **Chapter 2**, we delve into the emerging container virtualization and how we can facilitate cloud resource management and pricing leveraging containers. We identify there has been a constant battle over the billing options of between the public cloud providers

and their users. The users generally have to pay for the entire billing cycle even on fractional usage. Much like the house renting services Airbnb where house owners fractionally use their house, the work proposes a "cloudbnb" service for cloud users. We designed and implemented the service, HARV, leveraging the container virtualization. Our design achieves efficient resource consolidation with no need from provider-level support, and a prototype of HARV has been deployed over the Amazon EC2 public cloud. Evaluations on both micro-benchmarks and real-life workloads reveal that applications experience negligible performance overhead when hosted on HARV. In the most conservative case, HARV achieves over 20% cost savings as compared to the fixed-price billing options. The work has been accepted to IEEE/ACM IWQoS, 2017 [27].

In **Chapter 3**, we show that NFV powered middleboxes are prone to high energy overhead when delivering multimedia content. In particular, we find there exists energy inefficiency in the data forwarding component of major NFV platforms. We demonstrated this inefficiency is inherent to its design that excessively uses CPU cycles to attain high performance. Since multimedia traffic is persistent throughout the streaming session and usually impose additional QoS constraints, existing energy saving methods may not function well. Based on these observations, we outline power management framework for NFV-based multimedia content delivery. We show that CPU frequency scaling can achieve promising energy savings without compromising the performance of multimedia applications. The work has been accepted to publish in IEEE MultiMedia, 2017 [25].

We conclude the thesis in **Chapter 4**, where we sketch our thoughts about future research directions pertaining to this thesis.

1.2 Virtualization Technologies Covered in the Thesis

1.2.1 Container Virtualization

Container is a lightweight, flexible, and application-driven tool for fine-grained resource control and OS-level isolation. It offers cloud providers an alternative tool for resource multiplexing and control other than virtual machines. A closer relative to container is an operating system process since both of them essentially encapsulate a (single) application runtime. What the container offers additionally is the capabilities of controlling and isolating OS resources assigned to the runtime, meanwhile including complete dependencies in a container instance. As such, one may also refer to a container as a *virtual environment*.

The idea of the container and OS-level virtualization is not new. Linux-VServer [46] and OpenVZ[10] are two earlier container-based virtualization platforms. Yet container has only come to the fore in recent years, for two reasons. First, it has shifted from the original role as a "hypervisor-free" virtual machine, where a single container instance had to be built full-fledge to support a full OS (as in VServer and OpenVZ), to a lightweight runtime environment for applications. Second, recent platforms significantly simplify the procedure

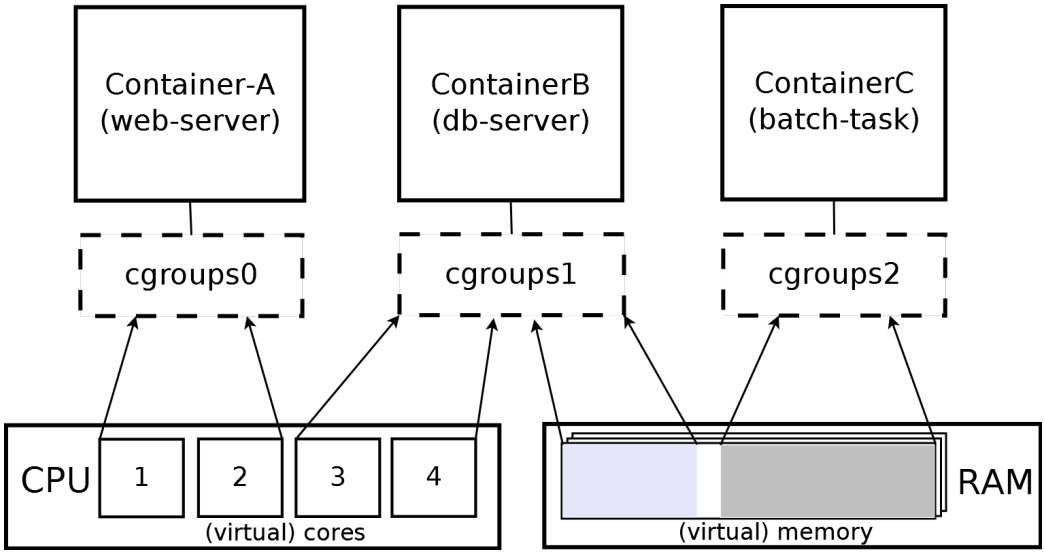


Figure 1.1: Control Groups (cgroups) for CPU and Memory

of container creation and management. These two advancements meet the growing need of deploying cloud-based distributed applications with "just enough" performance overhead and maintenance cost. They have been jointly achieved by Docker [5], the most established and popular container by far.

Docker relies on utilities of the modern Linux kernel to create and manage container runtime[1]. First and foremost, a *Control Groups* (cgroups) module defines a collection of kernel resource controllers for, including but not limited to CPU, memory, network and disk I/O. User-level code are allowed to customize these controllers through cgroups virtual file system. At the runtime, cgroups are assigned to a process through function hooking, by which resource accesses of the process will trigger the corresponding hooks. As such, without intervening performance-critical execution paths, cgroups is able to achieve resource tracking and control efficiently. Figure 1.1 gives an example showing the use of cgroups with container runtime. `cgroups1` defines the control groups of two CPU cores $3, 4$ and a limited amount of memory shaded in light gray. It is assigned to ContainerB that encapsulates a database server runtime (denoted by *db-server*). During its life-cycle, the CPU usage of db-server is limited to cores $3, 4$ (which in turn will be used exclusively by db-server) and the memory footprint is limited by the given amount. As `cgroups0` and `cgroups2` indicate, it is also possible to define the cgroups solely for CPU, memory and other manageable system resources, or arrange them together in different combinations.

Docker further leverages the namespaces isolation feature, enforcing processes to have separate namespaces for system resources including but not limited to PID, IPC, and network. The resources allocated to the application runtime insides a container cannot be

[1]Later on Docker developed a native implementation of these modules, as a solution for cross-platform support [9][13].

addressed by the other containers, and vice-versa. With the use of cgroups and namespaces isolation, a container runtime can readily be hosted. Docker has donated the implementation of these modules to the OCI project in a collection called *runC*, serving as the cornerstone to a standardized container runtime. Notably, both cgroups and namespaces isolation have been used independently and flexibly to achieve resource control [39] or isolation [35], and many other container platforms are built based on these modules [11][8][9], making the container techniques versatile.

To facilitate container creation and management, Docker has also designed and implemented the container (image) format. Each container runtime is created from an image predefined, which includes all the dependencies the target application requires. Besides, the images can be stored in publicly accessible repositories and conveniently distributed. Finally, Docker utilizes a layered file system to allow efficient sharing of container images, which significantly reduces the storage overhead.

1.2.2 Container vs. Virtual Machine

To date, machine virtualization remains the most common way of managing hardware resources for cloud providers (e.g., Xen [15] for Amazon EC2 public cloud), and has attracted significant standardization efforts (e.g., OVF). Containers share such common design goals and features with virtual machines as resource isolation and imaging. Yet the new generation of containers, represented by Docker, are built with important distinct tenets [46].

At the high-level, the container is upward-facing and application-driven, while the virtual machine is downward-facing and hardware-driven. Hypervisor-based virtualization, e.g., Xen, enables multiple users to create virtual machines that share the same physical hardware, where distinct OSes, ranging from the proprietary to open-sourced, are hosted in an isolated fashion. Containerization permits only applications to be encapsulated in containers, which leads to greatly reduced deployment overhead and much higher instance density on a single machine. It unfortunately disallows a full OS-stack to be run separately from the host OS, prohibiting a multi-OS setting.

At the low-level, the container leverages the host OS utilities to achieve resource encapsulation and management. Hypervisors, on the contrary, runs directly on top of the hardware in the most privilege mode, taking charge of accessing and managing the underlying hardware resources, akin to the role of an operating system kernel. The virtual machines/guest OS kernels now run in a less privileged mode, such that any privileged system calls from guest OSes will be trapped to the hypervisor's kernel and executed in isolation. This process can be done in two ways, either through modifying the Guest OS kernel and drivers to enforce privileged calls being sent to the hypervisor directly, which is known as the Paravirtualization (PV); or trapping those calls by special hardware extensions, known as the Hardware-assisted Virtualization (HVM). As such, virtualization functions at the border of hardware and OS. It is able to provide strong performance isolation and security guarantees

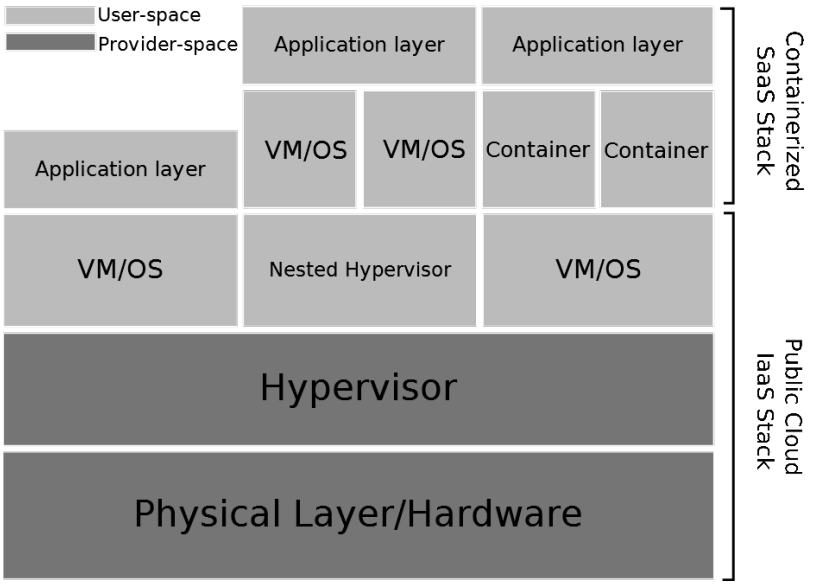


Figure 1.2: Hybrid virtualized layering in public cloud

with the narrowed interface between VMs and hypervisor. Containerization, which sits in between the OS and applications, incurs lower overhead, but potentially introduces greater security vulnerabilities such as namespace-agnostic system calls [23].

1.2.3 Hybrid Virtualization

The container is not a replacement to the virtual machine; rather, these two compliment each other, and are to be placed into a unified framework for cloud vendors and users. When they are used together (e.g. in a public cloud), with the underlying physical hardware and the OS in between, they form the hybrid-virtualization layers sitting at the bottom of the standard model (analogous to the link layers in the OSI model). This layering essentially places containers insides hypervisor-based virtual machines which are then run on top of the underlying hardware, and the rationale is that these two technologies, with distinct design goals and characteristics as illustrated earlier, are in fact complementary to each other. This hybrid layering is intuitive yet powerful, by which cloud users are allowed to orchestrate their resources provisioned without any assistance from the underlying infrastructure providers. As shown in Figure 1.2, the cloud resource stack is, more often than not, separated into user's space and provider's space. For security and overall cloud performance, (public) providers do not allow users to freely launch any operations in their space, making resource consolidation and orchestration in the user space a headache (left half in the user space). In the hybrid layering (right half), container adds another layer of abstraction in between the VM instance and applications, decoupling application-specific scheduling and VM scheduling. A direct use of this model is Platform-as-a-Service (PaaS), which will rely on the container to establish any service runtime environment effortlessly, in

the meantime, the PaaS providers are exempted from handling the physical infrastructure. IaaS providers can also benefit from this paradigm [1]. Without involving redundant and unnecessary OS processes, the scaling out is more efficient for containers than VMs, which enables finer-grained billing [7]. Noticeably, the hybrid layering is able to coexist with the virtualization-only solution, as the figure indicates.

1.2.4 Network Function Virtualization

Led by global network operators, Network Function Virtualization (NFV) is an on-going paradigm shift in the way people implement and deploy *middlebox*, i.e., a network device that performs functions other than packet forwarding/routing such as NAT, firewall, load balancer, and transcoders [17]. Traditionally, middleboxes are implemented on dedicated hardware devices with high infrastructure and management costs. NFV targets at embedding middleboxes[2] on the off-the-shelf, general-purpose server machines. Potential NFV users include cloud providers, network providers, and Telco CDNs (Content Distribution Network); transitioning to NFV can help them alleviate the capital expenditures (CAPEX) and operating expenses (OPEX). An attractive deployment target for NFV, for example, is the carrier network edge or their Central Offices.

NFV reuses cloud technologies, including virtual machines and containers, to deploy, manage, and scale middleboxes [44][41]. A middlebox is now a "software box," running as a normal cloud application inside a virtual machine, a container, or a hybrid-layer of both[3]. As such, the process it takes to manage middleboxes is greatly simplified: installing patches, monitoring, scaling up and out, etc., are much easier (and faster) to perform on software than hardware.

Such flexibility comes with a price, however. Hardware devices enjoy circuit-level specializations to attain both high performance and high energy efficiency; whereas virtual, software devices usually have to make a trade-off between these two objectives. We will explore such trade-off in this thesis.

[2] In the context of NFV, middleboxes are often referred to as the *network functions*.

[3] At the cost of losing the benefits of virtualization, it is possible to run network functions on bare-metal machines, too.

Chapter 2

Container Empowered Resource Management and Pricing

2.1 Overview

IaaS (Infrastructure as a Service) has been a major form of public cloud service deployment, and it is estimated the IaaS market will grow from $15.1B in 2014 to $126.2B by 2026 [24]. State-of-the-art IaaS cloud providers generally offer resources to users as *virtual machine* (VM) instances, and a user has to pay for the full billing cycle of an instance even if only a fraction of the cycle is to be used. Existing studies have shown that this partial usage issue exists extensively among cloud tasks [32]. As a matter of fact, 79.8% of cloud users use less than 20% of billing cycle according to the previous analysis [32]. There have been pioneer efforts toward fine-grained resource provisioning and pricing to offer instances that better match the user demands [32][33][38]. Unfortunately, as we will show later, there is a trade-off in terms of cost-effectiveness between a cloud provider and the users since the former generally resists to refining the instance granularity.

An attractive alternative is to allow users to re-sell their unused instances [14]. Having a cloud market allowing this not only improves the utilization of cloud resources but is beneficial for building a healthier cloud ecosystem [20]. This is however easy said than done. To generate re-usable resources, it is necessary to aggregate and consolidate the partially used instances. Early solutions on resource consolidation are mostly done from the provider-side, e.g., how to allocate virtual machines given the limited number of available physical machines [19][53][29]. To achieve similar goals from the user-side in the public cloud, global knowledge of the physical machine cluster will be needed, together with such operations as VM live migration. It is hardly possible or feasible for a public cloud provider to expose those low-level interfaces to its user's given concerns from security and network/system management. Such third-party solutions as *Cloud Brokerage* [49] suggest that a wholesaler may purchase a large volume of instances from the cloud provider and re-sell them to users

at discounted prices. While they do not require infrastructure changes to the public cloud provider, the broker still operates at the VM level; thereby the *usage waste problem* of the cloud instances remains to exist.

Moreover, maximizing the (re)usage efficiency demands effective resource-multiplexing, i.e., allowing workloads from more than one user to run together on an instance. Without a proper implementation, this will lead to nested virtualization that can introduce considerable performance overhead [51]. Maximizing the (re)usage efficiency also calls for novel pricing mechanism beyond those offered by the public cloud provider. Facing the ever changing availability of partially used instances and the arrival patterns of their potential users, a dynamic online solution is naturally expected.

In this chapter, we show strong evidence that the partially used instances are valuable resources, which, if properly recycled, can remarkably improve the cost-effectiveness of public cloud users. We also demonstrate that such an *instance recycling* service is doable with limited overhead to both cloud users and providers. In particular, we design and implement HARV, a third-party platform that **HAR**nesses hybrid **V**irtualization to both recycle cloud instances and manage their users' tasks. The hybrid virtualization seamlessly combines existing hypervisor-based virtualization and containerization and does not require any change to the infrastructure of the existing public cloud providers. We present a two-level scheduling policy in HARV to simplify cluster resource management and ensure its applicability with a public cloud. It also incorporates a truthful online auction mechanism to determine the allocation of requests and the corresponding recycling price. We have implemented HARV and deployed it with the Amazon EC2 public cloud. Extensive experiments with real-world benchmarks and large-scale simulations verify that HARV is highly scalable and cost-effective. It achieves cost savings up to 24% on a typical 1-hour billing cycle, and 19% of the 15-minute billing cycle.

The remainder of this chapter is organized as follows. We first explore the public cloud cost-effectiveness issues and present a system overview in Section 2.2. In Section 2.3, we present details about our system design, including its scheduling and pricing policies. Extensive experiments and evaluations can be found in Section 2.4. Finally, we review related literature in Section 2.6.

2.2 Background and Motivation

We start from investigating the (in)efficiency of state-of-the-art billing options offered by IaaS cloud and how it affects users' cost-effectiveness. We argue that a recycling mechanism is necessary for utilizing the residual time of cloud instances, and suggest that hybrid virtualization is the key toward real-world implementation and deployment.

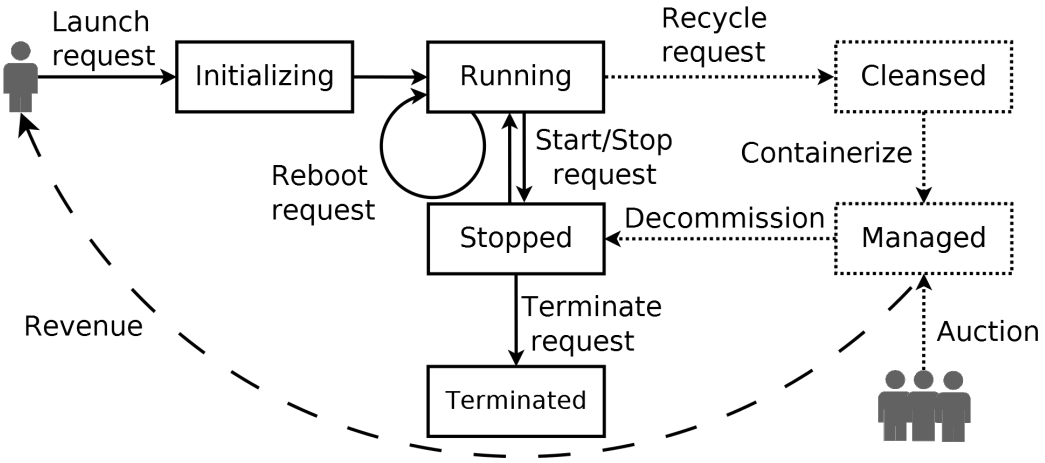


Figure 2.1: Augmented instance life cycle

2.2.1 Billing Inefficiency: Cause and Consequence

In Fig. 2.1, we depict the state transitions in a typical public cloud instance's life-cycle (the ones in solid lines). In general, the provisioned instance is considered in the same billing cycle as long as it stays in the *Running phase*; however, if the user *stops* a running instance, a new billing cycle will begin when it is restarted. It is because the cloud provider needs to release the computing resources held by the instance (CPU cores, memory, IP, etc.) when handling the stop request. As such, when the user "restarts" the instance, a new group of resources has to be re-provisioned for it. Consequently, the user will be charged for the newly provisioned resources, even if (in terms of time) it still falls into the same billing cycle. Let T_{actual} be the actual time a user utilizes an instance, and T_{cycle} be the duration of the billing cycle, the *residual instance time* T_{RI} can be calculated as:

$$T_{RI} = T_{cycle} - (T_{actual} \bmod T_{cycle}) \tag{2.1}$$

In Fig. 2.2, we depict the number of potential residual instances ($T_{cycle} = 1hr$) per time slot during a ≈ 6.5 hours record duration in a real-world cluster. Each of the four lines denotes an assumed T_{RI} range of the residual instances. The data are extracted from one of the Google's publicly accessible traces [30]. Those Google-cluster traces have also been widely used in other recent cloud resource provisioning studies as well [19][49][54]. As shown, despite the fluctuation, there is a constant supply of residual instances: about 10,000 to 20,000 with $T_{RI} \geq 15min$, and the ones with $T_{RI} \geq 30min$ account for nearly half of the total.

Define the *waste ratio*:

$$W_{ratio} = T_{RI}/T_{actual} \tag{2.2}$$

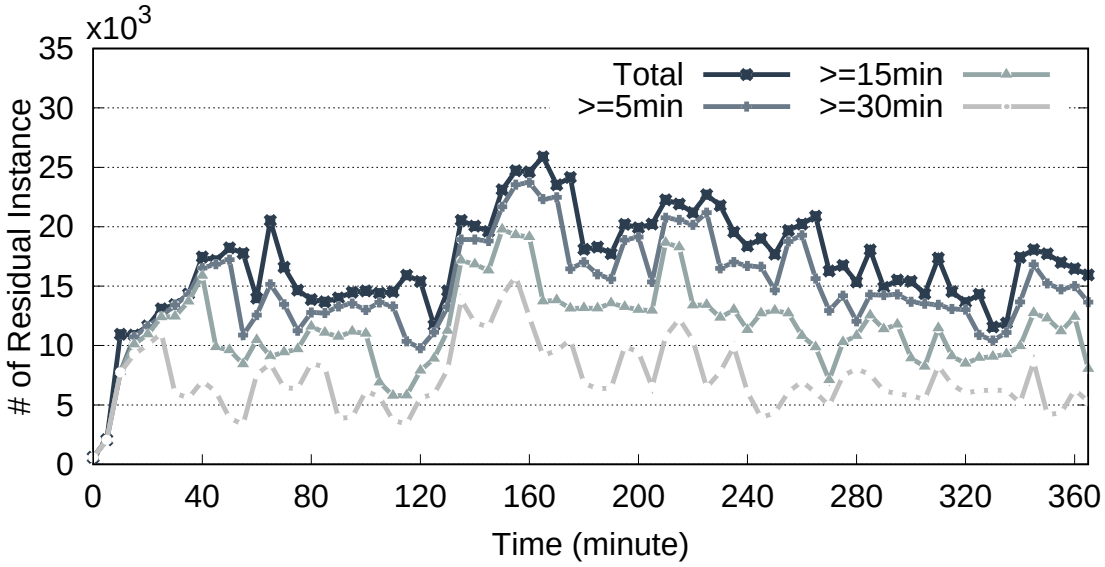


Figure 2.2: The number of residual instances

which is bounded by T_{cycle}/T_{actual}; given the actual usage time is usually unpredictable, the smaller the T_{cycle}, the less likely a cloud user will overpay the billing. Had the cloud providers adopted an ideal "per-second billing", the billing would have been efficient. Unfortunately, *per-hour* cycle is still the dominant billing model in the current IaaS market (e.g. Amazon EC2)[1]. Although there exist cloud providers who offer per-minute billing after an initial time interval[2] such as Microsoft Azure Cloud and Google Compute Engine, their resource offerings can vary compared to EC2's [4][3][6]. In addition, we conjecture it is a business decision for the per-minute billing cloud platforms to offer more competitive pricing schemes than their market opponents [2] even if those schemes could yield a lower profit margin as we will explain in what follows.

The above analysis raises the question: why IaaS providers favor long billing cycles? In addition to other potential reasons, we conjecture that a longer billing cycle will help compensate and reduce cloud providers' operational costs, especially the costs for instance provisioning (e.g., instance creation, decommission, VM image transfer, boot-time operations, scheduling costs, etc.). To be specific, first, we can infer from Fig. 2.2 that the duration of user jobs vary substantially with the majority being short-term ones. We then extract the history a user's job requests during a 75-minute interval from the trace as depicted in Fig. 2.3. Supposing this user will create an instance and runs several jobs spanning across our examined time interval; when the billing cycle is an hour, the user may subse-

[1]As shown later in this chapter, even when the billing cycle is much shortened (e.g. 15 minutes), our solution can still provide substantial cost savings.

[2]The initial time interval is usually 10 to 15 minutes, also leading to potential billing inefficiency problems.

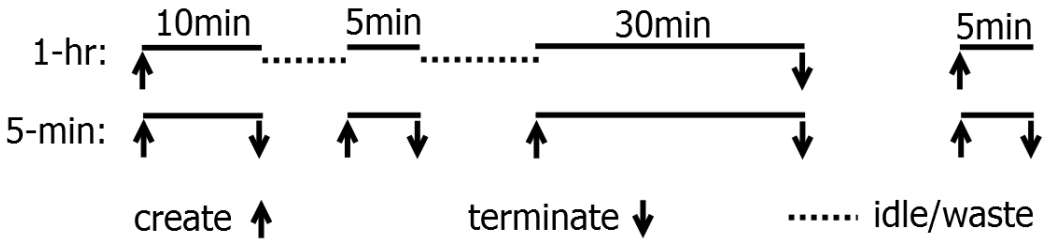


Figure 2.3: Billing cycle: 1-hour vs. 5-minute

quently create two instances with the first one covering three jobs in the first hour (with a waste ratio 1/3). When the billing cycle is shorter (e.g., 5 minutes), the user is allowed to timely terminate the instance to avoid unnecessary billing cycle charges and create a new instance upon the arrival of the next job (with a zero waste ratio). On the provider side, however, the shortened billing cycle leads to 2x more instance creations and thereby surged provisioning costs.

Despite it being an ideal case for users who have precise cost management, we can expect most of the users would follow such a pattern to avoid unnecessary billing if shorter billing cycle were available. Hence, a longer billing cycle could help reduce potential provisioning costs for cloud providers. It transfers the complexity of consolidating workloads in the time dimension to cloud users, which leads to the billing inefficiency.

2.2.2 Recycling Instances with Third Party

Given the resistance from the cloud service provider on shortening billing cycles, a better alternative is to consolidate user-supplied residual instances, trade their computing resources, and generate revenue. For brevity, we refer to the cloud users who "recycle" their instances as sellers; those who purchase resources as buyers. As illustrated in Fig. 2.1, the instance life-cycles can be augmented with additional states and transitions represented in dotted lines. Before a seller decides to stop an instance, it can launch a recycle request, with the necessary information to take over the instance and immediately clean its states. Once the instance reaches the *cleansed* state, it can be added back to the cluster management. Buyers can purchase resources from the cluster at a market-driven price, deploy their applications, and the resulting revenue goes to the sellers. Each managed instance is associated with a decommission deadline to ensure the seller not to be charged for another full billing cycle.

While there are instances available for recycling as shown earlier, the recycling is non-trivial to accomplish from both the system's perspective and the pricing perspective. As described in Fig. 1.2 with the hybrid virtualized layering structure in the public cloud, at the very bottom sits the physical layer with bare-metal machines, on top of which hypervisor is placed to abstract and manage the underlying hardware. These two layers are marked as the provider-space, as only the cloud providers have access to resource management on

these layers for security reasons. As such, public cloud users are not allowed to access these provider-space utilities, making resource consolidation difficult at the user-space. Even worse, the heterogeneity and highly transient nature of recycled instances may significantly limit the compatible workload types. In short, we are facing the following challenges:

- Managing a large amount of residual instances;
- Utilizing transient cloud resources efficiently;
- Identifying target workloads and providing platform-level supports accordingly;
- Determining the resource price and scheduling policies.

To address the first two challenges, there is a need for an additional virtualization layer on top of the existing one. It will allow tenant isolation on the same recycled instance as to achieve resource multiplexing. It is also a resource management layer where residual instances can be consolidated even with no support from the provider. We emphasize here that a *third-party* solution that does not rely on the provider for recycling is necessary: 1. The instance's billing cycle is fully paid regardless of whether the owner chooses to recycle it or not. 2. Providers could have higher operational costs when residual instances are recycled, since those instances will consume more resources as compared to when they are idle. As such, without explicit incentives, providers themselves are less likely to offer the recycling service on their own. More discussion on the incentives is offered in Sec. 2.5.

2.2.3 Why Hybrid-virtualization?

There are two potential candidates for building the additional layer, namely *nested virtualization*[3] and *hybrid-virtualization*. As depicted in Fig. 1.2, in the original user-space (left side of the figure), applications are run directly in the provider-managed VM. With nested virtualization, the applications are placed in the VMs managed by a nested hypervisor, which is run on top of the original VM. By doing so, each application in the same VM can now have their own virtualized resource pool and isolated runtime environment. This is seemingly a natural choice to facilitate resource consolidation in the user-space [45][16]. However, placing a hypervisor on top of another often results in excessive overhead and application performance penalties [51]. Further optimization would require tuning the underlying hypervisor resides in the provider space, which unfortunately is not allowed in the public cloud in general.

On the contrary, the alternative technique leverages both the traditional virtualization and the emerging lightweight containerization, which we refer to as hybrid-virtualization. As described in Chapter 1, in hybrid-virtualization, application containers are placed insides virtual machines. The container, in its simplest form, is a collection of OS kernel utilities

[3] `http://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen`

Resource Type (Benchmark)	Bare-VM	Hybrid-VM
CPU (7z Compression)	92.18Mbytes/s	92.26Mbytes/s
Memory (Sysbench, Read)	10.84Gbytes/s	10.85Gbytes/s
Memory (Sysbench, Write)	10.49Gbytes/s	10.08Gbytes/s
Disk (Bonnie++, Rewrite)	119.95Mbytes/s	118.22Mbytes/s
Network (Iperf, TCP Send)	126.60Mbytes/s	126.59Mbytes/s
Network (Iperf, TCP Recv)	126.53Mbytes/s	126.50Mbytes/s

Table 2.1: Performance of hybrid-virtualization

(e.g. `cgroups`) configured to manage the resources that an application uses. With containers, resources are monitored and managed through efficient function hooking devised in only the non-performance critical execution paths, thereby incurring much lower overhead.

To validate this, we provisioned four m4.2xlarge general purpose instances from Amazon EC2 cloud, powered by 8x vCPU on Intel Xeon Haswell processor, 32 GB memory, high-throughput SSD storage, and enhanced networking. In the non-containerized test (the baseline), benchmarks were run directly in the host VM. For container virtualization, we installed the latest version of *docker*[4], the mostly widely used container implementation. In Table. 2.1, we present the experimental results. As we can see, for CPU, containers are able to attain the compression speed within $\pm 0.1\%$ against the native VM. A closer look at the MIPS number confirmed that container does not consume more CPU cycles in compression. Similar observations can be made on the disk, memory, and network tests. These results indicate that hybrid-virtualization is able to complement today's public cloud infrastructure with another lightweight resource management layer, and thereby has the potential of supporting the instance recycling framework with limited performance penalties.

2.3 HARV: System Design and Implementation

We designed and implemented HARV, a third-party platform that **HAR**nesses hybrid **V**irtualization to realize the instance recycling mechanism. In this section, we first illustrate the design considerations of HARV. We show how HARV uses a two-level scheduling policy to simplify cluster resource management while improving its applicability. We show how it handles workloads with different persistence and duration requirements. Further, we design a truthful online auction mechanism to complement our system.

2.3.1 Cluster Architecture

In Fig. 2.4, we describe the architectural design of the HARV. Our cluster consists of recycled/residual instances from contributors, a *state manager* module in charge of cluster state updates, and a *Tier-1 scheduler* handles container allocation. *Tier-2 scheduler* and load

[4]Docker Container: https://www.docker.com/

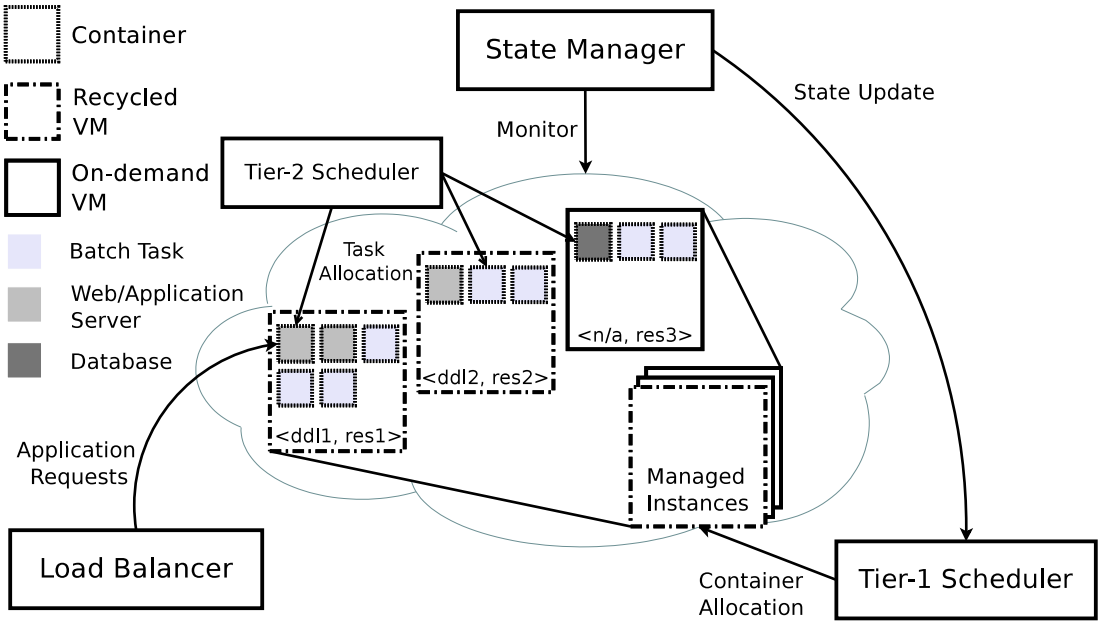


Figure 2.4: Cluster architecture with two-level schedulers

balancer are two complementary modules incorporating the two-level scheduling policies and can be customized by the buyers themselves. The state manager holds a consistent state information of the cluster, including details on each active residual instance, the available resources, and their decommission deadline. It is also in charge of detecting failures of residual instances and containers through keep-alive messaging. Upon the arrival, failure, or decommission of each residual instance, the state manager updates the state table and notifies Tier-1 scheduler.

Containers are allocated based on our auction mechanism. As shown in Fig. 2.4, five containers are placed in the leftmost instance, including two for web servers and three for batch tasks. Those containers may have different arrival time, duration, and ownerships. Meanwhile, each component of an application is encapsulated in different containers and scheduled across the cluster. By allowing such, we can achieve not only resource consolidation but also better resource multiplexing and statistical multiplexing in the user-space.

2.3.2 Two-level Scheduling

Another advantage brought by the hybrid-virtualization is allowing us to separate cluster-level scheduling and application-specific scheduling. Specifically, HARV does not schedule user-provided jobs directly. Instead, it only decides the container allocation on residual instances, improving resource utilization, and optimizing recycling efficiency. We employ the auction algorithm, described in Sec. 2.3.3, as the cluster-level scheduler (Tier-1) for this purpose.

Algorithm 1: Request dispatch algorithm

```
1: while Receiving request q_i do
2:    u_i = q_i.getUserID()
3:    if q_i.isContainerRequest() then
4:       S = MS.getCurrentInstances()
5:       τ_i = R_i.getResourceVector()
6:       if s_i = ∅ then
7:          T1.declineRequest(q_i)
8:       else
9:          MS.update(s_i, τ_i)
10:      end if
11:   else
12:      T2_i = MS.getT2Scheduler(u_i)
13:      T2_i.scheduleWorkload(q_i)
14:   end if
15: end while
```

Meanwhile, the application-specific scheduler (Tier-2) enables buyers to deploy customized scheduling policies. This is because, intuitively, users are the ones who ultimately decide how to effectively use their provisioned containers, since the usage pattern is best understood by themselves. Upon receiving a request, the cluster will run the dispatch algorithm to determine whether it is a container request or an application request, and consult to the (Tier-1 or -2) scheduler accordingly. We give the details of the request dispatch process in Algorithm. 1, with symbols described in Table 2.2.

Following the two-level scheduling, buyers need to submit their requests with specifications on containers, by which the Tier-1 scheduler decides where to allocate them, considering both the decommission deadline and resource constraint. Sample specifications are listed in Table 2.3 with specifications for database, web server, and batch job containers. Here CPU is expressed in relative units, where the higher the amount, the more CPU share the container can obtain. The maximum unit can be specified for a CPU/vCPU is 1024.

2.3.3 Tier-1 Scheduling and Instance Trading

An integral component of our cloud system is this container allocation scheduler (Tier-1) as well as a market *mechanism* to facilitate the trading of recycled instances. To this end, we built an auction-based instance trading module to meet both requirements. Current pricing scheme in cloud markets is still fixed price dominant which usually does not lead to an efficient market. Auctions have been widely used to determine the clearing prices that reflect the demand and supply relationship in the market [53]. Our system differs from the previously studied scenarios in that (1) the resources are inherently constrained by each instance that are holding them. Treating each type of resource as a monolithic resource pool like previous works did is not applicable to our system; (2) Instance pool in our system is dynamic. To this end, we carefully modify the state of art auction mechanism [22] into

Symbol	Description
MS	Cluster state management service
$T1$, $T2_i$	Tier-1 scheduler and Tier-2 scheduler supplied by user i
s_i	Instance i
τ_i	Resource vector $< ddl_i, r\vec{e}s_i >$
b_i	Bid with request i (when use auction-based scheduling)
u_i	The utility when b_i is satisfied
q_i	Request i: $< u_i, \tau_i, (b_i) >$
R_j^r	Capacity of resource type r in an instance j
S	Total number of instances
R	Number of resource types
T	The allowable running time of the auction
L_r, U_r	Lower and upper bound of per unit resource valuation
t_{min}	Minimum requested time of all bids
t_i, t_j	Requested time for bid i and residual time for instance j
d_i^r	Demand of resource type r in a bid i

Table 2.2: Table of notation

our problem. The detailed algorithm is presented in Algo. 2, where the symbols used are listed in Table. 2.2.

We set binary variable $x_{i,j}$ equal 1 if a container request i is allocated to instance j, otherwise, this container request will not be satisfied by HARV. Our unit price updating function is defined as $\lambda_{r,j} = U^r \frac{t_{min} L^r}{2TRSU^r}^{\beta}$, where $L^r = \min_i \frac{b_i}{d_i^r}$, $U^r = \max_i \frac{b_i}{d_i^r}$. The intuition behind this pricing function such is that the smaller the β, the fewer resources are left in the system. Once β equals zero, the marginal price is set to be the upper bound of the user's value per unit of resources. Under such circumstance, no bid can win the auction, guaranteeing the capacity constraint is satisfied. We are allowing as many requests as we can to be satisfied by the platform in the beginning, and becoming more conservative with the diminishing of resources.

Though the competitive ratio claimed in the original mechanism cannot be guaranteed anymore[5], our modified mechanism can still guarantee *truthfulness* and *individual rationality*, two important economical properties of a good auction. The individual rationality is guaranteed by our designed algorithm by ensuring that the utility for each selected requests is nonnegative. Since the pricing scheme of this mechanism falls into the family of *sequential posted price mechanisms* [18] in which truthful bid reporting is a dominant strategy, our algorithm guarantees the truthfulness in bid value.

[5]While we leave the design of a more competitive mechanism as future works, our current mechanism is able to achieve high social welfare and cost savings, as we will show in Sec. 2.4.

Algorithm 2: Online auction algorithm (OA)

Initiate $\lambda_{r,j} = \frac{t_{min} L^r}{2TRS}, x_{i,j} = 0, L^r, U^r$
while Receiving bid i **do**
  Calculate utility:$u_i = b_i - \sum_r \lambda_{r,j} d_i^r$
  if $u_i > 0$ **and** $t_i \leq t_j$ **then**
    $j^* = \arg\max_j (b_i - \sum_r \lambda_{r,j} d_i^r)$, $x_{i,j^*} = 1$
    $p_i = \sum_r \lambda_{r,j^*} d_i^r$
    Update dual variable: $\beta_{j^*}^r = \frac{R_j^r - \sum_i x_{i,j} d_i^r}{R_j^r}$,
    $\lambda_{r,j^*} = U^r \frac{t_{min} L^r}{2TRSU^r}^{\beta}$
  else
    $x_{i,j} = 0$
  end if
end while

2.3.4 Details in Resource Allocation and Sharing

HARV relies on the `cgroups` kernel feature to enforce the resource allocation decision made by the Tier-1 scheduler, and the `namespace isolation` kernel feature to enable sharing of resources among multiple buyers on the same residual instance. Specifically, each buyer's workload is encapsulated in a container which is associated with a resource vector $\vec{res}_i$ given in the buyer's request. Through container management tools, HARV translates the resource vector into corresponding *control groups* (cgroups), a collection of kernel controllers for system resources including CPU, memory, network and disk I/O. These controllers are assigned to the container runtime in the form of function hooking. When the container starts running, its resource access will trigger the corresponding hooks to ensure that the container does not use more than its resource share. Further, each container will be assigned a unique set of resource identifiers for its PID, IPC, network, and file system etc., providing it a runtime environment isolated from other co-located containers'. HARV creates a software bridge to allow co-located containers to share the host VM's network (with packet forwarding, NAT, and DNS configured).

2.3.5 Relocating/Migrating Containers

HARV is able to handle long-term task/containers through container migration. It configures the Linux `CRIU` (Checkpoint/Restore In Userspace) utility to checkpoint a running container, creates image files, sends those files to the next running destination, rebuilds and restarts the container. An advantage of this approach is that applications usually have dependencies such as OS binaries, third-party packages etc., while the container is able to encapsulate those runtime dependencies, making it convenient to restart an application without manually reconfiguring underlying host instance. Besides, buyers themselves (or the Tier-2 scheduler) can handle the instance decommission through data migration.

Job Type	Duration (min)	Persis-tence	CPU (Units)	Memory (MB)	Network (Mbps)	Storage (GB)	Port
Web Front-end	35	No	256	100	200	0.1	80
Database	n/a	Yes	256	500	200	50	n/a
Sysbench	35	Yes	1024	512	100	0.5	n/a

Table 2.3: Sample container request specifications given job types

For batch tasks, migration can be efficient given it preserves computed results. For applications such as web front-end server, when instance decommission occurs, instead of migrating containers, a perhaps more efficient way is to simply treat it as container failures, and reassign the job to containers launched in other instances. It is worth noting that, in current version of our platform, there will be a service downtime from a few seconds to minutes depending on the check-pointed image size. Although batch tasks should not be affected much, service downtime may not be tolerable for other user-facing applications.

2.3.6 Target Workloads

HARV is an ideal platform for running short jobs or the ones with limited persistent data. A variety of applications fit in this category, either in data processing including MapReduce accelerator [21], or the web front-end servers. We categorize the potential workloads for HARV and handling approaches based on their persistence and duration (long-term when user-specified duration exceeds the maximum allowable residual hour) as follows.

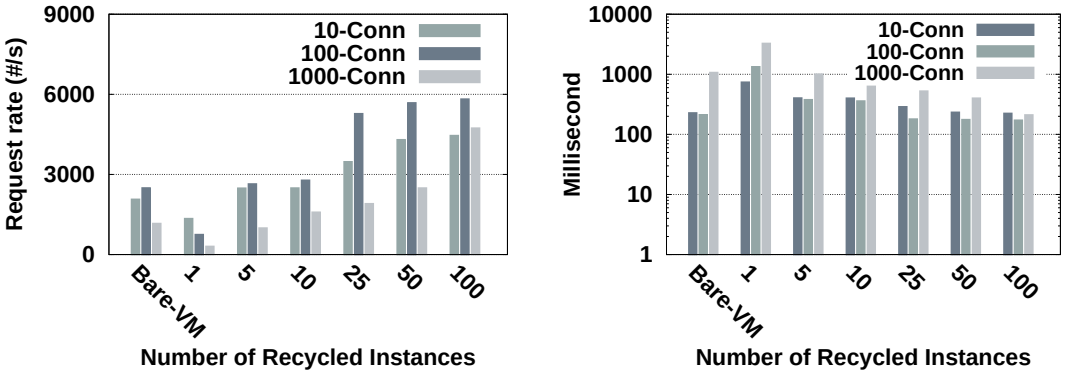
Long-term Stateless and Short-term Stateless HARV runs them in recycled instances and handles instance decommission through migrating or replicating containers (treats the decommission as an instance failure).

Short-term Stateful A migration deadline will be set for tasks of this kind. The larger the amount of state data, the earlier it is set prior to decommission deadline.

Long-term Stateful Since frequently migrating these jobs can be cost-prohibitive, our current version of HARV handles them by provisioning on-demand or reserved instances from the cloud provider, e.g. the database server shaded in dark gray in Fig. 2.4. Buyers can also launch those jobs in their own (non-recycled) public cloud instances while linking them to the accelerators deployed in recycled instances.

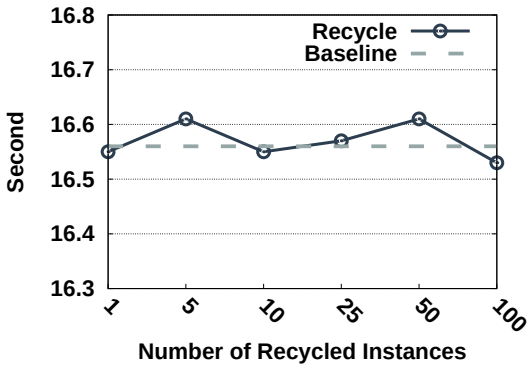
2.4 Evaluation

In this section, we present the results of system-level benchmarking and trace-driven simulation on HARV. We show that HARV is able to attain high application performance with substantial cost savings.

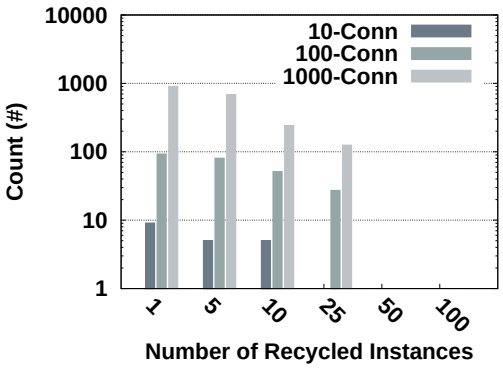


(a) Average Request Rate

(b) Average Request Completion Time



(c) Average Batch Task Completion Time



(d) Average Queue Length in Load Balancer

Figure 2.5: Real-world web application performance on HARV

2.4.1 System-level Evaluation

Prototype and Benchmarks Setup

We deployed a prototype of HARV with Amazon EC2 public cloud. We used *docker* as the containerization tool for the hybrid-virtualization setup. We run the master node that accepts instance recycle requests and hosts Tier-1 scheduler in an On-demand `m4.2xlarge` instance. We configured Amazon ECS service[6] to handle cluster state management (namely the state manager module). We modified its agent program to integrate it with the Tier-1 scheduler. Notably, except for the state management module, no other EC2 services were used in our system. Since such a module is commonly available in major cloud providers[7], HARV can be easily ported to other cloud platforms. Our testing cluster contains a maximum of a hundred residual `m4.large` instances. We chose the following two representative types of workloads:

Multi-tier Web Service: We used the RuBBoS[8] on-line forum benchmark to model the multi-tier application service. The number of containers provisioned for running the web and application servers is equal to the initial amount of recycled instance whose specifica-

[6]Amazon ECS: https://aws.amazon.com/ecs/

[7]Azure Container Service Cluster: https://azure.microsoft.com/en-us/documentation/articles/container-service-deployment/

[8]RUBBoS Bulletin Board Benchmark: http://jmob.ow2.org/

tions are shown in Table 2.3. We set up a load balancer for the web servers and deployed an emulated HTTP client[9] on an on-demand `m4.large` instance to request web pages from the server with different numbers of concurrent connections. We selected the average request rate and average request completion time as the performance metrics. We also sampled and calculated the average queue length in the load balancer.

Batch Task: We created batch workloads by devising a script that runs `sysbench` multi-threaded benchmark repeatedly (with a short sleep time between each run). We provisioned its container as specified in Table 2.3.

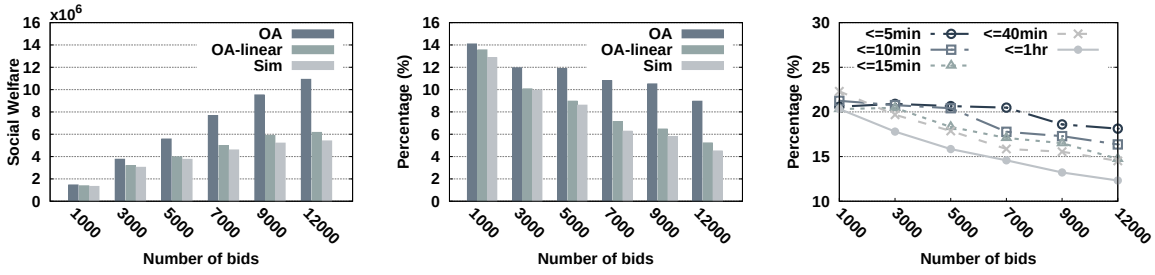
We began the test by running only one recycled instance with one web server container and one batch task container requested. We used the HTTP client to launch page requests with 10, 100, 1000 concurrent connections consecutively. We collected the average request rate, throughput, request completion time as well as the number of queued requests in the load balancer. To ensure fairness, we waited until the load balancer queue was emptied before starting each new test. The client emulator is placed on an on-demand instance within the same cloud region in order to minimize the interference from the network. We then changed the number of recycled instance, the number of server container, and the number of batch task container to 5, 10, 25, 50, and 100, and repeat these tests. Finally, we obtained the baseline performance for both benchmarks by running each of them in a single, non-containerized `m4.large` instance.

Results

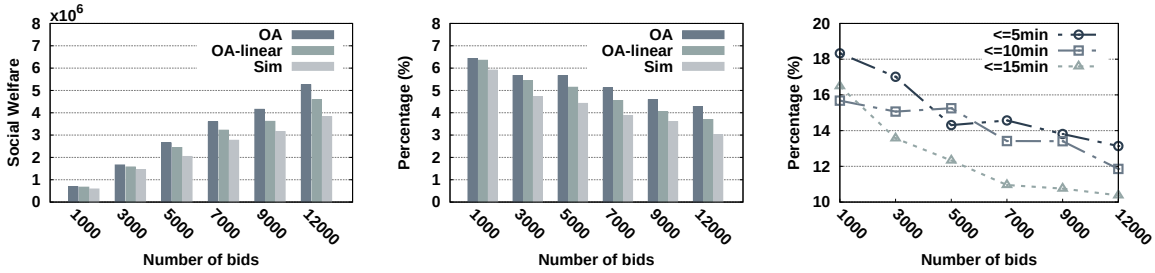
We present the benchmark results in Fig. 2.5. As shown in Fig. 2.5a, the baseline performance with a single `m4.large` instance (the "Bare-VM") is considerably higher than the single recycled instance case. This is due to, in the former case, the web servers being allowed to use all of the VM resources; whereas in the latter the servers are run in containers, and they have to share the resources with other co-located containers, thereby experiencing lower performance. Nonetheless, the average request rate immediately catches up with the baseline when there are five recycled instances and more. The effect of scaling out is also significant when there are more concurrent connections. For example, when the connection is 1000, an additional 50 recycled instances doubles request rate from around 2500 per second with 50 instances to near 5000 with 1000 instances.

Similarly, in Fig. 2.5b, the more recycled instances joining the HARV cluster, the less the time to complete requests, with the average request completion time dropping from above 3000 ms the highest to 300 ms the lowest for 1000 connections. Particularly, when the number of recycled instances is higher than 10, the request completion time stays lower than the baseline across all concurrent connection settings. To confirm the effect of scaling, we depicted the average queue length in the load balancer in Fig. 2.5d. As can be

[9]Apache Benchmark: https://httpd.apache.org/docs/2.4/programs/ab.html



(a) 1 hour Lifecycle Social Welfare (b) 1 hour Lifecycle Cost Saving (c) 1 hour Lifecycle Cost Saving for Short Jobs with OA



(d) 15 minutes Lifecycle Social Welfare (e) 15 minutes Lifecycle Cost Saving (f) 15 minutes Lifecycle Cost Saving for Short Jobs with OA

Figure 2.6: Comparison of different online mechanisms in social welfare and cost saving

seen, when the number of recycled instances is lower than 10, a substantial amount of requests are buffered in the load balancer's queue, especially when the system experiences high concurrent connections. This high-buffering leads to the excessive delay in the request completion as observed in Fig. 2.5b. With more recycled instances and more web server container allocated, the queue length plummets, because requests can be immediately dispatched to available or idle servers. These results indicate that the additional container layer poses a minimal impact on user-perceived application performance. Considering the low (monetary) cost of HARV containers, HARV is a good choice for web service providers to provision for demand peaks.

For the performance of the batch tasks in Fig. 2.5c, the average job completion time stays almost unaffected throughout the experiments. As aforementioned, we assigned a better part of VM resources to the Sysbench container. This shows that in HARV, even if applications (with different resource usage patterns) share the same residual instances, HARV can still maintain their performance through differentiating the resources usage priorities. We attribute this achieved performance isolation to the use of containerization with each running container assigned an independent OS namespace and resource control groups (e.g. cgroups in Linux).

Finally, we measured the management module overhead. We found that when the recycled instance cluster is not trivially small, the overhead of running HARV is negligible. The management overhead (excluding the user-level overhead, e.g., the load balancer and Tier-2

scheduler) originates from the master node, state management module, Tier-1 scheduler, and the agent program on each recycled instance. In terms of monetary costs, the master node (running on `m4.2xlarge` On-demand instance in our prototype) costs $0.479 per hour, an affordable price (in the real deployment, HARV can transfer some of the revenue to cover this cost) that can be further reduced by provisioning the cheaper reserved instance as the module will be running constantly. There is no additional charge for the state management module from EC2[10]. In terms of performance overhead, HARV takes 67.5 seconds (averaged over 100 instances) to setup a recycled instance, a process that includes instance cleansing, agent program installation, and containerization; and 4.2 seconds to handle a request (the time between receiving the request and starting the container; averaged over 5000 requests). For the agent program, HARV consumes less than 5% of CPU, limited memory footprint, and network bandwidth. The results indicate that the time HARV takes to manage a recycled instance is considerably shorter than the billing cycle, leaving most of the residual instance time available to recycle.

2.4.2 Large-scale Trace-driven Simulations

Experimental Settings

In this part, we conduct simulations to evaluate the effectiveness and scalability of our trading module (Tier-1 scheduler). We select the publicly accessible Google Cluster trace [30] (also used in Sec. 2.2), consisting of 3,535,030 entries, reporting each tasks' ID, active time, normalized resource demand (CPU, Memory), as well as task types, in an approximately 6 hours period. The time interval between each report update is 5 minutes. We identified 176,580 unique tasks after removing the reported anomalies, combining different entries that belong to the same task and calculating their durations.

To simulate the residual instances, we firstly assume each task request will be handled by a single on-demand VM/instance, and compute the corresponding residual instance information, including the time it being recycled as well as the residual hour according to Formula 2.1. Our event-driven simulator read the entries sequentially while checking the "current time" of the cluster; it adds and removes a residual instance to simulate the recycling and decommissioning process. Each request is submitted to the scheduler; unsatisfiable requests are simply omitted.

Performance Metrics

We use cost saving and social welfare as our performance metrics. Cost saving is defined as the percentage of saving can be achieved by using our trading system compared with directly buying on-demand instances. Social welfare is the sum of utilities of all users and

[10]Amazon ECS Pricing: https://aws.amazon.com/ecs/pricing/

the auctioneer (i.e., $\sum_i (b_i - \sum_r \lambda_{r,j} d_i^r + \sum_r \lambda_{r,j} d_i^r) = \sum_i b_i$), as defined in Algo. 2, an indicator on how efficiently our system allocates resources to users who want them most. We test the system performance under different instance lifecycle. We choose 1 hour since it is one of the current prevalent instance life time settings. We also choose 15 minutes as the lifecycle to reflect current trends in designing fine-grained resource provisioning scheme in academia.

Trading Systems Compared

We compare our instance trading system with other one-off sale markets where an instance will only be sold once (i.e., no recycling will be involved). We implemented the online auction algorithm (Sec. 2.3.3) in our trading system, whereas we adopted two other allocation algorithms in the one-off market. First, we adopted the similarity-based scheduling policy (Sim). In Sim, the vector similarity is computed between the container request vector and the instance vectors, and the instance with the highest similarity score is chosen to satisfy the request. Sim is a representative heuristic that being frequently used in designing cluster scheduling algorithms [28]. The second algorithm is the online auction mechanism with linear dual variable updates (OA-linear). In OA-linear, we change the dual variable update function in Algo. 2 to a linear function to validate the effectiveness of the original, exponential function. Finally, the pricing scheme in Sim is fixed, whereas OA and OA-linear both implement dynamic pricing.

Results

We present the experimental results in Fig. 2.6. First, as shown in Fig. 2.6a, given 1-hour lifecycle (current EC2 billing cycle setting) the proposed trading system with OA algorithm consistently achieves higher social welfare than the other two methods in a flat-rate market. Further, similar observations can be made in Fig. 2.6d where the lifecycle is reduced to 15 minutes. Notice that social welfare using 15 minutes lifecycle decreases when compared with the 1-hour counterpart is because we have fewer requests in the trace to be satisfied by the 15 minutes long instance. Though smaller billing cycle results in fewer resources available on the market to accommodate container requests, OA still achieves considerably higher performance than the other two methods.

Second, OA is able to maintain the cost savings even when the resource contention is high. In Fig. 2.6b, the cost saving of OA achieves 14% at 1,000 bids and sustains over 10% except for the 12,000 bids scenario. For the other two methods, however, the cost savings drop from nearly 14% (OA-linear) and 13% (Sim) to below 10% when there are more than 3,000 bids, and plummet to around 5% at 12,000 bids. Cost savings of OA in 15 minutes lifecycle in Fig. 2.6e also exhibit similar superiority. Notably, even OA-linear implements a dynamic pricing scheme and Sim a fixed one, OA-linear achieves no better social welfare

and cost savings than Sim. This confirms the importance and superiority of the pricing function in OA.

Third, in Fig. 2.6c and Fig. 2.6f, we intend to show the cost saving effects of our system to the jobs with different lengths. In the 1 hour lifecycle, our system constantly brings over 15% cost saving gains to all jobs with less than 40 minutes duration in all tested scenarios. Jobs with less than 5 minutes duration maintain around 20% cost saving. As we have explained before, the reduction of lifecycle brings less space for requests consolidating, which leads to smaller cost savings. However, jobs with less than 10 minutes duration still can benefit from 15% cost saving in 1000 bids to 12% cost savings in 12000 bids. As a conclusion, our auction-based trading system can achieve significant performance gain as compared to those one-off markets with either flat-rate or dynamic-rate.

2.5 Discussion

Before concluding our paper, we discuss the following issues pertaining to the adoption and practicality of the instance recycling service.

Provider's Incentives and Support: Although HARV tackles the general, third-party instance recycling problem where we assume the absence of cloud providers' support, there are indeed incentives for providers to support such service. Similar to Amazon EC2's spot instances (or Google Compute Engine's preemptible VMs), recycled instances is a cost-effective choice for certain types of workloads (see Sec. 2.3.6). They both allow users to buy non-standard computing resources with a (likely) much lower price. On the other hand, there are major differences between recycled and spot instances. First, recycled instances can not only save costs for users who want to buy resources but also those who sell them, i.e., the residual instances owners. Second, unlike spot instances, recycled instances do not preempt workloads. They allow users to run workloads that are not interruptible. Third, the resource offering of recycled instances is container as opposed to VM in spot instances. As such, while spot instance has become a widely used service, cloud providers can exploit recycled instances as another form of differentiated, value-added service to attract diverse user groups, gain extra revenue, and further improve their resource utilization. Cloud providers can either cooperate with a third-party instance recycling platform or build one themselves. Within the extent of our knowledge, HARV is the first work that addresses the motivation and technical challenges for building such service.

Trust and security issues: Trust and security issues have been one of the biggest concerns over cloud computing in general [42]. On the one hand, HARV targets for major public cloud deployment only and thereby these issues are not as pronounced as in other platforms relying on private, customer-supplied resources (e.g., from a private cloud or PCs) [48]. On the other hand, instance recycling indeed introduces new trust and security challenges. For example, residual instances suppliers and buyers should not have each

other's data. In addition, malicious workloads should be prevented from sabotaging other co-located workloads. HARV relies on containers to provide resource isolation as discussed in Sec. 2.3.4. At the policy level, the supplier is required to grant root privileges to HARV in order to successfully submit a residual instance to HARV (and they can choose to wipe out their data beforehand). HARV will then drop the supplier's root privileges and limit the local container manager to root access only (i.e., the docker daemon in our prototype). Notably, even if suppliers give up the root privileges, they will still be able to terminate the instance or modify its running through the cloud providers' API. Although HARV can blacklist the untrustworthy instance suppliers, a well-rounded solution would require cloud providers' support. Moreover, advancements on cryptography allow more types of privacy-sensitive workloads to be run on third-party platforms such as HARV. For example, Order Preserving Encryption has been effectively used to preserve client's confidentiality for middlebox workloads running on the third-party cloud [34]. While in this chapter we focus on other design dimensions of instance recycling, we will continue to address the trust and security issues in our future works.

2.6 Related Work

Both cloud providers and users are faced with the resource inefficiency problem. While cloud providers have the luxury of improving their resource provisioning methods, cloud users may only leverage existing pricing options or application-level scheduling to alleviate the issue. HARV provides an alternative solution for cloud users by enabling them to resell their underutilized resources.

Resource Provisioning: Facing the resource inefficiency in current IaaS cloud, a substantial works have been done on designing fine-grained resource provisioning methods [53][50][37]. Most of these works focus on solving resource allocation problems from the provider's perspective, and they all operate on VM level. HARV can be treated as a cloud provider. Different from these works, HARV operates on containers for resource provisioning and does not belong to the IaaS category. The advancement of containerization techniques opens opportunities for cloud providers to supply flexible and efficient cloud resource offering. Containers bring less CPU consumption, less reboot time, smaller image size as compared to hypervisor-based VMs [36]. They also introduce little application performance overhead as shown in our paper.

Pricing Options: Extensive works tried to improve cost-effectiveness for cloud users by leveraging and improving the existing pricing options [21][55][49]. Chohan *et al.* [21] explored the Spot Instance option to accelerate MapReduce jobs with greatly reduced monetary cost. Wang *et al.* in [49] exploited the Reserve option and proposed a dynamic instance acquisition scheme that minimizes the broker's cost to accommodate given de-

mands. Instead of exploiting existing billing options, we tackle this issue by introducing a new cloud instance type, which can be used jointly with those existing frameworks, too.

Customer-supplied Cloud: Wang *et al.* [48] studied a customer-supplied cloud (SpotCloud), where resources are provided from user's physical machine instead of the public cloud. HARV can be viewed as a customer-supplied cloud, too. The major difference between HARV and SpotCloud is that HARV's resources are from the public cloud only. Compared to the residual instance we studied, a SpotCloud machine could introduce greater performance variance and trust issues.

Chapter 3

Energy Efficiency in NFV Empowered Content Distribution

3.1 Overview

Multimedia traffic, in particular video traffic, accounts for the largest share of all traffic in today's Internet. By 2020, an estimation of 82% of consumer Internet traffic will be attributed to video streaming according to Cisco [12]. Multimedia traffic is highlighted by its volume, variety, multicast nature and additional QoS constraints. Downstream towards consumers, multimedia traffic often traverse through middleboxes (i.e., *network functions*; in this chapter, we use terms middlebox, network function, and network appliance interchangeably), such as WAN (Wide Area Network) optimizers, transcoders, content caches, NATs (Network Address Translator) and traffic shapers, undergoing additional data processing imposed by content providers and/or distributors.

Traditionally, middleboxes are implemented as dedicated, vendor-specific hardware, an approach that leads to escalated management costs and an inefficient use of infrastructural resources [44]. Network Function Virtualization (NFV) is an on-going movement led by global network operators that aims to migrate network functions from dedicated hardware to off-the-shelf, general-purpose servers. Potential NFV operators include cloud providers, network providers, and Telco CDNs (Content Distribution Network); transitioning to NFV can help them alleviate the capital expenditures (CAPEX) and operating expenses (OPEX). For instance, an attractive deployment target for NFV is carrier network edge or their Central Offices (see: `http://opencord.org/`).

While there are comprehensive works surrounding NFV's performance and architectural design [44][31][41], its energy cost has rarely been studied. Given the extensive presence of middleboxes, energy cost (as a major contributor of OPEX) is likely to become one of the deciding factors in the operators' adoption of NFV. In this chapter, we show that NFV powered middleboxes are prone to high energy overhead when delivering multimedia content.

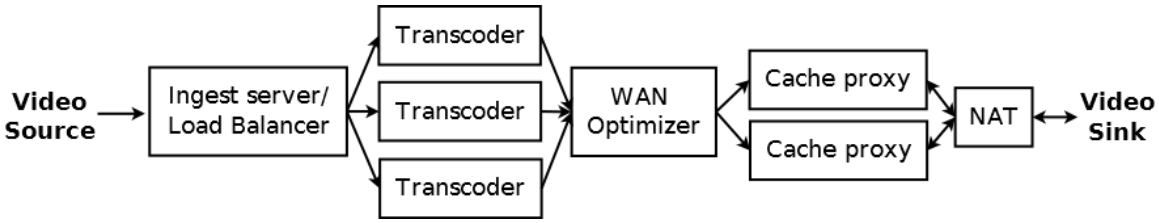


Figure 3.1: A holistic view of middlebox traversal of a video stream.

In particular, we find there exists energy inefficiency in the data forwarding component of major NFV platforms. We demonstrated this inefficiency is inherent to its design that excessively uses CPU cycles to attain high performance. Since multimedia traffic is persistent throughout the streaming session and usually impose additional QoS constraints, existing energy saving methods may not function well. Based on these observations, we propose an outline of a power management framework for NFV-based multimedia content delivery. We show that CPU frequency scaling can achieve promising energy savings without compromising the performance of multimedia applications.

3.2 NFV-based Multimedia Content Delivery: the Energy Cost

NFV is often considered as an extension of cloud computing to the networking domain. As a multimedia content provider or distributor, however, transitioning to NFV may not be as easy as deploying a cloud application on its datacenter. Multimedia traffic often traverses through a series of middleboxes, as described in Figure 3.1. This simplified example captures how popular live broadcasting service providers, e.g., Twitch.tv, collect content (video broadcasters send streams to the ingest server), process content (video transcoding) and deliver content (via CDN) [52]. Two observations can be made here: first, there are different types of middleboxes [17] along the path, and they may possess various runtime characteristics such as resource usage and energy cost. Second, middleboxes are likely to be owned by different operators. When transitioning to NFV, the operators need to reevaluate their power budgets with the knowledge of energy costs for different middleboxes deployed from place to place.

To better understand the energy cost, we measured the power and bandwidth consumption of a NAT box with multimedia traffic is passing through. A typical NAT box modifies the IP header of in-transit packets to remap one IP address space into another. We chose the NAT box for its prevalence in today's Internet as well as its simplicity: being a network function with little application-layer processing, which provides a cleaner baseline for our future measurements. We set up an ASUS RT-AC68U router with NAT acceleration turned on for the hardware NAT, and the Linux `iptables` utility for the virtual NAT. We used

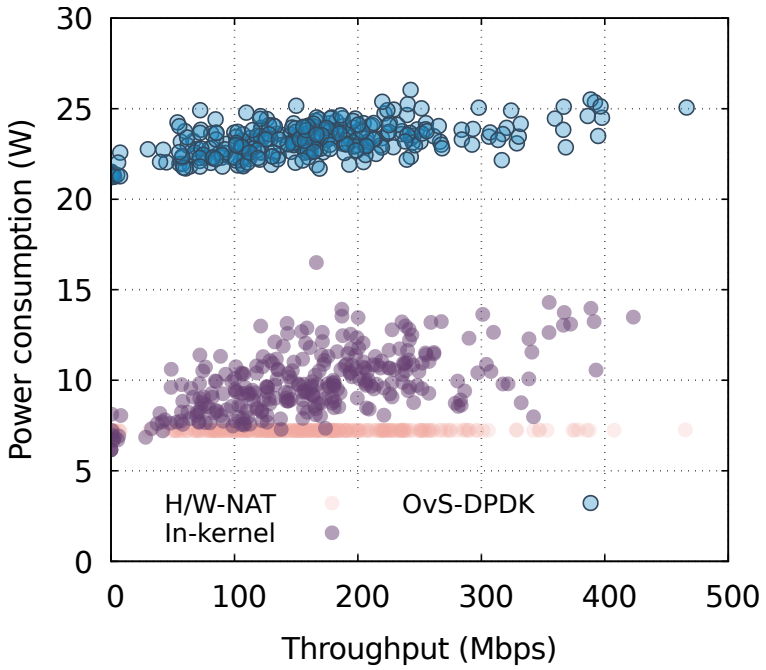


Figure 3.2: Throughput and power consumption of a NAT box when multimedia traffic is passing through. Three middlebox deployments in comparison: hardware NAT, virtual NAT with Linux in-kernel bridge, and virtual NAT with Open vSwitch and DPDK forwarding engine.

the `ffserver` (`https://trac.ffmpeg.org/wiki/ffserver`) to set up a multimedia broadcasting server and `ffplay` (`https://ffmpeg.org/ffplay.html`) as the client. The sample multimedia traffic consists of a 1080p video stream (H.264 encoded, 3 Mbps bitrate), two audio streams (320 kbps and 160 kbps bitrate) and meta data. We devised a script that consecutively creates 300 sessions (starting from 0 session and every 10 seconds 10 more sessions are added) in 5 minutes. More details on the testbed setup and data collection can be found in Section 3.3.

We plot the results in Figure 3.2. As shown, the hardware NAT, sitting at the bottom, sustains the power consumption of around 7.2 watts regardless of the network load. Sitting in the middle is the virtual NAT implemented with Linux in-kernel bridge, whose power consumption is roughly in proportion to the load. Although the virtual NAT achieves lower power consumption than the hardware NAT when idle (6.16 W vs. 7.18 W), its power consumption is considerably higher (average 9.84 W vs. 7.20 W) with the highest readings doubling the power cost. Finally, the virtual NAT implemented with Open vSwitch and Intel DPDK forwarding engine, a popular module used in NFV platforms [31][41], consumes significantly more energy than the other two setups (average 23.24 W), tripling the cost of hardware NAT. Moreover, *it consumes around* 22 *watts even when there is no traffic passing through.* The results indicate that energy cost may indeed become a concern for multimedia content provider and distributors when they transition to NFV.

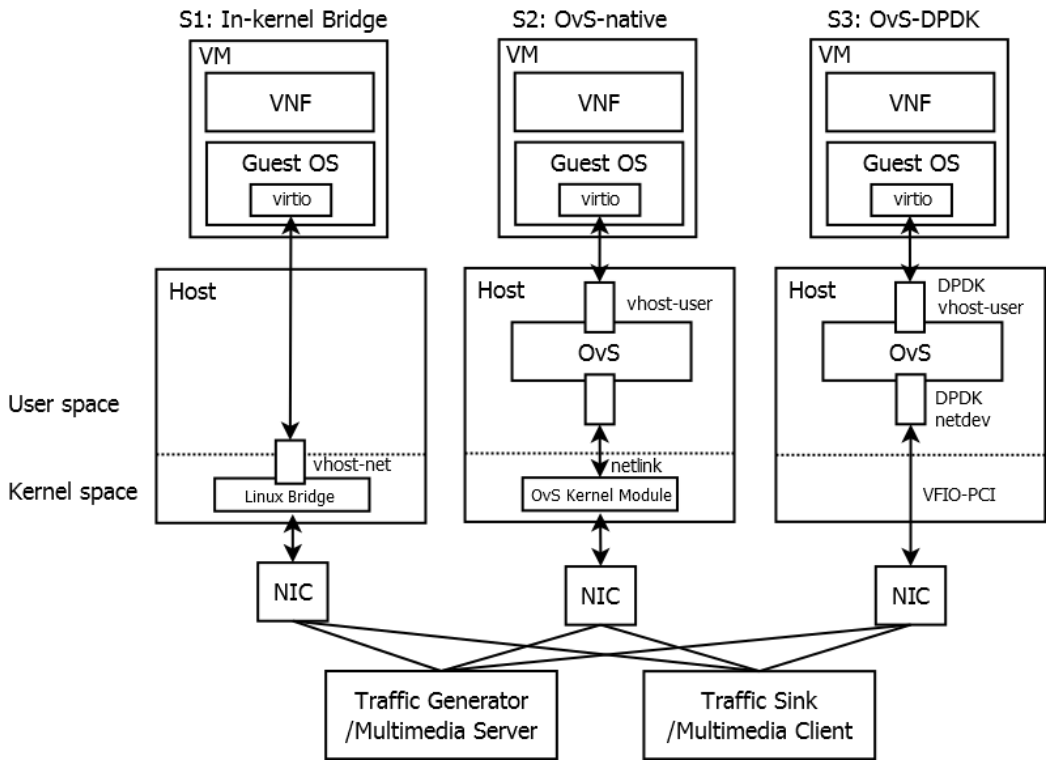


Figure 3.3: The NFV testbed consists of three dataplane setups: off-the-box in-kernel bridge, Open vSwitch with native and DPDK forwarding engines, and two servers to inject traffic. Each virtual machine (VM) emulates a physical machine, providing functionalities to run an OS and hence network functions.

3.3 Background and Testbed

By drawing a comparison with the hardware middlebox, one can see that the NFV products may greatly raise the energy costs when delivering multimedia content. To track the cause, we built a single-host NFV testbed as summarized in Figure 3.3. We use the following terms to refer to each component of the testbed:

- **Virtual Network Function (VNF)** A network function implemented by software and embedded in commodity server.
- **Virtualization Layer** An isolated runtime environment for VNFs, e.g., virtual machines (VM) or application containers; we chose the former and used Linux qemu-kvm utilities to create VMs.
- **Dataplane** The term originates from Software-defined Networking (SDN), which addresses the separation of network control plane (deciding how packets are routed) and the data plane (forwarding and/or processing packets). In some literature a dataplane may refer to any network component that operates on the traffic, including a VNF.

In this chapter, we find it convenient use the term to describe the combined module of a vSwitch and its forwarding engine *only*.

- **Virtual Switch (vSwitch)** A software program that facilitates network communications among VMs. Virtual switches often support SDN flow rule interface (e.g., OpenFlow), and can be thereby more versatile than a software bridge.

- **Forwarding Engine** A set of drivers and libraries that perform packet transmission between a physical NIC and the virtual NIC (and thus to the VNF).

We set up a midrange server with a 3.4GHz Intel i7 quad-core CPU and an Intel I350 Gigabit Ethernet Network Interface Card (NIC) as the host machine. We installed an additional Ethernet card to have an "out-of-band" control of the server, as in some of our configurations, the access of NIC will bypass the host's kernel entirely and cause the host to loss its IP stack functionality. We set up two physical servers as the traffic generator and receiver. The three physical machines are connected via a Netgear GS116NA Gigabit switch. Finally, we created the following dataplane setups (as shown in Figure 3.3):

Linux in-kernel Bridge The bridge is available in most of mainline Linux distributions, making it an ideal out-of-the-box solution for setting up an NFV platform.

Open vSwitch Open vSwitch (OvS) is one of the most widely deployed software switch in the market. It features advanced flow caching, fast packet classification supports, and compatibility with OpenFlow. In addition to its native kernel forwarding engine, OvS also supports third-party forwarding engines such as Intel DPDK (`http://dpdk.org/`), targeting to better line-rate performance. We built two versions of OvS from the source (`https://github.com/openvswitch/ovs`), one with the native forwarding engine (OvS-native) and the other with DPDK (OvS-DPDK).

Having two OvS versions with different allows us to track down the energy consumption in vSwitch and forwarding engine separately. Across all setups, we configured `virtio` (`http://www.linux-kvm.org/page/Virtio`) as the datapath between the host and VM.

3.4 Tracking the Energy Inefficiency

To begin with, we consider the following characteristics of multimedia traffic:

- **Continuous** Multimedia traffic are persistent, continuous and sizable flows that occupy the link bandwidth during a time period.

- **QoS constraints** Multimedia applications are *user-facing*. They impose additional and possibly time-varying QoS requirements such as end-to-end delay, bandwidth, and packet loss rate.

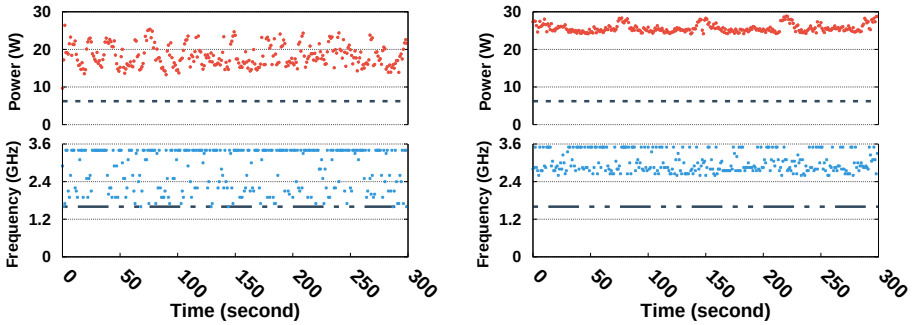


Figure 3.4: Power consumption and CPU frequency measurements when multimedia traffic traverse transcoding box on in-kernel bridge (top) and OvS-DPDK (bottom), with the common power manager of OS turned on. The baseline power consumption (6.22 W) is the host server running without any VNF or dataplane; the baseline CPU frequency is the lowest supported P-state frequency.

These characteristics are likely to contribute to the raised NFV power consumption and affect the efficacy of power management methods. For instance, due to the continuity of multimedia traffic, it is difficult to spot an idle time of the traversed middleboxes. Hence it may become prohibitive to use sleep modes and deprovisioning unused middleboxes to achieve energy savings. Due to the QoS constraints, power management frameworks must also take into account the application performance when making power tuning decisions, a problem that is unfortunately non-trivial [47]; whereas, for those background, non-user facing applications, the middleboxes and network traffic can be better scheduled to achieve energy savings with fewer concerns. In addition, multimedia traffic can trigger intensive computations at the middleboxes and lead to high energy consumption. In the previous experiment we evaluated the NAT box, a network function that does not touch the traffic payload (i.e. the content) and involves little application-level processing; while there exist middleboxes that are computational-intensive and pervasive such as the *transcoding boxes* and *content distribution boxes* [52][17].

In Figure 3.4, we depict the experiment of multimedia traffic traversing a transcoding box. We turned on the Linux `on-demand power governor`, a power saving module available and used in mainline Linux distributions. It tunes the CPU frequency based on the CPU utilization to conserve energy. As compared to the NAT box traversal, the energy cost is raised considerably: when the in-kernel bridge is used, the average power consumption is now close to 20 watts, doubled from the previous case ($\approx$ 10 W); for OvS-DPDK, it is now close to a staggering 30 watts. Besides, the majority of frequency is set to the maximum frequency (3.6 GHz) for both cases, indicating that the power manager may have limited influence on the overall energy savings (we offer discussion over this issue in Section 3.5.2). Moreover, we can see that the combined effect of dataplane and application-level processing on frequency and power consumption. The CPU frequency with OvS-DPDK stays above 2.4 GHz, hovering around 3.0 GHz and close to the maximum frequency. As a result, the consumed power is consistently higher than 20 watts, more than four times of the baseline.

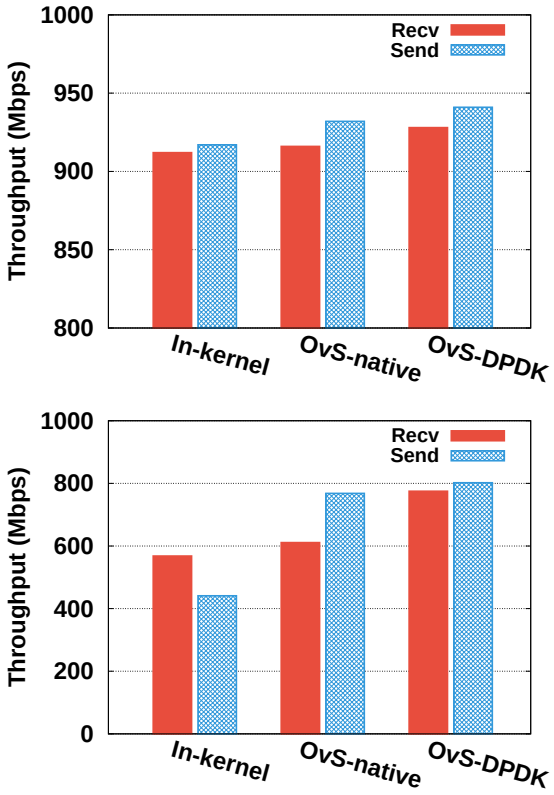


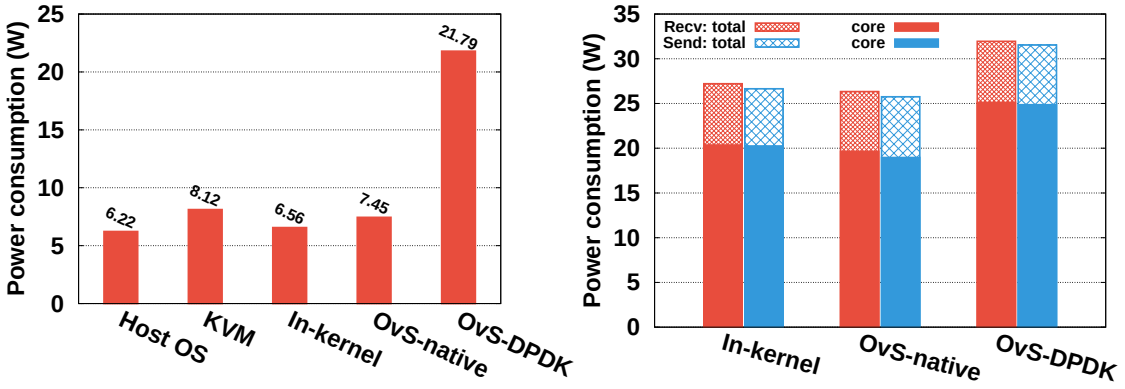
Figure 3.5: Performance comparison of three dataplane setups under different packet rates: MTU 1500 (Left) and MTU 500 (Right). Note the difference in the Y-axis scale.

3.4.1 Benchmarking the Dataplane

The experiment with real-world multimedia traffic indicates that NFV may introduce high energy overhead for content providers and distributors. They also hint us on potential energy inefficiency: the dataplane. Supporting the same VNF, the OvS-DPDK dataplane consumes much more power than the in-kernel counterpart (Figure 3.2). Yet it is not clear whether the vSwitch or forwarding engine is to blame.

We ran the following experiments to investigate these issues. We used `iperf` (iperf: `https://iperf.fr/`) network benchmark to generate traffic between the source and the sink. Using the network benchmark allows the traffic generator to conveniently saturate the link capacity and put the dataplane under heavy load. It also allows us to evaluate the send and receive behavior separately. Besides, we were interested in the effect of different packet rates. We adjusted the packet rate by changing the MTU at the traffic generator.

We used Intel RAPL (Running Average Power Limit) (`https://01.org/rapl-power-meter`) to measure power consumption. RAPL provides both the holistic power readings (cores, caches, and memory controller) as well as core-only power readings, allowing us to determine the energy contributions of core and non-core components.



(a) Standby (b) Busy

Figure 3.6: Power consumption comparisons of three dataplane setups in standby (without traffic) and busy (with traffic) scenarios and baseline for host OS and VM. MTU is set to 500.

We present the network performance results with three dataplane setups in Figure 3.5. At the default 1500-byte MTU, both OvS-native and OvS-DPDK achieve higher throughput in TCP sending and receiving than the out-of-the-box in-kernel bridge, although the performance difference is only marginal ($< 5\%$). When the MTU is 500, with roughly three times more packets to process, the performance of three setups now differs prominently. The TCP send throughput of in-kernel bridge plummets to 441 Mbps, less than half of the original performance. Although the other two switches also suffer from performance loss, they are able to attain much higher throughput. In particular, OvS-DPDK performs 2x better on TCP send and 1.5x better on receive than the in-kernel switch. Moreover, comparing the results of OvS-native and OvS-DPDK reals that the choice of forwarding engine has considerable impact over *packet receiving* performance, where OvS-DPDK increases the throughput by about 30% from OvS-native.

The power consumption results are depicted in Figure 3.6. The baseline power performance of our NFV testbed is about 6.22 watts when running the host OS alone; running a virtual machine in it will add about extra 2 watts. In terms of the standby power, the in-kernel bridge adds a negligible amount of energy and the OvS-native about 1 watt. The staggering number appears in the case of OvS-DPDK. Even when no traffic is in transit, OvS-DPDK consumes about 3 times more energy than the other two dataplanes, adding 250% energy overhead (15.57 W) on top of the host OS.

When they are used to forward traffic, the discrepancy between OvS-DPDK and the other two dataplanes shrinks, despite the fact that OvS-DPDK still draws about 25% more energy than the in-kernel bridge (400% of the idle host OS). From the core-only power readings (the total accounts for other subsystems such as memory controller, last-level cache, and graphic processor) we found that the CPU cores are the major contributor to the energy discrepancy. OvS-DPDK brings in about 5 watts more consumption in its use

of the core. In fact, without using DPDK, OvS-based dataplane achieves similar power consumption at standby and even less when running as compared to the in-kernel bridge.

3.4.2 Forwarding Engine is the Energy Hog

Multimedia applications are often throughput demanding; using the advanced forwarding engines such as DPDK can benefit the application performance. We confirmed this performance speed-up in the experiments, yet we also showed that high-speed forwarding engine leads to an undue amount of energy consumption. To better understand this issue, we conducted a profiling analysis of OvS-DPDK using `Intel VTune Amplifier` (`https://software.intel.com/en-us/intel-vtune-amplifier-xe`), a binary instrumentation toolset. Thanks to the open-source nature of DPDK, we are able to conduct detailed, code-level analysis.

To put the dataplane under full load, we launched `hping3` (hping3: `https://linux.die.net/man/8/hping3`) on both the guest VM and the traffic generator to inject bidirectional traffic at the maximum link speed. First, we found that there is a pmd_thread_main accounts for around 99.7% CPU occupancy (i.e., a full core utilization). By cross referencing to the source code, we found the hot-spots of this thread reside in the dpif−netdev.c:

```
2,850: error = netdev_rxq_recv(rxq, &batch);
```

This function call accounts for 51.6% of the CPU usage. There are also other hot-spots in the same function which we will omit here. By tracing back to the function's caller in pmd_thread_main(void ∗f_):

```
3,113 for (;;) {
3,114   for (i = 0; i < poll_cnt; i++) {
3,115     dp_netdev_process_rxq_port(pmd,
              list[i].port, poll_list[i].rx);
3,116   }
3,135 }
```

We found that the thread keeps spinning on this code block, indicating it is continuously monitoring and executing the receiving data path. When we repeat the experiment with OvS-native, we did not find such high CPU utilization and thread spinning. We then collected other statistics including the CPI value (*cycles per instructions*) and branch mis-prediction rate for both OvS-native and OvS-DPDK. We found that OvS-DPDK spends more CPU cycles in fetching and decoding the instructions than actually doing computations (CPI value: 0.693). And the majority of instructions are a result of bad speculation with a lot of mis-predicted branches (i.e., the "if" statements). On the other hand, OvS-native gives lower mis-prediction rate and yields much higher CPI value (1.364). The results demonstrate

that OvS-native, as compared to OvS-DPDK, uses CPU more efficiently by spending cycles on useful computations, although on average it spends more time to complete an instruction (due to the use of interrupts).

As a conclusion, we confirm that OvS-DPDK improves its performance by running a spinning thread that keeps polling for any newly received packets in the device buffer to attain better performance. This approach yields higher performance than the traditional, CPU-efficient interrupt based approach. On the other hand, the polling model also leads to an inefficient use of CPU cycles that causes the significantly higher power consumption than the interrupt based approach.

3.5 How to Manage the Power?

Our single-host experiments reveal energy inefficiency issues that lie in current NFV dataplane implementations. In reality, dataplanes are often deployed in a multi-host, multi-hop fashion, e.g., multimedia distribution (Figure 3.1), where energy inefficiency is likely to be amplified or even propagated, making it a network-wide problem. We summarize the following challenges in designing power management frameworks for NFV-based multimedia content delivery:

- At the single-host level, the framework should exploit the commonly available power tuning interfaces on the host machines. Since dataplanes and VNFs are real-time, data-intensive applications, the power tuning must be done carefully without risking performance.
- At the multi-host level, the framework should preserve the Quality of Service (QoS) of multimedia applications, which demands a global view of all transit VNFs performance when applying power tuning.

In this section, we provide the outline of a performance-aware power management framework. We then examined the energy and performance impact of CPU frequency scaling as the power tuning method.

3.5.1 Overview of a Power Management Framework

In Figure 3.7, we depict the outline of a proposed power management framework. The framework consists of an agent program deployed on each host server and a centralized power manager. The power manager can be viewed as a control plane service. It should query the NFV manager for the most updated VNF and the host server information. The manager should take charge of generating a *global policy* for network-wide power saving, based on the performance requirements of multimedia applications, stream characteristics, and the current NF status.

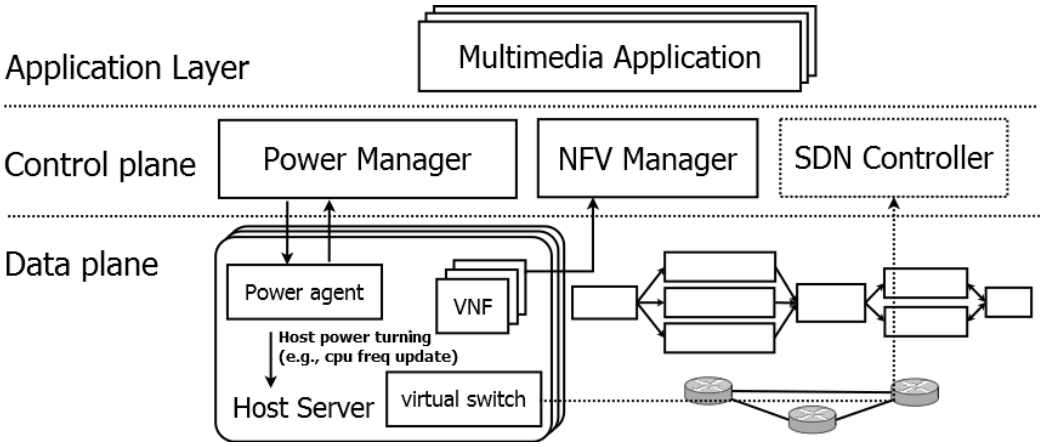


Figure 3.7: The proposed power management framework. It complies with the control and data plane separation of SDN. The power agent is deployed on each VNF host server at the data plane and a centralized manager that makes global power-performance tuning decision based on application-supplied policies. Despite sitting at the control plane, the power manager may as well leverage the APIs of other controllers.

3.5.2 Energy Savings Through CPU Frequency Scaling

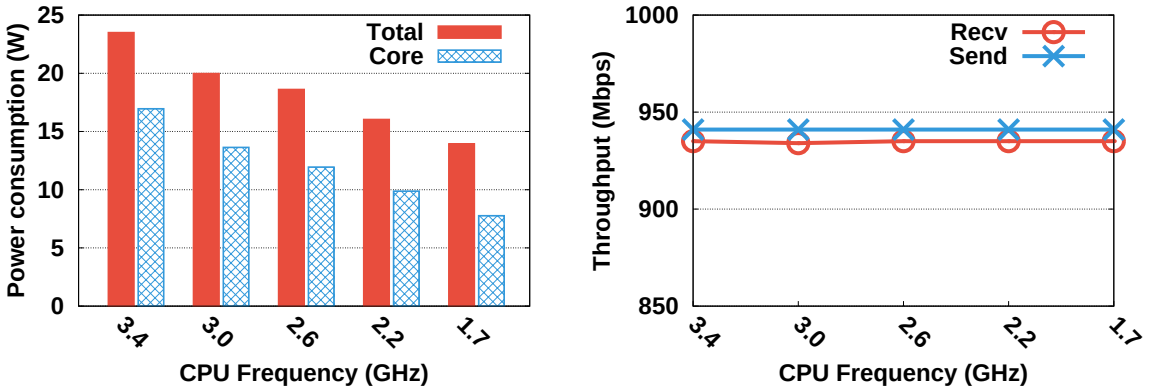
Given multimedia traffic are continuous and multimedia middleboxes usually shared by multiple streaming sessions, it can be rare to spot an idle time for the middlebox and its underlying dataplane. As such, power tuning methods that rely on OS sleep modes may not function well since they squeeze energy savings from the application idle time. To incorporate the QoS requirements, power tuning methods should also enable fine-grained power-performance trade-off and avoid QoS violations.

We decided that CPU frequency scaling could be a promising power tuning method for NFV host server energy saving. First, the interface has long been supported by modern CPUs and OSes, including most of the server-class machines. Second, it allows frequency tuning at per-core granularity and permits access in the userspace [47][43]. Although there exist some OS-level power management tools that exploit this interface (e.g., Linux power governors), they cannot be directly used in our case because computation-intensive applications such as DPDK's polling driver may run on a spinning thread and the OS will not be able to distinguish whether it is under heavy load or not.

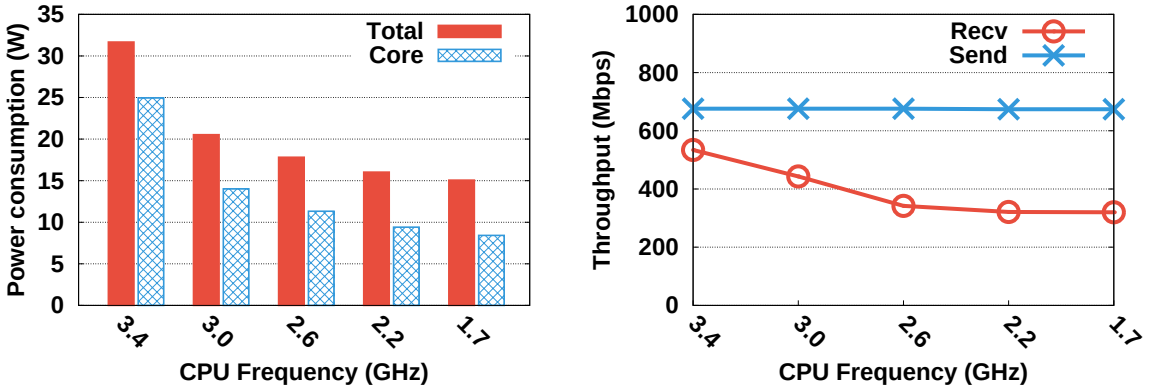
We repeated the iperf experiments in Section 3.4 with CPU frequency scaling. We devised a script to provide simple frequency control. As a sample power agent, the script probes the target frequency setting by gradually reducing the frequency until it sees the target throughput drops for more than a given threshold, i.e., finding the *Pareto frontier* of the power consumption and throughput. We leave the detailed design and implementation of the full-fledge power agent and manager in our future work.

To illustrate the effect of frequency scaling, we configured the script to iterate through the frequency range and collect the intermediate results. As shown in Figure 3.8, at the

default 1500 MTU, we see a power reduction from 23 watts at 3.4 GHz to under 20 watts at 2.6 GHz and 13 watts at 1.7 GHz with over 43% energy saving. As for the performance, the throughput for both sending and receiving stay the highest across the frequency range. When we change the MTU to 200, the power saving becomes more prominent with the total power consumption halved at 2.6 GHz. On the other hand, the frequency tuning starts to have an impact on the performance on the packet receiving. The throughput drops from around 534 Mbps at the maximum frequency to 320 Mbps at the lowest. We conjecture the performance penalty is caused by the overloaded PMD thread on the packet receiving path. The sending performance remains unaffected. During the experiment, the frequency control loop consumed less than 1% of CPU and negligible power consumption.



(a) MTU 1500



(b) MTU 200

Figure 3.8: Impact of CPU frequency scaling over power consumption and application performance at low packet rate (top, MTU 1500) and high packet rate (bottom, MTU 200).

Based on these preliminary results, we conclude that when the network load on data-plane is moderate or send-bound, it is performance-friendly to lower the CPU frequency to conserve energy; when network load becomes high and receive-bound, the frequency should be raised accordingly to avoid performance penalty. This leads to the idea of *energy proportionality*, the desired property stating that the power consumed in computer systems should

be in proportion to how much useful work they complete [43]. To incorporate this property in our framework, the power manager needs to extract the multimedia stream information and forward it to the power agents. A power agent will then exploit this information to estimate the workload in the next time frame and adjust the CPU frequency accordingly.

3.6 Discussion

Traditional middleboxes rely on hardware specialization, e.g., Application-Specific Integrated Circuit (ASIC), to attain high performance. At the computer architecture level, ASICs differ from general purpose processors in their use of *dataflow* model. It minimizes the overhead of instruction loading, where the program execution solely depends on the availability of input data at the logic gates. Dataflow model is thus well-suited for real-time, data-intensive applications such as packet forwarding and signal processing. On the other hand, hardware specialization often imposes a lacking of *flexibility*. It has longer time-to-market, higher (one-off) provisioning costs, and inconvenient patches and updates, whereas its virtualized counterpart, run on general purpose processors, excel in addressing those issues.

A main focus of the current NFV research is how to make reasonable trade-offs between performance and flexibility [31]. Based on our measurement study, we found *energy efficiency* an equally important dimension in the design space of NFV, particularly when NFV is integrated into Internet-scale services such as multimedia content delivery. Our experiments have shown that performance optimizations in the dataplane could result in substantial energy costs. NFV vendors and customers need to consider these three dimensions together transitioning to NFV.

3.7 Related Work

Multimedia content delivery has attracted many research interests in recent years. Studies have shown that multimedia broadcasting providers such as Twitch.tv place content processing and distribution servers along the traffic flow path [52], and a majority of them can be in fact categorized as middleboxes [17]. The rise of NFV will make it easier to deploy and manage these middleboxes. It allows network functions to be provisioned as part of the consolidated infrastructure in modern datacenter [44][41]. For instance, E2 [41] is one of the first frameworks that provide common NFV-related mechanisms for creating and managing network functions in compute clusters.

Performance and flexibility remain the central topic of current NFV research. For example, NetVM [31] and ClickOS [40] allow network functions to be implemented on commodity servers while preserving their line-rate performance. It employs several software-level optimizations such as zero-copy data transmission between VMs to achieve efficient packet

processing. Energy efficiency has not drawn full attention in NFV research so far, particularly in the context of multimedia distribution. Among the existing works, Prekas *et al.* [43] designed OS-level mechanisms to achieve energy proportionality for latency-sensitive, data-processing workloads. Song *et al.* [47] proposed a CPU frequency scaling based scheme to conserve energy for video transcoding workloads in the non-NFV context. this chapter takes the first step towards addressing the energy efficiency in NFV-based multimedia content distribution.

Chapter 4

Concluding Remarks and Future Directions

4.1 Summary of this Thesis

In this thesis, we proposed an instance recycling mechanism to address the prevalent partial usage waste problem faced by users in public IaaS cloud. We designed and implemented a system (HARV) to enable efficient instance recycling. HARV incorporates a container-virtualized layer to enable resource orchestration without provider-level supports. HARV adopts a two-level scheduling policy which offloads application-specific scheduling to its buyers while it only handles container allocations. Further, we designed an instance trading module, with an online auction to determine the market price. Evaluation on real-world workloads demonstrates HARV's practicality and scalability; the large-scale simulation shows it achieves considerable cost savings, even when the life-cycle is shortened.

We investigated the energy cost of advanced NFV platforms for multimedia content delivery. We found that the upsurge of energy consumption is due to the characteristics of multimedia traffic, expensive computations of multimedia network functions and the dataplane energy inefficiency. We proposed a power management framework that leverages CPU frequency scaling to achieve energy saving.

We believe our findings apply to other major NFV platforms. The software components of our testbed are the commonly used building blocks for NFV platforms (e.g., DPDK, KVM), and we have identified the energy issues arise from the software stack as opposed to the specific hardware configuration. Moreover, our measurement tools and methodology can be reused on other hardware setups for obtaining quantitative results.

4.2 Future Directions

We plan to extend HARV to handle those non-residual instances. Cloud instances are often underutilized [19], if not completely idle, and their underused portions of resources can as well being reused. This will result in a generalized definition of residual instances, i.e. the instances with residual resources. New challenges will emerge from this more general problem, including how to estimate residual resources with precision and design pricing scheme with finer granularity. Nevertheless, with HARV demonstrating the feasibility and benefits of instance recycling, we believe those challenges are worth addressing.

Further, we are going to implement a prototype of the NFV power management framework. Based on the prototype, we will explore the integration with other control plane services including SDN and NFV controllers.

More broadly, we plan to explore the following research topics inspired by this thesis:

Content distribution with software packet processing While NFV is gaining its traction, it is foreseeable that a growing amount of multimedia/video traffic will flow through software middleboxes and dataplanes, which are spread across multiple servers and/or datacenters (e.g., at the origin, CDN, ISP, or client edge). Since multimedia applications are timid to network conditions, replacing or adding additional software packet processing devices at any place of the path is likely to have an impact on the user-perceived stream quality. It is critical to understand the performance and deployment nuances of these devices. Existing QoE optimization schemes (e.g., bitrate adaptation) should be extended accordingly to adapt to this trend.

Network function orchestration The cluster orchestration platforms are becoming "the OS for the datacenter." However, existing platforms may fall short-handed when they lack the toolchain to express middlebox-related semantics (e.g., service composition). How do we *reuse* the services of these platforms and extend them to handle middlebox operations? The framework should facilitate cross-server coordination and ensure network-wide policy enforcement. Customizing VM or container for middlebox (with the properties such as fault-tolerance). For example, container checkpoint/restore can be reused for providing migration for middleboxes. It may also increase the operational costs (e.g., energy) for the network function providers, who need to make smart trade-off among flexibility, performance, and energy efficiency. Such tools can benefit other cloud services deployed following the micro-services architecture (e.g., Netflix's design), as it bears similarities with NFV.

Bibliography

[1] Amazon EC2 Container Service. http://aws.amazon.com/ecs/.

[2] Aws's a leader in the iaas market, gartnermq, 2016. https://aws.amazon.com/resources/gartner-2016-mq-learn-more/.

[3] Azure virtual machines, microsoft azure, 2017. https://azure.microsoft.com/en-ca/services/virtual-machines/.

[4] Compute engine, google cloud platform, 2017. https://cloud.google.com/compute/.

[5] Docker. https://www.docker.com/.

[6] Ec2 product details, amazon web services, 2017. https://aws.amazon.com/ec2/details/.

[7] Google Container Engine. https://goo.gl/oxBS2e.

[8] lxc. https://linuxcontainers.org/.

[9] Open Container Initiative. https://www.opencontainers.org/.

[10] OpenVZ Containers. https://openvz.org/.

[11] rkt app-container runtime. https://github.com/coreos/rkt.

[12] White paper: Cisco vni forecast and methodology, 2015-2020. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

[13] Windows Server Containers. https://goo.gl/Vxkf6h.

[14] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The rise of raas: the resource-as-a-service cloud. *Communications of the ACM*, 57(7), 2014.

[15] Paul Barham et al. Xen and the art of virtualization. In *Proc. ACM SOSP*, pages 164–177, 2003.

[16] M. Ben-Yehuda et al. The turtles project: Design and implementation of nested virtualization. In *Proc. USENIX OSDI*, volume 10, 2010.

[17] B Carpenter and S Brim. Rfc 3234-middleboxes: Taxonomy and issues. *Network Working Group. Ietf*, 2002.

[18] Shuchi Chawla, Jason D. Hartline, David L. Malec, and Balasubramanian Sivan. Multi-parameter mechanism design and sequential posted pricing. In *Proc. ACM Symposium on Theory of Computing (STOC)*, 2010.

[19] Liuhua Chen and Haiying Shen. Consolidating complementary vms with spatial/temporal-awareness in cloud datacenters. In *Proc. IEEE INFOCOM*, 2014.

[20] Sergei Chichin, Quoc Bao Vo, and Ryszard Kowalczyk. Towards efficient and truthful market mechanisms for double-sided cloud markets. *IEEE Transactions on Services Computing*, 10(1):37–51, 2017.

[21] N. Chohan et al. See spot run: Using spot instances for mapreduce workflows. In *Proc. USENIX HotCloud*, 2010.

[22] Nikhil R Devanur and Zhiyi Huang. Primal dual gives almost optimal energy efficient online algorithms. In *Proc. ACM-SIAM SODA*, 2014.

[23] W. Felter et al. An updated performance comparison of virtual machines and linux containers. *IBM Research Report*, 2014.

[24] Ralph Finos. Public cloud market forecast 2015-2026. `http://wikibon.com/public-cloud-market-forecast-2015-2026/`.

[25] Silvery Fu, Jiangchuan Liu, and Wenwu Zhu. Multimedia content delivery with nfv: From an energy perspective. *to appear in IEEE MultiMedia*, 2017.

[26] Silvery Fu, Jiangchuan Liu Liu, Xiaowen Chu Chu, and Yueming Hu. Toward a standard interface for cloud providers: The container as the narrow waist. *IEEE Internet Computing*, 20(2):66–71, March 2016.

[27] Silvery Fu, Yifei Zhu, and Jiangchuan Liu. Harv: Harnessing hybrid virtualization to improve instance (re)usage in public cloud. In *Proc. IEEE/ACM IWQoS*, 2017.

[28] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, 2014.

[29] Fang Hao, Murali Kodialam, TV Lakshman, and Sayan Mukherjee. Online allocation of virtual machines in a distributed cloud. In *Proc. IEEE INFOCOM*, 2014.

[30] Joseph L. Hellerstein. Google cluster data. Google research blog, Jan 2010. Posted at `http://googleresearch.blogspot.com/2010/01/google-cluster-data.html`.

[31] Jinho Hwang, KK Ramakrishnan, and Timothy Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. In *Proc. USENIX Networked Systems Design and Implementation (NSDI)*, pages 445–458, 2014.

[32] Hai Jin, Xinhou Wang, Song Wu, Sheng Di, and Xuanhua Shi. Towards optimized fine-grained pricing of iaas cloud platform. *IEEE Transactions on Cloud Computing*, 3(4), 2015.

[33] Ian A Kash and Peter B Key. Pricing the cloud. *IEEE Internet Computing*, 20(1):36–43, 2016.

[34] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proc. USENIX NSDI*, 2016.

[35] Bob Lantz et al. A network in a laptop: rapid prototyping for software-defined networks. In *SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[36] Li Li, Tony Tang, and Wu Chou. A rest service framework for fine-grained resource management in container-based cloud. In *Proc. IEEE CLOUD*, 2015.

[37] Yusen Li, Xueyan Tang, and Wentong Cai. Dynamic bin packing for on-demand cloud resource allocation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(1):157–170, 2016.

[38] Haikun Liu and Bingsheng He. F2c: Enabling fair and fine-grained resource sharing in multi-tenant iaas clouds. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(9):2589–2602, 2016.

[39] David Lo et al. Heracles: improving resource efficiency at scale. In *ISCA*. ACM, 2015.

[40] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[41] Shoumik Palkar et al. E2: a framework for nfv applications. In *Proc. ACM SOSP*, pages 121–136, 2015.

[42] Siani Pearson and Azzedine Benameur. Privacy, security and trust issues arising from cloud computing. In *Proc. IEEE CloudCom*, 2010.

[43] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, pages 342–355, 2015.

[44] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. USENIX Networked Systems Design and Implementation (NSDI)*, pages 323–336, 2012.

[45] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proc. ACM Eurosys*, 2015.

[46] Stephen Soltesz et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys*, 2007.

[47] Minseok Song, Yeongju Lee, and Jinhan Park. Scheduling a video transcoding server to save energy. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 11(2s):45, 2015.

[48] Haiyang Wang, Feng Wang, Jiangchuan Liu, and Justin Groen. Measurement and utilization of customer-provided resources for cloud computing. In *Proc. IEEE INFOCOM*, 2012.

[49] Wei Wang, Di Niu, Baochun Li, and Ben Liang. Dynamic cloud resource reservation via cloud brokerage. In *Proc. IEEE ICDCS*, 2013.

[50] Zhuoyao Wang, Majeed Hayat, Nasir Ghani, and Khalid Shaaban. Optimizing cloud-service performance: Efficient resource provisioning via optimal workload allocation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2016.

[51] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: virtualize once, run everywhere. In *Proc. ACM EuroSys*, 2012.

[52] Cong Zhang and Jiangchuan Liu. On crowdsourced interactive live streaming: A twitch.tv-based measurement study. In *Proc. ACM NOSSDAV*, pages 55–60, 2015.

[53] Linquan Zhang, Zongpeng Li, and Chuan Wu. Dynamic resource provisioning in cloud computing: A randomized auction approach. In *Proc. IEEE INFOCOM*, 2014.

[54] Xiaoxi Zhang, Zhiyi Huang, Chuan Wu, Zongpeng Li, and Francis CM Lau. Online auctions in iaas clouds: Welfare and profit maximization with server costs. In *Proc. ACM SIGMETRICS*, 2015.

[55] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to bid the cloud. In *Proc. ACM SIGCOMM*, 2015.

Appendix A

List of Publications

While the author has some other publications studying cloud computing and virtualization, they were not included in this thesis. In what follows, we provide a complete list of these publications for reference, including the ones constitute the thesis ([1],[2],[5]). We hope the list suffice to offer a complete picture of the author's research on cloud computing and virtualization during his bachelor's and master's:

[1] Silvery Fu, Jiangchuan Liu, and Wenwu Zhu, "Multimedia content delivery with nfv: From an energy perspective." *IEEE Multimedia*, 2017.

[2] Silvery Fu, Yifei Zhu, and Jiangchuan Liu, "Harv: Harnessing hybrid virtualization to improve instance (re)usage in public cloud." In *Proc. IEEE/ACM IWQoS*, 2017.

[3] Yifei Zhu, Silvery Fu, and Jiangchuan Liu, "Truthful online auction for cloud instance subletting," In *Proc. IEEE ICDCS*, 2017.

[4] Zhang Lei, Silvery Fu, Jiangchuan Liu, Edith Cheuk-Han Ngai, and Wenwu Zhu, "On energy efficient offloading in mobile cloud for realtime video applications," *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT)*, vol.27, no.1, pp.170-181, January 2017.

[5] Silvery Fu, Jiangchuan Liu, Xiaowen Chu, and Yueming Hu, "Toward a standard interface for cloud providers: The container as the narrow waist," *IEEE Internet Computing*, vol.20, no.2, pp.66-71, March 2016.

[6] Ryan Shea, Silvery Fu, and Jiangchuan Liu, "Cloud gaming: Understanding the support from advanced virtualization and hardware," *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT)*, vol.25, no.12, pp.2026-2037, December 2015.

[7] Ryan Shea, Silvery Fu, and Jiangchuan Liu, "Towards bridging online game playing and live broadcasting: Design and optimization," In *Proc. ACM NOSSDAV*, March 2015.

[8] Ryan Shea, Silvery Fu, and Jiangchuan Liu, "Rhizome: Utilizing the public cloud to provide 3d gaming infrastructure," In *Proc. ACM Multimedia Systems Conference (MMSys)*, March 2015.