NetKernel: Making Network Stack Part of the Virtualized Infrastructure

Zhixiong Niu
City University of Hong Kong

Hong Xu
City University of Hong Kong

Peng Cheng
Microsoft Research

Yongqiang Xiong
Microsoft Research

Tao Wang
City University of Hong Kong

Dongsu Han
KAIST

Keith Winstein
Stanford University

ABSTRACT

The network stack is implemented inside virtual machines (VMs) in today's cloud. This paper presents a system called NetKernel that decouples the network stack from the guest, and offers it as an independent module implemented by the cloud operator. NetKernel represents a new paradigm where network stack is managed by the operator as part of the virtualized infrastructure. It provides important efficiency benefits: By gaining control and visibility of the network stack, operator can perform network management more directly and flexibly, such as multiplexing VMs running different applications to the same network stack module to save CPU cores, and enforcing fair bandwidth sharing with distributed congestion control. Users also benefit from the simplified stack deployment and better performance. For example mTCP can be deployed without API change to support nginx and redis natively, and shared memory networking can be readily enabled to improve performance of colocating VMs. Testbed evaluation using 100G NICs shows that NetKernel preserves the performance and scalability of both kernel and userspace network stacks, and provides the same isolation as the current architecture.

1 INTRODUCTION

Virtual machine (VM) is the predominant virtualization form in today's cloud due to its strong isolation guarantees. VMs allow customers to run applications in a wide variety of operating systems (OSes) and configurations. VMs are also heavily used by cloud operators to deploy internal services, such as load balancing, proxy, VPN, etc., both in a public cloud for tenants and in a private cloud for supporting various business units of an organization. Lightweight virtualization technologies such as containers are also provisioned inside VMs in many production settings for isolation, security, and management reasons [2, 3, 6].

VM based virtualization largely follows traditional OS design. In particular, the TCP/IP network stack is encapsulated inside the VM as part of the guest OS as shown in Figure 1(a). Applications own the network stack, which is separated from the network infrastructure that operators own; they interface using the virtual NIC abstraction. This architecture preserves the familiar hardware and OS abstractions so a vast array of workloads can be easily moved into the cloud. It provides high flexibility to applications to customize the entire network stack.

We argue that the current division of labor between application and network infrastructure is becoming increasingly inadequate. The central issue is that the operator has almost zero visibility and control over the network stack. This leads to many efficiency problems that manifest in various aspects of running the cloud network.

Many network management tasks like monitoring, diagnosis, and troubleshooting have to be done in an extra layer outside the VMs, which requires significant effort in design and implementation [23, 54, 55]. Since these network functions need to process packets at the end-host [29, 37, 45, 61], they can be done more efficiently if the network stack were opened up to the operator. More importantly, the operator is unable to orchestrate resource allocation at the end-points of the network fabric, resulting in low resource utilization. It remains difficult today for the operator to meet or define performance SLAs despite much prior work [17, 28, 34, 39, 52, 53], as she cannot precisely provision resources just for the network stack or control how the stack consumes these resources. Further, resources (e.g. CPU) have to be provisioned on a per-VM basis based on the peak traffic; it is impossible to coordinate across VM boundaries. This degrades the overall utilization of the network stack since in practice traffic to individual VMs is extremely bursty.

Even the simple task of maintaining or deploying a network stack suffers from inefficiency today. Network stack has critical impact on performance, and many optimizations have been studied with numerous effective solutions, ranging from congestion control [13, 19, 47], scalability [33, 40], zerocopy datapath [5, 33, 51, 59, 60], NIC multiqueue scheduling [57], etc. Yet the operator, with sufficient expertise

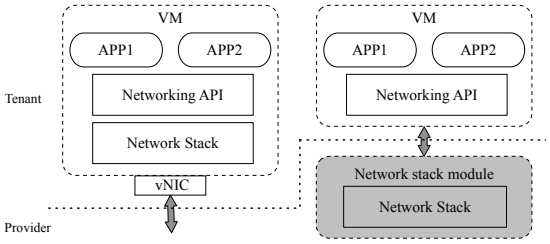

Figure 1: Decoupling network stack from the guest, and making it part of the virtualized infrastructure.

and resources, still cannot deploy these extensions to improve performance and reduce overheads. As a result, our community is still finding ways to deploy DCTCP in the public cloud [20, 31]. On the other hand, applications without much knowledge of the underlying network or expertise on networking are forced to juggle the deployment and maintenance details. For example if one wants to deploy a new stack like mTCP [33], a host of problems arise such as setting up kernel bypass, testing with kernel versions and NIC drivers, and porting applications to the new APIs. Given the intricacy of implementation and the velocity of development, it is a daunting task if not impossible to expect users, whether tenants in a public cloud or first-party services in a private cloud, to individually maintain the network stack themselves.

We thus advocate a new division of labor in a VM-based cloud in this paper. We believe that network stack should be managed as part of the virtualized infrastructure instead of in the VM by application. The operator is naturally in a better position to own the last mile of packet delivery, so it can directly deploy, manage, and optimize the network stack, and comprehensively improve the efficiency of running the entire network fabric. Applications' functionality and performance requirements can be consolidated and satisfied with several different network stacks provided by the operator. As the heavy-lifting is taken care of, applications can just use network stack as a basic service of the infrastructure and focus on their business logic.

Specifically, we propose to decouple the VM network stack from the guest as shown in Figure 1(b). We keep the network APIs such as BSD sockets intact, and use them as the abstraction boundary between application and infrastructure. Each VM is served by a network stack module (NSM) that runs the network stack chosen by the user. Application data are handled outside the VM in the NSM, whose design and implementation are managed by the operator. Various network stacks can be provided as different NSMs to ensure applications with diverse requirements can be properly satisfied. We do not enforce a single transport design, or trade off flexibility of the existing architecture in our approach.

We make three specific contributions.

- We design and implement a system called NetKernel to show that this new division of labor is feasible without radical changes to application or infrastructure (§3–§5). NetKernel provides transparent BSD socket redirection so existing applications can run directly. The socket semantics from the application are encapsulated into small queue elements and transmitted to the corresponding NSM via lockless shared memory queues.
- We present new use cases that are difficult to realize today to show NetKernel's potential benefits (§6). For example, we show that NetKernel enables multiplexing: one NSM can serve multiple VMs at the same time and save over 40% CPU cores without degrading performance using traces from a production cloud.
- We conduct comprehensive testbed evaluation with commodity 100G NICs to show that NetKernel achieves the same performance, scalability, and isolation as the current architecture (§7). For example, the kernel stack NSM achieves 100G send throughput with 3 cores; the mTCP NSM achieves 1.1M RPS with 8 cores.

2 MOTIVATION

Decoupling the network stack from the guest OS, hence making it part of the infrastructure, marks a clear departure from the way networking is provided to VMs nowadays. In this section we elaborate why this is a better architectural design by presenting its benefits and contrasting with alternative solutions. We discuss its potential issues in §8.

2.1 Benefits

We highlight key benefits of our vision with new use cases we experimentally realize with NetKernel in §6.

Better efficiency in management for the operator. Gaining control over the network stack, the operator can now perform network management more efficiently. For example it can orchestrate the resource provisioning strategies much more flexibly: For mission-critical workloads, it can dedicate CPU resources to their NSMs to offer performance SLAs in terms of throughput and RPS (requests per second) guarantees. For elastic workloads, on the other hand, it can consolidate their VMs to the same NSM (if they use the same network stack) to improve its resource utilization. The operator can also directly implement management functions as an integral part of user's network stack and improve the effectiveness of management, compared to doing them in an extra layer outside the guests.

Use case 1: Multiplexing (§6.1). Utilization of network stack in VMs is very low most of the time in practice. Using a real trace from a large cloud, we show that NetKernel enables

multiple VMs to be multiplexed onto one NSM to serve the aggregated traffic and saves over 40% CPU cores for the operator without performance degradation.

Use case 2: Fair bandwidth sharing (§6.2). TCP's notion of flow-level fairness leads to poor bandwidth sharing in data centers [55]. We show that NetKernel allows us to readily implement VM-level congestion control [55] as an NSM to achieve fair sharing regardless of number of flows and destinations.

Deployment and performance gains for users. Making network stack part of the virtualized infrastructure is also beneficial for users in both public and private clouds. Operator can directly optimize the network stack design and implementation. Various kernel stack optimizations [40, 59], high-performance userspace stacks [1, 18, 33, 51], and even designs using advanced hardware [7, 9, 10, 41] can now be deployed and maintained transparently without user involvement or application code change. Since the BSD socket is the only abstraction exposed to the applications, it is now feasible to adopt new stack designs independent of the guest kernel or the network API. Our vision also opens up new design space by allowing the network stack to exploit the visibility into the infrastructure for performance benefits.

Use case 3: Deploying mTCP without API change (§6.3). We show that NetKernel enables unmodified applications in the VM to use mTCP [33] in the NSM, and improves performance greatly due to mTCP's kernel bypass design. mTCP is a userspace stack with new APIs (including modified `epoll`/`kqueue`). During the process, we also find and fix a compatibility issue between mTCP and our NIC driver, and save significant maintenance time and effort for users.

Use case 4: Shared memory networking (§6.4). When two VMs of the same user are colocated on the same host, NetKernel can directly detect this and copy their data via shared memory to bypass TCP stack processing and improve throughput. This is difficult to achieve today as VMs have no knowledge about the underlying infrastructure [38, 62].

And beyond. We focus on efficiency benefits in this paper since they seem most immediate. Making network stack part of the virtualized infrastructure also brings additional benefits that are more far-reaching. For example, it facilitates innovation by allowing new protocols in different layers of the stack to be rapidly prototyped and experimented. It provides a direct path for enforcing centralized control, so network functions like failure detection [29] and monitoring [37, 45] can be integrated into the network stack implementation. It opens up new design space to more freely exploit endpoint coordination [25, 50], software-hardware co-design, and programmable data planes [15, 41]. We encourage the community to fully explore these opportunities in the future.

2.2 Alternative Solutions

We now discuss several alternative solutions and why they are inadequate.

Why not just use containers? Containers are gaining popularity as a lightweight and portable alternative to VMs [4]. A container is essentially a process with namespace isolation. Using containers can address some efficiency problems because the network stack is in the hypervisor instead of in the containers. Without the guest OS, however, containers have poor isolation [36] and are difficult to manage. Moreover, containers are constrained to using the host network stack, whereas NetKernel provides choices for applications on the same host. This is important as data center applications have diverse requirements that cannot be satisfied with a single design.

In a word, containers or other lightweight virtualization represent a more radical approach of removing the guest kernel, which leads to several practical issues. Thus they are commonly deployed inside VMs in production settings. In fact we find that all major public clouds [2, 3, 6] require users to launch containers inside VMs. Thus, our discussion is centered around VMs that cover the vast majority of usage scenarios in a cloud. NetKernel readily benefits containers running inside VMs as well.

Why not on the hypervisor? Another possible approach is to keep the VM intact, and add the network stack implementation outside on the hypervisor. Some existing work takes this approach to realize a uniform congestion control without changing VMs [20, 31]. This does allow the operator to gain control on network stack. Yet the performance and efficiency of this approach is even lower than the current architecture because data are then processed twice in two independent stacks, first by the VM network stack and then the stack outside.

Why not use customized OS images? Operators can build customized OS images with the required network stacks for users, which remedies the maintenance and deployment issues. Yet this approach has many downsides. It is not transparent: customers need to update these images on their own, and deploying images causes downtime and disrupts applications. But more importantly, since even just a single user will have vastly different workloads that require different environments (Linux or FreeBSD or Windows, kernel versions, driver versions, etc.), the cost of testing and maintenance for all these possibilities is prohibitive.

In contrast, NetKernel does not have these issues because it breaks the coupling of the network stack to the guest OS. Architecturally a network stack module can be used by VMs with different guest OSes since BSD socket APIs are widely supported, thereby greatly reducing development resources required for operators. Maintenance is also transparent and

non-disruptive to customers as operators can roll out updates in the background.

3 DESIGN PHILOSOPHY

NetKernel imposes three fundamental design questions around the separation of network stack and the guest OS:

(1) How to transparently redirect socket API calls without changing applications?
(2) How to transmit the socket semantics between the VM and NSM whose implementation of the stack may vary?
(3) How to ensure high performance with semantics transmission (e.g., 100 Gbps)?

These questions touch upon largely uncharted territory in the design space. Thus our main objective in this paper is to demonstrate feasibility of our approach on existing virtualization platforms and showcase its potential. Performance and overhead are not our primary goals. It is also not our goal to improve any particular network stack design.

In answering the questions above, NetKernel's design has the following highlights.

Transparent Socket API Redirection. NetKernel needs to redirect BSD socket calls to the NSM instead of the tenant network stack. This is done by inserting into the guest a library called GuestLib. The GuestLib provides a new socket type called NetKernel socket with a complete implementation of BSD socket APIs. It replaces all TCP and UDP sockets when they are created with NetKernel sockets, effectively redirecting them without changing applications.

A Lightweight Semantics Channel. Different network stacks may run as different NSMs, so NetKernel needs to ensure socket semantics from the VM work properly with the actual NSM stack implementation. For this purpose NetKernel builds a lightweight socket semantics channel between VM and its NSM. The channel relies on small fix-sized queue elements as intermediate representations of socket semantics: each socket API call in the VM is encapsulated into a queue element and sent to the NSM, who would effectively translate the queue element into the corresponding API call of its network stack.

Scalable Lockless Queues. As NIC speed in cloud evolves from 40G/50G to 100G [24] and higher, the NSM has to use multiple cores for the network stack to achieve line rate. NetKernel thus adopts scalable lockless queues to ensure VM-NSM socket semantics transmission is not a bottleneck. Each core services a dedicated set of queues so performance is scalable with number of cores. More importantly, each queue is memory shared with a software switch, so it can be lockless with only a single producer and a single consumer to avoid expensive lock contention [32, 33, 40].

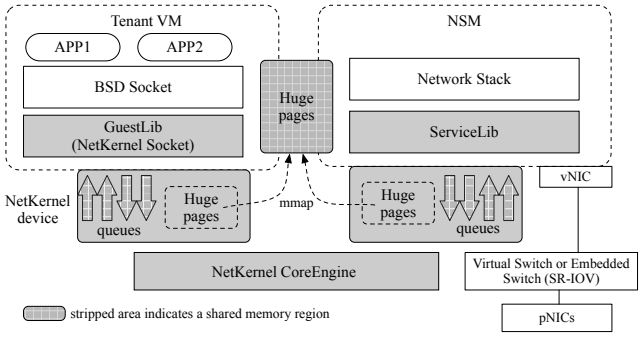

Figure 2: NetKernel design overview.

Switching the queue elements offers important benefits beyond lockless queues. It facilitates a flexible mapping between VM and NSM: a NSM can support multiple VMs without adding more queues compared to binding the queues directly between VM and NSM. In addition, it allows dynamic resource management: cores can be readily added to or removed from a NSM, and a user can switch her NSM on the fly. The CPU overhead of software switching can be addressed by hardware offloading [24, 27], which we discuss in §7.8 in more detail.

VM Based NSM. Lastly we discuss an important design choice regarding the NSM. The NSM can take various forms. It may be a full-fledged VM with a monolithic kernel. Or it can be a container or module running on the hypervisor, which is appealing because it consumes less resource and offers better performance. Yet it entails porting a complete TCP/IP stack to the hypervisor. Achieving memory isolation among containers or modules are also difficult [48]. More importantly, it introduces another coupling between the network stack and the hypervisor, which defeats the purpose of NetKernel. Thus we choose to use a VM for NSM. VM based NSM readily supports existing kernel and userspace stacks from various OSes. VMs also provide good isolation and we can dedicate resources to a NSM to guarantee performance. VM based NSM is the most flexible: we can run stacks independent of the hypervisor.

4 DESIGN

Figure 2 depicts NetKernel's architecture. The BSD socket APIs are transparently redirected to a complete NetKernel socket implementation in GuestLib in the guest kernel (§4.1). The GuestLib can be deployed as a kernel patch and is the only change we make to the user VM. Network stacks are implemented by the provider on the same host as Network Stack Modules (NSMs), which are individual VMs in our current design. Inside the NSM, a ServiceLib interfaces with the network stack. The NSM connects to the vSwitch, be it a software or a hardware switch, and then the pNICs. Thus our design also supports SR-IOV.

All socket operations and their results are translated into NetKernel Queue Elements (NQEs) by GuestLib and ServiceLib (§4.2). For NQE transmission, GuestLib and ServiceLib each has a NetKernel device, or NK device in the following, consisting of one or more sets of lockless queues. Each queue set has a *send queue* and *receive queue* for operations with data transfer (e.g. `send()`), and a *job queue* and *completion queue* for control operations without data transfer (e.g. `setsockopt()`). Each NK device connects to a software switch called CoreEngine, which runs on the hypervisor and performs actual NQE switching (§4.3). The CoreEngine is also responsible for various management tasks such as setting up the NK devices, ensuring isolation among VMs, etc. (§4.4) A unique set of hugepages are shared between each VM-NSM tuple for application data exchange. A NK device also maintains a hugepage region that is memory mapped to the corresponding application hugepages as shown in Figure 2 (§4.5).

For ease of presentation, we assume both the user VM and NSM run Linux, and the NSM uses the kernel stack.

4.1 Transparent Socket API Redirection

We first describe how NetKernel's GuestLib interacts with applications to support BSD socket semantics transparently.

Kernel Space API Redirection. There are essentially two approaches to redirect BSD socket calls to NSM, each with its unique tradeoffs. One is to implement it in userspace using `LD_PRELOAD` for example. The advantages are: (1) It is efficient without syscall overheads and performance is high [33]; (2) It is easy to deploy without kernel modification. However, this implies each application needs to have its own redirection service, which limits the usage scenarios. Another way is kernel space redirection, which naturally supports multiple applications without IPC. The flip side is that performance may be lower due to context switching and syscall overheads.

We opt for kernel space API redirection to support most of the usage scenarios, and leave userspace redirection as future work. GuestLib is a kernel module deployed in the guest. This is feasible by distributing images of para-virtualizated guest kernels to users, a practice providers are already doing nowadays. Kernel space redirection also allows NetKernel to work directly with I/O event notification syscalls like epoll.

NetKernel Socket API. GuestLib creates a new type of sockets—`SOCK_NETKERNEL`, in addition to TCP (`SOCK_STREAM`) and UDP (`SOCK_DGRAM`) sockets. It registers a complete implementation of BSD socket APIs as shown in Table 1 to the guest kernel. When the guest kernel receives a `socket()` call to create a new TCP socket say, it replaces the socket type with `SOCK_NETKERNEL`, creates a new NetKernel socket, and initializes the socket data structure with function pointers to NetKernel socket implementation in GuestLib. The `sendmsg()` for example now points to `nk_sendmsg()` in GuestLib instead of `tcp_sendmsg()`.

Table 1: NetKernel socket implementation.

	inet_stream_ops	netkernel_pro
bind	`inet_bind()`	`nk_bind()`
connect	`inet_connect()`	`nk_connect()`
accept	`inet_accept()`	`nk_accept()`
poll	`tcp_poll()`	`nk_poll()`
ioctl	`inet_ioctl()`	`nk_ioctl()`
listen	`inet_listen()`	`nk_listen()`
shutdown	`inet_shutdown()`	`nk_shutdown()`
setsockopt	`sock_common_setsockopt()`	`nk_setsockopt()`
recvmsg	`tcp_recvmsg()`	`nk_recvmsg()`
sendmsg	`tcp_sendmsg()`	`nk_sendmsg()`

4.2 A Lightweight Socket Semantics Channel

Socket semantics are contained in NQEs and carried around between GuestLib and ServiceLib via their respective NK devices.

1B	1B	1B	4B	8B	8B	4B	5B
op type	VM ID	Queue set ID	VM socket ID	op_data	data pointer	size	rsved

Figure 3: Structure of a NQE. Here socket ID denotes a pointer to the `sock` struct in the user VM or NSM, and is used for NQE transmission with VM ID and queue set ID in §4.3; `op_data` contains data necessary for socket operations, such as ip address for bind; `data pointer` is a pointer to application data in hugepages; and `size` is the size of pointed data in hugepages.

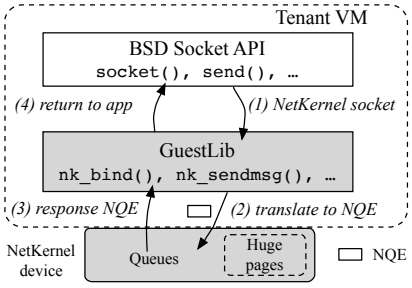

Figure 4: NetKernel socket implementation in GuestLib redirects socket API calls. GuestLib translates socket API calls to NQEs and ServiceLib translates results into NQEs as well (not shown here).

NQE and Socket Semantics Translation. Figure 3 shows the structure of a NQE with a fixed size of 32 bytes. Translation happens at both ends of the semantics channel: GuestLib

encapsulates the socket semantics into NQEs and sends to ServiceLib, which then invokes the corresponding API of its network stack to execute the operation; the execution result is again turned into a NQE in ServiceLib first, and then translated by GuestLib back into the corresponding response of socket APIs.

For example in Figure 4, to handle the `socket()` call in the VM, GuestLib creates a new NQE with the operation type and information such as its VM ID for NQE transmission. The NQE is transmitted by GuestLib's NK device. The `socket()` call now blocks until a response NQE is received. After receiving the NQE, ServiceLib parses the NQE from its NK device, invokes the `socket()` of the kernel stack to create a new TCP socket, prepares a new NQE with the execution result, and enqueues it to the NK device. GuestLib then receives and parses the response NQE and wakes up the `socket()` call. The `socket()` call now returns to application with the NetKernel socket file descriptor (fd) if a TCP socket is created at the NSM, or with an error number consistent with the execution result of the NSM.

We defer the handling of application data to §4.5.

Queues for NQE Transmission. NQEs are transmitted via one or more sets of queues in the NK devices. A queue set has four independent queues: a *job queue* for NQEs representing socket operations issued by the VM without data transfer, a *completion queue* for NQEs with execution results of control operations from the NSM, a *send queue* for NQEs representing operations issued by VM with data transfer; and a *receive queue* for NQEs representing events of newly received data from NSM. Queues of different NK devices have strict correspondence: the NQE for `socket()` for example is put in the job queue of GuestLib's NK device, and sent to the job queue of ServiceLib's NK device.

We now present the working of I/O event notification mechanisms like epoll with the receive queue. Suppose an application issues `epoll_wait()` to monitor some sockets. Since all sockets are now NetKernel sockets, the `nk_poll()` is invoked by `epoll_wait()` and checks the receive queue to see if there is any NQE for this socket. If yes, this means there are new data received, `epoll_wait()` then returns and the application issues a `recv()` call with the NetKernel socket fd of the event. This points to `nk_recvmsg()` which parses the NQE from receive queue for the data pointer, copies data from the hugepage directly to the userspace, and returns.

If `nk_poll()` does not find any relevant NQE, it sleeps until CoreEngine wakes up the NK device when new NQEs arrive to its receive queue. GuestLib then parses the NQEs to check if any sockets are in the epoll instances, and wakes up the epoll to return to application. An `epoll_wait()`can also be returned by a timeout.

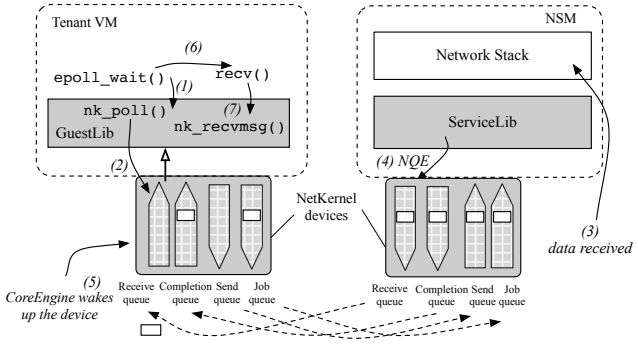

Figure 5: The socket semantics channel with epoll as an example. GuestLib and ServiceLib translate semantics to NQEs, and queues in the NK devices perform NQE transmission. Job and completion queues are for socket operations and execution results, send queues are for socket operations with data, and receive queues are for events of newly received data. Application data processing is not shown.

4.3 NQE Switching across Lockless Queues

We now elaborate how NQEs are switched by CoreEngine and how the NK devices interact with CoreEngine.

Scalable Queue Design. The queues in a NK device is scalable: there are one dedicated queue set per vCPU for both VM and NSM, so NetKernel performance scales with CPU resources. Each queue set is shared memory with the CoreEngine, essentially making it a single producer single consumer queue without lock contention. VM and NSM may have different numbers of queue sets.

Switching NQEs in CoreEngine. NQEs are load balanced across multiple queue sets with the CoreEngine acting as a switch. CoreEngine maintains a connection table as shown in Figure 6, which maps the tuple ⟨VM ID, queue set ID, socket ID⟩ to the corresponding ⟨NSM ID, queue set ID, socket ID⟩ and vice versa. Here a socket ID corresponds to a pointer to the `sock` struct in the user VM or NSM. We call them VM tuple and NSM tuple respectively. NQEs only contain VM tuple information.

Using the running example of the `socket()` call, we can see how CoreEngine uses the connection table. The process is also shown in Figure 6. (1) When CoreEngine processes the socket NQE from VM1's queue set 1, it realizes this is a new connection, and inserts a new entry to the table with the VM tuple from the NQE. (2) It checks which NSM should handle it,[1] performs hashing based on the three tuple to determine which queue set (say 2) to switch to if there are multiple queue sets, and copies the NQE to the NSM's corresponding job queue. CoreEngine adds the NSM ID and queue set ID to the new entry. (3) ServiceLib gets the NQE and copies the

[1]A user VM to NSM mapping is determined either by the users offline or some load balancing scheme dynamically by CoreEngine.

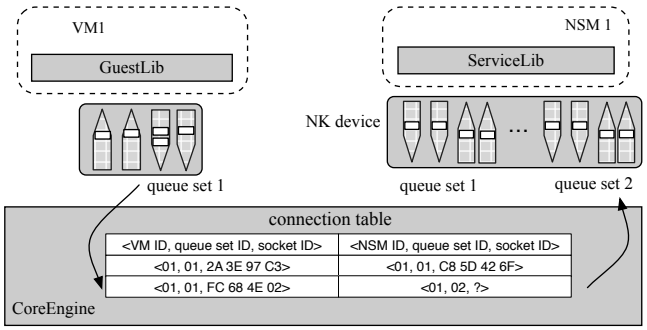

Figure 6: NQE switching with CoreEngine.

VM tuple to its response NQE, and adds the newly created connection ID in the NSM to the `op_data` field of response NQE. (4) CoreEngine parses the response NQE, matches the VM tuple to the entry and adds the NSM socket ID to complete it, and copies the response NQE to the completion queue 1 of the corresponding VM1 as instructed in the NQE. Later NQEs for this VM connection can be processed by the correct NSM connection and vice versa. Note ServiceLib pins its connections to its vCPUs and queue sets, thus processing the NQE and sending the response NQE is done on the same CPU.

The connection table allows flexible multiplexing and demultiplexing with the socket ID information. For example one NSM can serve multiple VMs using different sockets. CoreEngine uses polling across all queue sets to maximize performance.

4.4 Management with CoreEngine

CoreEngine acts as the control plane of NetKernel and carries out many control tasks beyond NQE switching.

NK Device and Queue Setup. CoreEngine allocates shared memory for the queue sets and setts up the NK devices accordingly when a VM or NSM starts up, and de-allocates when they shut down. Queues can also be dynamically added or removed with the number of vCPUs.

Isolation. CoreEngine sits in an ideal position to carry out isolation among VMs, a task essential in public clouds with VMs sharing one NSM. In our design CoreEngine polls each queue set in a round-robin fashion to ensure the basic fair sharing. Providers can implement other forms of isolation mechanisms to rate limit a VM in terms of bandwidth or the number of NQEs (i.e. operations) per second, which we also experimentally show in §7.6. Note that CoreEngine isolation happens for egress; ingress isolation at the NSM in general is more challenging and may need to resort to physical NIC queues [21].

4.5 Processing Application Data

So far we have covered API redirection, socket semantics transmission, NQE switching, and CoreEngine management in NetKernel. We now discuss the last missing piece: how application data are actually processed in the system.

Sending Data. Application data are transmitted by hugepages shared between the VM and NSM. Their NK devices maintain a hugepage region that is mmaped to the application hugepages. For sending data with `send()`, GuestLib copies data from userspace directly to the hugepage, and adds a data pointer to the send NQE. It also increases the send buffer usage for this socket similar to the send buffer size maintained in an OS. The `send()` now returns to the application. ServiceLib invokes the `tcp_sendmsg()` provided by the kernel stack upon receiving the send NQE. Data are obtained from the hugepage, processed by the network stack and sent via the vNIC. A new NQE is generated with the result of send at NSM and sent to GuestLib, who then decreases the send buffer usage accordingly.

Receiving Data. Now for receiving packets in the NSM, a normal network stack would send received data to userspace applications. In order to send received data to the user VM, ServiceLib then copies the data chunk to huge pages and create a new NQE to the receive queue, which is then sent to the VM. It also increases the receive buffer usage for this connection, similar to the send buffer maintained by GuestLib described above. The rest of the receive process is already explained in §4.2. Note that application uses `recv()` to copy data from hugepages to their own buffer.

ServiceLib. As discussed ServiceLib deals with much of data processing at the NSM side so the network stack works in concert with the rest of NetKernel. One thing to note is that unlike the kernel space GuestLib, ServiceLib should live in the same space as the network stack to ensure best performance. We have focused on a Linux kernel stack with a kernel space ServiceLib here. The design of a userspace ServiceLib for a userspace stack is similar in principle. We implement both types of stacks as NSMs in §5. ServiceLib polls all its queues whenever possible for maximum performance.

4.6 Optimization

We present several best-practice optimizations employed in NetKernel to improve efficiency.

Pipelining. NetKernel applies pipelining in general between VM and NSM for performance. For example on the VM side, a `send()` returns immediately after putting data to the hugepages, instead of waiting for the actual send result from the NSM. Similarly the NSM would handle the `accept()` by accepting a new connection and returning immediately, before the corresponding NQE is sent to GuestLib and then application to process. Doing so does not break BSD socket

semantics. Take `send()` for example. A successful `send()` does not guarantee delivery of the message [12]; it merely indicates the message is written to socket buffer successfully. In NetKernel a successful `send()` indicates the message is written to buffer in the hugepages successfully. As explained in §4.5 the NSM sends the result of send back to the VM to indicate if the socket buffer usage can be decreased or not.

Interrupt-Driven Polling. We adopt an interrupt-driven polling design for NQE event notification to GuestLib's NK device. This is to reduce the overhead of GuestLib and user VM. When an application is waiting for events e.g. the result of the `socket()` call or receive data for epoll, the device will first poll its completion queue and receive queue. If no new NQE comes after a short time period (20µs in our experiments), the device sends an interrupt to CoreEngine, notifying that it is expecting NQE, and stops polling. CoreEngine later wakes up the device, which goes back to polling mode to process new NQEs from the completion queue.

Interrupt-driven polling presents a favorable trade-off between overhead and performance compared to pure polling based or interrupt based design. It saves precious CPU cycles when load is low and ensures the overhead of NetKernel is very small to the user VM. Performance on the other hand is competent since the response NQE is received within the polling period in most cases for blocking calls, and when the load is high polling automatically drives the notification mechanism. As explained before CoreEngine and ServiceLib both use busy polling to maximize performance.

Batching. As a common best-practice, batching is used in many parts of NetKernel for better throughput. CoreEngine uses batching whenever possible for polling from and copying into the queues. The NK devices also receive NQEs in a batch for both GuestLib and ServiceLib.

5 IMPLEMENTATION

Our implementation is based on QEMU KVM 2.5.0 and Linux kernel 4.9 for both the host and the guest OSes, with over 11K LoC. We plan to open source our implementation.

GuestLib. We add the `SOCK_NETKERNEL` socket to the kernel (`net.h`), and modify `socket.c` to rewrite the `SOCK_STREAM` to `SOCK_NETKERNEL` during the socket creation. We implement GuestLib as a kernel module with two components: Guestlib_core and nk_driver. Guestlib_core is mainly for Netkernel sockets and NQE translation, and nk_driver is for NQE communications via queues. Guestlib_core and nk_driver communicate with each other using function calls.

ServiceLib and NSM. We also implement ServiceLib as two components: Servicelib_core and nk_driver. Servicelib_core translates NQEs to network stack APIs, and the nk_driver is identical with the one in GuestLib. For the kernel stack NSM, Servicelib_core calls the kernel APIs directly to handle socket operations without entering userspace. We create an independent `kthread` to poll the job queue and send queue for NQEs to avoid kernel stuck. Some BSD socket APIs can not be invoked in kernel space directly. We use `EXPORT_SYMBOLS` to export the functions for ServiceLib. Meanwhile, the boundary check between kernel space and userspace is disabled. We use per-core `epoll_wait()` to obtain incoming events from the kernel stack.

We also port mTCP [11] as a userspace stack NSM. It uses DPDK 17.08 as the packet I/O engine. The DPDK driver has not been tested for 100G NICs and we fixed a compatibility bug during the process; more details are in §6.3. For simplicity, we maintain the two-thread model and per-core data structure in mTCP. We implement the NSM in mTCP's application thread at each core. The per-core mTCP thread (1) translates NQEs polled from the NK device to mTCP socket APIs, and (2) responds NQEs to the tenant VM based on the network events collected by `mtcp_epoll_wait()`. Since mTCP works in non-blocking mode for performance enhancement, we buffer send operations at each core and set the `timeout` parameter to 1ms in `mtcp_epoll_wait()` to avoid starvation when polling NQE requests.

Queues and Huge Pages. The huge pages are implemented based on QEMU's IVSHMEM. The page size is 2 MB and we use 128 pages. The queues are ring buffers implemented as much smaller IVSHMEM devices. Together they form a NK device which is a virtual device to the VM and NSM.

CoreEngine. The CoreEngine is a daemon with two threads on the KVM hypervisor. One thread listens on a pre-defined port to handle NK device (de)allocation requests, namely 8-byte network messages of the tuples $\langle ce\_op, ce\_data \rangle$. When a VM (or NSM) starts (or terminates), it sends a request to CoreEngine for registering (or deregistering) a NK device. If the request is successfully handled, CoreEngine responds in the same message format. Otherwise, an error code is returned. The other thread polls NQEs in batches from all NK devices and switches them as described in §4.3.

6 NEW USE CASES

To demonstrate the potential of NetKernel, we present some new use cases that are realized in our implementation. Details of our testbed is presented in §7.1. The first two use cases show benefits for the operator, while the next two show benefits for users.

6.1 Multiplexing

Here we describe a new use case where the operator can optimize resource utilization by serving multiple bursty VMs with one NSM.

To make things concrete we draw upon a user traffic trace collected from a large cloud in September 2018. The trace contains statistics of tens of thousands of application gateways (AGs) that handle tenant (web) traffic in order to provide load balancing, proxy, and other services. The AGs are internally deployed as VMs by the operator. We find that the AG's average utilization is very low most of the time. Figure 7 shows normalized traffic processed by three most utilized AGs (in the same datacenter) in our trace with 1-minute intervals for a 1-hour period. We can clearly see the bursty nature of the traffic. Yet it is very difficult to consolidate their workloads in current cloud because they serve different customers using different configurations (proxy settings, LB strategies, etc.), and there is no way to separate the application logic with the underlying network stack. The operator has to deploy AGs as independent VMs, reserve resources for them, and charge customers accordingly.

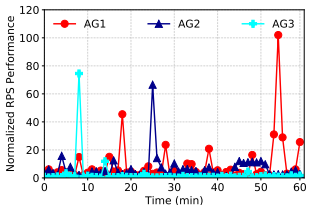

Figure 7: Traffic of three most utilized application gateways (AGs) in our trace. They are deployed as VMs.

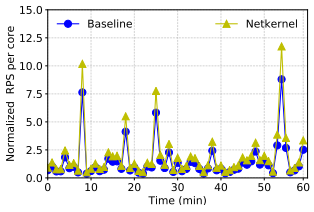

Figure 8: Per-core RPS comparison. Baseline uses 12 cores for 3 AGs, while NetKernel with multiplexing only needs 9 cores.

NetKernel enables multiplexing across AGs running distinct services, since the common TCP stack processing is now separated into the NSM. Using the three most utilized AGs which have the least benefit from multiplexing as an example, without NetKernel each needs 4 cores in our testbed to handle their peak traffic, and the total per-core requests per second (RPS) of the system is depicted in Figure 8 as Baseline. Then in NetKernel, we deploy 3 VMs each with 1 core to replay the trace as the AGs, and use a kernel stack NSM with 5 cores which is sufficient to handle the aggregate traffic. Totally 9 cores are used including CoreEngine, representing a saving of 3 cores in this case. The per core RPS is thus improved by 33% as shown in Figure 8. Each AG has exactly the same RPS performance without any packet loss.

In the general case multiplexing these AGs brings even more gains since their peak traffic is far from their capacity. For ease of exposition we assume the operator reserves 2 cores for each AG. A 32-core machine can host 16 AGs. If we use NetKernel with 1 core for CoreEngine and a 2-core NSM, we find that we can always pack 29 AGs each with 1 core for the application logic as depicted in Table 2, and the maximum utilization of the NSM would be well under 60% in the worst case for ~97% of the AGs in the trace. Thus one machine can run 13 or 81.25% more AGs now, which means the operator can save over 40% cores for supporting this workload. This implies salient financial gains for the operator: according to [24] one physical core has a maximum potential revenue of $900/yr.

	Baseline	NetKernel
Total # Cores	32	32
NSM	0	2
CoreEngine	0	1
# AGs	16	29

Table 2: NetKernel multiplexes more AGs per machine and saves over 40% cores.

6.2 Fair Bandwidth Sharing

TCP is designed to achieve flow-level fairness for bandwidth sharing in a network. This leads to poor fairness in a cloud where a misbehaved VM can hog the bandwidth by say using many TCP flows. Distributed congestion control at an entity-level (VM, process, etc.) such as Seawall [55] has been proposed and implemented in a non-virtualized setting. Yet using Seawall in a cloud has many difficulties: the provider has to implement it on the vSwitch or hypervisor and make it work for various guest OSes. The interaction with the VM's own congestion control logic makes it even harder [31].

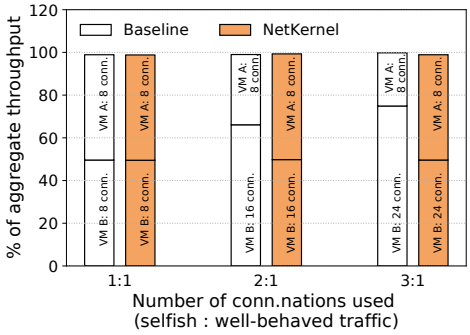

Figure 9: By sharing a unified congestion window to same destination, a NSM can achieve VM fairness.

NetKernel allows schemes like Seawall to be easily implemented as a new NSM and effectively enforce VM-level fair bandwidth sharing. Our proof-of-concept runs a simple VM-level congestion control in the NSM: One VM maintains a global congestion window shared among all its connections to different destinations. Each individual flow's ACK advances the shared congestion window, and when sending data, each flow cannot send more than $1/n$ of the shared window where n is the number of active flows. We then

# vCPUs	1	2	4
Kernel stack NSM	71.9K	133.6K	200.1K
mTCP NSM	98.1K	183.6K	379.2K

Table 3: Performance of unmodified nginx using ab with 64B html files, a concurrency of 100, and 10M requests in total. The NSM and VM use the same number of vCPUs.

run experiments with 2 VMs: a well-behaved VM that has 8 active flow, and a selfish VM that uses varying number of active flows. Figure 9 presents the results. NetKernel with our VM-level congestion control NSM is able to enforce an equal share of bandwidth between two VMs regardless of number of flows. We leave the implementation of a complete general solution such as Seawall in NetKernel as future work.

6.3 Deploying mTCP without API Change

We now focus on use cases of deployment and performance benefits for users.

Most userspace stacks use their own APIs and require applications to be ported [1, 5, 33]. For example, to use mTCP an application has to use `mtcp_epoll_wait()` to fetch events [33]. The semantics of these APIs are also different from socket APIs [33]. These factors lead to expensive code changes and make it difficult to use the stack in practice. Currently mTCP is ported for only a few applications, and does not support complex web servers like nginx.

With NetKernel, applications can directly take advantage of userspace stacks without any code change. To show this, we deploy unmodified nginx in the VM with the mTCP NSM we implement, and benchmark its performance using ab. Both VM and NSM use the same number of vCPUs. Table 3 depicts that mTCP provides 1.4x–1.9x improvements over the kernel stack NSM across various vCPU setting.

NetKernel also mitigates the maintenance efforts required from tenants. We provide another piece of evidence with mTCP here. When compiling the DPDK version required by mTCP on our testbed, we could not set the RSS (receive side scaling) key properly to the mlx5_core driver for our NIC and mTCP performance was very low. After discussing with mTCP developers, we were able to attribute this to the asymmetric RSS key used in the NIC, and fixed the problem by modifying the code in DPDK mlx5 driver. We have submitted our fix to mTCP community. Without NetKernel tenants would have to deal with such technical complication by themselves. Now they are taken care of transparently, saving much time and effort for many users.

6.4 Shared Memory Networking

In the existing architecture, a VM's traffic always goes though its network stack, then the vNIC, and the vSwitch, even when the other VM is on the same host. It is difficult for both users and operator to optimize for this case, because the VM has no information about where the other endpoint is. The hypervisor cannot help either as the data has already been processed by the TCP/IP stack. With NetKernel the NSM is part of the infrastructure, the operator can easily detect the on-host traffic and use shared memory to copy data for the two VMs. We build a prototype NSM to demonstrate this idea: When a socket pair is detected as an internal socket pair by the GuestLib, and the two VMs belong to the same user, a shared memory NSM takes over their traffic. This NSM simply copies the message chunks between their hugepages and bypasses the TCP stack processing. As shown in Figure 10, with 7 cores in total, NetKernel with shared memory NSM can achieve 100Gbps, which is ~2x of Baseline using TCP Cubic.

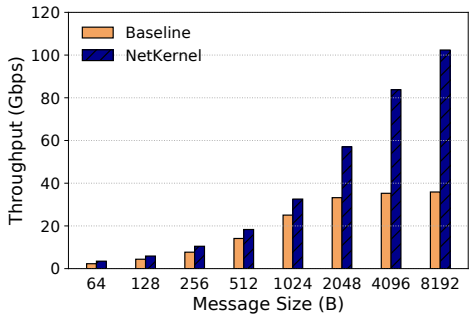

Figure 10: Using shared memory NSM for NetKernel for traffic between two colocating VMs of the same user. NetKernel uses 2 cores for each VM, 2 cores for the NSM, and 1 core for CoreEngine. Baseline uses 2 core for the sending VM, 5 cores for receiving VM, and runs TCP Cubic. Both schemes use 8 TCP connections.

7 EVALUATION

We seek to examine a few crucial aspects of NetKernel in our evaluation: (1) microbenchmarks of NQE switching and data copying §7.2; (2) basic performance with the kernel stack NSM §7.3; (3) scalability with multiple cores §7.4 and multiple NSMs §7.5; (4) isolation of multiple VMs §7.6; (5) latency of short connections §7.7; and (6) overhead of the system §7.8.

7.1 Setup

Our testbed servers each have two Xeon E5-2698 v3 16-core CPUs clocked at 2.3 GHz, 256 GB memory at 2133 MHz, and a Mellanox ConnectX-4 single port 100G NIC. Hyperthreading is disabled. We compare to the status quo where an application uses the kernel TCP stack in its VM, referred to as Baseline in the following. We designate NetKernel to refer to the common setting where we use the kernel stack NSM in our implementation. When mTCP NSM is used we explicitly

mark the setting in the figures. CoreEngine uses one core for NQE switching throughout the evaluation. Unless stated otherwise, Baseline and NetKernel use 1 vCPU for the VM, and NetKernel uses 1 vCPU for the NSM. The same TCP parameter settings are used for both systems.

7.2 Microbenchmarks

We first microbenchmark NetKernel regarding NQE and data transmission performance.

NQE switching. NQEs are transmitted by CoreEngine as a software switch. It is important that CoreEngine offers enough horsepower to ensure performance at 100G. We measure CoreEngine throughput defined as the number of 32-byte NQEs copied from GuestLib's NK device queues to the ServiceLib's NK device queues with two copy operations. Figure 11 shows the results with varying batch sizes. CoreEngine achieves ~8M NQEs/s throughput without batching. With a small batch size of 4 or 8 throughput reaches 41.4M NQEs/s and 65.9M NQEs/s, respectively, which is sufficient for most applications.[2] More aggressive batching provides throughput up to 198M NQEs/s. We use a batch size of 4 in all the following experiments.

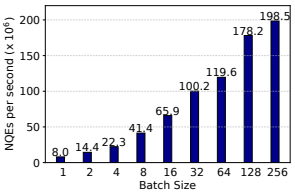

Figure 11: CoreEngine switching throughput using a single core with different batch sizes.

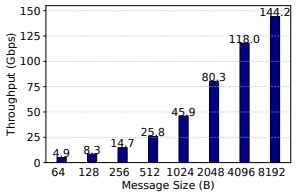

Figure 12: Message copy throughput via hugepages with different message sizes.

Memory copy. We also measure the memory copy throughput between GuestLib and ServiceLib via hugepages. A memory copy in this experiment includes the following: (0) application in the VM issues a `send()` with data, (1) GuestLib gets a pointer from the hugepages, (2) copies the message to hugepages, (3) prepares a NQE with the data pointer, (4) CoreEngine copies the NQE to ServiceLib, (5) ServiceLib obtains the data pointer and puts it back to the hugepages. Thus it measures the effective application-level throughput using NetKernel (including NQE transmission) without network stack processing.

Observe from Figure 12 that NetKernel delivers over 100G throughput with messages larger than 4KB: with 8KB messages 144G is achievable. Thus NetKernel provides enough raw performance to the network stack and is not a bottleneck to the emerging 100G deployment in public clouds.

[2] 64Mpps provides more than 100G bandwidth with an average message size of 192B.

7.3 Basic Performance with Kernel Stack

We now look at NetKernel's basic performance with Linux kernel stack. The results here are obtained with a 1-core VM and 1-core NSM; all other cores of the CPU are disabled. Baseline uses one core for the VM.

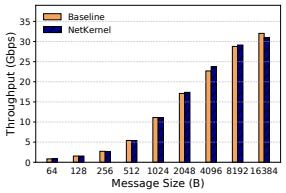

Figure 13: Single TCP stream send throughput with the kernel stack NSM. The NSM uses 1 vCPU.

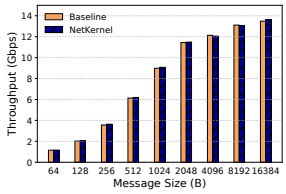

Figure 14: Single TCP stream receive throughput with the kernel stack NSM. The NSM uses 1 vCPU.

Single TCP Stream. We benchmark the single stream TCP throughput with different message sizes. The results are averaged over 5 runs each lasting 30 seconds. Figure 13 depicts the send throughput and Figure 14 receive throughput. We find that NetKernel performs on par with Baseline in all cases. Send throughput reaches 30.9Gbps and receive throughput tops at 13.6Gbps in NetKernel. Receive throughput is much lower because the kernel stack's RX processing is much more CPU-intensive with interrupts. Note that if the other cores of the NUMA node are not disabled, soft interrupts (softirq) may be sent to those cores instead of the one assigned to the NSM (or VM), thereby inflating the receive throughput.[3]

Multiple TCP Streams. We look at throughput for 8 TCP streams on the same single-core setup as above. Figures 15 and 16 show the results. Send throughput tops at 55.2Gbps, and receive throughput tops at 17.4Gbps with 16KB messages. NetKernel achieves the same performance with Baseline.

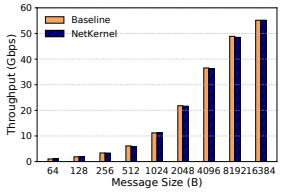

Figure 15: 8-stream TCP send throughput with the kernel stack NSM. The NSM uses 1 vCPU.

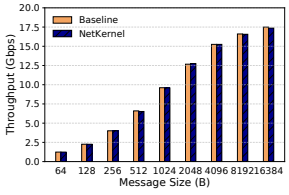

Figure 16: 8-stream TCP receive throughput with the kernel stack NSM. The NSM uses 1 vCPU.

Short TCP Connections. We also benchmark NetKernel's performance in handling short TCP connections using a

[3] We observe 30.6Gbps receive throughput with 16KB messages in both NetKernel and Baseline when leaving the other cores on.

server sending a short message as a response. The servers are multi-threaded using epoll with a single listening port. Our workload generates 10 million requests in total with a concurrency of 1000. The connections are non-keepalive. Observe from Figure 17 that NetKernel achieves ~70K requests per second (rps) similar to Baseline, when the messages are smaller than 1KB. For larger message sizes performance degrades slightly due to more expensive memory copies for both systems.

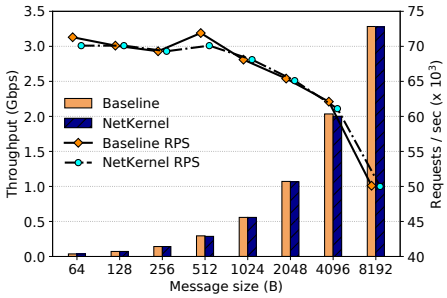

Figure 17: RPS with the kernel stack NSM using 1 vCPU.

7.4 Network Stack Scalability

Here we focus on the scalability of network stacks in NetKernel.

Throughput. We use 8 TCP streams with 8KB messages to evaluate the throughput scalability of the kernel stack NSM. Results are averaged over 5 runs each lasting 30 seconds. Figure 18 shows that both systems achieve the line rate of 100G using 3 vCPUs or more for send throughput. For receive, both achieve 91Gbps using 8 vCPUs as shown in Figure 19.

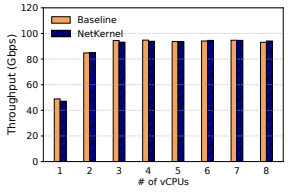

Figure 18: Send throughput of 8 TCP streams with varying numbers of vCPUs. Message size 8KB.

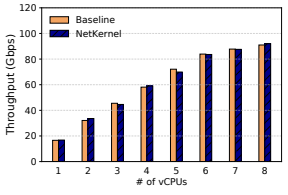

Figure 19: Receive throughput of 8 TCP streams with varying numbers of vCPUs. Message size 8KB.

Short TCP Connections. We also evaluate scalability of handling short connections. The same epoll servers described before are used here with 64B messages. Results are averaged over a total of 10 million requests with a concurrency of 1000. Socket option `SO_REUSEPORT` is always used.

Figure 20 shows that NetKernel has the same scalability as Baseline: performance increases to ~400Krps with 8 vCPUs, i.e. 5.7x the single core performance. More interestingly, to demonstrate NetKernel's full capability, we also run the mTCP NSM with 1, 2, 4, and 8 vCPUs.[4] NetKernel with mTCP offers 190Krps, 366Krps, 652Krps, and 1.1Mrps respectively, and shows better scalability than kernel stack.

The results show that NetKernel preserves the scalability of different network stacks, including high performance stacks like mTCP.

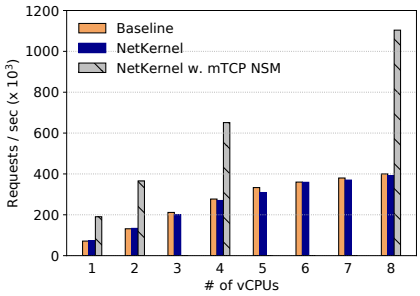

Figure 20: Performance of TCP short connections with varying number of vCPUs. Message size 64B. Both kernel stack and mTCP NSMs are used.

7.5 NetKernel Scalability

We now investigate the scalability of our design. In particular, we look at whether adding more NSMs can scale performance. Different from the previous section where we focus on the scalability of a network stack, here we aim to show the scalability of NetKernel's overall design.

We use the same epoll servers in this set of experiments. The methodology is the same as §7.4, with 8 connections and 8KB messages for throughput experiments and 10 millions of requests with 64B messages for short connections experiments. Each kernel stack NSM now uses 2 vCPUs. The servers in different NSMs listen on different ports and does not share an accept queue. We vary the number of NSMs to serve this 1-core VM.

# of 2-vCPU NSMs	1	2	3	4
Send throughput (Gbps)	85.1	94.0	94.1	94.2
Receive throughput (Gbps)	33.6	61.2	91.0	91.0
Requests per sec (x10^3)	131.6	260.4	399.1	520.1

Table 4: Throughput scaling and short connections with varying numbers of NetKernel with kernel stack NSM each with two vCPUs.

Table 4 shows the throughput scaling results. Throughput for send is already 85.1Gbps with 2 vCPUs (recall Figure 18), and adding NSMs does not improve it beyond 94.2Gbps. Throughput for receive shows almost linear scalability on the

[4]Using other numbers of vCPUs for mTCP causes stability problems even without NetKernel.

other hand. Performance of short connections also exhibits near linear scalability: One NSM provides 131.6Krps, 2 NSMs 260.4Krps, and 4 NSMs 520.1Krps which is 4x better. The results indicate that NetKernel's design is highly scalable; reflecting on results in §7.4, the network stack's scalability limits its multicore performance.

7.6 Isolation

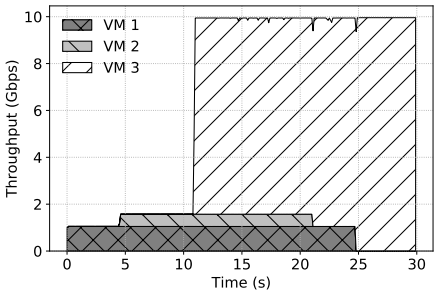

Figure 21: VM 1 is capped at 1Gbps, VM2 at 500Mbps, and VM3 uncapped. All VMs use the same kernel stack NSM. The NSM is assigned 10Gbps bandwidth. NetKernel isolates VM1 and VM2 successfully while allowing VM3 to obtain the remaining capacity.

Isolation is important in public clouds to ensure co-located tenants do not interfere with each other. We conduct an experiment to verify NetKernel's isolation guarantee. As discussed in §4.4, CoreEngine uses round-robin to poll each VM's NK device. In addition, for this experiment we implement token buckets in CoreEngine to limit the bandwidth of each VM, taking into account varying message sizes. There are 3 VMs now: VM1 is rated limited at 1Gbps, VM2 at 500Mbps, and VM3 has unlimited bandwidth. They arrive and depart at different times. They are colocated on the same host running a kernel stack NSM using 1 vCPU. The NSM is given a 10G VF for simplicity of showing work conservation.

Figure 21 shows the time series of each VM's throughput, measured by our epoll servers at 100ms intervals. VM1 joins the system at time 0 and leaves at 25s. VM2 comes later at 4.5s and leaves at 21s. VM3 joins last and stays until 30s. Observe that NetKernel throttles VM1's and VM2's throughput at their respective limits correctly despite the dynamics. VM3 is also able to use all the remaining capacity of the 10G NSM: it obtains 9Gbps after VM2 leaves and 10Gbps after VM1 leaves at 25s. Therefore, NetKernel is able to achieve the same isolation in today's public clouds with bandwidth caps. More complex isolation mechanisms can be applied in NetKernel, which is beyond the scope of this paper.

7.7 Latency

One may wonder if NetKernel with the NQE transmission would add delay to TCP processing, especially in handling short connections. Table 5 shows the latency statistics when we run ab to generate 1K concurrent connections to the epoll server for 64B messages. A total of 5 million requests are used. NetKernel achieves virtually the same latency as Baseline. Even for the mTCP NSM, NetKernel preserves its low latency due to the much simpler TCP stack processing and various optimization [33]. The standard deviation of mTCP latency is much smaller, implying that NetKernel itself provides stable performance to the network stacks.

	Min	Mean	Stddev	Median	Max
Baseline	0	16	105.6	2	7019
NetKernel	0	16	105.9	2	7019
NetKernel, mTCP NSM	3	4	0.23	4	11

Table 5: Distribution of response times (ms) for 64B messages with 5 million requests and 1K concurrency.

7.8 Overhead

We finally investigate NetKernel's CPU overhead. To quantify it, we use the epoll servers at the VM side, and run clients from a different machine with fixed throughput or requests per second for both NetKernel and Baseline with kernel TCP stack. We disable all unnecessary background system services in both the VM and NSM, and ensures the CPU usage is almost zero without running our servers. During the experiments, we measure the total number of cycles spent by the VM in Baseline, and the total cycles spent by the VM and the NSM together in NetKernel. We then report NetKernel's CPU usage normalized over Baseline's for the same performance level in Tables 6 and 7.

Throughput	20Gbps	40Gbps	60Gbps	80Gbps	100Gbps
Normalized CPU usage	1.14	1.28	1.42	1.56	1.70

Table 6: Overhead for throughput. The NSM runs the Linux kernel TCP stack. We use 8 TCP streams with 8KB messages. NetKernel's CPU usage is normalized over that of Baseline.

Requests per second (rps)	100K	200K	300K	400K	500K
Normalized CPU usage	1.06	1.05	1.08	1.08	1.09

Table 7: Overhead for short TCP connections. The NSM runs the Linux kernel TCP stack. We use 64B messages with a concurrency of 100.

We can see that to achieve the same throughput, NetKernel incurs relatively high overhead especially as throughput increases. This is due to the extra memory copy from the hugepages to the NSM. This overhead can be optimized away by implementing zerocopy between the hugepages and the NSM, which we are working on currently.

Table 7 shows NetKernel's overhead with short TCP connections. Observe that the overhead ranges from 5% to 9% in all cases and is fairly mild. As the message is only 64B here, the results verify that the NQE transmission overhead of the NK devices is small.

Lastly, throughout all experiments of our evaluation we dedicated one core to CoreEngine, which is another overhead. As we focus on showing feasibility and potential of NetKernel in this paper, we resort to software NQE switching which attributes to the polling overhead. It is possible to explore hardware offloading using FPGAs for example to attack this overhead, just like offloading the vSwitch processing to SmartNICs [23, 24]. This way CoreEngine does not consume CPU for the majority of the NQEs: only the first NQE of a new connection needs to be handled in CPU (direct to a proper NSM as in §4.3).

To quickly recap, current NetKernel implementation incurs CPU overheads especially for extra data copy and CoreEngine. We believe, however, they are not inevitable when separating the network stack from the guest OS. They can be largely mitigated using known implementation techniques which is left as future work. In addition, as shown in §6.1 multiplexing can be used in NetKernel's current implementation to actually save CPU compared to dedicating cores to individual VMs.

8 DISCUSSION

NetKernel marks a significant departure from the way networking is provided to VMs nowadays. One may have the following concerns which we address now.

How about security? One may have security concerns with NetKernel's approach of using the provider's NSM to handle tenant traffic. Security impact is minimal because most of the security protocols such as HTTPS/TLS work at the application layer. They can work as usual with NetKernel. One exception is IPSec. Due to the certificate exchange issue, IPSec does not work directly in our design. However, in practice IPSec is usually implemented at dedicated gateways instead of end-hosts [56]. Thus we believe the impact is not serious.

How about fate-sharing? Making network stack a service introduces some more additional fate-sharing, say when VMs share the same NSM. We believe this is not serious because cloud customers already have fate-sharing with the vSwitch, hypervisor, and the complete virtual infrastructure. The efficiency, performance, and convenience benefits of our approach as demonstrated before outweigh the marginal increase of fate-sharing; the success of cloud computing these years is another strong testament to this tradeoff.

How can I do netfilter now? Due to the removal of vNIC and redirection from the VM's own TCP stack, some networking tools like netfilter are affected. Though our current design does not address them, they may be supported by adding additional callback functions to the network stack in the NSM. When the NSM serves multiple VMs, it then becomes challenging to apply netfilter just for packets of a specific VM. We argue that this is acceptable since most tenants wish to focus on their applications instead of tuning a network stack. NetKernel does not aim to completely replace the current architecture. Tenants may still use the VMs without NetKernel if they wish to gain maximum flexibility on the network stack implementation.

Can hardware offloading be supported? Providers are exploring how to offload certain networking tasks, such as congestion control, to hardware like FPGA [15] or programmable NICs [46]. NetKernel is not at odds with this trend. It actually provides better support for hardware offloading compared to the legacy architecture. The provider can fully control how the NSM utilizes the underlying hardware capabilities. NetKernel can also exploit hardware acceleration for NQE switching as discussed in §7.8.

9 RELATED WORK

We survey several lines of closely related work.

There has been emerging interest on providing proper congestion control abstractions in our community. CCP [47] for examples puts forth a common API to expose various congestion control signals to congestion control algorithms independent of the data path. HotCocoa proposes abstractions for offloading congestion control to hardware [15]. They focus on congestion control while NetKernel focuses on stack architecture. They are thus orthogonal to our work and can be deployed as NSMs in NetKernel to reduce the effort of porting different congestion control algorithms.

Some work has looked at how to enforce a uniform congestion control logic across tenants without modifying VMs [20, 31]. The differences between this line of work and ours are clear: these approaches require packets to go through two different stacks, one in the guest kernel and another in the hypervisor, leading to performance and efficiency loss. NetKernel does not suffer from these problems. In addition, they also focus on congestion control while our work targets the entire network stack.

In a broader sense, our work is also related to the debate on how an OS should be architected in general, and microkernels [26] and unikernels [22, 42] in particular. Microkernels take a minimalist approach and only implement address space management, thread management, and IPC in the kernel. Other tasks such as file systems and I/O are done in userspace [58]. Unikernels [22, 42] aim to provide various

OS services as libraries that can be flexibly combined to construct an OS. Different from these works that require radical changes to the OS, we seek to flexibly provide the network stack as a service without re-writing the existing guest kernel or the hypervisor. In other words, our approach brings some key benefits of microkernels and unikernels without a complete overhaul of existing virtualization technology. Our work is also in line with the vision presented in the position paper [14]. We provide the complete design, implementation, and evaluation of a working system in addition to several new use cases compared to [14].

Lastly, there are many novel network stack designs that improve performance. The kernel TCP/IP stack continues to witness optimization efforts in various aspects [40, 49, 59]. On the other hand, since mTCP [33] userspace stacks based on high performance packet I/O have been quickly gaining momentum [1, 8, 38, 43, 44, 51, 60]. Beyond transport layer, novel flow scheduling [16] and end-host based load balancing schemes [30, 35] are developed to reduce flow completion times. These proposals are for specific problems of the stack, and can be potentially deployed as network stack modules in NetKernel. This paper takes on a broader and fundamental issue: how can we properly re-factor the VM network stack, so that different designs can be easily deployed, and operating them can be more efficient?

10 CONCLUSION

We have presented NetKernel, a system that decouples the network stack from the guest, therefore making it part of the virtualized infrastructure in the cloud. NetKernel improves network management efficiency for operator, and provides deployment and performance gains for users. We experimentally demonstrated new use cases enabled by NetKernel that are otherwise difficult to realize in the current architecture. Through testbed evaluation with 100G NICs, we showed that NetKernel achieves the same performance and isolation as today's cloud. We will open source our implementation after paper review.

NetKernel opens up new design space with many possibilities. As future work we are implementing zerocopy to the NSM, and exploring using hardware queues of a SmartNIC to offload CoreEngine and eliminate CPU overhead as in §7.8.

This work does not raise any ethical issues.

REFERENCES

[1] http://www.seastar-project.org/.

[2] Amazon EC2 Container Service. https://aws.amazon.com/ecs/details/.

[3] Azure Container Service. https://azure.microsoft.com/en-us/pricing/details/container-service/.

[4] Docker community passes two billion pulls. https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/.

[5] F-Stack: A high performance userspace stack based on FreeBSD 11.0 stable. http://www.f-stack.org/.

[6] Google container engine. https://cloud.google.com/container-engine/pricing.

[7] Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html.

[8] Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. http://www.moderntech.com.hk/sites/default/files/whitepaper/V10_Solarflare_OpenOnload_IntroPaper.pdf.

[9] Mellanox Smart Network Adaptors. http://www.mellanox.com/page/programmable_network_adapters?mtag=programmable_adapter_cards.

[10] Netronome. https://www.netronome.com/.

[11] https://github.com/eunyoung14/mtcp/tree/2385bf3a0e47428fa21e87e341480b6f232985bd, March 2018.

[12] The Open Group Base Specifications Issue 7, 2018 edition. IEEE Std 1003.1-2017. http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html, 2018.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[14] Anonymous. Details omitted for double-blind reviewing. 2017.

[15] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. HotCocoa: Hardware Congestion Control Abstractions. In *Proc. ACM HotNets*, 2017.

[16] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proc. USENIX NSDI*, 2015.

[17] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[18] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. USENIX OSDI*, 2014.

[19] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *Commun. ACM*, 60(2):58–66, February 2017.

[20] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized Congestion Control. In *Proc. ACM SIGCOMM*, 2016.

[21] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proc. USENIX NSDI*, 2018.

[22] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. ACM SOSP*, 1995.

[23] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Proc. NSDI*, 2017.

[24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proc. USENIX NSDI*, 2018.

[25] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proc. ACM CoNEXT*, 2015.

[26] D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Proc. the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[27] A. Greenberg. SDN in the Cloud. Keynote, ACM SIGCOMM 2015.

[28] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proc. ACM CoNEXT*, 2010.

[29] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proc. ACM SIGCOMM*, 2015.

[30] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM*, 2015.

[31] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *Proc. ACM SIGCOMM*, 2016.

[32] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *Proc. USENIX NSDI*, 2014.

[33] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. USENIX NSDI*, 2014.

[34] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proc. USENIX NSDI*, 2013.

[35] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford. CLOVE: How I Learned to Stop Worrying About the Core and Love the Edge. In *Proc. ACM HotNets*, 2016.

[36] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating Network-based CPU in Container Environments. In *Proc. USENIX NSDI*, 2018.

[37] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *Proc. USENIX NSDI*, 2019.

[38] D. Kim, T. Yu, H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *Proc. USENIX NSDI*, 2019.

[39] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *Proc. USENIX HotCloud*, 2014.

[40] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proc. ASPLOS*, 2016.

[41] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda. Multi-Path Transport for RDMA in Datacenters . In *Proc. USENIX NSDI*, 2018.

[42] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. ASPLOS*, 2013.

[43] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proc. ACM SIGCOMM*, 2014.

[44] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. ACM SIGCOMM*, 2015.

[45] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proc. SIGCOMM*, 2016.

[46] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan. The Case for Moving Congestion Control Out of the Datapath. In *Proc. ACM HotNets*, 2017.

[47] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. Restructuring Endpoint Congestion Control. In *Proc. ACM SIGCOMM*, 2018.

[48] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*, 2016.

[49] S. Pathak and V. S. Pai. ModNet: A Modular Approach to Network Stack Extension. In *Proc. USENIX NSDI*, 2015.

[50] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proc. ACM SIGCOMM*, 2014.

[51] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*, 2014.

[52] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *Proc. ACM SIGCOMM*, 2012.

[53] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. In *Proc. ACM SIGCOMM*, 2013.

[54] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *Proc. ACM SIGCOMM*, 2017.

[55] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proc. USENIX NSDI*, 2011.

[56] J. Son, Y. Xiong, K. Tan, P. Wang, Z. Gan, and S. Moon. Protego: Cloud-Scale Multitenant IPsec Gateway. In *Proc. USENIX ATC*, 2017.

[57] B. Stephens, A. Singhvi, A. Akella, and M. Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *Proc. USENIX ATC*, 2017.

[58] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proc. USENIX FAST*, 2017.

[59] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proc. USENIX ATC*, 2016.

[60] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proc. ACM SIGCOMM*, 2015.

[61] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *Proc. ACM SIGCOMM*, 2015.

[62] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proc. USENIX NSDI*, 2019.