A Performance Evaluation of Hypervisor, Unikernel, and Container Network I/O Virtualization

Pekka Enberg

Master's Thesis
University of Helsinki
Department of Computer Science

Helsinki, May 17, 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author
Pekka Enberg

Työn nimi — Arbetets titel — Title

A Performance Evaluation of Hypervisor, Unikernel, and Container Network I/O Virtualization

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level Master's Thesis	Aika — Datum — Month and year May 17, 2016	Sivumäärä — Sidoantal — Number of pages 64

Tiivistelmä — Referat — Abstract

Hypervisors and containers are the two main virtualization techniques that enable cloud computing. Both techniques have performance overheads on CPU, memory, networking, and disk performance compared to bare metal. Unikernels have recently been proposed as an optimization for hypervisor-based virtualization to reduce performance overheads. In this thesis, we evaluate network I/O performance overheads for hypervisor-based virtualization using Kernel-based Virtual Machine (KVM) and the OSv unikernel and for container-based virtualization using Docker comparing the different configurations and optimizations. We measure the raw networking latency and throughput and CPU utilization by using the Netperf benchmarking tool and measure network intensive application performance using the Memcached key-value store and the Mutilate benchmarking tool. We show that compared to bare metal Linux, Docker with bridged networking has the least performance overhead with OSv using vhost-net coming a close second.

ACM Computing Classification System (CCS):
Networks → Cloud Computing

Avainsanat — Nyckelord — Keywords
virtualization, performance, unikernels, library os

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Virtualization is the main technology that powers today's cloud computing systems. It is used in public clouds, such as Amazon Web Services (AWS) [8], Google Compute Engine (GCE) [14], and Microsoft Azure [12], and in private clouds that are provisioned, orchestrated, and managed by tools such as OpenStack [13] and Docker [16]. Virtualization provides isolation and resource control, which enables multiple workloads to run efficiently on a single shared machine [37] and thus allows servers that traditionally require multiple physical machines to be consolidated to a single cost effective physical machine using virtual machines or containers [60]. Virtualization is also proposed to be used in networking infrastructure for network function virtualization (NFV). Network appliances, or middleboxes, are traditionally implemented as hardware appliances. Deploying new functionality typically also requires deploying new hardware, which takes a long time and is expensive. NFV replaces hardware-based middleboxes with software-based systems that are less expensive and allow deploying new functionality as software updates. As a consequence, NFV reduces deployment time significantly [49].

While virtualization has many benefits, the performance overhead of virtualization compared to bare metal is still an open topic [50][37]. Both hypervisors and containers, two of the most popular virtualization techniques, have performance overhead in areas of CPU, memory management, and I/O, which all contribute to the performance penalty of network intensive applications under virtualization. Networking performance is particularly important for both cloud computing and NFV workloads. Cloud computing workloads like Memcached depend heavily on TCP/IP performance. NFV workloads involve very high packet rates, which makes them sensitive to latency introduced by any part of the system [57]. Various optimization and variations are proposed to minimize virtualization overheads but their effectiveness is still largely mystified.

In this thesis, we present our attempt to demystify virtualization overheads. We first provide an overview on hypervisors, containers, I/O virtualization, and unikernels to understand all the components that affect virtualization performance. We then detail known virtualization related performance issues and suggested solutions that are presented in literature. Finally, we evaluate the performance of networking I/O for latency and throughput using Netperf and Memcached for different combinations of virtualization techniques and compare them to baseline bare metal performance. The main difference to previous virtualization performance comparisons [50] [37] [42] is that we focus on networking performance across both hypervisors and containers by measuring the network processing latency distribution. This gives us more detailed understanding of the performance characteristics across various virtualization techniques.

1.1 Virtualization Techniques

The two main virtualization techniques that provide multitenancy on a single host machine are hypervisor-based virtualization and container-based virtualization. For hypervisor-based virtualization, I/O virtualization and unikernels are two important optimizations for achieving high performance. These virtualization techniques are detailed in Section 2.

Hypervisor-based virtualization lets you run multiple operating systems on a single host machine by creating isolated virtual machines (VMs) that each have a set of virtualized CPUs, memory, and I/O devices such as networking and storage devices. The virtualized resources are mapped to physical hardware resources either by sharing a physical resource between multiple VMs or by assigning a physical resource to a specific VM. As presented in Section 2, hypervisors can be broadly categorized into Type-1 and Type-2 depending on whether the hypervisor runs directly on top of the hardware or under a host operating system, respectively. The Xen hypervisor is an example of a Type-1 hypervisor whereas KVM-based hypervisors such as QEMU/KVM are typically categorized as Type-2 hypervisors.

Container-based virtualization lets you run multiple independent userspaces on a single host machine on a shared kernel image. On Linux, containers provide multitenancy by utilizing operating system kernel namespaces and control groups. Namespaces enable a process or a process group to see a virtualized view of the filesystem, processes, network stack, and other parts of the system. Control groups provide support for accounting and limiting resource usage for a process group. Docker is a high level tool that simplifies creation and management of Linux containers but still use the same namespace and cgroup mechanisms under the hood.

Unikernels are a light-weight alternative to general purpose operating system kernels such as Linux. They are optimized for running a single application per virtual machine. Unikernels follow the library OS model that structures operating system services such as memory management, scheduling, and networking as libraries that the application program links against. There is no split between kernel-space and userspace but rather all code runs in a single address space, which reduces guest operating system overhead by being more memory efficient and reducing context switch times. Various unikernel designs have been proposed in literature. We focus on OSv, a Linux compatible unikernel, Arrakis, an unikernel for SR-IOV capable systems, and ClickOS, a minimal unikernel for NFV middleboxes.

1.2 Virtualization Overheads

Virtualization imposes overhead on CPU, memory, networking and disk performance. These overheads are detailed in Section 3.

Hypervisors have CPU performance overheads from double scheduling [61] (hypervisors and guest OSes are unaware of each other's scheduling decisions), scheduler unfairness between VMs [55], asymmetric CPU speeds because virtual CPUs (vCPUs) run on shared physical CPUs [63], and interrupt handling [32]. Hypervisors also have memory performance overheads from memory reclamation and memory duplication. Memory reclamation overheads are caused by the hypervisor having limited information on what guest pages are good candidates for eviction, which can cause the hypervisor and the guest operating system to make bad paging decisions [64]. Memory duplication overheads come from each virtual machine having its own copy of the full operating system because of isolation requirements [64]. Networking performance has overheads from packet processing [57] and unstable network [66]. Storage performance has overheads from disk I/O scheduling [29]. Many of the overheads are amplified because guest operating systems are not fully aware that they are running in a virtualized environment or they have not been optimized to work with virtualized I/O access costs that are much higher than with real hardware.

Containers do not have the same kind of CPU performance overheads as hypervisors do because all applications run on the same host kernel. However, containers are affected by memory duplication problems – although to a lesser degree as executables and libraries can shared across containers. Container networking performance has overheads from the host operating system network stack performance because NICs are not virtualized and from network address translation (NAT) [37]. Container implementations like Docker also have disk performance overheads from layered filesystems that copy-on-write semantic by default [37].

1.3 Contributions

In this thesis, we evaluate the performance of different virtualization techniques compared to bare metal and show that:

- Containers have virtually no performance overhead compared to bare metal for network intensive applications when containers are configured appropriately.
- Unikernels are a promising optimization to hypervisor-based virtualization that reduces virtualization overhead and bringing performance close to containers.

We evaluate raw networking performance using the Netperf benchmarking tool [43] and network intensive application performance using the Memcached key-value store [7] and the Mutilate [15] benchmarking tool. The evaluation is run on Intel-based commodity hardware with a 1 Gigabit NIC using Linux as the host operating system. For hypervisor-based virtualization, we use KVM/QEMU [4] as the hypervisor using both virtio [58] and vhost [40] for networking and run Linux and OSv [46] as the guest operating system. For container-based virtualization, we use the Docker tool to provision Linux namespaces and cgroups-based containers.

1.4 Roadmap

The rest of this thesis is organized as follows. In Section 2, we detail virtualization techniques and their various optimizations, including I/O virtualization and unikernel design. In Section 3, we detail problems in virtualization techniques that impose performance overhead on CPU, memory, storage, and networking. In Section 4, we detail the evaluation methodology used to evaluate the effectiveness of various virtualization techniques for performance. In Section 5, we evaluate the performance of Netperf for bare metal, Docker, and Linux and OSv running on QEMU/KVM using both vhost and virtio and with TCP offload enabled and disabled. In Section 6, we evaluate the performance of Memcached also for bare metal, Docker, and Linux and OSv running on QEMU/KVM using just vhost with offload enabled. In Section 7, we present related work, and finally, we conclude in Section 8.

2 Background on Virtualization Techniques

In this section, we present background on virtualization techniques. We detail hypervisor-based virtualization that lets you create multiple isolated virtual machines on a single physical machine and I/O virtualization techniques for hypervisors. We also detail unikernels that are an operating system design approach that attempt to mitigate performance inefficiencies caused by hypervisors by eliminating the traditional kernelspace and userspace separation from the guest OS. Finally, we detail container-based virtualization lets you run multiple operating system userspaces on the same physical machine sharing a single kernel.

2.1 Hypervisor-Based Virtualization

In this section, we present an overview on hypervisor-based virtualization techniques: full virtualization, paravirtualization, and hardware-assisted virtualization. We also present a brief overview on nested virtualization for completeness.

A hypervisor[1] is a system software component that lets you run guest operating systems on a host machine in efficient and isolated virtual machines [52]. A hypervisor must exhibit the following three properties: the equivalence property, the efficiency property, and the the resource control property.

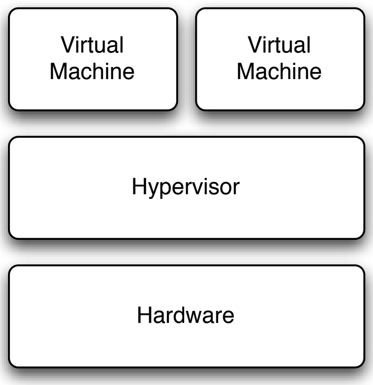
Equivalence property states that program execution has identical observable behavior on real hardware and under virtualization, except for differences caused by timing or resource availability. The exception is needed because virtual machines typically share the same physical resources. For example, multiple virtual CPUs that share the same physical CPU make it impossible to have the exact same timing on bare metal and under virtualization because thread execution is interleaved on the physical CPU.

Efficiency property states that the majority of virtual CPU machine instructions are executed directly by a physical CPU without interference from the hypervisor. This property separates hypervisors from emulators and interpreters that execute no virtual CPU instructions directly on physical CPU.

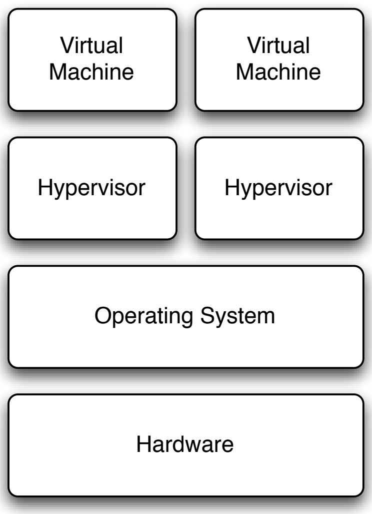
Resource control property states that the hypervisor manages all hardware resources. It must be impossible for an arbitrary program in a virtual machine to directly access hardware without permission from the hypervisor.

The architecture of a hypervisor be broadly categorized into two classes as illustrated in Figure 1: Type 1 and Type 2 hypervisors [38]. The main

[1]Hypervisor is also known as a virtual machine monitor (VMM).



(a) **Type-1** hypervisor runs directly on top of the hardware and manages hardware resources by itself. The direct hardware access gives Type-1 hypervisors a performance edge over Type-2 hypervisors.



(b) **Type-2** hypervisor runs on top of a host operating system that manages hardware resources. The decoupling of host operating system and hypervisor give Type-2 more flexibility and allows running and managing multiple versions of the hypervisor, which simplifies deployment.

Figure 1: **Hypervisor architecture classification.**

difference between Type-1 and Type-2 hypervisors is that a Type-1 hypervisor runs directly on the hardware and manages hardware sources by itself. Type-2 hypervisor runs on top of a host operating system and lets the OS manage hardware resources. The direct hardware access gives Type-1 hypervisors a performance edge over Type-2 hypervisors. However, the decoupling of host operating system and hypervisor give Type-2 more flexibility and allows running and managing multiple versions of the hypervisor, which simplifies deployment. Popular examples of Type-1 hypervisors are Xen [24] and VMware ESXi [1], while QEMU [26] and VirtualBox [10] are popular examples of Type-2 hypervisors.

Hypervisors can be implemented using different techniques that are summarized in Table 1. Full virtualization requires neither guest OS modifications or hardware support. However, hypervisors implemented using full virtualization are complex and require advanced techniques like binary translation for high performance [20]. Paravirtualization provides high performance by requiring the guest OS to be modified to to run on the hypervisor. Hardware-assisted virtualization is essentially full virtualization that utilize

Virtualization Technique	**Guest OS Modifications**	**Hardware Support**	**High Performance**
Full virtualization	–	–	–
Paravirtualization	✓	–	✓
Hardware-assisted	–	✓	✓
Nested virtualization	–	–	–

Table 1: **Summary of hypervisor-based virtualization techniques.** *Full virtualization requires neither guest OS modifications or hardware support. Paravirtualization requires the guest OS to be modified to run on the hypervisor. Hardware-assisted virtualization is similar to full virtualization but requires hardware support. Nested virtualization does not require either guest OS modifications or hardware support. A checkmark (✓) indicates that feature applies to the virtualization technique. A dash (–) indicates that the feature does not apply to the virtualization technique.*

hardware virtualization capabilities. Nested virtualization does not require guest modifications or hardware support.[2]

2.1.1 Full Virtualization

Full virtualization is a virtualization technique that requires the guest operating system to trap to the hypervisor for instructions that access CPU privileged state or issue I/O so that the hypervisor can emulate those instructions [20]. This trap-and-emulate approach is sometimes called *classical virtualization* because it has been such a prevalent implementation technique since it was first proposed [52]. The x86 architecture, however, has not been classically virtualizable until the introduction of virtualization hardware extensions [20] so full virtualization approaches have relied on binary translation to detect and trap non-virtualizable instructions. Hypervisors that implement full virtualization are able to run unmodified guest operating systems, which makes them a very practical virtualization technique.

In full virtualization, the guest operating system runs in unprivileged mode and each access to privileged state is trapped and emulated by the hypervisor [20]. As the physical hardware privileged state and the virtual machine privileged state differ from each other, the hypervisor maintains shadow structures that are based on guest-level primary structures. CPU data structures such as page table pointer or processor status word are maintained as a shadow copy of the guest registers that are accessed by the hypervisor when instructions are emulated. To maintain coherency of data structures like page tables that are stored in memory, the hypervisor uses MMU protection capabilities to trace memory accesses and update shadow

[2]Hardware support is required for high performance nested virtualization.

structures accordingly.

The trap-and-emulate approach presents a problem for traditional x86 architecture because of two main issues: privileged state visibility and lack of traps for privileged instructions that run in userspace [20]. The first problem is that the guest OS sees that it is running in state by reading its code segment selector (CS) register that has the current privilege level (CPL) encoded in the low two bits. This means that the hypervisor is unable to virtualize privileged state for the guest OS. The second problem is that instructions such as `popf` (pop flags) that modify both ALU flags and processor flags do not cause a trap in unprivileged mode and simply ignore the changes to the processor flags. The guest OS runs in unprivileged state and is therefore unable to catch access to processor flags such as the interrupt flag (IF) that controls interrupt delivery for such instructions to update the shadow processor flags accordingly.

To solve these issues on traditional x86 architecture, hypervisors use dynamic binary translation to replace privileged instructions with explicit hypervisor traps for instructions that are running in a VM [20]. Non-privileged instructions require no translation and can be translated to the original instruction. The binary translation approach to full virtualization outperformed hardware-assisted virtualization for the first generation hardware [20] but is no longer able to do so with newer hardware generations [21].

2.1.2 Paravirtualization

Paravirtualization is a virtualization technique that requires the guest OS to be modified to call hypervisor services instead of executing privileged hardware instructions [24]. The technique is particularly effective for architectures that are not classically virtualizable like traditional x86 for high performance virtualization. Paravirtualization does not require changes in applications that run fully unmodified in the virtual machines under the modified guest OS. The paravirtualization technique was popularized by the Xen hypervisor [24] that was first developed for the traditional x86 architecture to avoid binary translation and to efficiently virtualize the x86 MMU.

Xen is a hypervisor that supports both paravirtualization [24] and hardware-assisted virtualization [54]. Xen runs directly above the hardware and provides a minimal set of hypervisor services to virtual machines called domains as illustrated in Figure 2. Domain 0 is the initial domain that runs hypervisor management software that provides control interfaces for processors, memory, virtual network devices, and virtual block devices. Other domains are known as user domains and run operating systems that have been ported to Xen in unprivileged mode. The Xen hypervisor and domains have two mechanisms for communicating with each other: hypercalls and events.

Hypercalls allows domains to trap into the hypervisor synchronously to allow the hypervisor perform privileged operation for them, which resembles the mechanism of OS system calls. Events are a mechanism that allows the Xen hypervisor to asynchronously notify the domains in light-weight manner, replacing device interrupts. Data is transferred between domains and the Xen hypervisor using I/O rings that only contain descriptors to the data. The use of I/O rings allows efficient data transfer because data is not copied between the hypervisor and the domains.

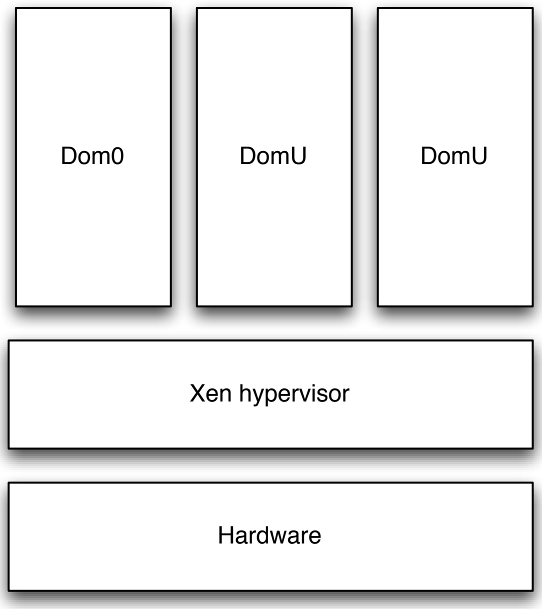


Figure 2: **The Xen hypervisor architecture.** *The Xen hypervisor runs directly on top of hardware and manages multiple domains. Domain 0 is a special domain that runs Xen control software. Other domains are known as user domains and run operating systems that have been ported to Xen in unprivileged mode.*

The x86 paravirtualization interface in Xen has interfaces for memory management, CPU, and device I/O [24]. In Xen, the guest OS is responsible for page table management. The Xen hypervisor updates hardware page tables but only on behalf of the guest OS when it calls the paravirtualization interfaces. Xen validates the page table updates to make sure that domains are isolated from each other. CPU privilege level virtualization on x86 is achieved by changing the guest OS to run on ring 1 instead of the usual ring 0 that prevents the guest OS from executing privileged instructions directly. CPU exception virtualization follows the x86 hardware model closely. The only interrupt related modification in the guest OS is in the page fault handler that needs to be modified to query for the faulting page address using Xen paravirtualization interface instead of reading the hardware `cr2` register. Device I/O virtualization is based on Xen specific device model with its own event notification mechanism that replaces hardware interrupts.

The Xen hypervisor was initially developed for the traditional x86 architecture but has since been ported to the x86-64, IA-64, and the PowerPC machine architectures [54].

2.1.3 Hardware-Assisted Virtualization

Hardware-assisted virtualization is essentially the same as full virtualization detailed in Section 2.1.1 when trap-and-emulate virtualization technique is supported directly by the hardware. This virtualization technique became possible on the x86 architecture with the introduction of CPU virtualization extensions [20].

The virtualization extensions on Intel CPUs introduce a new in-memory data structure called the *virtual machine control block* (VMCB) that maintains a subset state of a guest virtual CPU [20]. The extensions also introduce a new execution mode, guest mode, that supports execution of both unprivileged and privileged instructions. A new `vmrun` instruction transfers control from host mode to guest mode by loading guest state from the VMCB and resuming execution in the guest. Execution continues in the guest until some condition expressed as a control bit in the VMCB indicates a VM exit. The hardware then saves the guest state in VMCB, loads state provided by the hypervisor, and resumes execution in host mode. The VMCB also provides diagnostic fields to assist the hypervisor for indicating why the VM exit happened. For example, if the guest issued an I/O instruction, the port, width, and the direction of the I/O operation is available from VMCB. The hardware virtualization extensions did not support explicit MMU virtualization in the first generation hardware [20] but support for that has since been added.

The performance of hardware virtualization depends on the frequency of VM exits that the guest operating system issues [21]. A guest application that does not issue any I/O runs effectively at native speed. However, as I/O requires some amount of VM exists, the most important optimization for hardware-assisted virtualization is reducing them. Strategies for reducing VM exits for I/O are discussed for I/O virtualization techniques in Section 2.2. One further optimization to reduce VM exits is using binary translation technique to dynamically detect back-to-back instructions that both cause a VM exit and fuse the instructions [21].

Kernel-based Virtual Machine (KVM) is an operating system module that provides an abstraction of hardware virtualization extensions and exposes an ABI to userspace that can be used as a building block for implementing a

Type-2 hypervisor [45]. It abstracts the virtualization extension differences between machine architectures and CPUs. KVM is typically used in tandem with QEMU, which provides I/O device emulation for a variety of hardware, including paravirtualized virtio-based devices that are optimized for virtual machines [58]. KVM was originally developed as a Linux subsystem for Intel and AMD x86 virtualization extensions but has since then been ported to other machine architectures and operating systems [31]. In KVM, new virtual machines are created via a `/dev/kvm` device file. Each guest has its own memory that is separate from the userspace process that created the VM but vCPUs require a userspace thread for execution.

The Xen hypervisor also supports hardware-assisted virtualization with the introduction of the HVM mode [54]. The HVM mode allows users to create fully virtualized domains that support running an unmodified guest OS. In HVM mode, the hypervisor emulates x86 architecture services and I/O devices using QEMU [34].

2.1.4 Nested Virtualization

Nested virtualization is an extension to hypervisor-based virtualization that allows hypervisors to run other hypervisors and their virtual machines inside them [27]. Nested virtualization has become important as commodity operating system such as Linux and Windows 7 that are being used as guest OS'es have also gained hypervisor functionality. As there is no hardware support in x86 for nested virtualization, various optimizations have been proposed to implement nested virtualization efficiently. In the nested virtualization implementation of the Turtles project [27], the lowest hypervisor inspects VM exits and forwards them to hypervisors above it for emulation. For memory virtualization, a multi-dimensional paging approach is used that collapses the various page tables into one or two MMU-based page tables. For I/O virtualization, multilevel device assignment is used to bypass multiple levels of hypervisors.

2.2 I/O Virtualization

In this section, we give an overview on I/O virtualization techniques. We detail full device emulation, paravirtualization, and direct I/O (device assignment).

2.2.1 Full Device Emulation

In full device emulation, the hypervisor emulates the I/O registers and memory of a real hardware device I/O in full or in part [33]. The guest OS device driver I/O accesses are trapped to the hypervisor and emulated by it. Full device emulation lets a single physical I/O device be shared across multiple virtual machines without having to modify the guest OS device drivers. In full device emulation, every I/O access translates to a costly VM exit and the hardware emulation layer does not always implement all the functionality of physical devices, which can be a significant performance overhead. However, some I/O devices like modern NICs have similar capabilities as paravirtualized devices, which makes it is possible to implement device emulation to reduce VM exits and approach near native performance [57].

2.2.2 Paravirtualization

In paravirtualization, the hypervisor emulates an I/O device that is optimized for virtualization [57]. The guest OS requires custom device drivers to interact with the virtual I/O device, which makes it impossible to run unmodified guest operating systems. However, paravirtualization reduces performance overhead by reducing VM exits that are caused by I/O accesses by using shared memory between the hypervisor and the guest OS for communication. The performance of paravirtualization I/O is still worse than the performance of state-of-the-art I/O virtualization techniques such as SR-IOV [41].

Virtio is the native KVM/QEMU hypervisor paravirtualized I/O model [58]. It was originally developed for Linux to unify its various virtual I/O device models. The virtio model provides support for device probing, configuration, and I/O transport via the virtqueue API. A virtqueue is a queue that is shared by both the hypervisor and the guest OS. The guest OS posts buffers (scatter-gather arrays that have a readable and a writable part) that are consumed by the hypervisor. The hypervisor is notified about new buffers in the queue via a kick operation that typically requires an expensive VM exit. However, the guest OS is free to add new buffers in a batch and issue a kick operation for all of them to amortize VM exit cost. The hypervisor notifies the guest via a per-queue callback when buffers have been consumed but the guest OS can disable this callback notification if necessary (similar to OS disabling device hardware interrupts). Device probing and configuration are implemented using PCI [3] that is emulated by the hypervisor.

The main virtio transport implementation in Linux is called virtio_ring that consists of three parts: a descriptor array, an available ring, and an used ring [58]. The *descriptor array* contains buffer descriptors that contain a guest-physical address, length of the buffer, an optional next descriptor pointer, and flags that specify whether the buffer is for reading or writing, and whether the next descriptor is set. The *available ring contains* information of what descriptors are available. Every element in the available ring has a free-running index, a flag for suppressing interrupts, and an array of indices to the descriptor array to advertise unused entries. The available ring is separated from the descriptor array to make sure slow I/O operations that are waiting for completion do not block new operations when the ring is circled. The *used ring* is similar to the available ring but it is managed by the host when descriptors are consumed. Virtio device drivers like block device and network device are layered on top of virtio_ring by including a device specific header in the generic virtio_ring buffers.

Virtio-net is a paravirtualized network device that is implemented using the virtio device model. The architecture of virtio-net is illustrated in Figure 3a. In virtio-net, packets that arrive on the hardware NIC travel to the guest OS virtio queue through the host OS networking stack and QEMU device emulation layer. Packets that are transmitted from the guest OS are handled over to the QEMU device emulation layer via a virtqueue. QEMU then forwards the packets to host OS network stack from where they end up on the hardware NIC.

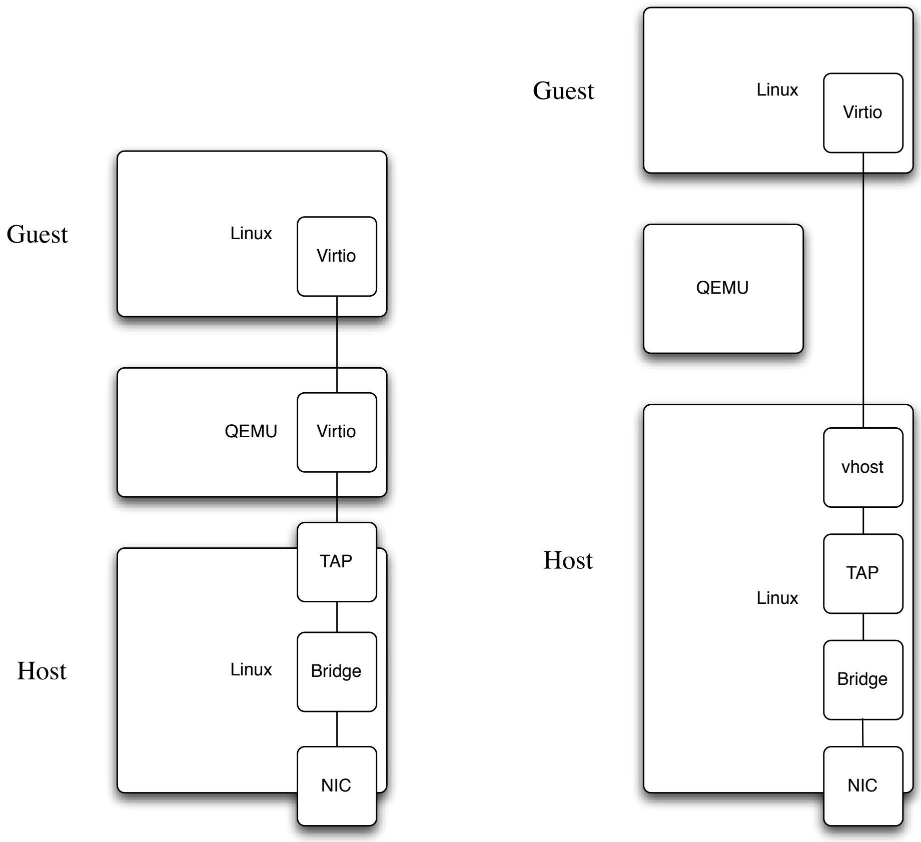
Vhost-net is an optimization that moves the virtio device emulation to the host kernel by-passing QEMU userspace process for the data plane. Vhost emulation is restricted to virtqueue operations and thus relies on QEMU userspace process to handle control plane operations such as virtio feature negotiation and live migration [40]. The vhost-net architecture is illustrated in Figure 3b.

2.2.3 Direct I/O

Single root I/O virtualization (SR-IOV) standard allows an I/O device to be efficiently shared by multiple VMs. SR-IOV is a PCIe specification extension that allows a PCIe device to separate physical functions from virtual functions. Physical functions are full-featured PCIe device functions whereas virtual functions only support data plane operations. Virtual functions can be assigned to specific VM to offer direct I/O access in a virtualized environment. SR-IOV is able to reach network line rate and scale up to tens of VMs with few percent additional per-VM CPU overhead [47] [35].

2.3 Unikernels

Unikernels are a new approach to operating system design that eliminates the traditional kernel and userspace split to improve the performance and manageability of applications that are deployed to cloud platforms [48]. Applications are linked against a unikernel core and the combination is packaged as a single application VM image. The unikernel concept builds upon library OS concept that structures OS services as libraries that applications link against [53] [36]. Previous library OS proposals have been impractical because they require a lot of device drivers to be implemented to support all the hardware devices that are in use. Hypervisors like KVM and Xen have a much limited device model, which makes unikernels a much more practical approach [48].



(a) **Virtio-net** (b) **Vhost-net**

Figure 3: **Virtio and vhost network I/O virtualization architectures.** *The vhost architecture moves virtio implementation to host kernel allowing I/O to by-pass the device emulation layer in QEMU.*

In this section, we will detail three unikernel designs: Arrakis, an unikernel for SR-IOV capable systems, ClickOS, a minimal unikernel for NFV middleboxes, and OSv, a Linux compatible unikernel.

2.3.1 Arrakis

Arrakis is an unikernel that allows applications to by-pass the kernel for most I/O operations while providing process isolation by utilizing I/O hardware virtualization such as SR-IOV [51]. For packet processing, Linux has four sources of overhead: network stack costs, scheduler overhead, kernel crossings, and copying of packet data. In Linux, 70% of packet processing overhead is spent in the network stack demultiplexing, security checks, and overhead caused by indirection between various layers. Scheduler overhead is 5% if the receiving process is already running but if it's not, the the extra context switch adds 2.2 µs overhead and a 0.6 µs slowdown in the network stack. On SMP, cache and lock contention issues can cause further slowdowns, especially as packets can arrive in different CPU core where the receiving server thread is currently running on. In Arrakis, packets are delivered directly to userspace that eliminates scheduling and kernel crossing overhead completely. Cache and lock contention overhead is also reduced because each application has its own network stack and thus each packet is delivered to the CPU core where application is running. By applying all these optimizations, Arrakis is able to achieve 2 times better read latency, 5 times better write latency, and 9 times better write throughput than Linux for Redis, a popular NoSQL store – a significant performance improvement.

Arrakis takes full advantage of SR-IOV to implement safe application-level access to I/O devices as illustrated in Figure 4. SR-IOV is a PCIe standard extension that allows multiple virtual machines to share the same physical device by letting the operating system dynamically create virtual PCI functions that are assigned to individual virtual machines. Access to these virtual functions can be protected using the hardware IOMMU. The device drivers in guest operating systems are then able to treat these virtual NICs as if they were physical NICs exclusively assigned to them. The host kernel still implements control plane functionality to configure the virtual NICs but does not participate in data plane operations.

Arrakis supports two sets of APIs: POSIX compatible socket API and Arrakis' own zero-copy API that eliminates the overhead from copying packet data from and to userspace that's required by POSIX. The POSIX APIs are designed to improve performance without requiring existing applications to be modified. However, further performance improvements are available if an application is ported to use Arrakis' native APIs that supports zero-copy I/O. The implementation of Arrakis is based on a fork of the Barrelfish, a SMP capable OS, which is extended to support SR-IOV, many POSIX socket APIs, and Linux `epoll()` interface that's used to poll large number of file

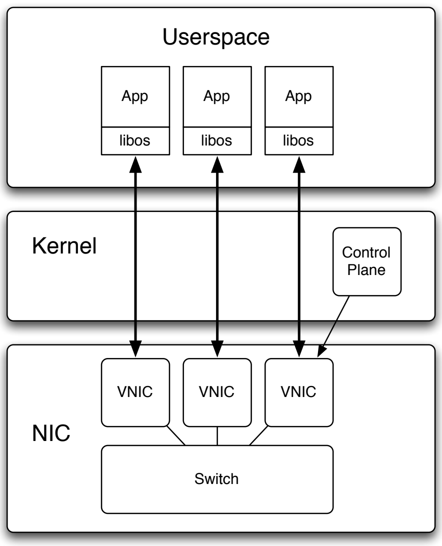


Figure 4: **Arrakis architecture.** *Hardware NIC provides virtual NICs via SR-IOV that are mapped directly to application address space.*

descriptors efficiently. Arrakis also implements its own user-level network stack, which leverages the packet processing core from the lwIP network stack. Barrelfish already provides a library OS that makes it easy to link it directly to Arrakis-based applications.

2.3.2 ClickOS

ClickOS is an unikernel optimized for middleboxes that runs exclusively on the Xen hypervisor with small virtual machine memory footprint overhead (5 MB), fast boot times (under 30 milliseconds), and high performance networking capabilities [49]. ClickOS-based middleboxes are capable of processing millions of packets per second and are able to saturate a 10 Gigabit hardware NIC while running 100 virtual machines on a single CPU core. The packet processing in ClickOS is designed for low latency and measurements show that ClickOS adds only a 45 microsecond delay per packet. When compared to a general purpose Linux also running on Xen, ClickOS network throughput is up to 1.5x times higher for MTU-sized packets and as much as 13.6x times higher for minimum-sized packets. ClickOS can run either as Linux kernel module at full performance or as a regular userspace process with less performance for debuggability.

ClickOS uses MiniOS, a tiny operating system that comes with Xen, as its basis that provides minimal set of operating system service: memory manager and a scheduler. Memory management is only needed for dynamic allocation of operating system data structures such as packet descriptors and there is no support for more complex memory management services like memory mapping or protection provided by the MMU. The scheduler only needs to switch between running middlebox application code and servicing interrupts and thus does not require complex features like preemptive scheduling or multithreading. MiniOS also has a Xen netfront driver that provides access to paravirtualized Xen network devices but it's performance is poor, especially for receive where MiniOS is barely able process 8K packets/second.

For high performance networking, ClickOS overhauls the Xen network I/O subsystem extensively by eliminating layers that are not essential to moving packets in and out of a VM. At the lowest level, ClickOS replaces the standard Open vSwitch back-end that is used by Xen with a high-speed switch, VALE, which allows mapping per-port ring buffers directly to the VM memory address space. ClickOS also improves upon VALE by adding support for connecting physical NICs directly to the switch, increasing maximum number of ports from 64 to 256, and adds support for per-VM configuration of packet ring buffer size up to 2048 slots. One layer above, ClickOS redesigns the Xen netback driver to function as primarily a control-plane driver that is only responsible for allocating memory for receive and transmit ring buffers, setting up memory grants so that the ring buffers can be mapped directly into VM memory address space, and setting up kernel threads for transferring packages between the switch and the netfront driver. Finally, the netfront driver is modified to map the switch ring buffers, support asynchronous transmit, enable re-use of memory grants, and MiniOS is extended to support the Netmap API [56] used by VALE switch. As the Xen network I/O modifications break non-MiniOS guests, a Linux driver has also been implemented that runs on the modified Xen hypervisor.

2.3.3 OSv

OSv is an unikernel that runs existing Linux cloud applications on various hypervisors and machine architectures [46]. Current OSv runs on 64-bit x86 and ARM architectures and supports the KVM/QEMU [4], Xen, VMware [1], and VirtualBox [10] hypervisors, and on Amazon EC2 [8] and Google Compute Engine (GCE) [14] platforms. OSv is able to improve network throughput up to 25% and decrease latency by 47% for unmodified applications and improve throughput up to 290% for memcached by using special purpose OSv APIs. OSv is heavily optimized to run Java Virtual Machine (JVM) [11] based applications fast on virtual machines.

OSv is designed as a drop-in replacement for applications that use a supported subset of the Linux application binary interface (ABI). The core of OSv implements an ELF loader and a dynamic linker, memory manager, thread scheduler, virtual filesystem (VFS) synchronization mechanisms (e.g. pthread mutexes and RCU), and device drivers for various hardware and virtual devices. The ELF dynamic linker makes it possible to run existing

Linux binaries. Linux system calls are translated into function calls to APIs that are implemented by OSv. The main limitation of the Linux ABI in OSv is lack of support for the `fork()` and `exec()` system calls, which are hard to implement with a single address space. OSv has adopted ZFS [5] as its primary filesystem but has since gained support also for the Network File System (NFS) [59] that is popular for HPC workloads [30]. The memory manager in OSv supports demand paging, the `mmap()` interface for memory mapping, and transparent huge pages for large memory mappings. OSv also implements a set of special purpose APIs, like memory shrinker API and a JVM balloon, that can be used by applications to further improve performance.

The network stack in OSv is optimized for network intensive applications. The stack is based on FreeBSD but is redesigned to follow a network channel design that reduces locking overhead. The network channel design is based on an observation that packets traverse the networking stack from different directions, top-down and bottom-up, causing lock contention on shared locks. The `send()` and `recv()` system calls traverse the network stack top-down from socket layer down to TCP/IP and packet layers at the NIC level. The NIC device drivers traverse the network stack in the reverse direction when packets are received starting from the NIC level and going up through the TCP/IP layer to the socket layer. In OSv, applications are able to perform almost all packet processing in their own threads because the receive path is handled separately using a network channel that has its own locking. When a packet arrives, a classifier is used to direct the network channel to a specific socket.

2.4 Container-Based Virtualization

Containers are a virtualization approach that does not involve a hypervisor. Instead, a machine runs only one kernel that is shared by all the guests. Virtualization is achieved with host operating system features such as namespaces and control groups [60].

2.4.1 Docker

Docker has emerged as the de facto standard runtime, image format, and build system for container-based virtualization on Linux [37]. Docker consists of two major components: Docker, the container platform, and Docker Hub, a software as a service (SaaS) that allows people to share and manage Docker containers [19]. Docker follows the client-server architecture where the Docker client (i.e. the `docker` command line tool) talks to a Docker daemon that is responsible for building, running, and distributing the containers. Both the client and the daemon can run on the same machine or the Docker client can connect to a remote node running the Docker daemon. Docker is further divided into three components internally: images, registries, and containers. Images are read-only templates for containers that contain detailed instructions how to build a container. For example, an image might contain instructions how to install an Ubuntu Linux operating system with the Apache HTTP server with a web application installed to it. Registries are managed by Docker Hub and are used for storing images so that they can be distributed among users. Containers are similar to directories and hold all the files needed to run an application and they can be started, stopped, moved, and deleted.

The container-based virtualization technique Docker uses on Linux is built on *kernel namespaces* and *control groups (cgroups)* [37]. Namespaces is a kernel feature that allows the creation of isolated namespaces for filesystem, PID, network, user, IPC, and host name via the `clone()` system call [28]. Namespaces are similar to `chroot()` but they are more powerful and allow full isolation. Cgroups are a feature that groups a set of processes and allows the kernel to manage and enforce resource limits on the group. They are typically used to limit the amount of CPUs and memory a container has access to.

2.4.2 Kubernetes

Kubernetes is an open source platform for deploying, scaling, and operating a cluster of containers, originally developed by Google [18]. It uses Docker under the hood to manage individual containers and provides support for managing a cluster of nodes. A node in a Kubernetes cluster is either a physical or a virtual machine that runs a `kubelet` node agent. The node agent manages *pods*, which are a set of closely related containers that are managed as a single unit that share the same storage volumes and IP address space. The control plane in Kubernetes consists of `etcd`, API server, scheduler, and a controller manager. Etcd [17] is a reliable distributed key-value store that Kubernetes uses to store all persistent configuration. The API server is the the main entry point to manage a cluster and allows users to configure and manage Kubernetes work units. The scheduler assigns workloads to nodes in the cluster, tracks resource utilization, and load balances the nodes when needed. The controller manager is an agent that controls the state of the system by watching cluster shared state and updating the cluster if needed.

2.5 Summary

Hypervisor-based virtualization and container-based virtualization provide isolation and resource control and let physical resources to be efficiently shared across multiple virtualized environments. Hypervisors can be implemented using different techniques: full virtualization, paravirtualization and hardware-assisted virtualization. Unikernels are an optimization for hypervisor-based virtualization, which reduces the overhead of the guest OS that runs in a VM. Containers provide isolation at the host operating system level and do not involve a hypervisor or a separate guest OS. Instead, all containers on a host machine run on the same host kernel in an isolated namespace with physical resources shared by a resource controller. The virtualization techniques have performance overheads, which are discussed in detail in Section 3.

3 Virtualization Overheads

In this section, we detail problems in virtualization techniques that impose performance overheads on CPU, memory, networking, and disk. The different overheads are summarized in Table 2 by system component and virtualization technique.

Containers are a relatively new virtualization technique compared to hypervisors. Although they have received widespread adoption in industry, they have not received significant research attention until recently. Therefore most of the overheads discussed in this section are focused on hypervisor-based virtualization.

System Component	Overhead Source	Virtualization Technique		
		HV	Unikernel	Container
CPU	Double Scheduling	✓	–	–
	Scheduling Fairness	✓	✓	–
	Asymmetric CPUs	✓	✓	–
	Interrupts	✓	✓	–
Memory	Memory Reclamation	✓	✓	–
	Memory Duplication	✓	✓	✓
Networking	Packet Processing	✓	✓	–
	NAT	–	–	–
	Unstable Network	–	–	✓
Disk	I/O Scheduling	✓	✓	–
	Layered filesystems	–	–	✓

Table 2: **Summary of virtualization overheads by virtualization technique.** *Overhead sources are grouped by system component (CPU, memory, networking, disk) and virtualization technique (hypervisor (HV), unikernel, container). A checkmark (✓) indicates that the overhead is applicable to the virtualization technique. A dash (–) indicates that the overhead is not applicable to the virtualization technique. For example, containers have performance overhead from memory duplication, NAT, and layered filesystems.*

3.1 CPU

As summarized in Table 2, virtualization causes performance overheads on CPU due to double scheduling, scheduler fairness, asymmetric CPUs, and interrupts. We now detail each of the causes for CPU overheads.

3.1.1 Double Scheduling

In hypervisor-based virtualization, there are two levels of scheduling: the guest operating system schedules process to virtual CPUs (vCPUs) that are implemented in the hypervisor as threads and the hypervisor schedules the vCPU threads to physical CPUs. Current hypervisor-level schedulers in both KVM and Xen are unaware of the second level of scheduling in the guest OS which can cause significant performance degradation for parallel applications running in a VM [61]. Two main issues with current hypervisor schedulers are vCPU preemption and vCPU stacking.

A vCPU preemption occurs when the hypervisor preempts a vCPU that is holding to a lock and then switches to another vCPU. The preemption extends vCPU synchronization time of significantly because the vCPU that is waiting for the lock is waiting for the lock to be released but the vCPU that is holding the lock is waiting to run. The problem is also known as *lock holder preemption* and is especially bad for spin-locks which are often used to protect data structures from concurrent access on multi-processor systems when a lock is needed for a short period of time. This type of locking works well on hardware but suffers from lock holder preemption problem on virtual machines: if a virtual machine is interrupted in the middle of a critical section, it can hold on to a spin-lock for a very long time which is a significant performance problem [63].

vCPU stacking happens because the hypervisor scheduler is allowed to schedule a vCPU on any physical CPU which can cause a lock waiter vCPU N to to be scheduled before a lock holder vCPU M on the same physical CPU which increases synchronization time on both vCPUs.

3.1.2 Scheduling Fairness

The double scheduling at hypervisor and guest OS level makes it difficult to achieve scheduling fairness for symmetric multiprocessing (SMP) virtual machines which have two or more vCPUs [55]. For example, one virtual machine may have all of its vCPU run on dedicated physical CPUs but another virtual machine may have all of its vCPUs multiplexed on the same physical CPU. Hypervisors attempt to enforce scheduling fairness by assigning equal number of physical CPU shares (amount of physical CPU time received by a vCPU) to each virtual machine. Virtual machines then distribute their physical shares among their own vCPUs. VMs with large number of vCPUs thus have a smaller portion of the physical CPUs assigned per vCPU than VMs with smaller amount of vCPUs. Hypervisors also attempt to prevent virtual machines from monopolizing the system by limiting the use of physical CPU even if the system is otherwise idle. In SMP configurations, CPUs are load balanced to improve throughput and responsiveness by distributing vCPUs evenly onto physical CPUs by using push and pull migration strategies. In push migration, the load balancer periodically pushes vCPUs away from busy physical CPUs. In pull migration, vCPUs that a ready to run are migrated from busy physical CPUs when a physical CPU is about to become idle.

Flex [55] is one proposed scheduling scheme that enforces fairness between VMs by dynamically adjusting vCPU weights and minimizing vCPU busy-waiting time that is able to reach CPU allocation that is no more than 5% in error of ideal fair allocation and outperforms standard Xen credit scheduler by 10x for parallel applications.

3.1.3 Asymmetric CPUs

The OS scheduling algorithm attempts to distribute work across CPUs to optimize overall system performance by exploiting information about CPU speed, cache sizes, and cache line migration costs to predict future performance based on past workload performance. However, in a virtualized environment where vCPUs is multiplexed on a physical CPU, the assumption that every CPU is identical to each other no longer holds. This leads the guest OS scheduler to distribute work across vCPUs in sub-optimal way which can lead to overcommitment or under utilization of CPUs. For the guest OS to make correct scheduling decisions, it must be able to distribute work equally based on physical CPU allocation [63].

Similar to the memory ballooning technique detailed in Section 3.2.1, *time ballooning* lets the hypervisor participate in load balancing decisions without changing the guest OS scheduling algorithm. A time balloon module is loaded into the guest OS as a pseudo device that periodically polls the hypervisor to determine how much physical CPU time the VM has received from the hypervisor and generates virtual workload so that the guest OS thinks it's busy executing a process when it has no physical CPU time which solves the imbalance caused by physical CPU allocation [63].

3.1.4 Interrupts

There are three main differences between interrupt handling on physical and virtual machines:

1. Interrupts received by the local APIC can be handled almost immediately in an interrupt handler on a CPU in a physical machine. In a virtual machine, the interrupt handler won't be executed until the hypervisor schedules the vCPU to run, which can introduce significant delays.

2. Most IO-APIC chips can route an interrupt to multiple CPUs in physical machines which makes interrupt delivery more flexible. Hypervisors

like Xen are, unfortunately, limited to delivering an interrupt to a specific CPU.

3. The OS idle process consumes all of the available CPU cycles in physical machines. In virtual machines, the guest OS can yield its CPU to the hypervisor which can cause delays in interrupt handling.

The above three differences can cause I/O virtualization techniques to have poor performance under I/O intensive workloads [32]. There have been recent hardware improvements to improve interrupt performance for hypervisors. For example, APICv is a new feature in the latest generation of Intel CPUs (the Broadwell microarchitecture) that supports *posted interrupt* mechanism that lets the hypervisor to inject virtual interrupts directly to the guest without involving a VM exit which reduces interrupt delivery overhead [62].

3.2 Memory

As summarized in Table 2, virtualization causes performance overheads on memory due to memory reclamation and memory duplication. We now detail each of the causes for memory overheads.

3.2.1 Memory Reclamation

When physical memory is overcommitted, the hypervisor needs reclaim memory from the virtual machines it manages and evict pages to physical storage. However, the hypervisor does not have a lot of information about which guest pages are good candidates for eviction because those pages are managed by the guest operating systems [64]. Furthermore, as the guest operating system also has a memory reclamation policy, a *double paging problem* is possible where under memory pressure, the guest operating system first evicts a page to its virtual backing storage only to have the hypervisor also evict the same page to a physical storage. One solution to the problem is to have the hypervisor communicate with the guest operating systems via a balloon process [64]. The balloon is a special device driver that runs in the guest operating system that allocates guest pages and pins them to memory by making them unusable to the guest operating system. These pages can be reclaimed by the hypervisor by inflating the balloon which increases memory pressure and forces the guest operating system to page out to its virtual storage. The hypervisor can decrease memory pressure by deflating the balloon which allows the guest to page in previously used pages. To track idle memory in in a guest VM, the hypervisor uses statistical sampling by invalidating the TLB entry for randomly selected pages. Access to invalidated pages increase their usage count which allows the hypervisor to estimate how many pages are currently in use and how many can be considered idle.

TLB invalidation comes with a significant performance penalty in a virtualized environment because it forces a context switch and requires multiple memory accesses to re-fill the TLB entries. To solve the TLB invalidation issue, Memory Pressure Aware (MPA) ballooning is proposed [44]. MPA ballooning compares the sum of anonymous pages across all VMs and the sum of file pages also across all VMs to the total available system memory and classifies system memory pressure to three regimes: low pressure, mild pressure, and heavy pressure. The low pressure regime occurs when the total memory demand is less than the total amount of available memory. No file pages need to be evicted because there is enough available memory to satisfy the memory demand. The mild pressure regime occurs when total memory demand exceeds the total amount of available memory. The hypervisor must have a paging policy because there is not enough memory for all the anonymous and file pages used by all the running VMs. Finally, the heavy pressure regime occurs when anonymous pages exceed the total amount of available memory which means that there's not enough memory to satisfy all anonymous pages used by running VMs which means guest OSes must swap pages to disk. The MPA balloon applies different memory management policy depending on memory pressure regime by implementing an adaptive memory cushion and an instant response mechanism that reacts to memory pressure changes via memory reclamation and reallocation. MPA ballooning improves performance by 18.2% when multiple VMs are running the same application and 13.2% when multiple VMs are running different applications in random order.

3.2.2 Memory Duplication

A general purpose operating system consists of a kernel image, shared libraries, and various OS services that have an overhead on memory footprint. As virtual machines are isolated from each other, every VM has a separate memory copy of the OS, even if two VMs are running the exact same version.

The hypervisor has two strategies for deduplicate memory: transparent page sharing and content-based page sharing [64]. Transparent page sharing eliminates duplicate memory such as executable code and read-only data by identifying copies and mapping guest pages to the same physical page and following copy-on-write semantics. This requires the guest operating system to be changed so that it is able to identify duplicate copies. Content-based page sharing on the other hand identifies copies based on page contents and does not therefore require changes to the guest operating system. The VMware ESX server uses content-based page sharing by using a hash value of page contents to look up pages that have been marked as copy-on-write in other virtual machines. If such a match is found, the full page contents are compared to ensure that the pages really are identical and then mapped to the guest address space as copy-on-write page.

Kernel Samepage Merging (KSM) in the Linux kernel is an implementation of content-based page sharing that lets applications flag memory regions to be scanned and deduplicated using the `MADV_MERGEABLE` of the `madvise()` system call. The KVM/QEMU hypervisor uses that capability to deduplicate memory between virtual machines [22]. The Xen hypervisor also has a memory deduplication called the Difference Engine which uses page sharing, patching, and compression to improve memory sharing between VMs [39].

An experiment by Waldspurger et al using identical Linux VMs that were running SPEC95 benchmarks demonstrated that up to 67% of memory is shared between virtual machines [64]. However, a more recent study by Barker et al show that absolute sharing sharing levels (which does not include zero pages) remains under 15% for real-world workloads which suggests that memory deduplication is not an effective optimization in many scenarios [25].

3.3 Networking

As summarized in Table 2, virtualization causes performance overheads on networking due to packet processing, NAT, and unstable network. We now detail each of the causes for networking.

3.3.1 Packet Processing

Hypervisor network I/O performance work has focused on making TCP/IP throughput as fast as possible but latency and per-packet overhead are largely unaddressed issues [57]. This is sufficient for traditional server-side network intensive applications but problematic for workloads introduced by NFV that involve very high packet rates. I/O at hypervisor level is implemented in two parts: there's a *frontend* component that exposes an device interface and a *backend* component that ties the frontend to an actual physical device. The guest operating system interacts with the frontend via device drivers to perform I/O. The frontend can either provide an interface that matches a hardware device or provide a paravirtualized device interface that's more efficient for virtualization. For networking I/O, the backend component can be implemented in various ways ranging from POSIX socket APIs to kernel-by-pass.

To perform I/O, the guest OS traps to the hypervisor via an VM exit by issuing an I/O request to a virtual I/O register. The hypervisor then interprets the I/O request, performs I/O as per device model semantics either in synchronous or asynchronous manner, and finally resumes execution in the guest OS. VM exit can be very expensive because the vCPU that's performing I/O may be descheduled, it may block on I/O, or may experience lock contention in device emulation while performing the I/O. On a modern

CPU, accessing an I/O register on bare metal takes 100 ns to 200 ns but in a virtualized environment, I/O register access can take up to 3 µs to 10 µs – a 50 fold increase in access time! As the cost of accessing an I/O register are completely different, OS device drivers need to take this into account when running in a virtualized environment. The NIC notifies the operating system on transmission completion and upon receiving a packet using interrupts. Interrupts are another source of VM exits and must be avoided as much as possible.

3.3.2 Network Address Translation (NAT)

Containers run in the same host machine behind a shared NIC which means the virtualization tool needs to virtualize network addresses. Docker uses network address translation (NAT) by default to provide an externally visible IP address and port to a container but it also supports networking by binding to a virtual Ethernet bridge on the host. NAT introduces a performance overhead because address translation consumes CPU cycles which increases packet round-trip latency as the number of connections increases. A study by Felter at al [37] shows that Docker's NAT significantly reduces the performance of Redis, a popular key-value store, and prevents its from reaching native peak performance.

3.3.3 Unstable Network

A performance evaluation on Amazon EC2 cloud service shows that CPU sharing can cause very unstable TCP and UDP throughput that jumps back and forth between zero and 1 GB/second on small instance types which receive only 40% to 50% of physical CPU time [65]. Furthermore, even when the data center network is not congested, packet delays in different EC2 instances can be hundreds of times larger than packet delays between two hosts which suggests that the Xen hypervisor on EC2 can have very large queues in the virtualized network. The abnormal variance in throughput and packet delays has significant effect on network performance. In a study by Whiteaker, high network utilization from virtual machines that compete on network resources is shown to increase RTT as much as 100 ms [66].

3.4 Disk

As summarized in Table 2, virtualization causes performance overheads on disk due to I/O scheduling and layered filesystems. We now detail each of the causes for disk.

3.4.1 I/O Scheduling

Disk I/O scheduling implemented as part of the operating system kernel has traditionally focused on optimizing throughput of rotational devices by re-ordering I/O requests to minimize seeks which reduces the time needed to wait for the disk head to rotate. As I/O patterns are very workload specific, there is no I/O scheduling policy that is best for all types of workloads. Various I/O scheduling policies have thus been proposed and implemented. Linux, for example, supports four different schedulers with different properties. I/O scheduling has recently been considered less important because seeks are not as expensive on non-rotating devices such as SSD and NVRAM. I/O scheduling is, however, still very relevant for obtaining maximum throughput with hypervisor-based virtualization [29]. With the layered approach of hypervisors, multiple tenants typically share the same physical disk and there are actually multiple parallel I/O scheduling policies at work: one at the hypervisor or host operating system level and one in the guest operating system of each virtual machine. If the I/O schedulers are incompatible with each other, I/O requests may be re-ordered in a way that actually reduces performance. For example, if both the host and guest operating systems are both using an elevator I/O scheduling algorithm, it is estimated that approximately 50% of the time, the two will schedule I/O in opposite order which has significant reduction on application performance. A study by Boutcher and Chandra [29] suggests that using a minimal scheduler at the hypervisor level and a workload specific I/O scheduler at the guest operating system level yields up to 72% I/O throughput improvement over default Linux I/O schedulers.

3.4.2 Layered Filesystems

Docker supports layered filesystems which allows users to stack filesystems on top of each other and reusing the layers to reduce storage space usage and simplify container filesystem management [37]. For example, the same operating system files can be used as the base for multiple application images which significantly reduces container image size compared to images which always contain a full copy of the operating system. Docker usually implements layered filesystem using Another UnionFS (AUFS) which unfortunately introduces a significant overhead on storage because filesystem operations have to travel through many layers [37]. Docker also supports mounting host filesystems directly to avoid layered filesystem performance penalty.

3.5 Summary

As summarized in Table 2, containers do not have CPU overheads but have both networking and disk overheads if NAT and layered filesystems are enabled, respectively. However, both NAT and layered filesystems can be disabled from container configuration which allows native performance. Unikernels are able to reduce some of the CPU overheads compared to running general purpose OS in a hypervisors but are they are still expected to perform worse than containers because of expensive VM exits that are not fully eliminated.

In this thesis, we quantify the impact of CPU and network overheads for hypervisors, unikernels, and containers in an experimental evaluation that is described in detail in Section 4.

4 Methodology

In this section, we detail our methodology for evaluating the overhead of hypervisor-based and container-based virtualization techniques for raw networking and network intensive applications. The overheads are measured in an experimental evaluation in a controlled testbed, which is detailed in Section 4.1. The focus of the evaluation is on CPU and networking virtualization overheads because they are critical for applications that are deployed on the cloud and for network function virtualization (NFV) that deals with very high packet rates [57]. For raw networking performance, we evaluate the latency of network packets traversing across the network stack using Netperf [43]. For network intensive application performance, we use a popular in-memory key-value store, Memcached [7], and measure its per-request latency using the Mutilate benchmarking tool [15]. The Netperf and Memcached test scenarios are outlined in Section 4.2 and Section 4.3, respectively.

4.1 Evaluation Setup

Our controlled testbed consists of two machines that have their NICs connected back-to back as illustrated in Figure 5, which eliminates the overhead from network switches. Firewall is disabled both in the virtualized environments, the host server, and the load generating client. The machines have no other load on them, which allows us to measure the lowest possible latency and isolate the measurement to virtualization overhead. We use commodity desktop class hardware (circa 2014) and Fedora 21 Linux distribution software. The host hardware configuration is enumerated in Table 3 and the software configuration is enumerated in Table 4. On the host, we run a single virtualized environment (a virtual machine or a container) so that we do not introduce noise to the measurements from multiple virtual environments competing from the same physical hardware resources.

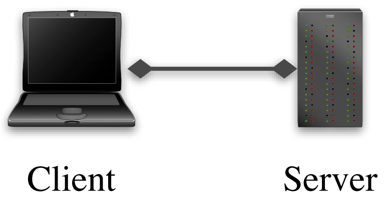


Figure 5: **Evaluation setup.** *The system under test (server) is connected back-to-back to the load generating client to eliminate overhead from a network switch. There are no other processes running on the system and the goal is to obtain lowest possible latency.*

Component	Description
CPU	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (Haswell)
Memory	8 GiB
NIC	Intel Ethernet Connection I218-V (1 Gigabit)

Table 3: **Host hardware configuration.** *The host server that runs Netperf and Memcached is a commodity desktop class machine circa 2014 that has a Intel Haswell CPU, 8 GiB of system memory, and an embedded 1 Gigabit Intel I218-V NIC. The author had complete control over all the processes running on system at all times.*

Component	Version
OS	Fedora 21
Kernel	Linux 4.0.8-200.fc21.x86_64
Hypervisor	QEMU 2.1.3
Container	Docker version 1.6.2.fc21, build c3ca5bb/1.6.2
Netperf	2.6.0 (with modifications)
Mutilate	commit d65c6ef (with modifications)

Table 4: **Host software configuration.** *Fedora 21 is selected as the operating system because it was the latest stable version when preliminary evaluation work started in May 2015. The Linux kernel, QEMU, and Docker versions were the latest versions packaged in Fedora 21 at the time. Netperf 2.6.0 is selected because it's the version OSv supported at the time. The Mutilate tool was cloned from the latest git repository HEAD when Memcached evaluation started in August 2015.*

4.2 Netperf

Netperf is a benchmarking tool that is able to measure various aspects of networking performance for both UDP and TCP [43]. We use Netperf to measure the effect of different virtualization techniques for network throughput and latency as well as CPU utilization. We select Netperf over another popular network performance benchmarking tools such as iPerf [2] because the former is known to run on OSv, an unikernel that is one of the virtualization techniques we are interested in. For network latency, Netperf reports the minimum, maximum, and average latency, including standard deviation, and latency percentiles. Netperf reports the 50^{th}, 90^{th}, and 99^{th} percentiles by default. We modified Netperf to also report the 1^{st}, 5^{th}, 10^{th}, and 95^{th} percentiles to obtain the full latency distribution.

Netperf follows a traditional client-server design as illustrated in Figure 6. There are two independent processes: Netserver that runs on the system under test (also known as remote system) and Netperf that runs as the load generating client (also known as local system). Netserver and Netperf can both act as transmitters or receivers depending on the test scenario. At startup, the Netperf process establishes a control connection to the Netserver process that is used to exchange information about the remote system as well as configure the remote process for a specific test scenario.

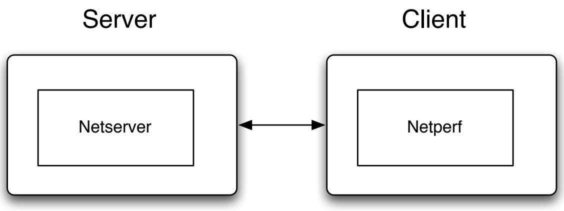


Figure 6: **Netserver and Netperf deployment.** *The system under test (server) runs the Netserver application and the load generating client runs the Netperf tool.*

Netperf supports measuring network throughput via bulk data transfer tests and latency via request/response tests. The Netperf test scenarios that we execute are summarized in Table 5.

Scenario	**Type**	**Connect**	**RX**	**TX**
TCP Stream	throughput	–	✓	–
TCP Reverse Stream	throughput	–	–	✓
TCP Request/Response	latency	–	✓	✓
TCP Connect/Request/Response	latency	✓	✓	✓
UDP Stream	throughput	–	✓	–
UDP Request/Response	latency	–	✓	✓

Table 5: **Netperf test scenarios.** *Netperf test scenarios measure either throughput or latency in different components of the networking stack: connection time (Connect), receive path (RX), and transmission path (TX). A checkmark (✓) indicates that the test scenario measures the component. A dash (–) indicates that the test scenario does not measure the component. Stream test scenarios measure throughput and request/response scenarios measure latency. The only scenario that measures connection establishment and teardown is TCP Connect/Request/Response. Please note that Netperf does not support a UDP Reverse Stream test scenario.*

TCP stream test measures the efficiency of the OS TCP receive path [43]. The test case sends a stream of data from a client running Netperf to a server running Netserver in a single TCP connection. The time to establish the initial connection is not included in the test time but the test waits for all data to reach the server before finishing. TCP stream test simulates a real-world scenario where a client is uploading a large file over TCP.

TCP reverse stream test measures the efficiency of the OS TCP transmission path [43]. The test case is similar to TCP stream except that the flow of data is reverse. The test case sends a stream of data from a server running Netserver to a client running Netperf to in a single TCP connection. The time to establish the initial connection is not included in the test time but the test waits for all data to reach the server before finishing. TCP stream test simulates a real-world scenario where a client is downloading a large file over TCP.

TCP request/response test measures the efficiency of both OS TCP transmission and receive paths [43]. The test case sends one request at a time from a client running Netperf to a server running Netserver and waits synchronously for the server to respond in a single connection. The time to establish a TCP connection is not included in the result. Both the request and response message payloads are 1 byte long in the test case. TCP request/response test simulates a real-world scenario where a client and a server are exchanging short messages.

TCP connect/request/response test measures the efficiency of both OS TCP transmission and receive paths as well as TCP connection establishment [43]. The test case opens a new connection, sends a request from a client running Netperf to a server running Netserver, waits synchronously for the server to respond, and finally closes the connection. Both the request and response message payloads are 1 byte long in the test case. TCP connect/request/response test simulates a real-world scenario where a client is sending HTTP requests to a server.

UDP stream test measures the efficiency of the OS UDP receive path [43]. The test case sends a stream of data from a client running Netperf to a server running Netserver in a single UDP connection. The time to establish the initial connection is not included in the test time but the test waits for all data to reach the server before finishing. UDP stream test simulates a real-world scenario where a client is streaming a large file over UDP.

UDP request/response test measures the efficiency of both OS UDP transmission and receive paths [43]. The test case sends one request at a time from a client running Netperf to a server running Netserver and waits synchronously for the server to respond in a single connection. The time to establish a TCP connection is not included in the result. Both the request and response message payloads are 1 byte long in the test case. UDP request/response test simulates a real-world scenario where a client and a server are exchanging messages.

Every test case is run for 50 seconds and every test case is repeated 10 times. The evaluation results of Netperf tests are presented in Section 5.

4.3 Memcached and Mutilate

Memcached is a high-performance, distributed in-memory key-value store that is typically used as an object caching system to improve the performance of web applications by reducing database load [7]. We use Memcached to measure the effect of different virtualization techniques for a network intensive application. Memcached is implemented as a client-server architecture where the server effectively provides a distributed hash table that is populated and queried by clients. When the memory allocated for Memcached server fills up, least recently used data is discarded from the cache.

Mutilate is a high-performance load generator client for Memcached that simulates realistic workloads and makes it easy to observe the request processing latency distribution [15]. Mutilate implements the full Memcache client-side protocol, which allows it to connect to a Memcached server and interact with it to measure request processing latency on the client-side. Mutilate supports multiple threads, multiple connections, request pipelining, and remote agents. Mutilate reports the 1^{st}, 5^{th}, 10^{th}, 90^{th}, 95^{th}, and 99^{th} percentiles by default. We modified Mutilate to also report the 50^{th} percentile to obtain the full latency distribution. The Mutilate test scenarios that we execute are summarized in Table 6.

Scenario	**Key Size**	**Value Size**	**set:get Ratio**	**Update**	**Read**
Update	fb_key	fb_value	0.0	✓	–
Read	fb_key	fb_value	1.0	–	✓
Mixed	fb_key	fb_value	0.03	✓	✓

Table 6: **Mutilate test scenarios.** *Mutilate test scenarios measure request/response throughput for update and read operations. A checkmark (✓) indicates that the test scenario measures operation. A dash (–) indicates that the test scenario does not measure the operation.*

Memcached is an interesting network intensive application to evaluate because it is so widely deployed and real-world Memcached workloads are well-documented in literature [23]. There are various server applications that implement the Memcache protocol but the canonical Memcached server that we test is implemented on top of an event-based, non-blocking networking library *libevent*. Libevent is an abstraction over OS specific non-blocking network interfaces like `epoll` on Linux and `kqueue` on OS X. The performance of Memcached is highly influenced by operating system networking stack, scheduling, threading, and memory management services. However, as

Memcached is an in-memory data store, it does not perform disk I/O, which makes it a good candidate for performance evaluation in virtualized environments.

Every test case is run for 50 seconds and every test case is repeated 10 times. The evaluation results of Memcached tests are presented in Section 6.

4.4 Summary

We evaluate the overhead of hypervisor-based and container-based virtualization techniques for raw networking and network intensive applications by conducting an experimental evaluation in a controlled testbed. We use Netperf to measure raw networking performance and Memcached and Mutilate to measure network intensive application performance with focus on CPU and networking virtualization overheads. In the evaluation setup, the system under test is connected back-to-back to the load generating client to eliminate overhead from a network switch. There are no other processes running on the system and the goal is to obtain lowest possible latency and CPU overhead of virtualization techniques. The host server that runs Netserver and Memcached is a commodity desktop class machine circa 2014 that has a Intel Haswell CPU, 8 GiB of system memory, and an embedded 1 Gigabit Intel I218-V NIC. The author had complete control over all the processes running on system at all times. Fedora 21 is selected as the operating system because it was the latest stable version when preliminary evaluation work started in May 2015. The Linux kernel, QEMU, and Docker versions were the latest versions packaged in Fedora 21 at the time. Netperf 2.6.0 is selected because it's the version OSv supported at the time. The Mutilate tool was cloned from the latest git repository HEAD when Memcached evaluation started in August 2015.

5 Evaluation Using Netperf

In this section, we present the evaluation results of raw networking performance for the different virtualization techniques enumerated in Table 7. For our experiments, we use the Netperf benchmarking tool that was detailed in Section 4.2. We compare hypervisor, unikernel, and container performance to bare metal Linux performance that is used as the baseline. In all of the combinations, hypervisor-based virtualization techniques use QEMU/KVM as the virtualization tool and OSv as the unikernel (where applicable) and container-based virtualization techniques use Docker as the virtualization tool. The full host software configuration is detailed in Table 4 and VM and container configurations in Table 8.

#	**Virtualization Technique**	**Guest OS**	**Tool**	**Network**	**Offload**
1	Hypervisor	Linux	KVM	virtio-net	–
2	Hypervisor	Linux	KVM	virtio-net	✓
3	Hypervisor + Unikernel	OSv	KVM	virtio-net	✓
4	Hypervisor	Linux	KVM	vhost-net	–
5	Hypervisor	Linux	KVM	vhost-net	✓
6	Hypervisor + Unikernel	OSv	KVM	vhost-net	✓
7	Container	Linux	Docker	NAT	✓
8	Container	Linux	Docker	Bridged	✓
	None *(bare metal Linux)*	N/A	N/A	N/A	✓

Table 7: **Virtualization techniques.** *Hypervisor-based virtualization techniques (1-6) use QEMU/KVM as the virtualization tool with combinations of Linux and OSv as the guest operating system and virtio-net and vhost-net as the network I/O virtualization technique with (virtual) NIC offloading capabilities enabled and disabled. Container-based virtualization techniques (7-8) use Docker as the virtualization tool with networking configured to either use the NAT (default configuration) and bridged mode.*

For evaluation, we execute a subset of Netperf test scenarios to measure latency and throughput for both TCP and UDP for all the different virtualization scenarios. The Netperf test scenarios are enumerated in Table 5. The Netserver server was started using its default configuration.[3] Each test case is executed 10 times and each iteration is run for 50 seconds. Processes, interrupts and vCPUs are not bound to specific CPUs to simplify system configuration. Although this *increases* noise in the results, it is a typical configuration for real deployments.

[3]Nagle's algorithm is enabled in the default Netperf configuration (i.e. `TCP_NODELAY` socket option is not set), which *increases* request/response latency because small packets are sent in batches.

Virtualization Technique	**Software Component**	**Version**
Hypervisor	OS	Fedora 21
Hypervisor	Kernel	Linux 4.0.8-200.fc21.x86_64
Hypervisor + Unikernel	OS	OSv 41dffab (2015-09-03)
Container	OS	Fedora 21
All	Netserver	2.6.0 (with modifications)

Table 8: **Virtual machine guest and container configurations.** *Fedora 21 is used as the operating system for as the Linux guest OS in VMs and in containers. OSv is used as the unikernel for hypervisor-based virtualization. Netserver 2.6.0 (with modifications) is installed on the system under test.*

The overheads for each virtualization technique are presented as percentage overhead to running the the same workload running on bare metal configuration (no virtualization). The plots reports mean, median, 10^{th}, and the 90^{th} percentile for latency, throughput, and CPU utilization overhead by taking the average for each metric from the 10 test iterations.

5.1 TCP Stream

The TCP stream test (TCP_STREAM) measures the efficiency of the OS TCP receive path [43].

The time to establish the initial connection is not included in the test time but the test waits for all data to reach the server before finishing. *TCP stream test simulates a real-world scenario where a client is uploading a large file over TCP.*

TCP stream results are presented in Figure 7. Docker (techniques 7 and 8) has the least overhead of all the virtualization techniques. Docker using bridged networking mode (technique 8) is the fastest having less overhead than Docker using NAT (technique 7) as expected because Docker NAT is known to increase virtualization overhead [37]. OSv running with vhost-net (technique 6) has the least overhead of hypervisor-based virtualization techniques and its performance is very close to Docker. CPU utilization in the guests is lower for vhost-net than for virtio-net as expected because vhost-net offloads packet processing from the guest to the host. We did not measure host CPU utilization so the impact of vhost offloading for total CPU utilization is unknown.

However, contrary to a previously published Netperf benchmark using a 40 Gigabit NIC [46] that show a 25% throughput improvement for OSv, Linux has better throughput than OSv in our tests, which suggests that OSv's networking stack is suboptimal for slower NICs for TCP receive.[4]

[4]Our evaluation setup uses a less powerful 1 Gigabit NIC.

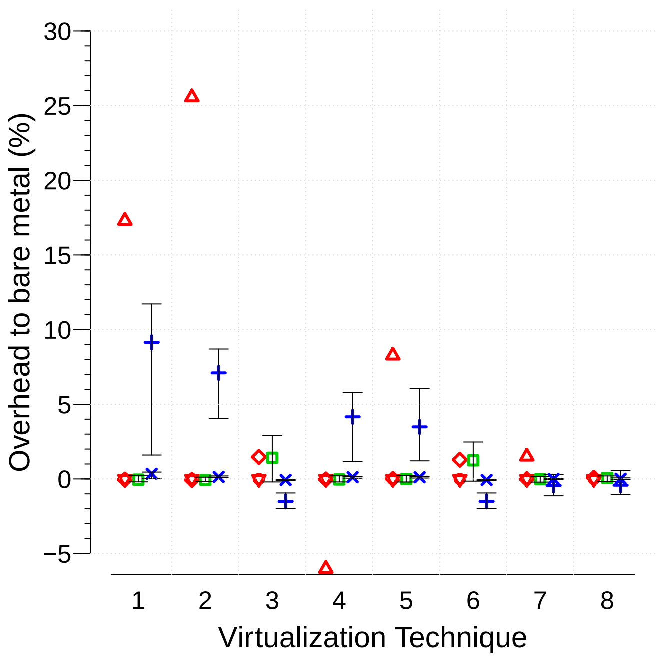


Figure 7: **Netperf TCP stream results.** *Docker (techniques 7 and 8) has the least overhead of all the virtualization techniques. Docker using bridged networking mode (technique 8) is the fastest having less overhead than Docker using NAT (technique 7) as expected because Docker NAT is known to increase virtualization overhead [37]. OSv running with vhost-net (technique 6) has the least overhead of hypervisor-based virtualization techniques and its performance is very close to Docker. The error bars indicate the minimum and maximum over the 10 iterations.*

Another published Netperf benchmark using 10 Gigabit NIC [50] reports a 28% and 26% *decrease in throughput* in TCP stream test for both Linux and OSv (both using virtio-net), respectively. We do not see such a performance drop for Linux in our tests compared to bare metal Linux, which suggests faster 10 Gigabit NICs suffer more from virtualization than the slower 1 Gigabit NICs.

5.2 TCP Reverse Stream (TCP_MAERTS)

The TCP reverse stream test (TCP_MAERTS) measures the efficiency of the OS TCP transmission path [43]. The test case is similar to TCP stream except that the data flows in the reverse direction. In this test case, the server sends a stream of data from a server running Netserver to a client running Netperf to in a single TCP connection. The time to establish the initial connection is not included in the test time but the test waits for

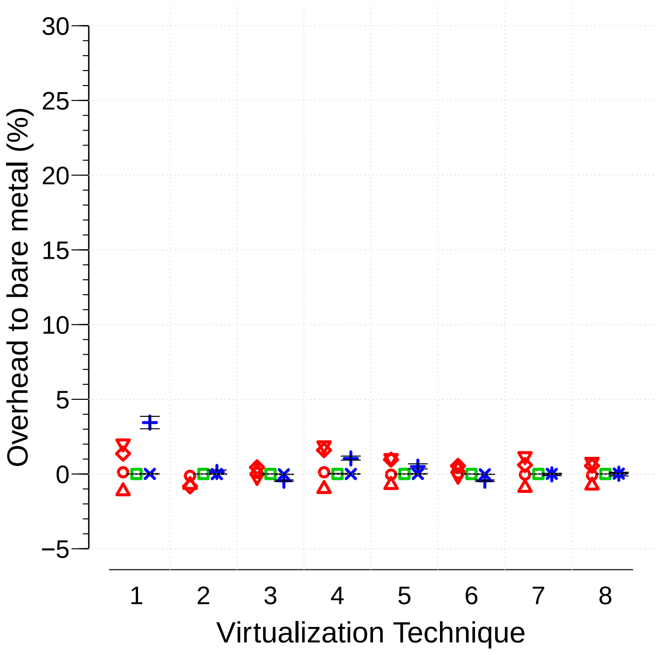


Figure 8: **Netperf TCP reverse stream results.** *OSv (techniques 3 and 6) has the least overhead of all the virtualization techniques. Docker and Linux running on QEMU/KVM do not have significant differences in overhead when offloading is enabled. Surprisingly, when offloading is enabled, Linux with virtio-net (technique 2) has less overhead than vhost-net (technique 5). The error bars indicate the minimum and maximum over the 10 iterations.*

all data to reach the server before finishing. *TCP stream test simulates a real-world scenario where a client is downloading a large file over TCP.*

TCP reverse stream results are presented in Figure 8. OSv (techniques 3 and 6) has the least overhead of all virtualization techniques for this test. Docker and Linux running on QEMU/KVM do not have significant differences in overhead when offloading is enabled. Surprisingly, when offloading is enabled, Linux with virtio-net (technique 2) has less overhead than vhost-net (technique 5), which suggests that there is an inefficiency in Linux vhost-net driver for offloading. Overall, TCP reverse stream tests have less performance differences across all the virtualization techniques than TCP stream test has, which suggests that TCP transmission does not suffer as much from virtualization as TCP receive does.

5.3 TCP Request/Response (TCP_RR)

The TCP request/response test (TCP_RR) measures the efficiency of both OS TCP transmission and receive paths [43]. The test case sends one request at a time from a client running Netperf to a server running Netserver and waits synchronously for the server to respond in a single connection. The time to establish a TCP connection is not included in the result. Both the request and response message payloads are 1 byte long in the test case. *TCP request/response test simulates a real-world scenario where a client and a server are exchanging short messages.*

TCP request/response results are presented in Figure 9. Docker using bridged networking (technique 8) has the least overhead and OSv with vhost-net (technique 6) comes to a close second beating Docker using NAT networking (technique 7). OSv with vhost-net (technique 6) has less overhead than Linux with vhost-net (technique 5), which is consistent with a previously published benchmarks using a 40 Gigabit NIC [46] that shows a 37%-47% reduction in latency. Surprisingly, OSv with virtio-net is slowest of all the virtualization techniques, which suggests an inefficiency in the virtio-net driver for TCP request/response. The results are similar to a previously published benchmark using 10 Gigabit NIC [50] that reports 10%, 47% and 43% *decrease in throughput* in TCP_RR for Docker, Linux and OSv (both using virtio-net), respectively, compared to bare metal Linux. Felter et al [37] report a 100% increase in request/response latency for Docker NAT and 80% increase for KVM/QEMU running Linux using a 10 Gigabit NIC. The results are inline with our evaluation results, although the overhead we observe is not as high.

5.4 TCP Connect/Request/Response (TCP_CRR)

The TCP connect/request/response test (TCP_CRR) measures the efficiency of both OS TCP transmission and receive paths as well as TCP connection establishment [43]. The test case opens a new connection, sends a request from a client running Netperf to a server running Netserver, waits synchronously for the server to respond, and finally closes the connection. Both the request and response message payloads are 1 byte long in the test case. *TCP connect/request/response test simulates a real-world scenario where a client is sending HTTP requests to a server.*

TCP connect/request/response results are presented in Figure 10. Docker (techniques 7 and 8) has the least overhead of all virtualization techniques. Surprisingly, Linux with offloading enabled has less overhead than OSv for virtio-net (technique 2 vs technique 3) and vhost-net (technique 5 vs technique 6), which suggests an inefficiency in OSv's TCP connection establishment because we do not see the same overhead in the TCP request/response test. We are unable to identify the exact source of the inefficiency but it is worth noting that because TCP request/receive does not have the inefficiency the overhead is unlikely to be in the virtio drivers and more likely to be in the TCP/IP stack, the network channel architecture, or the scheduler.

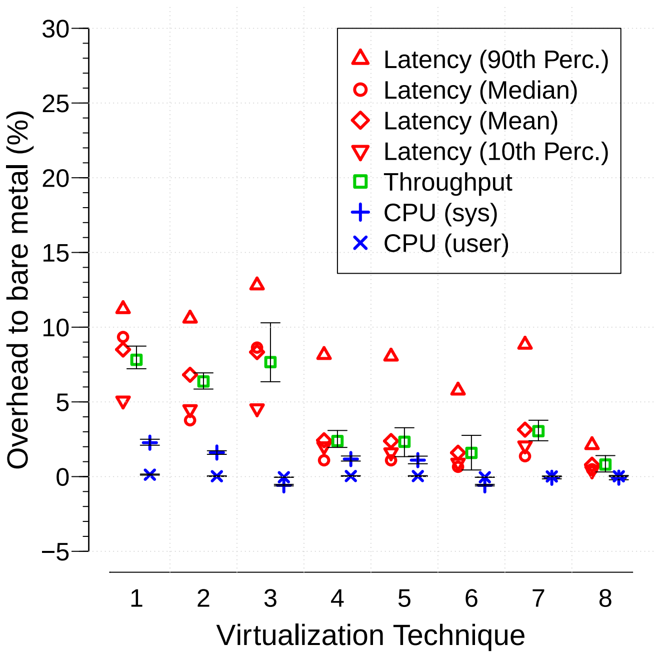


Figure 9: **Netperf TCP request/response results.** *Docker using bridged networking (technique 8) has the least overhead and OSv with vhost-net (technique 6) comes to a close second beating Docker using NAT networking (technique 7). OSv with vhost-net (technique 6) has less overhead than Linux with vhost-net (technique 5). The error bars indicate the minimum and maximum over the 10 iterations.*

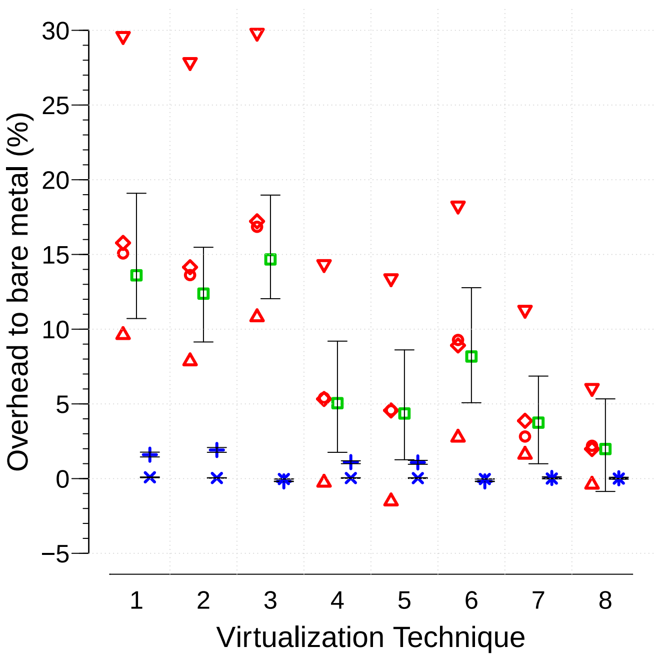


Figure 10: **Netperf TCP connect/request/response results.** *Docker (techniques 7 and 8) has the least overhead of all virtualization techniques. Surprisingly, Linux with offloading enabled has less overhead than OSv for virtio-net (technique 2 vs technique 3) and vhost-net (technique 5 vs technique 6), which suggests an inefficiency in OSv's TCP connection establishment. The error bars indicate the minimum and maximum over the 10 iterations.*

5.5 UDP Stream (UDP_STREAM)

The UDP stream test (UDP_STREAM) measures the efficiency of the OS UDP receive path [43]. The test case sends a stream of data from a client running Netperf to a server running Netserver in a single UDP connection. The time to establish the initial connection is not included in the test time but the test waits for all data to reach the server before finishing. *UDP stream test simulates a real-world scenario where a client is streaming a large file over UDP.*

UDP stream results are presented in Figure 11. Docker (techniques 7 and 8) has the least overhead of all virtualization techniques. Linux (technique 2) and OSv (technique 3) with virtio-net and Linux (technique 5) and OSv (technique 6) do not have significant differences, respectively. However, offloading increases overhead for both virtio-net (techniques 1 and 2) and vhost-net (techniques 4 and 5). The throughput numbers for OSv are missing, which suggests a bug in OSv in data gathering.

Contrary to a previously published Netperf benchmark using 10 Gigabit NIC [50] that reports 42%, 54% and 57% *decrease in throughput* for Docker, Linux and OSv (both using virtio-net) in UDP stream test, respectively, we do not see such a performance drop in our tests compared to bare metal Linux, which suggests faster 10 Gigabit NICs suffer more from virtualization than the slower 1 Gigabit NICs.

5.6 UDP Request/Response (UDP_RR)

The UDP request/response test (UDP_RR) measures the efficiency of both OS UDP transmission and receive paths [43]. The test case sends one request at a time from a client running Netperf to a server running Netserver and waits synchronously for the server to respond in a single connection. The time to establish a TCP connection is not included in the result. Both the request and response message payloads are 1 byte long in the test case. *UDP request/response test simulates a real-world scenario where a client and a server are exchanging messages.*

Docker using bridged networking (technique 8) has the least overhead and OSv with vhost-net (technique 6) comes to a close second. Docker using NAT networking (technique 7) has more overhead than Linux using vhost-net (techniques 4 and 5). It is worth nothing that OSv with vhost-net (technique 6) has the largest improvement to Linux with vhost-net (technique 5) over all the different Netperf test scenarios.

Our results are similar to another previously published benchmark using 40 Gigabit NIC [46] that reports a 47% reduction in latency for UDP request/response test for OSv compared to Linux. However, contrary to a previously published Netperf benchmark using 10 Gigabit NIC [50] that reports 12%, 45% and 43% *increase in latency* for Docker, Linux and OSv

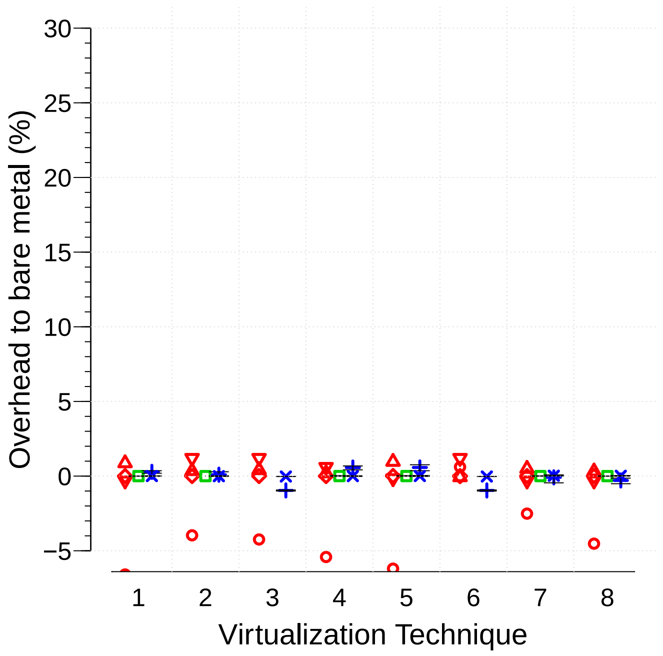


Figure 11: **Netperf UDP stream results.** *Docker (techniques 7 and 8) has the least overhead of all virtualization techniques. Linux (technique 2) and OSv (technique 3) with virtio-net and Linux (technique 5) and OSv (technique 6) with vhost-net do not have significant differences, respectively. However, offloading increases overhead for both virtio-net (techniques 1 and 2) and vhost-net (techniques 4 and 5). The throughput numbers for OSv are missing, which suggests a bug in OSv in data gathering. The error bars indicate the minimum and maximum over the 10 iterations.*

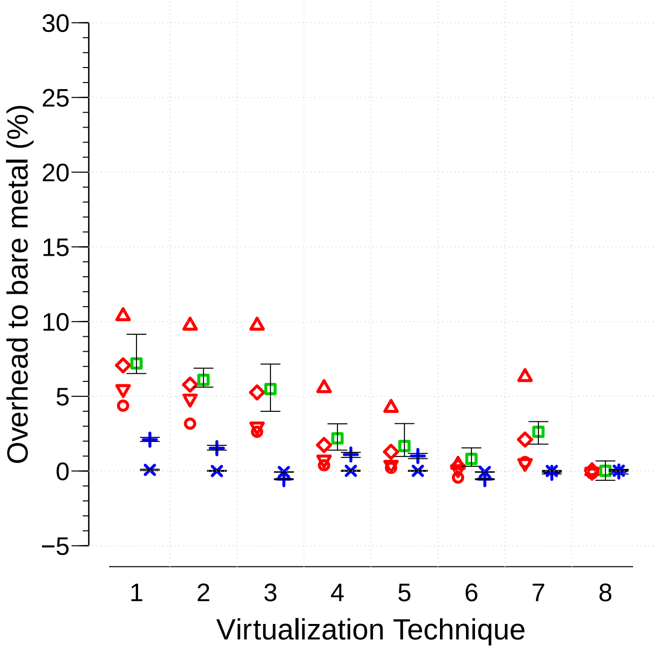


Figure 12: **Netperf UDP request/response results.** *Docker using bridged networking (technique 8) has the least overhead and OSv with vhost-net (technique 6) comes to a close second. Docker using NAT networking (technique 7) has more overhead than Linux using vhost-net (techniques 4 and 5). The error bars indicate the minimum and maximum over the 10 iterations.*

(both using virtio-net) in UDP stream test, respectively, we do not see such a performance drop in our tests compared to bare metal Linux. Felter et al [37] report a 100% increase in request/response latency for Docker NAT and 80% increase for Linux running on KVM/QEMU using a 10 Gigabit NIC. The results are inline with our evaluation results, although the overhead we observe is not as high.

5.7 Summary and Discussion

Container-based virtualization techniques appear to have less overhead than hypervisor-based techniques for raw networking performance. Docker using bridged networking (technique 8) has the least overhead in all the Netperf test scenarios except for TCP reverse stream test (that measures TCP transmission performance) in which OSv has the least overhead. OSv with vhost-net (technique 6) has the least overhead of hypervisor-based virtualization technique in all the Netperf test scenarios except for TCP connect/request/response test, which suggests an inefficiency in OSv's TCP/IP stack or the scheduler for TCP connection establishment. Overall, unikernels are a very promising optimization for hypervisor-based virtualization for raw networking performance but the technology needs to mature to really deliver the performance potential.

6 Evaluation Using Memcached

In this section, we present the performance evaluation results of network intensive application performance for the different virtualization techniques enumerated in Table 9. For our experiments, we use the Memcached key-value store and Mutilate benchmarking tool, which are detailed in Section 4.3. We selected virtualization techniques that performed the best for raw networking performance in Section 5 excluding configurations with offloading disabled and virtio-net to reduce the scope of the experimental evaluation. The full host software configuration is detailed in Table 4 and VM and container configurations in Table 10.

#	Virtualization Technique	Guest OS	Tool	Network	Offload
5	Hypervisor	Linux	KVM	vhost-net	✓
6	Hypervisor + Unikernel	OSv	KVM	vhost-net	✓
7	Container	Linux	Docker	NAT	✓
8	Container	Linux	Docker	Bridged	✓
-	**None** *(bare metal Linux)*	N/A	N/A	N/A	✓

Table 9: **Virtualization techniques.** *Hypervisor-based virtualization techniques (5-6) use QEMU/KVM as the virtualization tool with combinations of Linux and OSv as the guest operating system using vhost-net as the network I/O virtualization technique with (virtual) NIC offloading capabilities enabled. Container-based virtualization techniques (7-8) use Docker as the virtualization tool with networking configured to either use the NAT (default configuration) and bridged mode.*

Virtualization Technique	Software Component	Version
Hypervisor	OS	Fedora 21
Hypervisor	Kernel	Linux 4.0.8-200.fc21.x86_64
Hypervisor + Unikernel	OS	OSv git commit 41dffab (2015-09-03)
Container	OS	Fedora 21
All	Memcached	1.4.17

Table 10: **Virtual machine guest and container configurations.**

For evaluation, we execute Mutilate test scenarios enumerated in Table 6 to measure latency of TCP request/response for all the different virtualization scenarios. The Mutilate tool does not support UDP so that was excluded from the tests. The Memcached server was started using its default configuration: 1024 maximum simultaneous connections, 64 MB of memory to be used for items, and one thread per CPU. Mutilate was also started using its default configuration: 1 connection, no request pipelining, and `TCP_NODELAY` enabled. Each test case is executed 10 times and each iteration is run for 50 seconds. Furthermore, processes, interrupts and vCPUs are not bound to specific CPUs to simplify system configuration. Although this *increases* noise in the results, it is a typical configuration for real deployments. The key and value sizes are configured to follow Facebook's ETC workload distribution [23]. ETC workload was selected because it's a documented real-world workload and the largest Memcached workload type at Facebook.

6.1 Update Operations

The Mutilate update scenario measures OS TCP/IP stack and Memcached LRU cache performance. Mutilate is configured to only send update operations using key and value sizes that are generated using Facebook's ETC key and value distributions [23]. The total amount of key-value pairs transmitted during the test is approximately 70 MiB that exceeds the 64 MiB that is reserved for the Memcached in-memory store and therefore causes LRU cache evictions during the test. The response message payload generated by Memcache is approximately 6 bytes long so TCP receive is dominant in the test for OS overhead.

Mutilate update scenario results are presented in Figure 13. Docker using bridged networking (technique 8) has least overhead as expected and Docker using NAT networking (technique 7) is the second fastest technique. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5). The results are surprising as OSv (technique 6) has *less* overhead than Docker using NAT (technique 7) in the Netperf TCP request/response test detailed in Section 5.3. We speculate that this is a weakness in OSv's pthread mutex implementation that Memcached uses for synchronizing data structure updates between multiple threads. The results are similar to a previously published Memcached benchmark using a 40 Gigabit NIC [46] that show 22% throughput improvement for OSv compared to Linux using Memaslap [9], another Memcached benchmarking tool.

6.2 Read Operations

The Mutilate update scenario measures OS TCP/IP stack and Memcached LRU cache performance. The Memcached server is populated with data using key and value sizes that are generated using Facebook's ETC key and value distributions [23] and Mutilate is configured to only send read operations to the server. The total amount of keys sent during the test is approximately 12 MiB and the amount key-value pairs received during the test is approximately 70 MiB, which means that TCP transmission is

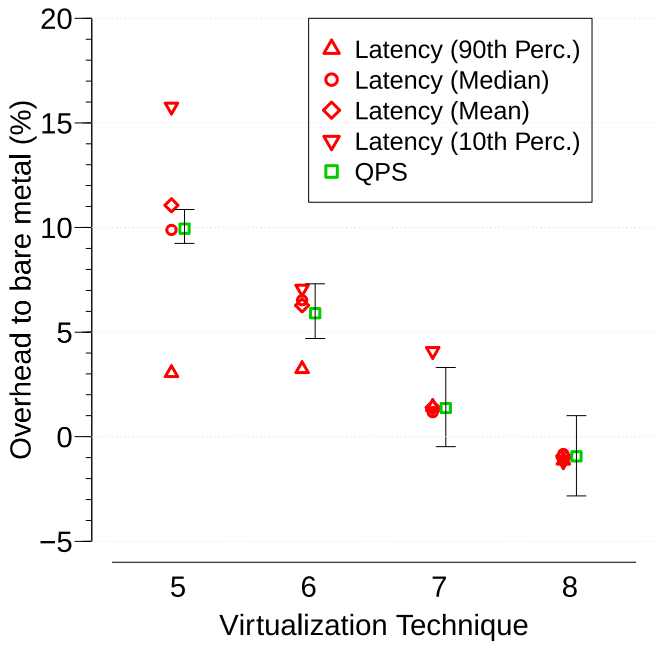


Figure 13: **Mutilate update scenario request/response latency.** *Docker using bridged networking (technique 8) has least overhead as expected and Docker using NAT networking (technique 7) is the second fastest technique. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5). The error bars indicate the minimum and maximum over the 10 iterations.*

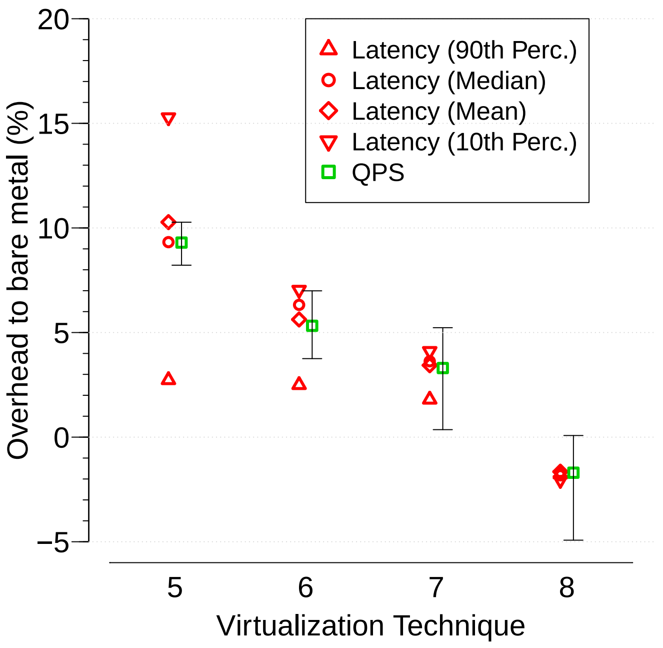


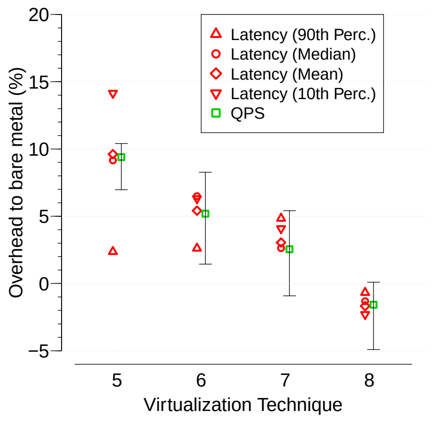
Figure 14: **Mutilate read scenario request/response latency.** *Docker using bridged networking (technique 8) has least overhead as expected and Docker using NAT networking (technique 7) is the second fastest technique. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5). The error bars indicate the minimum and maximum over the 10 iterations.*

dominant in the test for OS overhead.

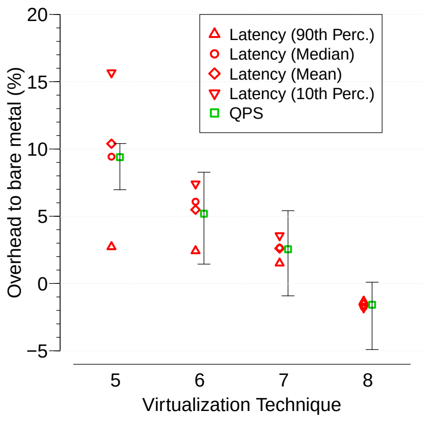
Mutilate read scenario results are presented in Figure 14. Docker using bridged networking (technique 8) has least overhead as expected and Docker using NAT networking (technique 7) is the second fastest technique. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5). The results are similar to Mutilate update scenario detailed in Section 6.1.

6.3 Mixed Read and Write Operations

The Mutilate mixed scenario measures OS TCP/IP stack and Memcached LRU cache performance. Mutilate is configured to send both update and read operations in ratio of 1:30 and using key and value sizes that are generated using Facebook's ETC key and value distributions [23]. The total amount of key-value pairs transmitted during the test is approximately 12 MiB and the amount of key-values received during the test is approximately 70 MiB so TCP transmission is dominant in the test for OS overhead.



(a) **Update latency.**



(b) **Read latency.**

Figure 15: **Memcached mixed workload (update and write).** *Docker using bridged networking (technique 8) has least overhead as expected and Docker using NAT networking (technique 7) is the second fastest technique. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5). The error bars indicate the minimum and maximum over the 10 iterations.*

Mutilate mixed scenario results are presented in Figure 15. Docker using bridged networking (technique 8) has least overhead as expected and Docker using NAT networking (technique 7) is the second fastest technique. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5). The results are similar to Mutilate update scenario detailed in Section 6.1 and read scenario detailed in Section 6.2.

6.4 Summary and Discussion

Container-based virtualization techniques appear to have less overhead than hypervisor-based techniques for network intensive application performance. Docker using bridged networking (technique 8) has the least overhead in all the Mutilate test scenarios with Docker using NAT (technique 7) coming to a close second. OSv (technique 6) has the least overhead of the two hypervisor-based virtualization techniques beating Linux (technique 5) but it's still ahead of Docker in performance. The results are surprising because OSv beats Docker NAT in raw networking performance as detailed in Section 5, which suggests that while networking in OSv, other components in the system are slowing down request/response processing in Memcached. More research is needed to understand where the overhead is and how to reduce it in OSv.

7 Related Work

In this section, we discuss three related studies that evaluate virtualization overheads by Morabito et al [50], Felter et al [37], and Hwang et al [42]. We also detail Netmap, a packet I/O framework for by-passing the host OS kernel [56], that significantly reduces networking overhead for bare metal performance.

7.1 Evaluations

Morabito et al [50] compare the performance of hypervisors, unikernels, and containers in an experimental evaluation that is similar to ours. They compare the performance of two container tools, LXC, and Docker, to the KVM/QEMU hypervisor running Linux and OSv as the guest OS. Both LXC and Docker are are based on Linux control groups (cgroups) and namespaces. The use several benchmarks to measure the performance of for CPU processing, memory, storage, and network for the various virtualization techniques and show that containers are able to achieve higher performance than virtual machines.

CPU processing performance is evaluated using the Y-cruncher, NBENCH, Geekbench, and Linpack benchmarks. Y-cruncher is a multi-threaded CPU benchmark that shows containers performing better than Linux running on KVM/QEMU. The NBENCH and Geekbench benchmarks show similar performance across all techniques except for memory performance that has 30% worse performance for KVM/QEMU than for other scenarios. Y-cruncher NBENCH, and Geekbench were not run on OSv because of benchmark application portability issues. Linpack, which was also run on OSv, had almost no difference in terms of performance across any of the techniques.

Memory performance is evaluated using the STREAM benchmark. The results show nearly identical performance across all configurations except for OSv, which shows half the performance compared to the rest. There is no architectural reason for OSv to be slower for such a simple benchmark, which suggests an issue in the benchmark while running on OSv (e.g. implementation bug in the timekeeping functionality of OSv used by the benchmark).

Storage performance is evaluated using the Bonnie++ benchmark. Their evaluation shows that both container-based tools have similar performance characteristics and are both close to bare metal performance. The storage performance of KVM/QEMU is the slowest with write throughput only 30% and read throughput only 20% of bare metal performance. However, The Bonnie++ results are inconclusive because they were not reproducible with two other storage benchmarks, Sysbench and IOzone. This highlights the fact that disk I/O performance measurement for virtualized environments is difficult. The Bonnie++ benchmark was not run on OSv because Bonnie++ requires the `fork()` system call that OSv does not support.

Network performance is evaluated using the Netperf benchmark. The results show TCP performance of container-based solutions and bare metal to be almost equal but 28% worse for Linux running on KVM/QEMU and 26% worse for OSv running on KVM/QEMU compared to bare metal. The authors do not specify what KVM/QEMU networking options are used so we do not know if vhost is enabled or not that makes a big difference according to our tests. UDP throughput was found to be 42.97% worse for Docker, 54.35% worse for KVM, and 46.88% worse for OSv compared to bare metal. For request-response netperf benchmarks, LXC and Docker were fastest of the virtualized configurations with 17.35% and 19.36% worse performance than bare metal, respectively. For the same benchmark, OSv shows an improvement over Linux as the guest OS with 43.11% worse performance to bare metal compared to 47.34% for Linux. It is important to note that their networking tests were run on 10 Gigabit NIC whereas our experiments were run on 1 Gigabit NIC.

The evaluated virtualization techniques in Morabito et al [50] are similar to ours but they do not specify whether they are using virtio or vhost and do not test Docker using bridged networking. However, they also evaluate CPU, memory, and disk I/O performance in isolation whereas we focus on combined CPU and networking performance for network intensive applications. Their Netperf evaluation results show increase in latency and decrease in throughput for both Docker and KVM/QEMU in TCP stream, and UDP stream, and UDP request/response. The results for TCP request/response are similar to ours.

Felter et al [37] have also compared the performance of containers and virtual machines to bare metal using Docker and KVM/QEMU.

CPU processing performance is evaluated using PXZ, a multi-threaded LZMA-based data compression utility, and Linpack, a HPC benchmark that that solves a dense system of linear equations. In their experiments, they found KVM/QEMU to be 22% slower than Docker for the PXZ compression benchmark even after tuning KVM by pining vCPUs to physical CPUs and exposing cache topology. They did not investigate the problem further but speculate that the performance difference might be caused by extra TLB pressure from nested paging on KVM/QEMU. Linpack performance is almost identical on both KVM/QEMU and Docker but they note that KVM/QEMU that has not been tuned properly shows a significant performance degradation. Memory performance is evaluated using the STREAM benchmark that shows nearly identical performance across bare metal, Docker, and KVM. Random memory access performance is also identical across configurations.

Block I/O performance is evaluated using the fio benchmark tool. Their results show that Docker and KVM/QEMU have almost no overhead compared to bare metal for sequential access. However, for random read, write, and mixed workloads, KVM/QEMU is only able to deliver half of the IOPS compared to bare metal and Docker.

Network throughput performance is measured using the nuttcp [6] tool. They use the tool to measure only TCP throughput and do not measure UDP throughput at all. This is different from our experiment that measures both TCP and UDP throughput using Netperf. Their results shows that bare metal, Docker, and KVM/QEMU all are able to reach 9.3 Gbps TCP throughput for receive and transmit, which is close to theoretical limit of the 10 Gigabit NIC. Docker NAT and KVM/QEMU have higher CPU overhead than bare metal. They also measure network latency using Netperf TCP request/response and UDP request/response tests and observe a 100% increase in latency for Docker NAT and 80% increase in latency for KVM/QEMU compared to the bare metal latency.

They also tested the performance of two database systems: Redis and MySQL. Redis is very network intensive so the higher network latency of Docker NAT and KVM hurt performance. However, as the number of clients increase, throughput approaches native performance as per Little's Law. For MySQL performance, KVM is shown to have 40% higher overhead in all measured cases. Docker's performance with AUFS enabled also shows significant overhead.

Hwang et al [42] compared the performance of four hardware-assisted hypervisors, Hyper-V, KVM, vSphere, and Xen, and found out that while there are significant differences across hypervisor performance, no single hypervisor was able to consistently outperform the others.

CPU processing performance is evaluated using the Bytemark benchmark. Their results show that all hypervisors were almost no overhead compared to bare metal because the benchmark itself does not trap to the hypervisor. Memory performance is evaluated using the Ramspeed benchmark that measures cache and memory bandwidth Their results show that all hypervisors have almost no overhead compared to bare metal for single vCPU scenario. However, for 4 vCPU scenario, KVM/QEMU performs worse than other hypervisors and is not able to utilize the available vCPUs completely.

Disk I/O performance is evaluated using the Bonnie++ benchmark. Their results show that every hypervisor performs similarly to each other, except for Xen that has 40% worse performance than other hypervisors. They also used the FileBench benchmark to confirm the results gathered from Bonnie++.

Network throughput performance is evaluated using the Netperf. Their results show that vSphere is the fastest hypervisor with 22% better throughput than Xen. Both KVM/QEMU and Hyper-V are close to vSphere in performance.

7.2 Netmap

Netmap is a packet I/O API for that allows applications to directly map NIC queues in a safe manner in their address space and perform I/O on them, by-passing the kernel completely for networking [56]. It solves the problem of high overhead from operating system networking stack for applications that deal with very high rate of packets. Netmap architecture reduces the cost of moving packets between hardware and the operating system networking stack and is orthogonal to virtualization networking techniques such as paravirtualization and SR-IOV and bare metal performance optimizations such as hardware checksumming, TCP segmentation offloading (TSO), and large receive offloading (LRO).

Netmap exposes hardware NIC receive and transmit queues as *netmap rings* that are memory mapped to userspace application virtual memory address space as illustrated in Figure 16. These netmap rings are circular queues of packet buffers where each queue entry represents a packet specified by a virtual address of the packet data start, lento of the packet, and special netmap flags. The shared memory accessible by userspace is always validated and the application is not able to directly access NIC hardware registers. An application is, however, able to corrupt the state of a netmap ring that's used by other applications.

To use Netmap, an application requests the operating system to put a NIC in netmap mode, which detaches the interface from the host networking stack. Applications are able to also communicate with the host TCP/IP stack via two special netmap rings that map to the host stack. Netmap does not require specific network hardware and is fully integrated to FreeBSD and Linux kernels with minimal modifications. Netmap reduces per-packet processing overhead significantly. A Netmap-based application that forwards packets between NICs is able send and receive at peak packet rate of 14.88 Mpps on a 10 Gigabit NIC running on a single CPU core that is 20 times higher than what POSIX APIs are able to achieve. For pktgen, a in-kernel Linux packet generator, the per packet cost is reduced from 250 ns needed by the skbuf/mbuf-based API to 20 ns to 30 ns when Netmap is used. Netmap achieves this improvement eliminating per-packet dynamic memory allocation, reducing system call overhead with batching, and eliminating copying by sharing buffers and metadata between kernel and userspace. Dynamic memory allocation is eliminated by preallocating linear fixed-size packet buffers when a device is moved to netmap mode. System call overhead is reduced by issuing netmap system calls for large batches of packets. Data copying is eliminated by allowing applications to directly access packet buffers via memory mapping.

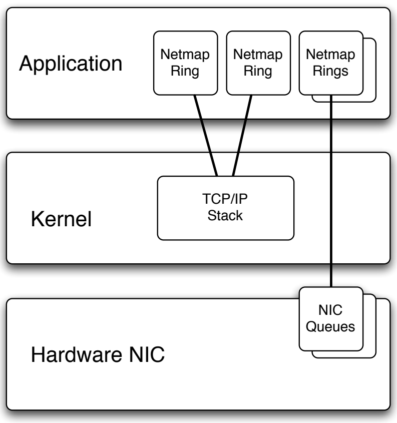


Figure 16: **Netmap architecture.** *Hardware NIC queues are mapped directly to application address space as netmap rings. Netmap exposes two additional rings to communicate with the host operating system TCP/IP stack.*

8 Conclusions

Containers have become a viable light-weight alternative to hypervisors for virtualizing applications over the past years, especially with the rise of Docker. As we can see from our experimental evaluation results, Docker has the least overhead of all virtualization techniques when configured to use bridged networking. Hypervisor I/O virtualization techniques have also improved over the years and the performance gap to bare metal and Docker is small on KVM/QEMU with vhost-net. Unikernels like OSv are able to reduce the performance gap even further making OSv running on KVM/QEMU the second fastest virtualization technique after Docker. However, as we can see from our OSv experimental evaluation results, unikernels are not mature enough and have more performance overheads for some TCP workloads. Deciding upon a virtualization technique depends on the application and deployment requirements. If the inflexibility of running a single shared kernel version is not an issue, containers are the fastest virtualization technique for network intensive applications. For hypervisor-based virtualization, the fastest technique for network intensive applications are unikernels.

Future work We would like to repeat the experiments on a 10 Gigabit NIC to measure how virtualization technique overheads change when the NIC performance improves. We also want to include a SR-IOV capable NIC as part of the test scenarios to see how hardware virtualization techniques compare to software virtualization techniques. It would also be interesting to compare virtualization overheads on current generation of ARM CPUs that support virtualization to see how machine architecture affect the overheads. We do not expect big differences in overheads for the Xen hypervisor compared to KVM/QEMU but it would be interesting to run the experiments also on Xen for completeness. To make reproducing our experiments easier, we are making the test scripts available in public at `https://github.com/penberg/virt-net-perf`.

References

[1] *VMware ESXi.* `www.vmware.com/products/esxi-and-esx/`, 2002 (accessed May 13, 2016).

[2] *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool.* `iperf.fr`, 2003 (accessed May 13, 2016).

[3] *PCI-SIG: Specifications.* `pcisig.com/specifications`, 2004 (accessed May 13, 2016).

[4] *QEMU.* `www.qemu.org`, 2005 (accessed May 13, 2016).

[5] *OpenZFS.* `open-zfs.org/wiki/Main_Page`, 2005 (accessed May 17, 2016).

[6] *The "nuttcp" Network Performance Benchmark.* `www.nuttcp.net`, 2005 (accessed May 17, 2016).

[7] *memcached - a distributed memory object caching system.* `memcached.org`, 2005 (accessed October 7, 2015).

[8] *Amazon Web Services (AWS) - Cloud Computing Services.* `aws.amazon.com`, 2006 (accessed May 13, 2016).

[9] *memaslap: Load testing and benchmarking a server.* `docs.libmemcached.org/bin/memaslap.html`, 2007 (accessed May 13, 2016).

[10] *VirtualBox.* `www.virtualbox.org`, 2007 (accessed May 13, 2016).

[11] *OpenJDK.* `openjdk.java.net`, 2007 (accessed May 17, 2016).

[12] *Microsoft Azure: Cloud Computing Platform & Services.* `azure.microsoft.com`, 2010 (accessed May 13, 2016).

[13] *OpenStack Open Source Cloud Computing Software.* `www.openstack.org`, 2010 (accessed May 17, 2016).

[14] *Compute Engine - IaaS - Google Cloud Platform.* `cloud.google.com/compute/`, 2012 (accessed May 13, 2016).

[15] *Mutilate: high-performance memcached load generator.* `github.com/leverich/mutilate`, 2012 (accessed September 25, 2015).

[16] *Docker - Build, Ship, and Run Any App, Anywhere.* `www.docker.com`, 2013 (accessed May 17, 2016).

[17] *etcd.* `coreos.com/etcd/`, 2013 (accessed May 17, 2016).

[18] *Kubernetes - Accelerate Your Delivery.* kubernetes.io, 2014 (accessed May 17, 2016).

[19] *Docker: Understand the architecture.* docs.docker.com/engine/understanding-docker/, 2016 (accessed May 13, 2016).

[20] Adams, Keith and Agesen, Ole: *A comparison of software and hardware techniques for x86 virtualization.* In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII.* Association for Computing Machinery (ACM), 2006. http://dx.doi.org/10.1145/1168857.1168860.

[21] Agesen, Ole, Mattson, Jim, Rugina, Radu, and Sheldon, Jeffrey: *Software techniques for avoiding hardware virtualization exits.* In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 373–385, Boston, MA, 2012. USENIX, ISBN 978-931971-93-5. https://www.usenix.org/conference/atc12/technical-sessions/presentation/agesen.

[22] Arcangeli, Andrea, Eidus, Izik, and Wright, Chris: *Increasing memory density by using ksm.* In *Proceedings of the 2009 Ottawa Linux Symposium*, pages 19–28. OLS, 2009.

[23] Atikoglu, Berk, Xu, Yuehai, Frachtenberg, Eitan, Jiang, Song, and Paleczny, Mike: *Workload analysis of a large-scale key-value store.* Volume 40, page 53. Association for Computing Machinery (ACM), jun 2012. http://dx.doi.org/10.1145/2318857.2254766.

[24] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew: *Xen and the art of virtualization.* 2003. http://dx.doi.org/10.1145/945445.945462.

[25] Barker, Sean, Wood, Timothy, Shenoy, Prashant, and Sitaraman, Ramesh: *An empirical study of memory sharing in virtual machines.* In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 273–284, Boston, MA, 2012. USENIX, ISBN 978-931971-93-5. https://www.usenix.org/conference/atc12/technical-sessions/presentation/barker.

[26] Bellard, Fabrice: *Qemu, a fast and portable dynamic translator.* In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1247360.1247401.

[27] Ben-Yehuda, Muli, Day, Michael D., Dubitzky, Zvi, Factor, Michael, Har'El, Nadav, Gordon, Abel, Liguori, Anthony, Wasserman, Orit, and

Yassour, Ben Ami: *The turtles project: Design and implementation of nested virtualization.* In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 423–436, Berkeley, CA, USA, 2010. USENIX Association. `http://dl.acm.org/citation.cfm?id=1924943.1924973`.

[28] Biederman, Eric W and Networx, Linux: *Multiple instances of the global linux namespaces.* In *Proceedings of the 2006 Ottawa Linux Symposium*, volume 1, pages 101–112. OLS, 2006.

[29] Boutcher, David and Chandra, Abhishek: *Does virtualization make disk scheduling passé?* ACM SIGOPS Operating Systems Review, 44(1):20, mar 2010. `http://dx.doi.org/10.1145/1740390.1740396`.

[30] Canet, Benoît and Marti, Don: *NFS on OSv or "How I Learned to Stop Worrying About Memory Allocations and Love the Unikernel".* `osv.io/blog/blog/2016/04/21/nfs-on-osv/`, 2016 (accessed May 17, 2016).

[31] Cantrill, Bryan: *KVM on illumos.* `http://dtrace.org/blogs/bmc/2011/08/15/kvm-on-illumos/`, 2011 (accessed May 13, 2016).

[32] Cheng, Luwei and Wang, Cho Li: *vBalance.* In *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC 12.* Association for Computing Machinery (ACM), 2012. `http://dx.doi.org/10.1145/2391229.2391231`.

[33] Dong, Yaozu, Dai, Jinquan, Huang, Zhiteng, Guan, Haibing, Tian, Kevin, and Jiang, Yunhong: *Towards high-quality I/O virtualization.* In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference on - SYSTOR 09.* Association for Computing Machinery (ACM), 2009. `http://dx.doi.org/10.1145/1534530.1534547`.

[34] Dong, Yaozu, Li, Shaofan, Mallick, Asit, Nakajima, Jun, Tian, Kun, Xu, Xuefei, Yang, Fred, and Yu, Wilfred: *Extending xen with intel virtualization technology.* Intel Technology Journal, 10(3), 2006.

[35] Dong, Yaozu, Yang, Xiaowei, Li, Xiaoyong, Li, Jianhui, Tian, Kun, and Guan, Haibing: *High performance network virtualization with SR-IOV.* In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture.* Institute of Electrical & Electronics Engineers (IEEE), jan 2010. `http://dx.doi.org/10.1109/HPCA.2010.5416637`.

[36] Engler, D. R., Kaashoek, M. F., and OToole, J.: *Exokernel: An operating system architecture for application-level resource management.* In *Proceedings of the fifteenth ACM symposium on Operating systems*

principles - SOSP 95. Association for Computing Machinery (ACM), 1995. `http://dx.doi.org/10.1145/224056.224076`.

[37] Felter, Wes, Ferreira, Alexandre, Rajamony, Ram, and Rubio, Juan: *An updated performance comparison of virtual machines and linux containers*. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Institute of Electrical & Electronics Engineers (IEEE), mar 2015. `http://dx.doi.org/10.1109/ISPASS.2015.7095802`.

[38] Goldberg, Robert P: *Architectural principles for virtual computer systems*. PhD thesis, Harvard University, 1973.

[39] Gupta, Diwaker, Lee, Sangmin, Vrable, Michael, Savage, Stefan, Snoeren, Alex C., Varghese, George, Voelker, Geoffrey M., and Vahdat, Amin: *Difference engine: Harnessing memory redundancy in virtual machines*. Communications of the ACM, 53(10):85, oct 2010. `http://dx.doi.org/10.1145/1831407.1831429`.

[40] Hajnoczi, Stefan: *QEMU Internals: vhost architecture*. `http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html`, 2011 (accessed May 13, 2016).

[41] Har'El, Nadav, Gordon, Abel, Landau, Alex, Ben-Yehuda, Muli, Traeger, Avishay, and Ladelsky, Razya: *Efficient and scalable paravirtual i/o system*. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 231–242, San Jose, CA, 2013. USENIX, ISBN 978-1-931971-01-0. `https://www.usenix.org/conference/atc13/technical-sessions/presentation/har'el`.

[42] Hwang, Jinho, Zeng, Sai, Wu, Frederick, and Wood, Timothy: *A component-based performance comparison of four hypervisors*. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013*, pages 269–276, 2013. `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6572995`.

[43] Jones, Rick: *Care and Feeding of Netperf 2.6.X*. `www.netperf.org/svn/netperf2/tags/netperf-2.6.0/doc/netperf.html`, 2012 (accessed July 30, 2015).

[44] Kim, Jinchun, Fedorov, Viacheslav, Gratz, Paul V., and Reddy, A. L. Narasimha: *Dynamic memory pressure aware ballooning*. In *Proceedings of the 2015 International Symposium on Memory Systems - MEMSYS 15*. Association for Computing Machinery (ACM), 2015. `http://dx.doi.org/10.1145/2818950.2818967`.

[45] Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, and Liguori, Anthony: *kvm: the linux virtual machine monitor.* In *Proceedings of the 2005 Ottawa Linux Symposium*, volume 1, pages 225–230, 2007.

[46] Kivity, Avi, Laor, Dor, Costa, Glauber, Enberg, Pekka, Har'El, Nadav, Marti, Don, and Zolotarov, Vlad: *Osv—optimizing the operating system for virtual machines.* In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association, ISBN 978-1-931971-10-2. `https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity`.

[47] Liu, Jiuxing: *Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support.* In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).* Institute of Electrical & Electronics Engineers (IEEE), 2010. `http://dx.doi.org/10.1109/IPDPS.2010.5470365`.

[48] Madhavapeddy, Anil, Mortier, Richard, Rotsos, Charalampos, Scott, David, Singh, Balraj, Gazagnaire, Thomas, Smith, Steven, Hand, Steven, and Crowcroft, Jon: *Unikernels: Library operating systems for the cloud.* ACM SIGPLAN Notices, 48(4):461, apr 2013. `http://dx.doi.org/10.1145/2499368.2451167`.

[49] Martins, Joao, Ahmed, Mohamed, Raiciu, Costin, Olteanu, Vladimir, Honda, Michio, Bifulco, Roberto, and Huici, Felipe: *Clickos and the art of network function virtualization.* In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association, ISBN 978-1-931971-09-6.

[50] Morabito, Roberto, Kjällman, Jimmy, and Komu, Miika: *Hypervisors vs. lightweight virtualization: A performance comparison.* In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, IC2E '15, pages 386–393, Washington, DC, USA, 2015. IEEE Computer Society, ISBN 978-1-4799-8218-9. `http://dx.doi.org/10.1109/IC2E.2015.74`.

[51] Peter, Simon, Li, Jialin, Zhang, Irene, Ports, Dan R. K., Woos, Doug, Krishnamurthy, Arvind, Anderson, Thomas, and Roscoe, Timothy: *Arrakis: The operating system is the control plane.* In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association, ISBN 978-1-931971-16-4. `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter`.

[52] Popek, Gerald J. and Goldberg, Robert P.: *Formal requirements for virtualizable third generation architectures.* Communications of the

ACM, 17(7):412–421, jul 1974. http://dx.doi.org/10.1145/361011.361073.

[53] Porter, Donald E., Boyd-Wickizer, Silas, Howell, Jon, Olinsky, Reuben, and Hunt, Galen C.: *Rethinking the library OS from the top down.* ACM SIGARCH Computer Architecture News, 39(1):291, mar 2011. http://dx.doi.org/10.1145/1961295.1950399.

[54] Pratt, Ian, Fraser, Keir, Hand, Steven, Limpach, Christian, Warfield, Andrew, Magenheimer, Dan, Nakajima, Jun, and Mallick, Asit: *Xen 3.0 and the art of virtualization.* In *Proceedings of the 2005 Ottawa Linux Symposium*, volume 2, pages 73–86. OLS, 2005.

[55] Rao, Jia and Zhou, Xiaobo: *Towards fair and efficient SMP virtual machine scheduling.* In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP 14.* Association for Computing Machinery (ACM), 2014. http://dx.doi.org/10.1145/2555243.2555246.

[56] Rizzo, Luigi: *netmap: A novel framework for fast packet i/o.* In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association, ISBN 978-931971-93-5. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[57] Rizzo, Luigi, Lettieri, Giuseppe, and Maffione, Vincenzo: *Speeding up packet i/o in virtual machines.* In *Architectures for Networking and Communications Systems.* Institute of Electrical & Electronics Engineers (IEEE), oct 2013. http://dx.doi.org/10.1109/ANCS.2013.6665175.

[58] Russell, Rusty: *Virtio: Towards a de-facto standard for virtual I/O devices.* ACM SIGOPS Operating Systems Review, 42(5):95–103, jul 2008. http://dx.doi.org/10.1145/1400097.1400108.

[59] Sandberg, Russel, Goldberg, David, Kleiman, Steve, Walsh, Dan, and Lyon, Bob: *Design and implementation of the sun network filesystem.* In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[60] Soltesz, Stephen, Pötzl, Herbert, Fiuczynski, Marc E., Bavier, Andy, and Peterson, Larry: *Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors.* Volume 41, page 275. Association for Computing Machinery (ACM), jun 2007. http://dx.doi.org/10.1145/1272998.1273025.

[61] Song, Xiang, Shi, Jicheng, Chen, Haibo, and Zang, Binyu: *Schedule processes, not VCPUs.* In *Proceedings of the 4th Asia-Pacific Workshop on Systems - APSys 13.* Association for Computing Machinery (ACM), 2013. http://dx.doi.org/10.1145/2500727.2500736.

[62] Tu, Cheng Chun, Ferdman, Michael, Lee, Chao tung, and Chiueh, Tzi cker: *A comprehensive implementation and evaluation of direct interrupt delivery.* In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE 15.* Association for Computing Machinery (ACM), 2015. `http://dx.doi.org/10.1145/2731186.2731189`.

[63] Uhlig, Volkmar, LeVasseur, Joshua, Skoglund, Espen, and Dannowski, Uwe: *Towards scalable multiprocessor virtual machines.* In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association. `http://dl.acm.org/citation.cfm?id=1267242.1267246`.

[64] Waldspurger, Carl A.: *Memory resource management in VMware ESX server.* ACM SIGOPS Operating Systems Review, 36(SI):181, dec 2002. `http://dx.doi.org/10.1145/844128.844146`.

[65] Wang, Guohui and Ng, T. S. Eugene: *The impact of virtualization on network performance of amazon EC2 data center.* In *2010 Proceedings IEEE INFOCOM.* Institute of Electrical & Electronics Engineers (IEEE), mar 2010. `http://dx.doi.org/10.1109/INFCOM.2010.5461931`.

[66] Whiteaker, Jon, Schneider, Fabian, and Teixeira, Renata: *Explaining packet delays under virtualization.* ACM SIGCOMM Computer Communication Review, 41(1):38, jan 2011. `http://dx.doi.org/10.1145/1925861.1925867`.