Unikernelized Driver Domain

A K M Fazla Mehrab

Preliminary Examination Proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Daphne Yao
Paul Plassmann
Haibo Zeng
Ruslan Nikolaev

December 16, 2020
Blacksburg, Virginia

Keywords: Operating Systems, Unikernels, Virtualization, Heterogeneous Systems

Unikernelized Driver Domain

A K M Fazla Mehrab

(ABSTRACT)

Modern hypervisors such as Xen, which provide specialized and faster I/O drivers to enlightened guest operating systems, allow housing physical device drivers in separate, driver domain OSs for better performance and enhanced isolation. Traditionally, driver domains run full-blown Linux or other general-purpose OSs. In this dissertation, we argue that implementing driver domain OSs using unikernel operating systems yields many benefits. These include reduced attack surface, smaller memory footprints, easier deployment, and faster failure recovery. These benefits are particularly significant when using the rumprun unikernel, which internally uses highly-customized NetBSD code and enables access to high-quality device drivers. Although rumprun can already be used in place of an ordinary (unikernel) OS for user applications, it lacks Xen backend drivers and other components that are necessary for driver domain OSs. We overcome these challenges. We develop rumprun-based driver domain OSs that can be used for storage and network driver domains, and compare them against traditional Linux-based storage and network driver domains, respectively. We evaluate performance using widely used storage and network applications. Our experimental results demonstrate high overall performance, comparable to that of Linux. They also highlight the benefits of unikernelized driver domains by showing a reduced number of gadgets, smaller image size, and faster boot time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtualization is the backbone technology in modern enterprise systems, especially those running in cloud infrastructures. Cloud environments such as Amazon EC2 rely on a hypervisor to strongly isolate client virtual machine (VM) instances from each other for better security. Such hypervisors must provide efficient and secure access to network and storage I/O devices. To that end, hypervisors such as Xen [21], which is used by EC2, have developed a number of solutions over the years. First, network and storage I/O are exposed to guest operating systems (OSs) via efficient paravirtualized (PV) drivers which communicate with the corresponding physical device drivers. Second, a physical device driver (e.g., a network or storage driver) can be executed in a separate *driver domain* [1] for more effective load balancing and enhanced security. Isolating drivers in separate virtual machines is especially important as the number of common vulnerabilities and exposures (CVEs) for drivers continue to surge across different OSs (see Figure 1.1).

Driver domains are special guest OSs, which run inside isolated VMs. They are often advocated for enhanced isolation and security [56]: they can be used to house certain classes of device drivers. Thus, instead of running these drivers in privileged VMs such as Xen's Dom-0, where a potential driver vulnerability can be catastrophic for the entire system, they

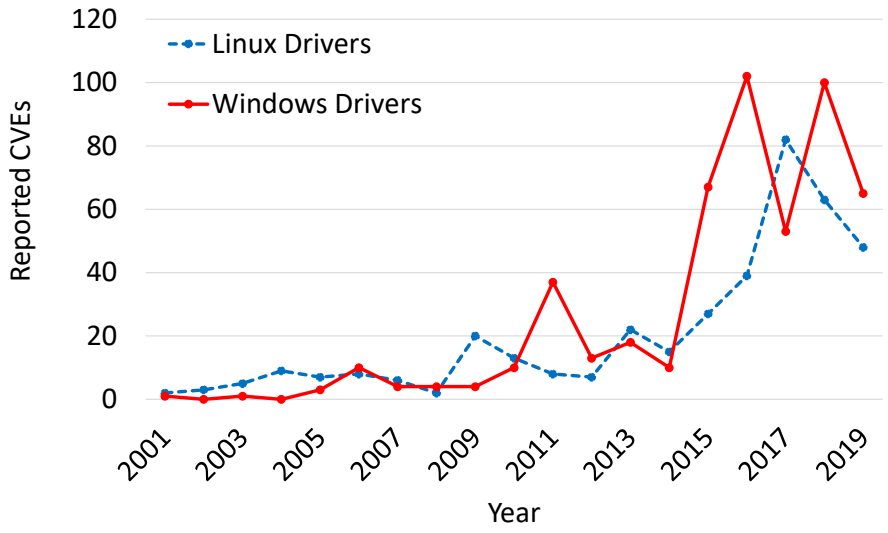


Figure 1.1: CVEs for drivers during 2001–2019 (from https://cve.mitre.org)

are isolated in driver domains. By removing the drivers from Dom-0, Dom-0's attack surface is also reduced, further increasing security.

A downside of driver domains is that they are relatively heavy-weight as they usually run a general-purpose, full-fledged OS such as Linux. Such domains are cumbersome for deployment and upgrades. Linux contains many device drivers and subsystems (e.g., audio, video, USB, etc) as well as user-space libraries, tools, and configuration scripts that are irrelevant to networking and storage, and yet, they all need to be properly maintained when Linux is deployed as a driver domain. Even a minimalistic Linux distribution has a fairly large memory footprint, which adds up in enterprise-scale cloud systems that handle many I/O devices. For example, a Linux kernel image size alone for the default configuration is about 50MB. A full-fledged distribution of Linux has a footprint of several gigabytes. (Ubuntu Linux's 18.04, x86-64 server image size is almost 1GB.) Using a general-purpose OS for driver domains is also not ideal for security as it exposes a potentially large attack surface, which is undesirable for systems with a greater degree of resource sharing such as cloud infrastructures like Amazon EC2.

1.1 The Unikernel Operating System Model

Unikernels [24, 34, 36, 37, 38, 41, 44, 46, 52, 68] are lightweight OSs designed specifically for cloud systems and run atop a hypervisor in separate VMs. They are, by design, capable of running only a single application. In contrast to full-fledged OSs, in unikernels, a single application is statically compiled together with the minimum necessary kernel code and libraries to produce a single address-space image – a form of library OS [20]. In contrast, the traditional VM model includes a full-fledged guest OS, which includes a large number of subsystems that may not be used by guest applications. Figure 1.2 illustrates this contrast.

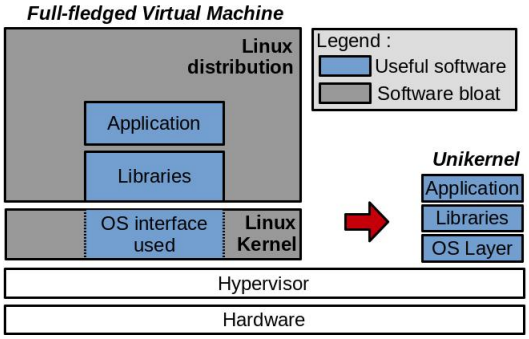


Figure 1.2: Traditional VM model versus the unikernel VM model.

The literature presents a number of unikernels. Examples include HermitCore [38], MirageOS [41, 42], rumprun [34], HermiTux [50], Lupine Linux [37], and IncludeOS [24], among others.

The unikernel model has several advantages. Since only select OS components are included with the application, their code size and memory footprints are significantly smaller, which also reduces their boot times. (This is potentially important if fast failure recovery through VM restarts is desired.) Figure 1.3 compares the memory image size and boot times of the rumprun unikernel [34] against Ubuntu Linux. In this figure, rumprun encapsulates a network driver domain that we designed (Chapter 6) and implemented (Chapter 7), and Ubuntu Linux does not run any application. The figure reveals rumprun's significant savings on memory image size and boot times – by one order of magnitude.

(a) Memory image size (b) Boot time

Figure 1.3: Memory image size and boot times of rumprun and Ubuntu Linux network driver domain .

Unikernels also have a performance advantage. Since OS components are compiled together with the application in a single address-space, system calls are converted to common function calls, a feature that can reduce system call latency by up to one order of magnitude [50], leading to performance improvement of up to 33% for system intensive applications [37].

Unfortunately, unikernels are designed only for user-space applications and are unsuitable for driver domains. In this dissertation, we explore the use of unikernels for driver domains. We use the rumprun unikernel [34] for this purpose. The key feature of rumprun is that it is directly based on NetBSD's code [16], which makes it possible to leverage NetBSD's large collection of device drivers, including highly specialized ones such as Amazon ENA. In fact, NetBSD device drivers can run inside a rumprun unikernel instance out-of-the-box.

Compared to Linux, a rumprun unikernel image is minimalistic: it can be as small as 22MB. This small code size yields an important security advantage: potentially less security vulnerabilities. Figure 1.4 compares the number of ROP gadgets [59], measured using the Ropper tool [14], for rumprun, vanilla Linux kernel with the default configuration, CentOS 8, Fedora Rawhide (05/2020), Ubuntu 18.04, and Debian 10.4 kernel images with their associated kernel modules. The number of ROP gadgets is one quantitative metric that is often used to evaluate security vulnerability [29]: smaller the number of gadgets, greater is the difficulty for an attacker to find appropriate gadgets to launch an exploit. Figure 1.4 demonstrates that rumprun has a substantially smaller number of gadgets than any of the Linux configurations. Assuming that NetBSD's code quality is on par with that of Linux, this figure also

indicates a potential for improved security more generally, as rumprun's attack surface is also proportionally reduced compared to that of full-fledged OSs.

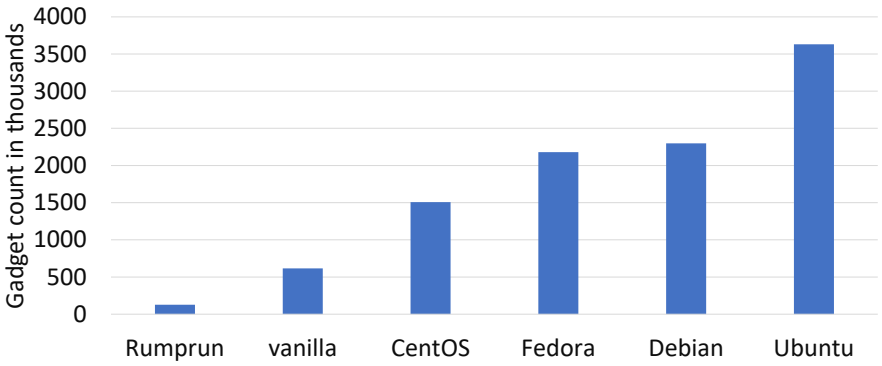


Figure 1.4: ROP gadget comparison.

Recent versions of rumprun [49] were extended to support multi-core systems and hardware-assisted virtualization in Xen, which makes it even more attractive as a driver domain OS. However, using rumprun as a driver domain OS requires overcoming several significant challenges. Xen's PV I/O support is very minimalistic in rumprun. For instance, rumprun lacks Xen backend drivers. Thus, rumprun can be used only as a regular unikernel, but not as a driver domain. Moreover, NetBSD itself is not known to be used for driver domains in Xen.

We overcome these challenges. We focus on the driver domains for the two most used devices – storage and network. We develop rumprun-based, storage and network driver domain OSs by designing and implementing the blkback and netback drivers, respectively. To understand the effectiveness of our implementations, we conduct extensive experimental studies using well-known benchmarks and applications. Our results reveal that our rumprun driver domains provide competitive performance to that of a Linux-based driver domain, while retaining all the benefits of unikernels such as reduced number of ROP gadgets, smaller image and code sizes, and faster boot time.

1.2 Summary of Research Contributions

This dissertation presents unikernelized driver domains for the two most basic and necessary devices: storage and network. An unikernelized driver domain isolates the device driver from the privileged domain, which is used for hypervisor services and guest VM management. The minimalist nature of unikernel enables us to develop lightweight and resource-efficient driver domains. Our lightweight driver domains significantly reduce the attack surface compared to commodity OSs. Moreover, they have faster deployment time and lesser resource consumption while not loosing any performance benefits.

1.2.1 Unikernelized Storage Domain

Most guest virtual machines need storage device drivers. PV storage drivers are one of the most common drivers used by the guest virtual machines because of faster I/O processing than emulation. We present the design and implementation process for unikernelizing a storage domain that yields security advantages while preserving performance benefits.

We use the rumprun unikernel, which allows us to leverage physical storage drivers from NetBSD. However, a PV storage device needs a backend (blkback) in the driver domain to communicate and transfer data between the physical storage and the guest machine through its storage frontend (blkfront). We design and implement the blkback driver in rumprun. Our extensive evaluations show that our rumprun-based storage domain is as performant as Linux, one of the most widely used commodity OS for servers and desktops. However, the exposed attack-surface of our storage domain is only a fraction of a counterpart Linux domain.

1.2.2 Unikernelized Network Domain

Networking is one of the most used functionality for guest VMs. PV networking is the most efficient method for multiplexing one network device between several guests. We design and

implement a rumprun-based network driver domain, which works as backends for PV networking and routes networking traffic from guest machines to the physical network interface card (NIC). We leverage the NIC drivers from NetBSD but implement netback driver, which rumprun lacks.

The network backend (netback) and frontend (netfront) communicate using two shared memory ring buffers for receiving and transmitting network packets, respectively. Since rumprun lacks rich work queues and interrupts, we present a multi-threaded model for efficiently handling network packets without incurring any performance penalty. We also design and implement network bridging support for netback, which is necessary for forwarding packets from multiple guest OSs to the physical network adapter. Our evaluations show that our rumprun-based network domain is as performant as Linux, while exposing only a fraction of a corresponding Linux domain's attack-surface.

1.3 Summary of Proposed Post-Preliminary Exam Work

Using our rumprun-based network domain, we were able to saturate 1GB NIC. However, due to Xen's grant table mechanism for shared memory, hypercalls, and other PV design aspects, driver domains incur more overheads than in Xen's Dom-0. Therefore, no existing network domain achieves high-throughput for NIC devices such as 10GbE and 40GbE. Past works [45, 53, 57] propose several changes to the driver domain, which does not allow compatibility with existing frontends. We propose to design and implement a backward-compatible, rumprun-based high-throughput network domain as post-preliminary exam work.

This dissertation leverages the PVHVM mode of rumprun unikernel from LibrettOS [49], which allows the system to benefit from IOMMU [19, 33]. However, PVH is the latest addition to Xen's virtualization modes. PVH is a lightweight virtualization mode and benefits from hardware acceleration for interrupts and timers, whereas PVHVM mode relies on PV for these features. Since the dissertation's goal is to provide lightweight, secure, and performant driver domains, we propose to modify rumprun to support Xen's PVH mode as

another direction for post-preliminary exam work.

We specifically worked on the Xen hypervisor for the work presented so far because it is the only hypervisor that supports the driver domain concept. However, rumprun works on other popular hypervisors such as KVM. KVM supports PV device drivers, and various technologies are being developed based on the KVM PV model to take advantage of high-throughput devices. Separating the device drivers from the host OS and running them ina unikernel would benefit such technologies because of added security benefits and better resource utilization. Therefore, we propose to develop the driver domain concept in other hypervisors such as KVM as part of post-preliminary exam work

1.4 Dissertation Organization

The rest of the dissertation proposal is organized as follows. Chapter 2 discusses relevant background information on Xen, its PV I/O driver models for network and storage devices, rump kernels, and the rumprun unikernel. Chapter 3 summarizes related work on hypervisor disaggregation techniques, unikernels, and driver domains.

Chapter 4 presents the design of unikernelized storage domain using rumprun and Chapter 5 presents its implementation details. Chapter 6 presents the design of unikernelized network domain using rumprun and Chapter 7 presents its implementation details.

Chapters 8 and 9 evaluate unikernelized storage domain and network domain, respectively. Finally, Chapter 10 concludes and discusses post-preliminary examination work.

Chapter 2

Background

In this chapter, we discuss concepts that are relevant to the design of our unikernelized driver domains for Xen. We also touch upon Xen itself as well as its virtualization approaches.

First, we discuss the concept of PV I/O device driver in section 2.1, followed by Xen's driver domain model in section 2.2 and 2.5. We then discuss the concept of unikernels, followed by a discussion of using unikernels for the driver domain and the rationale for choosing the rumprun unikernel in section 2.6.

2.1 Xen HVM

Xen is a popular Type I hypervisor [35], which pioneered a concept of "paravirtualization" by leveraging protection rings of x86-32. These CPUs historically lacked virtualization capabilities [55], and Xen's approach was to modify the target OS kernel, so that it can run on top of a hypervisor. Later CPUs enabled hardware-assisted virtualization support through VT-x and AMD-V (or other extensions by non-x86 vendors). *Hardware Virtualization Mode* (HVM) was proposed and widely adopted by Xen since then. HVM is also more preferable nowadays since it is not affected by recently discovered security vulnerabilities such as Meltdown [40].

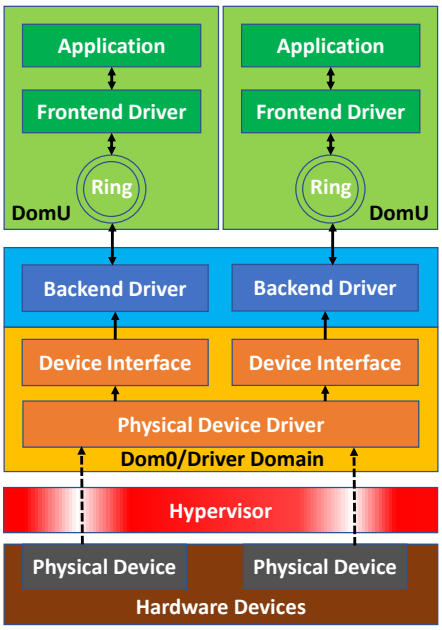


Figure 2.1: Xen's PV I/O device driver model.

2.2 Xen I/O Drivers

Traditional I/O device emulation is inefficient due to substantial performance overheads [25]. Xen's original paravirtualization method proposed to handle I/O through special PV drivers. PV drivers are also used in the HVM mode as long as the corresponding guest OS is *enlightened* about Xen's presence. Other hypervisors, e.g., VMware, Hyper-V, and KVM, similarly implement faster I/O drivers.

PV drivers in Xen are typically divided into two parts: *frontend* and *backend*. The frontend driver runs in the guest OS, denoted as DomU. The backend driver runs either in Dom-0, the very first (privileged) guest, or in a dedicated driver domain (see Section 2.5).

2.3 Blkfront and Blkback

The PV storage frontend (*blkfront*) provides an interface to abstract secondary storage. Therefore, the other part of the OS and applications can use this interface to issue regular block device operations such as read, write, etc. On the other hand, the PV storage backend

(*blkback*) is responsible for performing these operations on the real device using the physical storage device driver. A storage domain should be able to support multiple blkfront from the same or different guest machines as shown in 2.1.

Both ends of the PV driver get to know each other's configurations using the Xenstore database. The communication between the blkfront and blkback is maintained using Xen's shared memory and ring buffer mechanism. The blkfront allocates the shared memory and a ring buffer, which are used by both ends to transfer data. Using the ring buffer, the blkfront sends requests with information, like sector number, size, etc., to the blkback for performing specific kinds of operation on the storage device. On the other hand, the blkback sends responses upon performing such operations. The storage data is transferred between these ends using direct memory access (DMA) transfers, known as the grant table mechanism.

2.4 Netfront and Netback

In the case of network interface card (NIC), the frontend (*netfront*) exposes a virtual network interface to the network stack of DomU, whereas the backend (*netback*) talks to the actual physical NIC. The netfront and netback drivers establish a connection between each other using Xen-specific mechanisms such as *Xenstore* and *Xenbus*. Netfront and netback transfer data between each other using shared memory ring buffers. As a result, DomU can send packets to the physical NIC through netfront.

For each netfront, there should be one corresponding netback as shown in Figure 2.1. The netback and netfront drivers use two shared ring buffers for data exchange, which are allocated by netfront. One ring buffer, Tx, is used for sending packets from netfront to netback. Another ring buffer, Rx, is used for sending packets from netback to netfront.

Network bridge

As shown in Figure 6.1, there are multiple netbacks with multiple DomUs. One way to share the same physical NIC is to use *bridging* by connecting all netbacks to a bridge. The network bridge, which is connected to the physical NIC, routes packets between netbacks and the physical NIC as well as across different netbacks. Each netback is assigned its own IP address on the network.

2.5 Driver Domain

Xen's privileged Dom-0 domain runs device drivers and performs many critical system tasks. To offload Dom-0, *driver domains*, special unprivileged guest OSs which run device drivers, are often used. Driver domains also increase isolation and overall system security since potentially vulnerable drivers are isolated from Dom-0. Driver domains have direct access to the underlying hardware by using PCI passthrough capabilities of the hypervisor. They are more typical for networking, where the netback driver runs inside a separate VM rather than Dom-0. Although storage also has corresponding blkfront-blkback I/O drivers, they seem to be deployed in Dom-0 for typical setups.

The use of driver domains is exemplified by Qubes OS [56], an OS which runs individual Linux instances for each group of applications in VMs atop of Xen to provide very strong security. Qubes OS uses network driver domains to strongly isolate network device drivers from the system. For stronger isolation, Qubes OS also takes advantage of I/O memory management unit (IOMMU) [19, 33], which we also support in our design. Full-fledged IOMMU support needs HVM; it safely remaps interrupts and memory addresses to protect against both malicious (or faulty) devices and vulnerable (or buggy) device drivers.

By moving device drivers to a separate domain, Dom-0's attack surface reduces. As a result, any error or exploit in the corresponding code will not affect Dom-0, and the Xen administrative interface to other guest OSs remains uninterrupted. Though other open-source virtualization technologies such as KVM support faster I/O, they do not yet implement

driver domains, unlike Xen.

In this dissertation, we propose to use a unikernel for Xen driver domains. By using a light-weight unikernel, we are able to reduce memory footprints as well as reduce the attack surface of driver domains.

2.6 Rump Kernels and Rumprun

For the network and storage driver domains, we need a unikernel capable of running many device drivers. A fair number of non-compatible network (Ethernet [39], Wireless LAN [18], etc) and storage (PATA, SATA, SCSI, NVMe) controllers must be supported. Because porting incurs non-trivial engineering effort, we need a unikernel that can reuse existing drivers.

NetBSD [16], a well-known general-purpose OS, has a unique property in that all its core kernel components are refactored into *anykernel* components. The anykernel concept implies that these components can be used in any context, e.g., a device driver can be executed in a user thread. A special *rump kernel* glue layer enables the reuse of the anykernel components outside of the NetBSD kernel.

Rumprun is a unikernel that leverages rump kernels such that it can potentially reuse any NetBSD device driver. Figure 2.2 shows the rumprun software stack, which consists of the platform-specific layer (Xen) and *bare metal kernel* (BMK) layer, which implements thread management, scheduling, interrupts, and memory management. A special *rumpuser* layer implements an interface (known as "hypercalls", which are not to be confused with Xen's hypercalls) for the rump kernel components to communicate with the BMK layer. This dissertation reuses other NetBSD components, such as the TCP stack and vnode block device interface that are denoted as 'Faction.'

The layers above the rump kernel consist of relevant libraries and their interface to the rump kernel. A unikernel application runs on top of the stack. NetBSD system calls from LibC

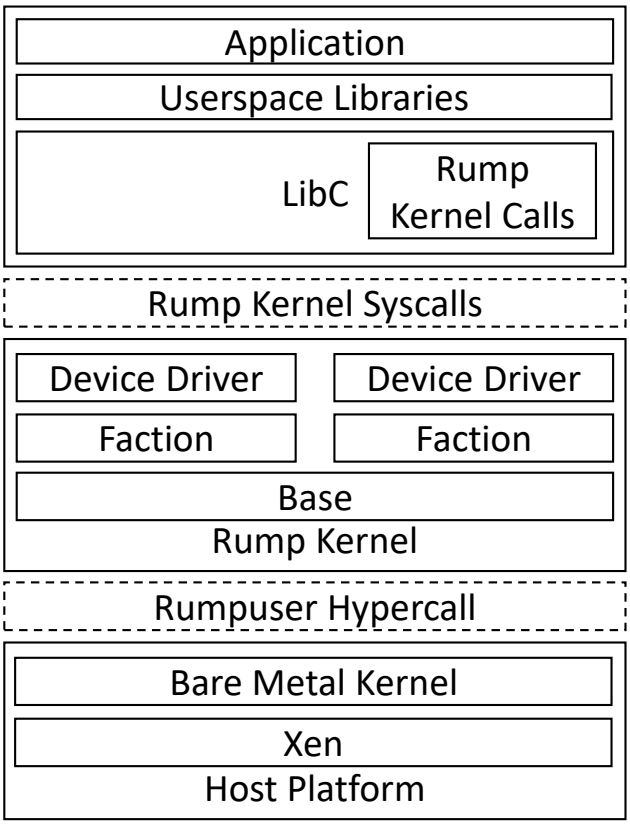


Figure 2.2: Rumprun stack on Xen.

are replaced with ordinary function calls. Since drivers need semantically similar support routines they use in NetBSD, the rump kernel contains 'Base' which provides support for memory allocation, thread handling, and locking.

Although a number of embedded Linux systems exist, we are not aware of a comparable minimal system that can readily be used for driver domains. Linux-based unikernels [37, 54] currently lack maturity and flexibility. In contrast, rumprun is stable, and rump kernels are upstreamed to NetBSD.

Chapter 3

Related Work

Several state-of-the-art works share motivations with our work in some aspects. Moreover, different existing approaches previously attempted to secure virtualization platforms in different ways. On the other hand, unikernels are being used on different hypervisors and proposed to solve various interesting problems. In this chapter, we discuss some existing relevant works on hypervisor security and unikernel usages.

Section 3.1 presents a discussion on different techniques that break the hypervisor into parts for more secured virtualization environments. Section 3.2 discusses how the usage of unikernel is helping improve cloud infrastructure.

3.1 Hypervisor Disaggregation Approaches

Hypervisors, which make it possible to run multiple VMs on the same physical machine, are counted towards a trusted computing base (TCB) for cloud infrastructures. The Xen hypervisor uses Dom-0 as a control VM. Dom-0 is a fully-fledged OS that runs on top of Xen. Unexpected behavior from Dom-0 or Xen can immediately and adversely affect any (DomU) guest OS. Therefore, there have been several efforts [27, 48, 60] for splitting Xen responsibilities, so that an exploited or failed component does not affect other components.

Xoar [27] disaggregates Dom-0 functionality into nine types of service VMs, each having different responsibilities. Two of them, PCI backend and bootstrapper, run on top of nanOS, a lightweight OS, destroyed after initialization. Among other service VMs are network driver domain and block driver domain. Driver domains execute corresponding backend drivers. Xoar allocates one backend driver for each frontend driver so that one frontend does not adversely affect another frontend. Every driver domain runs a fully-fledged OS such as Linux, which still has a potentially large attack surface.

Qubes OS [56] implements a protected Xen-based user environment using four types of VMs: apps VMs, network VM, storage VM, and administrative/graphical user interface (GUI) VM. The apps VMs are domains for running corresponding types of applications. The network VM runs netback and serves as a network driver domain for the apps VMs. The storage VM provides access to the disk for the apps VMs. The administrative/GUI VM provides GUI to users. For all types of VMs, Qubes OS runs Linux. Since Qubes OS uses standard Xen driver domains, our unikernelized driver domains can also be integrated into Qubes OS easily for reducing attack surface and memory footprint of driver domains. Reducing memory overheads is crucial for desktop environments that Qubes OS primarily targets.

The NeXen [60] architecture decomposes hypervisor into three parts using paged-based isolation mechanisms: security monitor, shared service domain, and Xen slices. The security monitor provides isolation between internal domains and manages privileges by controlling all updates to the memory management unit (MMU). Xen slices are composed of highly vulnerable hypervisor functionalities and data needed by the DomU. Each slice serves only one DomU. The shared service domain provides the functionalities that could not be decomposed into slices. One limitation of this work is that NeXen does not manage I/O devices. Therefore, NeXen relies on the native Linux PV (e.g., network and disk) device drivers and cannot prevent abuses on the drivers.

Murray et al. [48] disaggregate the hypervisor by extracting the domain building process, called domain builder, from Dom-0 and porting it to a light-weight OS such that the TCB at-

tack surface remains small. However, for I/O calls, the domain builder relies on Dom-0 which runs a backend driver as well as a physical driver. Therefore, this disaggregation does not secure the I/O path. SSC [26] describes a modified Xen architecture for reducing TCB by distributing DomU responsibilities to multiple user-level service domains called UDom-0. Each DomU belongs to one UDom-0, which enforces isolation. Apart from UDom-0, this design has a system-wide administrative domain, called SDom-0, and a domain builder. SDom-0 has multiple responsibilities, including scheduling and I/O device virtualization. Therefore, SDom-0 has a relatively larger attack surface and errors can affect core functionalities.

3.2 Unikernels for Cloud Infrastructures

Every workload in the cloud runs inside VMs. Unikernels are light-weight OSs that claimed to have various desirable properties by the researchers, which make them a good fit for the cloud infrastructure over the existing setup. In terms of boot time, some works [43, 65, 66] show that deployment of scaled cloud workload in remote data centers is faster and smoother compared to Linux. Moreover, memcached was shown to have a throughput that is twice as faster using a unikernel [58] compared to Linux.

On the other hand, because of the portability and consistency, application containerization became very popular for deployment in clouds. However, containers provide process-level isolation where unikernels themselves are VMs, often supported by hardware-assisted virtualization, which offers better isolation. To provide better isolation, containers often run inside VMs that use fully-fledged OSs [28, 31]. These OSs have a bigger attack surface than unikernels. Moreover, Goethals [32] et al. show that for microservice applications, the OSv [36] unikernel performs up to 38% faster than Docker [47]. LibrettOS [49] shows that rumprun-based unikernels can be used as servers in a single OS. LibrettOS's NFS server is 9% faster than that of either Linux or NetBSD, and Ngnix HTTP server is up to 66% and 27% faster than NetBSD and Linux, respectively. We use and extend LibrettOS's multi-core and Xen HVM support in our work.

HEXO [51] takes advantage of the low resource requirement nature of a unikernel, named HermitCore [38]. Authors were able to offload server workloads to light-weight, low-cost embedded computer boards such as Raspberry Pi [4]. This way, they were able to improve the throughputs for compute-intensive workloads in servers up to 67%, costing negligible amount of money for infrastructure and energy.

The faster boot time, lower resource requirement, and better performance of unikernels are inspiring the researcher in the academia and industry [2, 54] to adapt unikernel architecture for improving the cloud infrastructure. However, none of these prior works attempted to improve the hypervisor itself. In this dissertation, we show how adapting the unikernel concept can benefit hypervisors and make them more secure and performant, while reducing resource consumption.

3.3 Driver Domain and Backend Improvement

Several approaches were proposed by the researchers previously to improve the Xen PV device drivers and driver domains. Sushrut Shirole came up with the idea to replace the interrupt-based event handler with a polling-based request and response handler for both front and backends [61] to improve driver domain performance. This approach requires modification to Linux frontend and backend drivers for compatibility, where our work results in a lightweight driver domain without losing any compatibility with the existing frontends.

XCollOpts [67] focuses on the existing VM scheduling scheme that supposedly favors CPU-intensive workloads over the latency-sensitive I/O workloads running in a VM. In contrast, the authors proposes premature preemption prevention techniques for driver domains and CPU load balancing-based optimizations for credit schedulers to benefit I/O virtualization. Moreover, XCollOpts suggests multiple Tx and Rx buffer pairs in the network driver domain to leverage multicore systems along with methods to reduce grant page accesses for the small-sized packets.

Bourguiba et. el [22, 23] proposes a I/O virtualization model based on packet aggregation to transfer packets between the driver domain and the VMs. Their work shows network throughput improvement at the cost of latency. To mitigate latency degradation, the proposes a dimensioning tool that helps to dynamically balance between throughput and latency in different situations.

All of the above-mentioned prior works regarding I/O virtualization and driver domain focuses particularly on performance (especially for network) rather than other important aspects such as security and resource utilization. Moreover, it seems they consider Linux as the base OS for the driver domain. One exception would be the LibrettOS Network Server [49] that implements a network server in a unikernel. This network server runs as a network backend where the communication with the frontend is done using improvised ring buffers, which are only compatible with network clients with such ring support. Therefore, unlike our proposal, this model is not compatible with the frontends from existing commodity OSs such as Linux, NetBSD, and Windows.

Chapter 4

Unikernelized Storage Domain Design

Unikernelizing a storage domain is a non-trivial problem to solve because of several aspects related to unikernel and Xen's storage domain architecture. A careful design can provide a more compact storage domain than the state-of-the-art, without losing any performance benefit. In this chapter, we discuss the design choices we made to achieve such qualities.

In section 4.1, we discuss the associated challenges for unikernelizing a Xen storage domain. In section 4.3, we discuss the design of our storage domain for rumprun unikernel, keeping the discussed challenges in mind.

4.1 Challenges

Both Xen PV storage frontend and backend drivers are available in different commodity OSs, such as Linux and NetBSD. Therefore, it is easier to take one of these OSs as they are and deploy it as a storage driver domain. On the other hand, there is no Xen-based unikernel that has both ends of the PV storage driver implemented. To be specific, rumprun leverages Mini-OS for Xen interfaces, such as Xenbus, and has the frontend driver implementation of Xen PV storage driver. Neither rumprun nor Mini-OS has the blkback implemented, and it cannot be leveraged from NetBSD because the backend is associated with platform-specific implementation. This situation poses the challenge of design and implementation of blkback

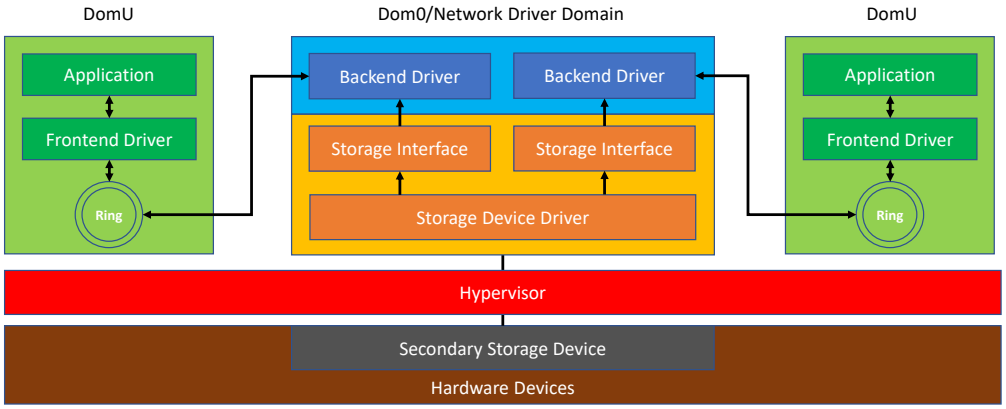


Figure 4.1: Xen's PV storage driver model.

in the unikernel context, which is fundamental for getting a unikernelized storage domain.

Now, a rumprun blkback needs some specific attention, unlike the commodity OSs, because of its any kernel philosophy, which allows us to use physical storage drivers from NetBSD. To access the physical device driver from NetBSD from rumprun, we need to access the storage device interfaces provided by NetBSD. At the same time, since our goal is to leverage the unikernel concept, we cannot use core kernel functionalities from NetBSD such as memory management and scheduling. For these functionalities, we depend on rump kernel. Therefore, it requires a careful separation and interaction between what we can leverage from NetBSD and what we cannot.

Xen provides some device-specific scripts for commodity OSs. The execution of these scripts helps the backend domain (Dom-0 or driver domain) to accomplish specific operations necessary for paravirtualization. Xen invokes these scripts in Dom-0 or the driver domain upon the corresponding front-end device driver's request for a backend device driver. For instance, the block script retrieves requested storage device-specific information and writes them to the Xenbus database. Rumprun is a single address space OS, which lacks the luxury of running scripts. Therefore, accomplishing these block script-specific tasks for the rumprun driver domain is another challenge.

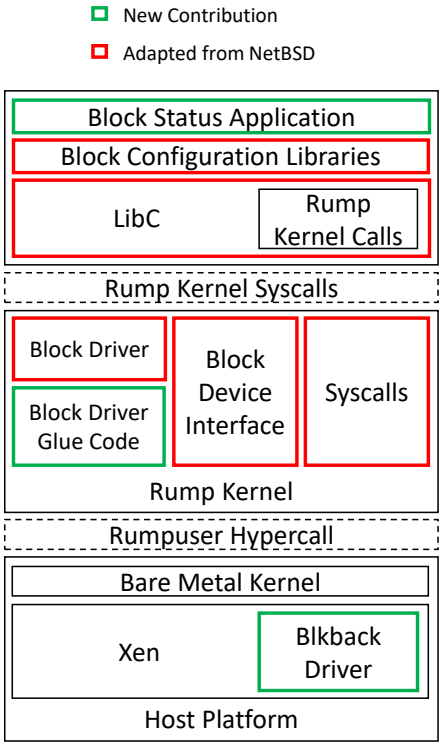


Figure 4.2: Rumprun storage domain.

4.2 Storage Device Driver

The first step to establishing a driver domain for a particular device is to provide full control of that device to a Xen domain. This device assignment can be done using Xen's PCI passthrough mechanism from the Dom-0. This way, for the unikernelized storage domain, we assign a storage device to a rumprun unikernel. Now, as we have a storage device assigned to the rumprun unikernel, we need a corresponding storage device driver. Therefore, the next step is to get a suitable storage driver that works for the rumprun unikernel.

From the discussion in chapter 2, we know that this driver belongs to the rump kernel layer in the rumprun stack as shown in Figure 4.1. Inspired from rump kernel's philosophy, we aim to import the storage device driver from another OS (i.e., NetBSD) and use it unmodified. Therefore, it benefits us saving development and maintenance efforts. However, to realize this vision and ensure that the driver works on the underlying platform and exports its functionalities, we will need to provide glue code, which belongs to the rump kernel layer.

Though we import the storage driver from another OS, the driver relies on the underlying kernel support from the target OS. Rump kernel provides an unmodified system call (syscalls) interface, which lets the imported driver use rump kernel functionalities without any modification occurring to it. Moreover, to interact with the device driver for storage operations, such as read or write on the disk, we need block device interfaces. These driver interfaces can be derived from other OSs, and they belong to the rump kernel layer like syscalls. In rump kernel, both syscall and device driver interfaces are denoted as factions (see Figure 4.2).

4.3 Storage Backend Driver

In a virtualization setup, multiple guest VMs should be able to use the same or different storage devices. Therefore, the storage backend driver is the most crucial part of a storage driver domain because it makes fast storage virtualization possible. Each guest machine is responsible for providing a storage frontend (blkfront) for each of its PV storage. On the other hand, the storage domain is responsible for providing a blkback against each of these blkfronts as shown in Figure 4.1.

To realize when to populate a blkback instance, our storage domain monitors all guest VMs and their properties all the time. If any guest VM seeks a PV storage, the storage domain negotiates with that VM and create a blkback instance. The blkback and blkfront communicate and transfer data between each other using Xen's shared memory and event channel mechanisms. Since the blkback driver is very hypervisor-specific, it belongs to rumprun's platform layer as shown in Figure 4.2.

We divide the blkback into platform-dependent and platform-independent layers (upper and bottom) according to the rumprun's philosophy. The bottom layer handles requests for block data from the blkfront and sends responses using Xen shared ring buffer. Depending on the type of the request (read, write, etc.), the bottom layer communicates with the upper layer. The upper layer performs the read and write operations on the storage device using the block

device interface. The upper layer is also responsible for forwarding the feedback from the device driver to the bottom layer.

Each request is consists of multiple block segment information to perform a particular operation, such as read, write, etc., on those segments of the block device. A write-request requires reading block data from the shared memory and writing them to the storage device. In contrast, a read-request requires reading from the storage device and writing them to the shared memory. In Xen, the shared memory access is performed using the grant table operations, which requires hypercall execution, and they are time-consuming.

The blkfront notifies the blkback about the requests using a Xen event channel. Therefore, to receiver those notifications, the blkback has a notification handler. To prevent the requests from accumulating and to accelerate the request-handling, we introduce a thread that is dedicated to reading all pending requests, performing operations on the storage device accordingly, and then it goes to sleep. The notification handler only wakes up this thread when there is a notification. This way notification handler can respond as soon as there is any new request.

A response against a request is sent to the blkfront when requested operations are performed on the storage device. However, performing block operations on the storage device is also time-consuming. Therefore, if a request is handled only when the response for the previous request is sent to the blkfront, it is highly likely that the number of pending requests will keep accumulating, and soon the ring will be full of requests. Therefore, the overall response time to the operations will be very high and results in low throughput and high latency, which is not desirable. Therefore, A response is sent to the blkfront once the corresponding requested operation is performed on the storage device by the device driver. Instead of waiting for the operation to be done and then sending the response and handle the next request afterward, we send the responses asynchronously. It means we keep handling the requests and submitting the requested operations. We send responses only when the storage device accomplishes those operations. This way subsequent requests do not have to wait for

the response to be sent for the current request.

There are few other optimization techniques we have used in out blkback driver. They are discussed below.

- **Batching:** A request may have multiple segment information to perform operations. It is possible that one segment consecutive to the next segment. Similarly, multiple requests may have consecutive segments where the first segment of a request starts from the last segment of the previous request. In such cases, if the same operation is supposed to be performed on these segments, we combine these operations into one and perform a single storage operation.
- **Persistent reference:** A request may have multiple grant references where each grant reference points to a shared memory location. The corresponding segment data is read from or written to these locations by the blkfront and blkback. However, the shared memories are allocated by the blkfront. Therefore, to access these memories, the blkback needs to map the grant references, and after finish accessing these memories, the blkback needs to unmap grant references. This grant reference mapping and unmapping requires cost significant time due to context switching for hypercall. To reduce this cost, we take advantage of Xen's persistent grant referencing feature, which lets blkfront and blkback to reuse already mapped references. It means a grant reference does not necessarily need to be unmapped, and later, if the same reference is used for segment data, the corresponding memory can be accessed using previously mapped pages. This optimization significantly saves processing time.
- **Indirect segments:** One request can contain up to 11 grant references, and each of these references points to one 4K page of shared memory. Therefore, one request may help transfer a maximum of 44K data between blkfront and blkback. This limitation may act as a potential bottleneck for paravirtualizing high throughput storage devices such as NVMe SSD. Xen supports a feature called indirect referencing to overcome this limitation. Here, instead of holding block data, indirect grant references point to pages that contain segment information such as sector numbers and grant references to

the corresponding shared memories. This way, each page can hold up to 512 segment information. Each request may contain a maximum of 8 indirect segments, which makes a total of 4096 segment information that each request can transfer. As each of these segments can point to one 4K page of data, each request can help transfer up to 16MB of data between blkfront and blkback. In our blkback design, we leverage this feature for improved storage throughput.

4.4 Application

Xen provides block device scripts for NetBSD and Linux, where these scripts retrieve some storage device information and write them to the Xen store database. The blkback reads this information from the database to learn about the device. Since we cannot run shell scripts in rumprun, we design an application to retrieve device-specific information and pass them to the blkback. Since rumprun is a single address space OS, we schedule context switching, between the blkback and this application, in a time-sharing manner.

Chapter 5

Unikernelized Storage Domain Implementation

The existing rumprun unikernel is not designed for storage domains and does not have any backend implemented for I/O devices. Therefore, along with the core PV storage backend driver, the unikernelized storage domain requires other implementations such as physical device interface, backend invocation, and application. In this chapter, we present the implementation detail for the unikernelized storage domain based on the design choices as described in chapter 4.

In section 5.1, we discuss how the blkback driver access the physical storage driver. In section 5.2 and 5.3, we discuss backend driver invocation, initialization, and its connection setup with the frontend driver. Section 5.4, 5.5, 5.6, and 5.7 contains blkback implementation detail. Finally, section 5.8 describes the application implementation for replacing Xen storage script.

5.1 Block Device Interface

There are a few block device interfaces/functions in NetBSD that the storage domain needs to use for performing operations on the storage devices. The upper layer is responsible for invoking these functions.

For each block device, NetBSD has a constant data structure called Block Device Switch (bdevsw). This data structure is defined in the physical driver with the mandatory entry points such as open, close, strategy, dump, size, flags, ioctl, etc. For performing any operation on the device, we heavily depend on the interfaces that let us use these entry points.

The step before performing any operation on a device is to check the device's availability and status. The physical device driver names each device. Therefore, as a part of the sanity check, we first check if a device has a name using devsw_blk2name(). If it does not return any name, then the device is assumed to be non-existent. Next, we search for the device's bdevsw structure using the bdevsw_lookup() function.

In UNIX-based OSs, a device is treated as a file. Therefore, we create a virtual node (vnode) for the storage device, which represents an inode of a storage device file. As a result, we can perform open, close, read, and write operations on it like files. We get a vnode for a block device using the bdevvp() function provided by NetBSD. The next obvious step is to perform the open operation on the device followed by read/write, ioctl, and close operations device done using VOP_OPEN(), bdev_strategy, VOP_IOCTL, and vn_close, respectively, as necessary.

Reading and writing on the block device is done using constructing buffers first and then submitting these buffers using the bdev_strategy function. Along with being the data place holder, the buffer holds various information such as source address, operation type, a callback function, etc. We initialize this buffer before constructing it with the necessary information and destroy the buffer after we are done using the buffer. We use the device driver interface to initialize and destroy the buffers. The device driver reads data from the buffer and writes them to the storage device for write operations. For read operations, upon finish reading data from the storage device, the device driver writes the data on the buffer. After a completed operation, the device driver invokes the aforementioned-callback function.

The usage of these above-discussed block device interfaces from the upper layer is mostly, except for the callback function, triggered by the lower layer. However, the lower layer is

the Xen-specific layer executed in the rump kernel context, where the upper layer uses the device interface executes in NetBSD codes. Since rumprun is a single process kernel, we need to context switching between rump kernel and NetBSD code. Therefore, when we switch from the bottom layer to the upper layer, we use cooperative scheduling by exiting from the rump kernel code using rumpuser___hyp.hyp_schedule() function. Later, after upper layer execution we get back to the rump kernel context using rumpuser___hyp.hyp_unschedule() function.

5.2 Blkback Invocation

A guest VM (DomU) creation is a discrete event. Therefore, a blkfront invocation can happen anytime, which will need a blkback to be invoked on the driver domain side to pair and make PV I/O happening. To detect blkfront creations, we create a dedicated thread for monitoring Xenstore database changes regarding the storage domain. Xenstore adds a new set of blkfront information to the storage domain's backend path if a new VM wants to use this storage domain.

We maintain a watch list of Xenstore paths that we are interested in and set a callback function for each of them. The dedicated thread runs xbdw_thread_func(), which continuously watches for changes in the Xenstore database, and if there is any change for any listed path, the corresponding callback function is invoked. This way, we invoke the xbdback_instance_search() function when there is a change in the storage domain's backend path. This function investigates the changes in the path by traversing all blkfront subdirectories. The probe_xbdback_device() function detects blkfronts with XenbusStateInitialising status, which essentially means a new blkfront instance is created that is waiting for a blkback to pair.

In the xbdback_Xenbus_create() function, we first make sure there is no blkback instance created previously for this blkfront by checking the list for blkback instances. Then we create an instance of blkback for this blkfront. A blkfront instance is an object of xbd-

back_instance structure containing various information about the requested storage device, blkfront, blkback, shared ring, etc. We add blkfront and blkback absolute paths to the watch list so that for any changes in those ends, xbdback_frontend_changed() and xbdback_backend_changed() methods are invoked, respectively. Lastly, in this function, we read the device's major-minor (physical-device) number and add them to the blkback path on the Xenstore database.

5.3 Blkback Initialization and Connection

In the xbdback_backend_changed() function, we first do the sanity check on the requested storage device as described in section 5.1 requested by the corresponding blkfront. Then, we add the blkback properties on the Xenstore database. They include sector information such as the number of sectors and each sector size, read/write mode, features such as flush support for cache, persistent grant reference, the maximum number of indirect segments. Finally, the blkback connection status is changed to connected (XenbusStateConnected).

Upon reading these properties from Xenstore, the blkfront also changes its connection status to XenbusStateConnected, which causes the xbdback_frontend_changed() function to be invoked. As a result, xbdback_connect() routine finishes the rest of the steps to complete the connection between blkfront and blkback. This function reads the blkfront properties, which include ring reference, event channel number, persistent grant reference support, and I/O protocol. In Xen, the communication between the frontend and backend parts happens through special I/O ring buffers, which are built on top of a shared memory mechanism called the grant table [3]. The ring reference is a reference to the location of the shared memory buffer. After reading the properties from Xenstore, the ring is mapped to a page so that blkback can access the ring, and an event handler is set to handle notifications from the blkfront using the event channel.

5.4 Event Handler and Request Handler Thread

As discussed in section 4.3, to keep the event handler available to the request notifications, we create a request handler thread (`xbdback_thread`)in `xbdback_connect()` routine. The event handler routine wakes up the xbdback_thread if it is not already awake upon a new notification.

The xbdback_function handles the I/O processing using a call graph adapted from NetBSD's blkback implementation. The xbdback_co_main_loop() copy the request into the blkback object, and the xbdback_co_io_loop keep reading segment information from the copied requests. The xbdback_co_io_gotfrag2() function construct the buffers in batches, which are initialized by xbdback_co_io_gotio(), with the segment information. The xbdback_co_map_io() function maps the grant references to the pages, from the storage domain's address space, which is used as the block data place holder in the buffer.

From the xbdback_co_do_io() function, the device driver interface is invoked using the upper layer to perform I/O operations on the disk. VOP_IOCTL interface is used for performing cache flush operations, and bdev_strategy function is used for submitting read/write operations.

5.5 Handling Device Driver Responses

As requests are received from blkfront, they are processed, and requested operations are submitted to the physical device driver. However, no response against the requests is sent to the blkfront until the lower layer hears from the device driver regarding the submitted operations. NetBSD buffer structure lets us set a callback function, which is invoked by the device driver when it ends performing the submitted operations.

We set a common callback function in the upper layer for all buffers. This function eventually invokes a function in the bottom layer, so it can unmap grant references, unless they are persistent, send the responses to the blkfront, and destroys the used buffers.

Based on the success or failure of the operations submitted to the device driver, the xbdback_iodone() function sends responses to the blkfront by putting them in the ring. Blkback sends notifications using the event channel to let the blkfront know about the placed responses.

Once all responses regarding batched operations are prepared and sent to the blkfront, it is crucial to release all used data structures, so we can efficiently reuse the memory. Therefore, we destroy the used buffers by invoking the device driver interface and put back other used data structure to the pools.

5.6 Persistent Reference

For all segments in one request, it is possible to batch the grant mapping so that we can map the references on consecutive pages. The benefit of doing that is we can use only one buffer to perform operations on the device for all segments in one request. For persistent referencing, we do not unmap grant references and tend to reuse the previously mapped pages if the corresponding grant references appear in a new request. Otherwise, a grant reference is mapped to a newly allocated page, if not mapped before. The issue with the persistent referencing is that there is no guaranty that the same set of grant references will appear

in another request. As a result, we cannot reuse the consecutive pages from the batched mapping of those references. Therefore, we discretely map pages for grant references. One mapped page acts as a place holder for data in a buffer. Thus, ultimately, we use one buffer for each segment. However, we batch the buffers to submit all segment operations in one request.

To maintain the list of mapped pages, we implement a simple array of mapped page addresses. The indices of this array represent the grant references. When a buffer is constructed for a segment, we first check if the corresponding grant reference already has a mapped page in the array. If we get a mapped page, we reuse that page. Otherwise, we map the grant reference in a newly allocated page. On the other hand, before destroying the buffers, we put the pages back in the array instead of unmapping them.

5.7 Indirect Segments

Indirect segment implementation requires an extra parsing of the requests if it indicates the operation type is indirect. This parsing is done in xbdback_co_main_loop(). First, we map the grant references to pages. Then we parse these pages, where each of them may contain up to 512 indirect segments. However, by default Linux support maximum 32 indirect segments. Therefore, we also limit the maximum number of indirect segments to 32.

We store the parsed indirect segments in `bdi_indirect_segments` in the blkback object so that we can read their information in xbdback_co_io_loop() where prepare segments for I/O operations.

5.8 Application

Each device has a unique major-minor number pair, which is used for identifying devices separately. The Xen block scripts in NetBSD and Linux reads this number for the requested devices and add them to the Xenstore database. Since we do not have this script, we read

the major-minor number pair for all of the available devices using a C application. This application constructs a list of devices with their major-minor numbers and availability, indicates if the device is already being used by a guest VM. The list is shared between the PV storage driver and the application.

If the application ends execution, the unikernel will halt. Therefore, we need to run the application until we want to run the storage domain. To do that, once the application finishes constructing the list, it unschedule itself by yielding periodically so that the other part of the unikernel gets context to run.

Chapter 6

Unikernelized Network Domain Design

Network devices are regarded as one of the two core devices for a guest machine. Like the PV storage devices, PV network devices use shared memory and ring buffers for the backend and frontend communication. However, unlike storage, networking requires two rings instead of one, as shown in Figure 6.1 which brings new challenges that require some new design decisions for unikernelizing a network domain. In this chapter, we discuss these challenges, our ideas to overcome them, and a high-level overview of our approach.

In section 6.1, we discuss the design challenges involved in building a rumprun-based network driver domain. In section 6.2 and 6.3, we discuss physical network driver and PV network backend association and design for rumprun architecture. In section 6.4, we discuss the method for linking the physical NIC and PV backends together to allow a guest to communicate with the rest of the world.

6.1 Challenges

As we have discussed in 4.1, rumprun lacks any I/O backend drivers in rumprun; i.e., there is no netback implemented, which is an integral part of a network driver domain. Therefore, the biggest obstacle for a unikernelized network domain is to design and implement a netback

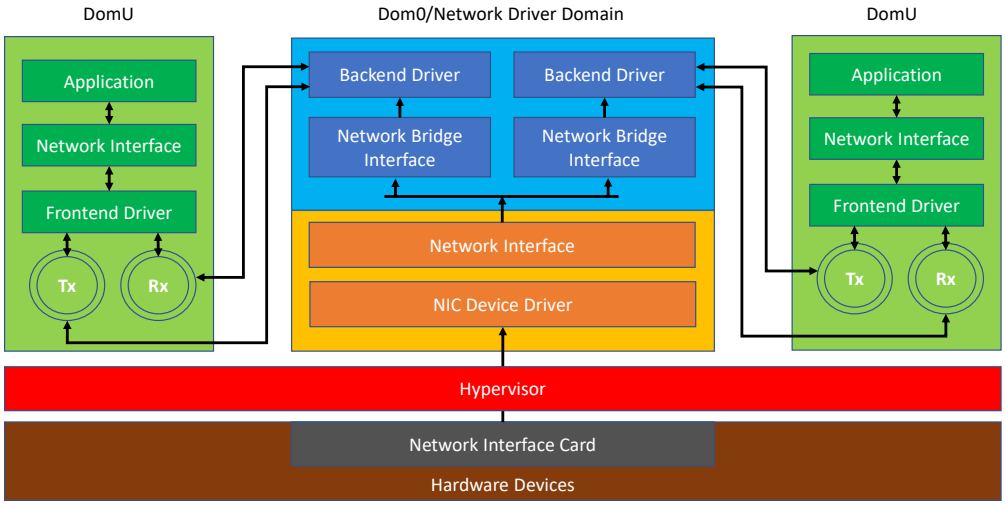


Figure 6.1: Xen's PV network driver model.

for rumprun that is compatible with any other OS with the network frontend. Moreover, like the storage domain in commodity OSs, netback heavily relies on various configuration scripts (e.g., network bridging across different backend instances), which cannot be used in rumprun-based environments.

Unikernelization of the network driver domain requires more than porting an existing netback driver from a commodity OS like NetBSD or Linux. On the one hand, unikernels are single-purpose and single address-space OSs. On the other hand, a netback driver must consistently maintain the data flow for both receive and transmit rings which takes place asynchronously. For smooth receive and transmit operations, netback must communicate with the TCP/IP stack in rumprun and also with the frontend. Therefore, we redesigned the netback driver as a single-process unikernel that is capable of seamless duplex communication without sacrificing performance or existing frontend compatibility.

Depending on the number of netfronts in DomUs, there can be several netback instances in the driver domain. Usually, to link these netbacks to a physical NIC, different techniques are used such as bridging, switching, routing, network address translation, and virtual local area networks. These techniques are independent of netback and run as a separate process that is triggered when necessary. Since rumprun is a single-process application, we cannot run

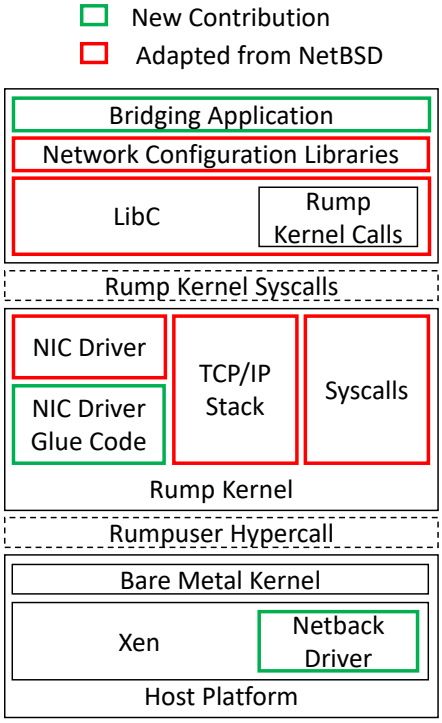


Figure 6.2: Rumprun network driver domain design.

the linking mechanism as a separate process, unlike Linux or NetBSD. Rather, our rumprun instance runs a special program in a dedicated thread, so that netback, integrated with the unikernel, links to a physical NIC with a minimal performance penalty.

6.2 Network Device Driver and Interface

Similar to the storage domain driver, our goal is to import physical NIC drivers from NetBSD and use it unmodified to save on development and maintenance efforts (see Figure 6.2) for the network domain. And likewise, to make it happen and leverage the drivers' functionalities for the underlying netback driver, we will need to provide corresponding glue code in the rump kernel layer.

Besides the network device driver, we also need other components in the rump kernel. One of them is the interface to the device driver, so the netback can communicate, like sending and receiving network packets, with the NIC. Therefore, we use a TCP/IP stack that provides

a network interface for a way of communication between netback and the device driver. TCP/IP stack, which is considered as a *faction* in the rump kernel layer along with the syscalls faction, which provides an unmodified system call interface to drivers as described in 4.2.

6.3 Netback Driver

A vital part of the rumprun network driver domain is the netback driver. There should be one corresponding netback in the driver domain for each virtual network interface from the netfront at guest virtual machines. Netback and netfront communicate and transfer data between each other using Xen's shared memory and event channel mechanisms.

Like the unikernelized storage domain, we create platform-dependent and platform-independent layers for the network domain. We divide the netback driver into two layers. The upper layer is responsible for communicating with rumprun's network stack. For any incoming packets (from the network stack) destined for a DomU, this layer places the packets into a memory buffer and forwards them to the Xen-specific bottom layer. The bottom layer places these memory buffers into a corresponding Xen I/O ring buffer and notifies netfront.

Similarly, for a stream of network packets originating from DomU, the bottom layer receives them in memory buffers through the ring buffer and forwards them to the upper layer. Then, the upper layer pushes these memory buffers into the network stack. Aside from ring buffer operations, other hypervisor-specific operations between netback and netfront, such as interrupts through an event channel, are done at the bottom layer.

The aforementioned duplex communication occurs as asynchronous events. The netback driver is expected to complete the interactions as soon as there is any data available from the netfront driver. However, rumprun lacks rich OS support for interrupts and work queues. In our design, we exploit multi-threading so that netback does not block any hypervisor-based event mechanisms and processes data fast.

Therefore, our design includes an event handler that is invoked when there is a notification from netfront regarding data request or response. Often, these notifications require operations involving hypercalls, which are time-expensive. Spending significant time in the handler may result in a bottleneck, blocking other incoming notifications. Thus, we introduce a dedicated thread, activated by the handler, to take over and perform necessary actions in response to notifications.

In contrast to netfront notifications, netback needs to respond to the network stack operations. For example, any data received from the stack should be forwarded to netfront. Thus, netback uses network stack callback routines to respond to the network stack operations. Spending too much time in these routines may delay subsequent operations because a response may require hypercall execution, which is expensive. To minimize this response time and prevent the stack operations from blocking, we introduce another thread which is activated by the routines. This thread performs the necessary processing and execution for the corresponding operations so that the network stack routines can respond to operations without blocking.

Like the unikernelized storage domain, we create platform-dependent and platform-independent layers for the network domain. We divide the netback driver into two layers. The upper layer is responsible for communicating with rumprun's network stack. For any incoming packets (from the network stack) destined for a DomU, this layer places the packets into a memory buffer and forwards them to the Xen-specific bottom layer. The bottom layer places these memory buffers into a corresponding Xen I/O ring buffer and notifies netfront.

Similarly, for a stream of network packets originating from DomU, the bottom layer receives them in memory buffers through the ring buffer and forwards them to the upper layer. Then, the upper layer pushes these memory buffers into the network stack. Aside from ring buffer operations, other hypervisor-specific operations between netback and netfront, such as interrupts through an event channel, are done at the bottom layer.

The aforementioned duplex communication occurs as asynchronous events. The netback

driver is expected to complete the interactions as soon as there is any data available from the netfront driver. However, rumprun lacks rich OS support for interrupts and work queues. In our design, we exploit multi-threading so that netback does not block any hypervisor-based event mechanisms and processes data fast.

Therefore, our design includes an event handler that is invoked when there is a notification from netfront regarding data request or response. Often, these notifications require operations involving hypercalls, which are time-expensive. Spending significant time in the handler may result in a bottleneck, blocking other incoming notifications. Thus, we introduce a dedicated thread, activated by the handler, to take over and perform necessary actions in response to notifications.

In contrast to netfront notifications, netback needs to respond to the network stack operations. For example, any data received from the stack should be forwarded to netfront. Thus, netback uses network stack callback routines to respond to the network stack operations. Spending too much time in these routines may delay subsequent operations because a response may require hypercall execution, which is expensive. To minimize this response time and prevent the stack operations from blocking, we introduce another thread which is activated by the routines. This thread performs the necessary processing and execution for the corresponding operations so that the network stack routines can respond to operations without blocking.

6.4 Linking Netback with a Physical Device

There are several approaches that can be used to connect netback to a physical NIC and thus link netfront with a wider network. Bridging is one of the popular methods used for Xen PV networking. How and when bridging should take place is decided by a given driver domain OS. For instance, Linux runs a driver domain *daemon* for starting services needed

in the corresponding Xen driver domain. Along with other responsibilities, this service runs networking scripts that set up a bridge.

A single-process unikernel cannot run multiple services or scripts. We therefore introduce a *unified application* that keeps looking for requests by DomUs, creates new netback instances, and connects netbacks to the physical NIC using the bridge when necessary. This application runs in a dedicated thread, so that it does not block the netback driver.

Chapter 7

Unikernelized Network Domain Implementation

The details for Xen PV network driver domain implementation is discussed in this chapter. A netback instance is needed for each netfront that uses this driver domain for its PV network devices. Netback driver implementation provides virtual network interfaces. The bridge implementation connects these interfaces with the physical network device.

In section 7.1, we discuss about the interface between netback and NIC driver. In section 7.2, we discuss the process of the creation and pairing of netback with netfront. We talk about the communication between the NIC, netback, and netfront in section 7.3, 7.4, and 7.5. In Section 7.6, we describe how we use an unmodified NIC device driver from NetBSD [16] in our rumprun-based network domain. Finally, how we implement the bridge for connecting NIC and netbacks is discussed in section 7.7.

7.1 Virtual Network Interface

Each netback driver instance creates a virtual network interface to use as a gateway to communicate with the outside world through the NIC. To do that, we leverage the TCP/IP stack from NetBSD, which offers a set of functionalities to talk to the device.

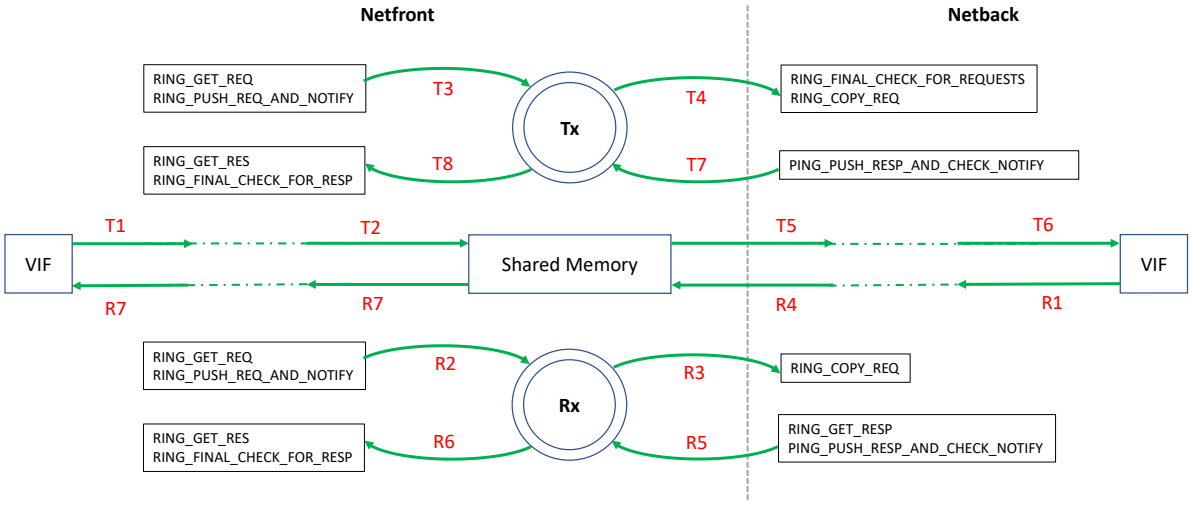


Figure 7.1: Data exchange between netback and netfront.

Netback uses if_init() function to Initialize and mark the interface running. If any network device-specific IOCTL operation needs to be performed, if_ioctl() function is used by netback. The if_start() function is invoked when any packet from the outside world reaches the netback through the interface. On the other hand, netback enqueues outgoing packets to the network stack using if_percpuq_enqueue() function.

7.2 Netback Invocation, Initialization, and Connection

In Xen's PV driver model, a separate netback instance must be created for every netfront counterpart in DomU. Similar to blkback invocation as described in section 5.2, a netback instance is created by network domain for each new netfront instance. We add the backend path of network domain to watch list for monitoting new netfront instances. A change in that path triggers functions to investigate the change and invoke corresponding netback instance.

Netfront and netback drivers use two ring buffers: Tx and Rx. Tx is used by netfront to send data to netback, which eventually forwards this data to the bridge through the netback virtual interface (VIF). On the other hand, when the netback virtual interface receives any data from the bridge, netback forwards it to netfront using Rx. Netfront does the ring initialization and corresponding shared memory allocations. Netback reads Tx and Rx ring

grant references from Xenstore. To be able to access Tx and Rx ring buffers, netback maps their grant references in its memory address space and keeps track of ring buffer accesses using producer-consumer indices.

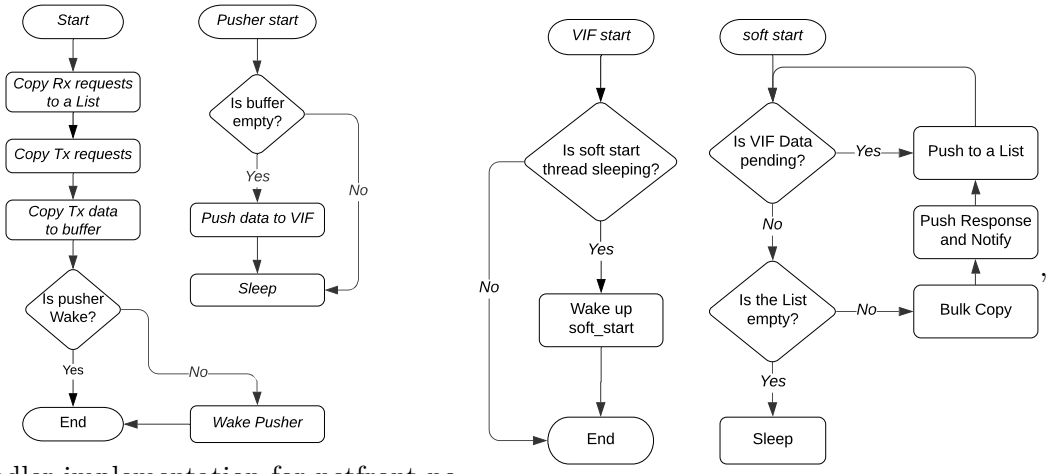
The communication between netback and netfront happens asynchronously. Therefore, event channels (virtual interrupts) are used for asynchronous notifications to communicate between front- and back- ends effectively in Xen. Netback binds the event channel to the event handler so that relevant notifications can be handled. For both transmit and receive, a network PV driver can use the same event channel. However, Xen supports separate event channels for transmit and receive as well. Our netback implementation supports both types.

Since hypervisor has all machine memory mapped for all domains, it is faster to copy data from one domain to another using hypervisor. To provide faster data transfer between netfront and netback, Xen supports hypervisor-based data copy, and nowadays, most of the netfronts from OSs such as Linux and NetBSD utilize this advantage. Therefore, in our rumprun network driver domain, we implement this feature.

Netback creates a VIF for sending data to the network stack. To distinguish one netback VIF from another in the network driver domain, each VIF is assigned a unique name. The first step for initializing a netback instance is to update the Xenstore database so that this netback's properties, such as VIF name, event channel type, support for data copy, etc., are advertised to other domains.

7.3 Transmit

In a DomU, any data pushed to a corresponding interface is received by the netfront. Netfront transmits the network data to the netback using the Tx ring buffer. Upon receiving the data, netback pushes its VIF. The ring buffer is consists of multiple slots that can be used for requests as well as responses. A Tx request means a request to transmit data from netfront through netback. Each request is consists of the address of the page containing data, corresponding offset, relevant flags, size of data, and ID of the request. As shown in



(a) Handler implementation for netfront notifications.

(b) Handling incoming data from VIF.

Figure 7.2: Threaded implementation of netback for efficient interaction with netfront.

figure 7.1, after inserting request(s) in the ring buffer (T3), netfront sends a notification via an event channe. The notification handler in netback copies the unhandled request(s) (T4) and maps the pages in memory using the grant table mechanism (T5). The memory contents are then delivered to the VIFs (T6).

Finally, netback sends responses to netfront on the accomplished Tx operations by utilizing the slots from already served requests. Once these slots are filled with responses, netback pushes them to the Tx ring buffer (T7) and updates netfront by sending an event notification if required. During the whole request serving process, netback keeps updating its Tx request consumer and response producer indices.

7.4 Receive

Upon receiving network data from netback, netfront delivers them to its associated VIF. The role of netback is to receive the data from its attached VIF and push it to the Rx ring buffer. Like Tx, the Rx ring buffer consists of multiple slots that can be used for requests and responses. However, the front and back end interaction for the Rx ring is not inverse of the Tx ring but similar in some aspects.

If the netfront sends any Rx requests (R2), netback copies the request (R3) when convenient but cannot send any data until it receives any from the VIF. A function in the netback, which gets invoked (R1) when the VIF receives any network data so that netback knows there are pending incoming data. Upon such invocations, netback copies the data in the pages associated with the copied Rx requests using the grant table mechanism. Next, to let netfront know the result of the copy operation, netback reuses the served request slots for responses. After pushing the responses to the Rx ring, netback notifies the netfront if required. Like transmit operation, netback updates all Rx request and response indices.

7.5 Threaded Implementation

Rumprun is a single process unikernel. We make use of multithreading for faster Tx and Rx operations.

The event handler in netback is invoked when netback receives a notification. Netback must not block the handler for a long time so that a notification can be received and handled as soon as possible. However, handling notification requires manipulation of shared pages, which involves Xen hypercalls. Since hypercalls are expensive and inappropriate for latency-sensitive contexts, we use a separate thread, called *pusher*, which is responsible for pushing data to netfront using such hypercalls.

As shown in figure 7.2a, the handler copies pending Rx requests from netfront to a list so that they can be read when any data is available at the VIF. Next, it reads the Tx requests and copies the corresponding data to a buffer. Then, the handler wakes up the pusher thread if it is sleeping (because it goes to sleep once it pushes the buffer contents to the VIF). This way, the handler does not have to wait for the data to be copied by the hypervisor to finish its routine.

On the other hand, when any data is available at the VIF, a callback function called *start*, is invoked. Here, netback's responsibility is to send this data to netfront using the Xen grant

table copy mechanism if there is any pending Rx request, which is time-consuming. Thus, if the start function does the copy operation, consecutive VIF data may have to wait until the copying is done, and netback may have to do additional copy operations. Therefore, we introduced another thread called `soft_start`, which performs copy operations in a batch using only one hypercall. As shown in figure 7.2b, in our netback implementation, the start function's only responsibility is to wake up the `soft_start` thread, which usually goes to sleep after a copy operation, if it is sleeping. As a result, data at the VIF is handled as soon as possible and efficiently.

7.6 Physical Device Driver

The design of rump kernels enables us to leverage existing NetBSD drivers with no modifications. Using the PCI passthrough mechanism (and IOMMU for better protection and security), Xen delegates a corresponding PCI device (NIC) to our rumprun network driver domain. Apart from the NIC driver itself, we use the TCP/IP stack and other network stack components from rump kernels.

7.7 Bridging Application

Using a network bridge, we connect the VIF interfaces from netbacks to the physical NIC. To that end, we developed a bridging application in our rumprun network driver domain. When this application is launched, it creates a bridge interface. Next, the application assigns an IP address to the physical interface; the physical interface works as a gateway for incoming and outgoing packets across all VIFs. Then, the application keeps scanning for new network interface creations. When a new VIF appears, the application adds the new interface to the bridge.

We ported the *ifconfig* utility from NetBSD to initialize bridge interfaces and assign IP addresses. Along with that, we also ported the *brconfig* utility from NetBSD, which is used for adding interfaces to a bridge. In order to give CPU context to the other factions such

as netback driver, physical driver, and network stack, the bridging application yields its schedule in a time-shared manner.

Chapter 8

Storage Domain Evaluation

In this chapter, we evaluate our rumprun-based storage driver domain. The evaluation results help to determine if the storage domain implementation incurs any performance overhead compared to existing counterpart solutions. Since all existing driver domains use Linux, and there does not exist NetBSD-based driver domains, we only compare against a Linux-based storage driver domain.

Our evaluation uses both micro- and macro- benchmarks. We do microbenchmarking using DD [6] and Filebench [63] to measure overall storage device throughput. We use macrobenchmarking tools such as Sysbench [62] and Filebench [63] to measure the performance of real-life applications like MySQL [11] and MongoDB [10] database server, fileserver, webserver, etc.

Table 8.1 shows the hardware configuration for the machine that we use for setting up Xen 4.9 and running the Dom-0, storage driver domains, and the guest domain (DomU). The configurations for these domains are shown in table 8.2. The NVMe SSD is assigned to the storage driver domain using the PCI passthrough. The mentioned applications run on the DomU and access the NVMe SSD using the Linux storage frontend. VM configurations are kept similar to get a fair performance comparison.

Table 8.1: Hardware configuration.

CPU	1 x Intel(R) Xeon(R) CPU E5-2695, 2.20GHz
Cores	24 per CPU (HT)
L1/L2 cache	32 KB / 256 KB per core
L3 cache	30720 KB
Main Memory	64 GB
Storage	Samsung 970 EVO Plus 500GB NVMe SSD

Table 8.2: Configuration of Xen domains on the server side.

	Dom-0	DomU	Linux Blkback	Rumprun Blkback
OS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Rumprun-SMP [34, 49]
Kernel	5.3.0-40-generic	4.15.0-88-generic	5.0.0-23-generic	Based on NetBSD 8.0
Memory	8 GB	5 GB	1 GB	1 GB
vCPUs	1	4	1	1

8.1 DD

dd is a command line tool in Linux, which lets us to perform read and write operations directly on the storage device. Therefore, we use this tool in DomU to measure and compare read-write performances on the PV storage device while using Linux and our Rumprun for storage domain. To keep the reading overhead minimal during write performance measurement for the disk, we use `/dev/zero` as the source device for write content. On the other hand, to keep the writing overhead minimal while measuring read performance, we use the `/dev/null` as the destination disk. Each experiments are run three times and each time 10GB of data is read-write from/to the device.

The experimental results are shown in Figure 8.1. It shows the storage device access throughput in MB per second. As one can see, for both read and write operations, Linux and Rumprun storage domain exhibits similar performance.

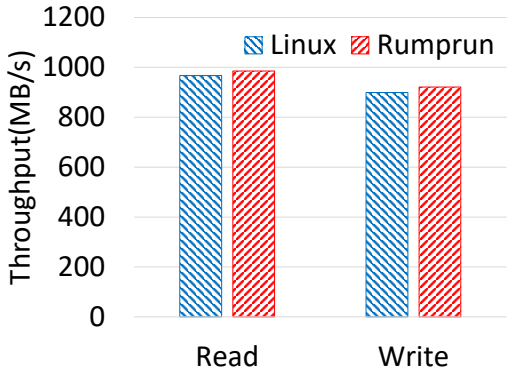


Figure 8.1: Throughput measurements using DD.

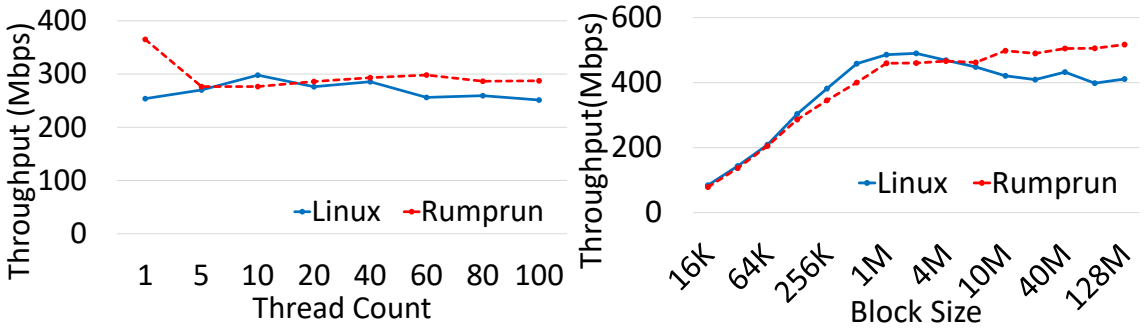
8.2 Filebench Microbenchmark

As we measure throughput for direct read-write operations on the storage device, we also measure read-write performance on a filesystem installed on the storage device. We install Ext4 filesystem on the storage device and run microbenchmarks from Filebench.

Filebench is a tool suite that is heavily used by researchers in academia and industry for benchmarking works associated with filesystem and storage benchmarking. Though filebench is a benchmarking framework, it comes with a few microbenchmarks intended for filesystems performance measurement. We take some of these microbenchmarks and modify them to measure maximum throughput for several file operations.

8.3 SysBench

SysBench [62] is a popular system component benchmarking tool, which is capable of running requests in threads so that multiple requests can run in parallel. For storage evaluation, we leverage SysBench (v1.1.0) for getting ideas on real-life application performance, running them on DomU that is using Linux and rumprun storage domain.



(a) Throughput varying number of threads. (b) Throughput for different block sizes.

Figure 8.2: File I/O Throughput measurements using sysbench.

8.3.1 SysBench File I/O

We measure the file I/O performance using the SysBench benchmark. For this experiment, SysBench uses 192 files totaling 15GB and performs random operations on these files with a read-write ratio of 3:2. We run the same experiment for a different number of threads, ranging from 1 to 100, and block sizes, ranging from 16KB to 128MB. Each experiment is run for 5 minutes, and we make sure to clear the read buffer cache between each run so that we can get actual storage throughput, not primary memory throughput.

Figure 8.2a shows throughput for runs with a different number of threads performing I/O operations of 256KB block size. On the other hand, Figure 8.2b shows throughputs for a fixed number of threads (20) but runs with a varying number of block sizes. As one can see from these figures, the throughput for rumprun storage domain is very comparable with Linux and even better than Linux for higher number of threads and block sizes.

8.3.2 SysBench MySQL

We setup the widely used MySQL database on the storage device in DomU and evaluate the database performance using SysBench. This databas contains 100 tables, each with 1,000,000 records, which totals 20GB of disk space. Since the primary memory size for DomU is 5GB, full database cannot fit in there, which reduces read buffering effect. We ran the benchmark varying number of threads (from 5 to 100) for performing complex SQL

queries on the database.

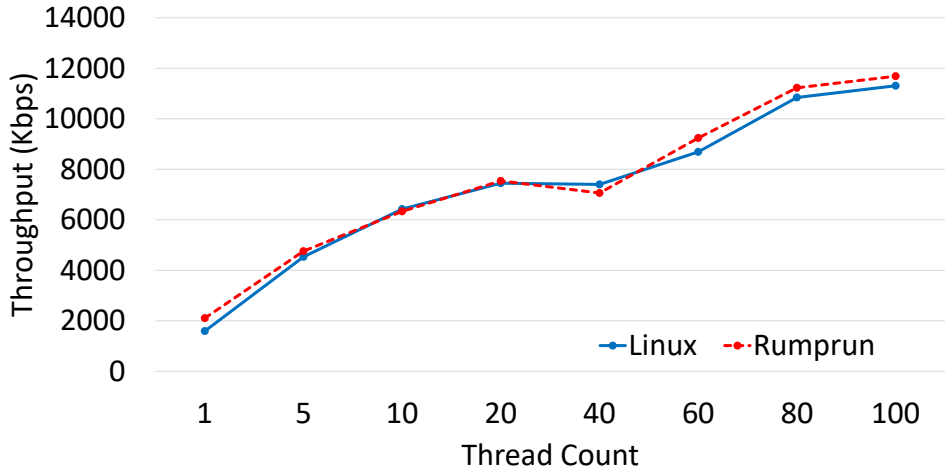


Figure 8.3: MySQL throughput measurements using filebench.

Figure 8.3 presents the throughput for the database operations for different number of threads. It shows both Linux and rumprun has overlapping throughput as storage domain.

8.4 Filebench Macrobenchmark

Along with the microbenchmarks, Filebench is shipped with a several macrobenchmarks, which are written in workload modeling language (WML) and are capable of creating workloads of real-life applications. We took few of them and modified as necessary to benchmark the storage domain that is virtualizing high speed NVMe SSD.

8.4.1 Filebench Fileserver

To generate a fileserver workload, we ran 50 threads in parallel each performing series of operation including create, read, write, append, close, stat, delete, etc. Before running the workload, filebench create 100000 files with average size of 128KB, which makes total around 13GB, i.e. at least twice the bigger than the assigned primary memory. The mean append size is 1KB where the I/O sizes are varied, from 16KB 8MB, for each run of 5 minutes.

The throughput for the described fileserver workload is presented against different number

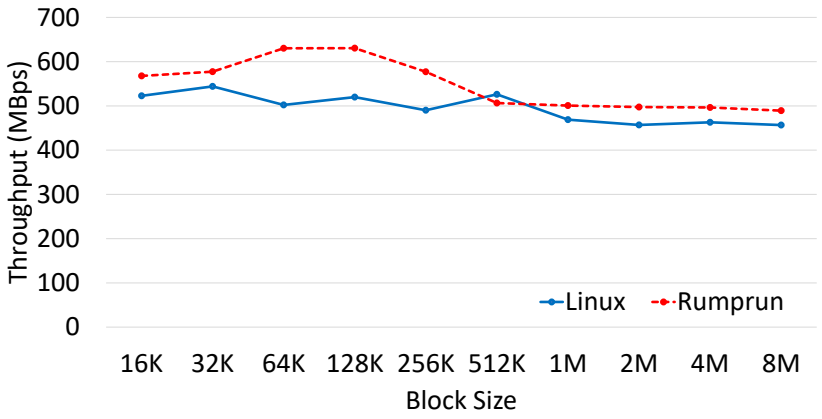


Figure 8.4: Fileserver throughput measurements using filebench.

of I/O block sizes in Figure 8.5. As one can see, rumprun storage often domain performs slightly better than Linux.

8.4.2 Filebench MongoDB Server

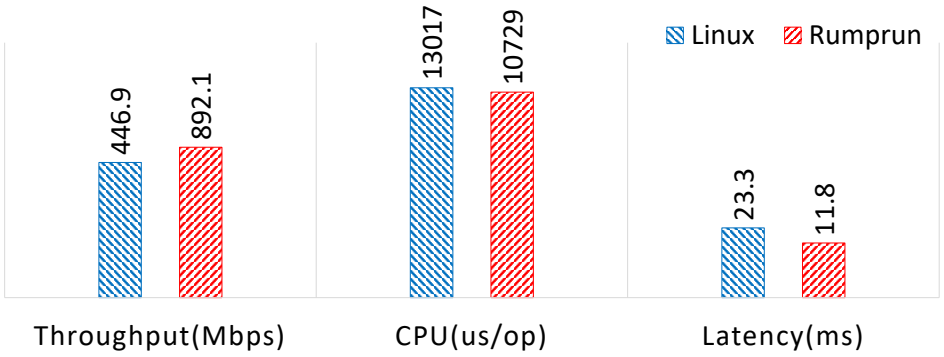


Figure 8.5: MogoDB server performance measurement using filebench.

8.4.3 Filebench Webserver

We generate the webserver workload running 50 threads in parallel, where each perfors series of operation combining open, read, and close. First, filebench create 200000 files with an average size of 64KB totaling around 13GB to make it at least twice the bigger than the assigned primary memory. Therefore, the effect of read buffer is mitigated. The mean append size is 16KB and the I/O size is 1MB where each operation is run for 5 minutes.

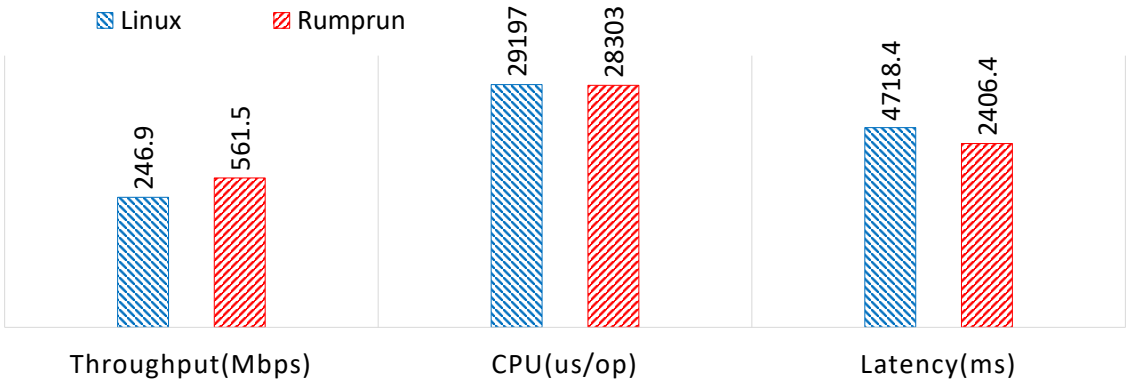


Figure 8.6: Webserver performance measurement using filebench.

Figure 8.6 shows the webserver throughput (TP), execution time per operation, and latency. It shows the rumprun storage domain takes a little less time than Linux for executing each operation and thus provides slightly better throughput. At the same time, the rumprun storage domain exhibits faster latency than the Linux counterpart.

Chapter 9

Network Domain Evaluation

In this chapter, we evaluate our rumprun-based network driver domain. The primary goal of our evaluation is to determine performance overheads, if any, due to our implementation. Like the sotrage domain evaluation in Chapter 8, we only compare against a Linux-based network driver domain.

Our evaluation uses both micro- and macro- benchmarks. We run the iPerf [5] and Wget [8] microbenchmarks to measure overall network throughput. We use macrobenchmarks including ApacheBench [64], Redis [13], Memcached [9], and MySQL [11] to measure the performance of real-life applications.

Table 9.1 shows our experimental setup. Our client and server machines are connected di-

Table 9.1: Hardware configuration.

	Server	Client
CPU	1 x Intel(R) Xeon(R) CPU E5-2695, 2.20GHz	Intel(R) Core(TM) i5-6600K
Cores	24 per CPU (HT)	4 per CPU
L1/L2 cache	32 KB / 256 KB per core	32 KB / 256 KB per core
L3 cache	30720 KB	6114 KB
Main Memory	64 GB	16 GB
Network	Intel Corporation I350 Gigabit adapter	RTL8111/8168/8411 PCIe Gigabit adapter

Table 9.2: Configuration of Xen domains on the server side.

	Dom-0	DomU	Linux Netback	Rumprun Netback
OS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Ubuntu 18.04.3 LTS	Rumprun-SMP [34, 49]
Kernel	5.3.0-40-generic	4.15.0-88-generic	5.0.0-23-generic	Based on NetBSD 8.0
Memory	8 GB	8 GB	2 GB	1 GB
vCPUs	1	22	1	1

rectly using a switch. Driver domains are tested on the server side. Our server runs Xen (Dom-0 is only used for the control path). Each tested application runs inside DomU. We create a Linux-based and unikernelized driver domains which have direct, PCI passthrough access to the NIC. Our rumprun environment also incorporates recent changes from LibrettOS [49] to run in Xen's HVM mode. Table 9.2 shows our configuration and software versions. We use out-of-the-box Xen 4.9 from Ubuntu. Driver domains are I/O-intensive and do not need to allocate more than one virtual CPU (vCPU) in our experiments.

9.1 iPerf

We use iPerf [5] (v3.1.3) to measure the network bandwidth, for TCP packets, that DomU can achieve when using our rumprun-based network driver domain and compare it with the Linux-based driver domain.

Figure 9.1 shows the receive and transmit bandwidth for DomU and the client machine communication. This microbenchmark result shows that our unikernelized network domain has very similar performance to that of Linux.

9.2 GNU Wget

To measure throughput for large file transfers, we used GNU Wget [8] (version 1.19.4). We transfer 16MB, 512MB, and 2GB files from DomU to the client machine. In separate runs, we saved these files onto the disk as well as `/dev/null`, so that we can measure throughput

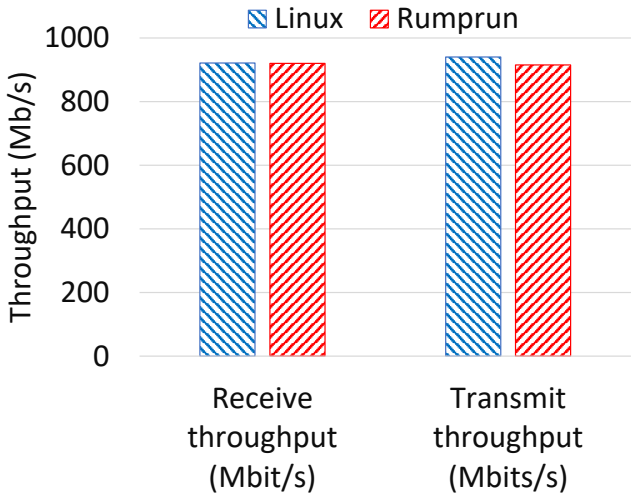


Figure 9.1: Throughput measurements using iPerf.

independent of any potential disk I/O.

Figure 9.2 shows that for rumprun and Linux driver domains, the throughput remains largely the same. Moreover, the throughput is unaffected by disk I/O in this test.

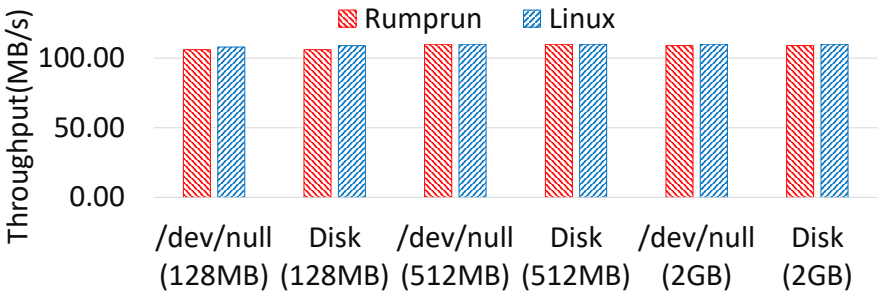
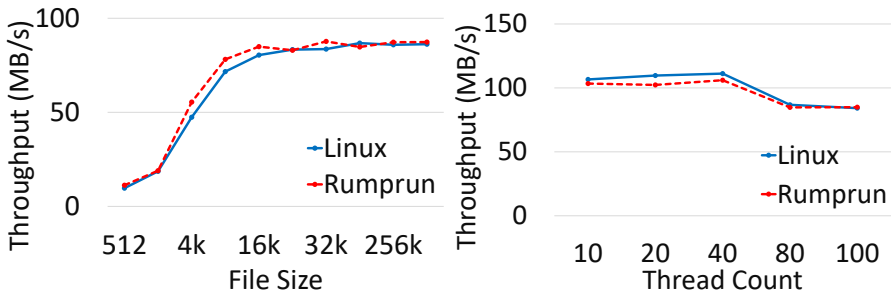


Figure 9.2: Throughput measurements using Wget for large TCP file transfer.

9.3 Apache

To evaluate performance with a real-life HTTP server, we run the Apache server (v2.4.29) in DomU and Apache benchmark in the client machine. The server data (files) are randomly generated. The benchmarking tool sends 100,000 requests and measures the server-side throughput, and each experiment is repeated 3 times. Figure 9.3a shows Apache server throughputs for different file sizes, ranging from 512B to 512KB, with 80 concurrent requests



(a) Varying file size. (b) Varying concurrency.

Figure 9.3: Apache server throughput.

for each run. Figure 9.3b shows throughputs for different number of concurrent requests, ranging from 10 to 100, for 128KB files. These numbers show the Apache server does not incur any overhead due to Cubicle.

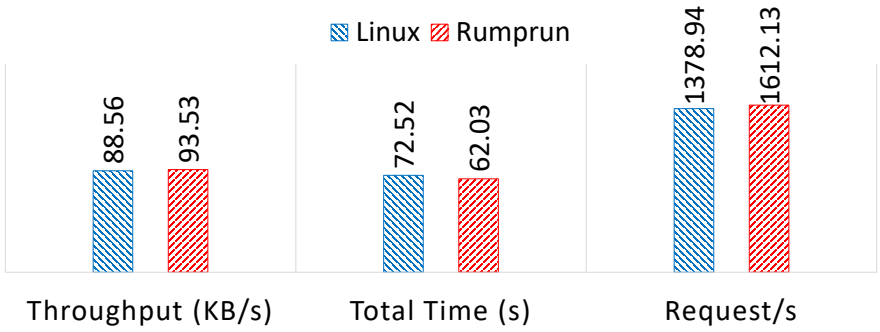


Figure 9.4: Apache throughput, transfer time, and request rate.

We show a specific example with transfer time, throughput, and request handling rate for 64KB in Figure 9.4. The experiment sends 100,000 requests (with 40 concurrent requests) using the Apache benchmark. The performance is marginally higher for rumprun. The maximum relative standard deviation (RSD) is 1.22% and 1.34% for Linux and rumprun, respectively, for the presented throughputs.

9.4 Redis

Nowadays, in-memory key-value databases are extensively used in cloud infrastructures. We ran Redis [13], a well-known key-value store to compare rumprun-based and Linux driver domains. We execute Redis (v4.0.9) with millions of SET/GET operations in the pipeline

Table 9.3: Relative standard deviation for different benchmarks.

	Apache	Redis	Memtier	Sysbench
Linux	1.22%	0.0007%	0.0029%	0.0167%
Rumprun	1.34%	0.0111%	0.0230%	0.1496%

mode. We set the pipeline size to 1000. Each run is repeated for different levels of concurrency wherein each GET/SET operation reads/writes 128KB of data.

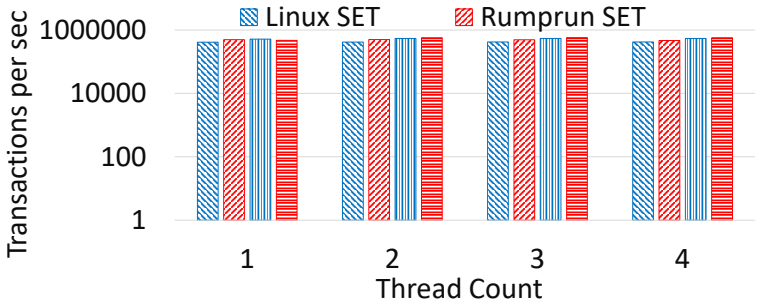


Figure 9.5: Redis key-value store throughput.

Figure 9.5 shows the number of SET/GET operations per second. Overall, rumprun and Linux netback exhibit very similar performance. The relative standard deviation for Linux's and rumprun's netback is 0.0007% and 0.0111%, respectively, as shown in Table 9.3.

9.5 Memcached

Memcached [9] is another widely used in-memory key-value store. We ran a memcached server (v1.5.6) in DomU, and compared its performance while using Linux and rumprun driver domain. On the client machine, we ran the memtier [12] benchmark (v1.2.17), which acts as a load generator and performs SET and GET operations using the memcache_binary protocol. We used a read-dominated workload with a 1:10 ratio for SET and GET operations.

Figure 9.6 shows the number of operations per second for different number of threads (1-20), wherein each thread ran 10 clients, each performing 100,000 operations. We repeated the

experiment with different block sizes (32B and 8KB). The relative standard deviation is 0.0029% and 0.023% for throughput measurement respectively for Linux's and rumprun's netback, as shown in Table 9.3.

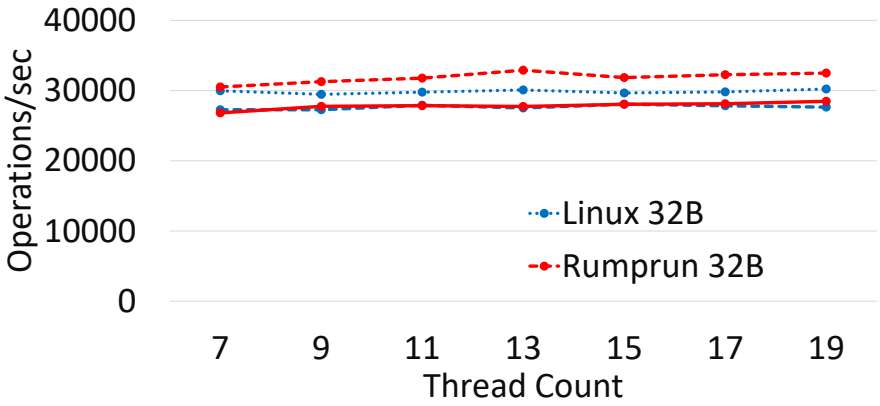


Figure 9.6: Memcached throughput.

We observe that memcached's throughput is similar for both Linux's and rumprun's netback. This indicates that rumprun netback does not add any performance overhead to the guest domain running memcached.

9.6 MySQL

Along with NoSQL databases like redis and memcached, relational databases are also widely used. We compared the performance of MySQL [11] server (v5.7.29), a popular SQL database, running on DomU, when it uses Linux and rumprun as the network driver domain. On the client machine, we ran Sysbench (v1.1.0), which sends SQL queries, triggering database transactions.

We created a database on the server with 10 tables, each with 1,000,000 records. We ran the benchmark from the client machine with the number of threads varied from 5 to 100. All data fits in memory, i.e., the workload is CPU-intensive and there is no I/O storage overhead. The benchmark sends read-only SQL queries to the server, which allows us to stress-test the network path.

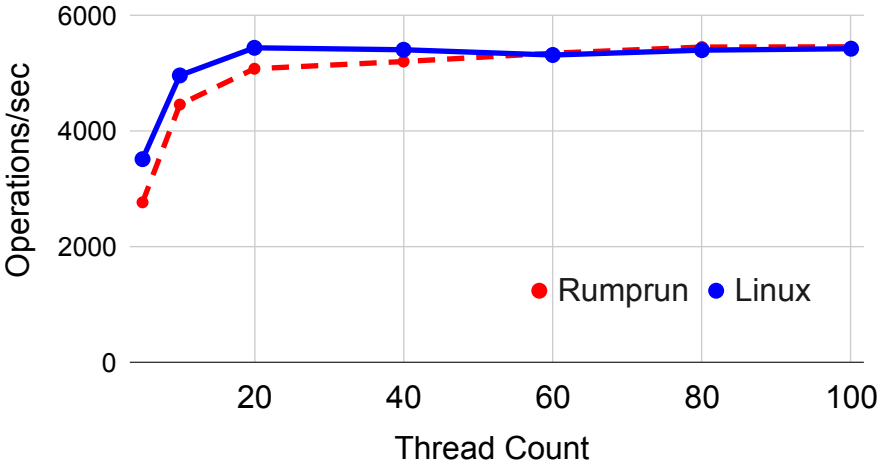


Figure 9.7: MySQL throughput.

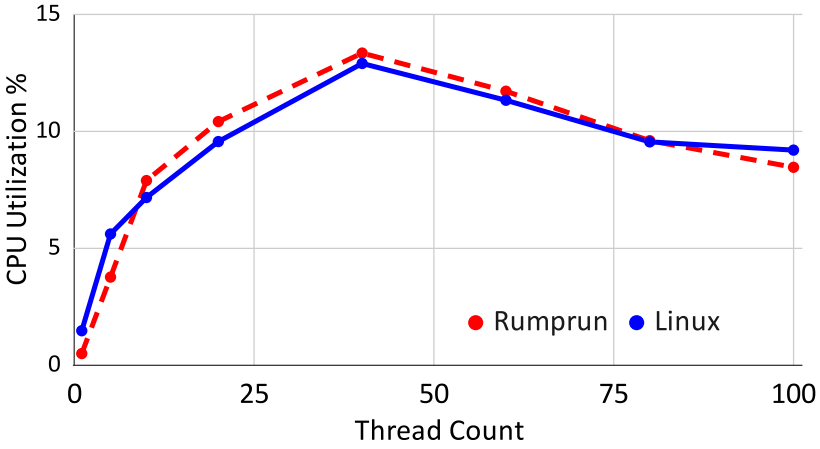


Figure 9.8: CPU utilization for the MySQL test.

Figure 9.7 shows the number of operations (queries and transactions) performed per second for different number of threads. We observe that there is almost no performance difference when using Linux's or rumprun's netback. The relative standard deviation is 0.0167% and 0.1496% when using Linux's and rumprun's netback, respectively.

We also measured CPU utilization. Figure 9.8 shows the average CPU utilization of DomU, measured using the sysstat utility [15], during the aforementioned benchmark execution. We observe that the DomU's CPU utilization for both Linux and rumprun is very similar. Therefore, the CPU utilization time is preserved.

9.7 Latency

Along with throughput and CPU utilization, we measured network latency for different applications including ping, memcached, and MySQL. We ran Linux ping for 100 times with one-second interval. Latency for memcached and MySQL is retrieved from the above-described benchmark runs of memtier and sysbench. Figure 9.9 shows the latency comparison when using Linux and rumprun network domains. It shows that the rumprun network domain achieves similar latency to that of Linux across different applications. Therefore, using rumprun-based driver domains for running latency-sensitive applications, we can achieve similar performance to that of Linux.

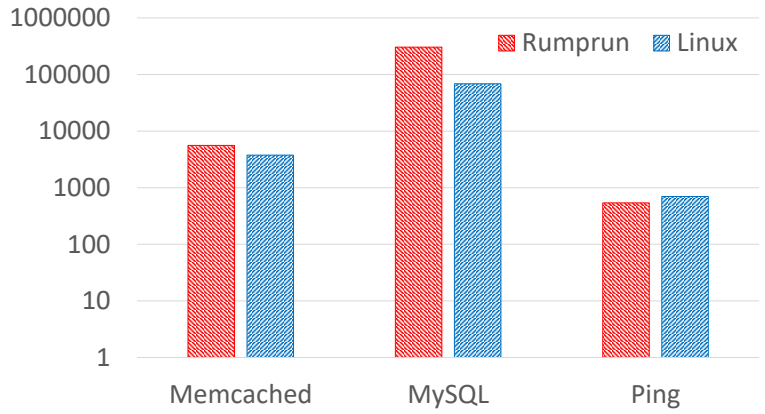
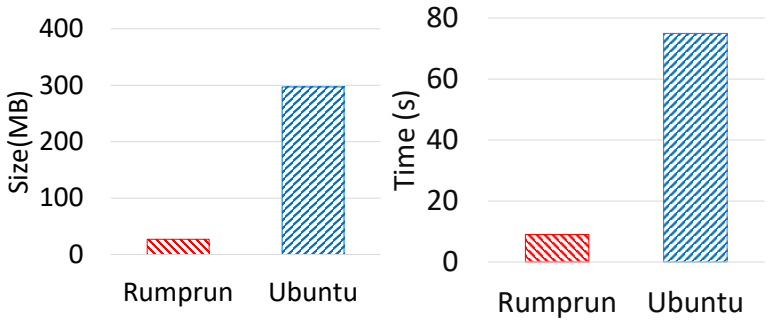


Figure 9.9: Latency comparison for Linux and rumprun network driver domains.

9.8 Image Size and Boot Time

We also compare the image size of rumprun vs. Ubuntu 18.04, the Linux distribution we used in the presented experiments. For rumprun, we measured the size of the entire rumprun binary. For Linux, we measured only the size of the kernel and its modules, i.e., did not include the size of user-space programs and libraries such as libc. As Figure 9.10a shows, the Linux image is about 10x bigger than the rumprun image.

Since the boot time directly affects deployment in the cloud infrastructure, it is important to reduce it as much as possible. Moreover, driver domains can potentially be restarted when recovering from failures, where faster boot times are equally important. As Figure 9.10b shows, rumprun only takes 13 seconds to boot the network domain. In contrast, Ubuntu needs 75 seconds.



(a) Size.

(b) Boot time.

Figure 9.10: Image size and boot time comparison.

9.9 Security Vulnerability

The number of ROP gadgets is one concrete, quantitative metric that can be used to evaluate security. A smaller number of gadgets indicates potential obstacles for an attacker since the attacker will have a hard time finding appropriate gadgets to exploit a known vulnerability. Moreover, the attack surface is proportionally reduced in rumprun, which also helps in achieving better security. Using the methodology from [30], we count gadgets belonging to different categories: Data move, Arithmetic, Logic, Control flow, Shift & Rotate, Setting

flags, String, Floating point, Misc, MMX, NOP, and RET. Each category represents a class of operations.

Figure 9.11 shows the quantity of ROP gadgets for the rumprun, vanilla Linux, and popular Linux distributions such as CentOS 8, Fedora Rawhide (05/2020), Debian 10.4, and Ubuntu 18.04. For rumprun, we measured the number of instructions from different categories present in the rumprun network domain image. For the vanilla Linux kernel, we did the same for the kernel image built with the default configuration (defconfig), which is a fairly minimalistic configuration. The vanilla image alone already has four times more gadgets than the rumprun image, but for using Linux as a network domain, we generally need kernel modules. (Not to mention user-space libraries.) Therefore, we measured the quantity of ROP gadgets for associated kernel modules along with the kernel image for the presented Linux distributions that are capable of running as the network driver domain.

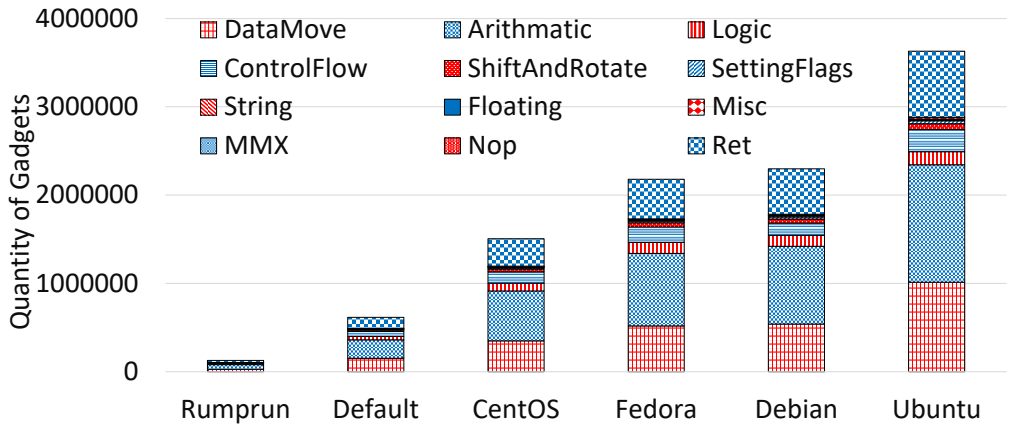


Figure 9.11: ROP gadget comparison.

We used Ropper [14] to measure the number of ROP gadgets. We found that the number of ROP gadgets for rumprun is 4x and 24x smaller than that of the Linux default configuration and Ubuntu, respectively. Rumprun's substantially smaller number of gadgets indicates its great potential for better security when using as a driver domain.

Chapter 10

Conclusions and Proposed Post-Preliminary Exam Work

In this dissertation, we presented the first unikernelized driver domain for Xen. While past efforts have explored reducing the attack surface of hypervisors, our work is the first to focus on improving memory footprint, isolation, and security of privileged components such as device drivers that run outside of the hypervisor. Our rumprun-based driver domains have several advantages compared to traditional Linux-based driver domains: a reduced number of gadgets, smaller image size, and faster boot time.

To realize the vision of unikernelized driver domains, we have designed and implemented separate storage and network driver domains. Each driver has several associated development challenges that we had to overcome. No existent unikernel works as a driver domain. Though rumprun benefits from NetBSD's rich driver base, it lacks backend drivers, the crucial component of any driver domain, implemented for Xen PV I/O devices. Therefore, we had to develop separate backends for network and storage domains that work with rumprun. Rumprun lacks work queues, full xen-tools support, and the ability to execute scripts. These challenges did not receive adequate attention by prior researchers. We overcame these challenges by designing multi-threaded backends, implementing an alternative to missing tools, and developing utility applications used for routing.

We evaluate our storage and network domains. We use a rich set of well-known benchmarking tools to perform storage and network evaluation. Performance results presented in this dissertation proposal are comparable to that of Linux-based driver domains. We also measure security and deployment properties and compare them with existing Linux-based OSs. Our evaluation shows that our rumprun driver domain provides competitive performance to that of a Linux-based driver domain while retaining all the benefits of unikernels.

10.1 Post-Preliminary Exam Proposed Work

We propose three directions for the post-preliminary exam work. These are discussed in the subsections that follow.

10.1.1 Backward Compatible High-speed Network

In this dissertation, we present a unikernelized network domain, which is compatible with guest VMs running commodity OSs, and evaluate performance for 1GbE NIC. Our implementation is as performant as Linux and able to achieve saturate the link. However, higher-speed NICs such as 10GbE and 40GbE are increasingly popular for server machines. As in Linux, though our netback implementation is compatible with these higher-speed NICs, we could not saturate them. Exploring the potential bottlenecks in the existing PV design and improving the unikernelized network domain for high-throughput NICs would be a prospective direction for post-preliminary exam work.

Several reasons behind the bottlenecks in Linux are already discussed, and certain approaches were proposed [45, 53, 57] to improve Xen's PV network driver for achieving 10 Gbps. According to Santos et al. [53, 57], the PV driver model itself is a significant bottleneck, which requires more CPU cycles to process network packets than regular device drivers due to copying overheads, kernel overheads, and hypervisor overheads associated with the added operations for PV drivers. They propose some optimizations including disabling the netfilter bridge, moving data copies to guests, extending the grant table mechanism, and leveraging

multi-queue NICs. Some of these optimizations require PV design changes. Riddoch et al. [45] propose an accelerator plugin for network PV drivers. This plugin allows to bypass netback and lets netfront communicate directly with the NIC to receive and transmit packets.

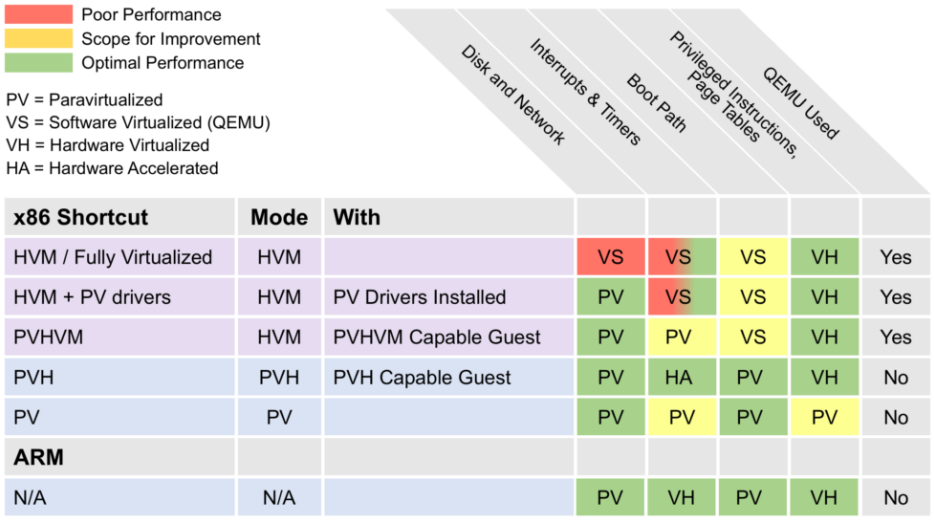
These prior works require modifications to Linux (netback and netfront) and Xen, which will break compatibility. None of these proposals are yet accepted by the Linux and Xen communities. Therefore, there is scope for further research on how to improve PV network drivers for achieving saturated throughput for high-speed NICs and develop an unikernelized driver domain without losing compatibility.

There are several challenges associated with this proposed work. One of them includes determining a better alternative to solving the already explored bottlenecks and identifying other bottlenecks that are contributing to poor performance. Another challenge is to propose a design that is backward compatible with the existing PV driver implementations so that no VM is forced to adapt the modification in its OS. Lastly, because of architectural differences with commodity OSs, meeting additional requirements to unikernelize this new driver domain would be another challenge.

10.1.2 Rumprun PVH

As shown in Figure 10.1, Xen offers multiple virtualization modes. For the work presented in this dissertation, we leveraged rumprun with HVM (or PVHVM to be specific) support from LibrettOS [49]. However, the last addition to this list is PVH, which is a lightweight virtualization mode that does not require any QEMU emulation and benefits from hardware-accelerated virtualization. The PVH support in rumprun is yet to be implemented, which can be another potential post-preliminary exam direction.

There are similarities between PVHVM and PVH mode. Both use hardware virtualization, such as Intel VT or AMD-V extensions of the host CPU, for virtualizing. For I/O device virtualization, such as network and storage, both PVHVM and PVH rely on PV device drivers. Hardware virtualization features are used for trapping privileged instructions and



x86 Shortcut	Mode	With					
HVM / Fully Virtualized	HVM		VS	VS	VS	VH	Yes
HVM + PV drivers	HVM	PV Drivers Installed	PV	VS	VS	VH	Yes
PVHVM	HVM	PVHVM Capable Guest	PV	PV	VS	VH	Yes
PVH	PVH	PVH Capable Guest	PV	HA	PV	VH	No
PV	PV		PV	PV	PV	PV	No
ARM							
N/A	N/A		PV	VH	PV	VH	No

Figure 10.1: Overview of the various virtualization modes implemented in Xen. (Courtesy: Xen Project [17])

manipulating page tables by both modes.

PVHVM uses Software Acceleration, such as Local APIC, Posted Interrupts, Viridian (Hyper-V) enlightenments, and makes use of guest PV interfaces because they are faster. In contrast, PVH leverages hardware acceleration support instead of the PV interface. Unlike HVM guests, PVH guests do not require QEMU to emulate devices, which makes the PVH guests lighter than PVHVM.

We expect that enabling rumprun with the PVH mode would make it lightweight and faster than it is now with the PVHVM mode. However, there are challenges in doing that, such as adopting PVH for rumprun, which we did not need for PVHVM. Moreover, PVH integration is still a work-in-progress, and several features are yet to be added to Xen. For instance, PCI passthrough support is still absent, but it is essential for driver domain implementation. We anticipate that these missing features will be supported soon and propose to upgrade rumprun and our rumprun-based driver domains in the post-preliminary exam work.

10.1.3 KVM Driver Domain

This dissertation presents unikernelized driver domains for Xen, which is a widely used hypervisor. KVM is another hypervisor, which is also very popular and supported by the Linux kernel. There are some fundamental architectural differences between these two hypervisors. For instance, the OS used for KVM installation is treated as a host VM, and other VMs are treated as guests. In contrast, Xen treats every VM as a guest VM, but Dom-0 is treated as a privileged guest VM. KVM does not yet support the concept of distributing core device (network and storage) responsibilities to driver domains, unlike Xen. However, since KVM is a popular hypervisor, and the driver domain abstraction improves security by imposing isolation, leveraging the driver domain concept for KVM would be another direction for post-preliminary exam work.

Like Xen, KVM supports its own PV device drivers, and the communication between frontend and backend is maintained using shared memory ring buffers. This mechanism is known as virtIO in KVM. Nowadays, an extension of the virtIO protocol, named vhost (virtual host), is used for device virtualization. To achieve an improved PV throughput, vhost uses QEMU processes for control plane communication and kernel-to-kernel shared memory for the data plane communication.

Several technologies, such as DPDK (for network) and SPDK (for storage), leverage vhost in KVM. At the same time, recent approaches, such as Demikernel [7], leverage unikernels for providing better and user-friendly interfaces to DPDK and SPDK applications. We expect that separate driver domains for network and storage will provide better isolation and security for such services and other guest VMs. Like Xen, KVM supports PCI passthrough, but introducing a separate domain running virtIO and vhost backends along with the drivers would be the most challenging part. Therefore, we propose the improvisation of virtIO and vhost protocols for driver domains in KVM as another potential direction for post-preliminary exam work.

Bibliography

[1] 2013. Driver Domain. https://wiki.xenproject.org/wiki/Driver_Domain. Online, accessed 02/19/2020.

[2] 2016. Docker Acquires Unikernel Systems to Extend the Breadth of the Docker Platform. https://www.docker.com/docker-news-and-press/docker-acquires-unikernel-systems-extend-breadth-docker-platform Online, accessed 02/28/2020.

[3] 2018. Grant Table. https://wiki.xen.org/wiki/Grant_Table. Online, accessed 02/19/2020.

[4] 2018. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ Online, accessed 09/15/2018.

[5] 2019. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/ Online, accessed 03/03/2020.

[6] 2020. dd - convert and copy a file. https://man7.org/linux/man-pages/man1/dd.1.html. Online, accessed 12/02/2020.

[7] 2020. Demikernel. https://www.microsoft.com/en-us/research/project/demikernel/ Online, accessed 05/26/2020.

[8] 2020. GNU Wget. https://www.gnu.org/software/wget/ Online, accessed 05/26/2020.

[9] 2020. Memcached. http://memcached.org/ Online, accessed 05/26/2020.

[10] 2020. mongoDB. The database for modern applications. https://www.mongodb.com/ Online, accessed 05/26/2020.

[11] 2020. MySQL. https://www.mysql.com/ Online, accessed 05/26/2020.

[12] 2020. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark/ Online, accessed 05/26/2020.

[13] 2020. Redis. https://redis.io/ Online, accessed 03/03/2020.

[14] 2020. Ropper. https://github.com/sashs/Ropper Online, accessed 05/26/2020.

[15] 2020. SYSSTAT Utilities. http://sebastien.godard.pagesperso-orange.fr/ Online, accessed 05/26/2020.

[16] 2020. The NetBSD Project. https://netbsd.org. Online, accessed 02/19/2020.

[17] 2020. Xen Project. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview Online, accessed 05/26/2020.

[18] IEEE Std 802.11 a. 1999. Wireless LAN medium access control (MAC) and physical layer (PHY) specification: high-speed physical layer in the 5GHz band. (1999).

[19] AMD, Inc. [n.d.]. AMD I/O Virtualization Technology (IOMMU) Specification. http://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf.

[20] T. E. Anderson. 1992. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*. 92–94. https://doi.org/10.1109/WWOS.1992.275682

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP'03)*. 164–177. https://doi.org/10.1145/945445.945462

[22] Manel Bourguiba, Kamel Haddadou, Ines KORBI, and Guy Pujolle. 2014. Improving Network I/O Virtualization for Cloud Computing. *Parallel and Distributed Systems,*

IEEE Transactions on 25 (03 2014), 673–681. https://doi.org/10.1109/TPDS.2013.29

[23] Manel Bourguiba, Kamel Haddadou, and Guy Pujolle. 2012. Packet Aggregation Based Network I/O Virtualization for Cloud Computing. *Computer Communications* 35 (02 2012), 309–319. https://doi.org/10.1016/j.comcom.2011.10.002

[24] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, 250–257.

[25] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture* 12, 1 (2017), 1–206.

[26] Shakeel Butt, H Andrés Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. 2012. Self-service cloud computing. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 253–264.

[27] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 189–202. https://doi.org/10.1145/2043556.2043575

[28] Intel Corp. 2018. Intel Clear Containers. https://clearlinux.org/documentation/clear-containers. Online, accessed 08/04/2018.

[29] Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the Gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (London, UK) *(ESSoS 2016)*. Springer-Verlag, Berlin, Heidelberg, 155–172. https://doi.org/10.1007/978-3-319-30806-7_10

[30] Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the Gadgets. In *Proceedings of the 8th International Symposium on Engineering Secure Software and*

Systems - Volume 9639 (London, UK) *(ESSoS 2016)*. Springer-Verlag, Berlin, Heidelberg, 155–172. https://doi.org/10.1007/978-3-319-30806-7_10

[31] Cloud Native Computing Foundation. 2020. Production-Grade Container Orchestration. https://kubernetes.io/. Online, accessed 02/28/2020.

[32] Tom Goethals, Merlijn Sebrechts, Ankita Atrey, Bruno Volckaert, and Filip Turck. 2018. Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications. 1–8. https://doi.org/10.1109/SC2.2018.00008

[33] Intel Corporation. [n.d.]. Intel's Virtualization for Directed I/O. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[34] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[35] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2003. Operating system support for virtual machines. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*. 71–84.

[36] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[37] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. https://doi.org/10.1145/3342195.3387526

[38] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*.

[39] David Law. 2019. IEEE Standard for Ethernet-Amendment 1: Physical Layer Specification and Management Parameters for 2.5 Gb/s and 5 Gb/s Operation over Backplane. *IEEE Std 802.3 cb-2018 (Amendment to IEEE Std 802.3-2018)* (2019).

[40] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[41] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 461–472.

[42] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 461–472.

[43] Anil Madhavapeddy and David J Scott. 2013. Unikernels: Rise of the virtual library operating system. *Queue* 11, 11 (2013), 30–44.

[44] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[45] Kieran Mansley, Greg Law, David Riddoch, Guido Barzini, Neil Turton, and Steven Pope. 2007. Getting 10 Gb/s from Xen: Safe and Fast Device Access from Unprivileged Domains, Vol. 4854. 224–233. https://doi.org/10.1007/978-3-540-78474-6_27

[46] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems*

Design and Implementation (Seattle, WA) *(NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[47] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[48] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. 2008. Improving Xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 151–160.

[49] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. 2020. LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 114–128. https://doi.org/10.1145/3381052.3381316

[50] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[51] Pierre Olivier, AKM Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. 2019. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 85–96.

[52] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 291–304. https://doi.org/10.1145/1961295.1950399

[53] Kaushik Ram, Jose Santos, Yoshio Turner, Alan Cox, and Scott Rixner. 2009. Achieving 10 Gb/s using safe and transparent network interface virtualization. *Proceedings*

of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'09, 61–70. https://doi.org/10.1145/1508293.1508303

[54] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS'19)*. ACM, New York, NY, USA, 7–13. https://doi.org/10.1145/3317550.3321445

[55] John Scott Robin and Cynthia E. Irvine. 2000. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*. 129–144.

[56] Joanna Rutkowska and Rafal Wojtczuk. 2010. Qubes OS architecture. *Invisible Things Lab Tech Rep* 54 (2010).

[57] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. 2008. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 29–42.

[58] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 671–688. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg

[59] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[60] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen.. In *NDSS*.

[61] Sushrut Shirole. 2014. Performance Optimizations for Isolated Driver Domains.

[62] Sysbench Contributors. [n.d.]. SysBench 1.0: A System Performance Benchmark. http://sysbench.sourceforge.net/.

[63] V. Tarasov, E. Zadok, and S. Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *login Usenix Mag.* 41 (2016).

[64] The Apache Software Foundation. [n.d.]. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[65] D. Williams and R. Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO, USA. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[66] Bruno Xavier, Tiago Ferreto, and Luis Jersak. 2016. Time provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2016)*. IEEE, 277–280.

[67] Lingfang Zeng, Yang Wang, Dan Feng, and Kenneth Kent. 2015. XCollOpts: A Novel Improvement of Network Virtualization in Xen for I/O-Latency Sensitive Applications on Multicores. *IEEE Transactions on Network and Service Management* 12 (06 2015), 1–1. https://doi.org/10.1109/TNSM.2015.2432066

[68] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*.