NetKernel: Making Network Stack Part of the Virtualized Infrastructure

Zhixiong Niu, *Microsoft Research;* Hong Xu, *City University of Hong Kong;*
Peng Cheng, *Microsoft Research;* Qiang Su, *City University of Hong Kong;*
Yongqiang Xiong, *Microsoft Research;* Tao Wang, *New York University;*
Dongsu Han, *KAIST;* Keith Winstein, *Stanford University*

https://www.usenix.org/conference/atc20/presentation/niu

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

NetKernel: Making Network Stack Part of the Virtualized Infrastructure

Zhixiong Niu
Microsoft Research

Hong Xu
City University of Hong Kong

Peng Cheng
Microsoft Research

Qiang Su
City University of Hong Kong

Yongqiang Xiong
Microsoft Research

Tao Wang
New York University

Dongsu Han
KAIST

Keith Winstein
Stanford University

Abstract

This paper presents a system called NetKernel that decouples the network stack from the guest virtual machine and offers it as an independent module. NetKernel represents a new paradigm where network stack can be managed 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. Users also benefit from the simplified stack deployment and better performance. For example mTCP can be deployed without API change to support nginx natively, and shared memory networking can be readily enabled to improve performance of colocated 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.

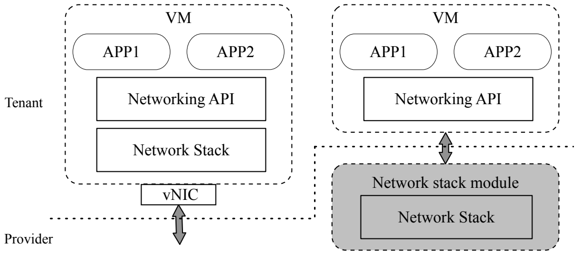
1 Introduction

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 various business units of an organization.

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 also 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, especially in a private cloud setting. The central issue is that the network stack is controlled solely by individual guest VM; the operator has almost zero visibility or control. This leads to efficiency problems that manifest in various aspects of running the cloud network. Firstly, 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 [17,28,35,41,56,57], 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. Also, many network management tasks like monitoring, diagnosis, and troubleshooting have to be performed in an extra layer outside the guest VMs, which requires significant efforts in design and implementation [23,59,60]. They can be done more efficiently if the network stack is opened up to the operator.

Even the simple task of maintaining or deploying a network stack suffers from much inefficiency today. Numerous new stack designs and optimizations ranging from congestion control [14,19,50], scalability [34,42], zerocopy datapath [4,34,55,64,65], NIC multiqueue scheduling [63], etc. have been proposed in our community. Yet the operator, with sufficient expertise and resources, could not easily deploy these solutions in a virtualized cloud to improve performance and reduce overheads because it does not own or control the network stack. As a result, our community is still finding ways to deploy DCTCP in the public cloud [20,31,36]. 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 [34], he faces a host of problems such as setting up kernel bypass, testing



(a). Existing architecture (b). Decoupling network stack from the guest

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

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 for individual users, whether tenants in a public cloud or first-party services in a private cloud, to maintain the network stack all by themselves.

To address these limitations, we advocate the *separation* of network stack from the guest OS as a new paradigm, in which the network stack is managed as part of the virtualized infrastructure 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.

More concretely, as shown in Figure 1(b), we propose to decouple the VM network stack from the guest OS. We keep the network APIs such as BSD sockets intact, and use them (instead of vNIC) as the abstraction boundary between application and infrastructure. Each VM is served by an external network stack module (NSM) that runs the network stack chosen by the user , *e.g.,* the kernel-bypass stack mTCP or the improved kernel stack FastSocket [42] . Application data are handled 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 work. This new paradigm does not necessarily enforce a single transport design, or trade off such flexibility of the existing architecture.

We make three specific contributions in this paper.

- We design and implement NetKernel that demonstrates our new approach is feasible on existing KVM virtualization platforms (§3–§5). NetKernel provides transparent BSD socket redirection so existing applications can run directly.
- We present NetKernel's benefits by showcasing novel use cases that are difficult to realize today (§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 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 979Krps with 8 cores.

NetKernel's official website is https://netkernel.net.

2 Motivation

Decoupling the network stack from the guest OS and 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 tradeoffs, and contrasting it with alternative solutions.

2.1 Benefits and Tradeoffs

We highlight the key benefits of our vision with several new use cases that 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 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, compared to doing them in an extra layer outside the guest OS.

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.

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. Various kernel stack optimizations [42, 64], high-performance userspace stacks [11, 18, 34, 55], and even designs using advanced hardware [6, 8, 9, 43] can now be deployed and maintained transparently without user involvement or application code change. For instance, DCTCP can now be deployed across the board easily in a public cloud. 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 visibility of the infrastructure for performance benefits.

Use case 2: Deploying mTCP without API change (§6.2). We show that NetKernel enables unmodified applications in the VM to use mTCP [34] in the NSM, and improves

Paradigm	Scenario	Multiplexing	New Stack Deployment	Performance Opt. with Infrastructure
Guest-based	VM	✗	Require user effort	✗
Host-based	Container	✓	Limited by host OS	✓
Application-based	Library OS	✗	Require user effort	✗
NetKernel	VM + NSM	✓	✓	✓

Table 1: Comparison of different network stack architectures depending on where the stack is. The current architecture is a guest-based paradigm where the network stack is part of the guest OS of a VM.

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 3: Shared memory networking (§6.3). 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 [40, 66].

Tradeoffs. We are conscientious of the tradeoffs our approach brings about. For example, due to the removal of vNIC and redirection from the VM's own network stack, some networking tools like netfilter are affected. This is acceptable since most users wish to focus on their applications instead of tuning a network stack. If they wish to gain maximum control over the network stack they can still use VMs without NetKernel. Also, additional fate-sharing may be introduced by our approach say when multiple VMs share the same NSM. We believe this is not serious because cloud users already have fate-sharing with the vSwitch, hypervisor, and the complete virtual infrastructure. The efficiency benefits of our approach as demonstrated outweigh the marginal increase of fate-sharing; the success of cloud computing these years is another strong testament to this tradeoff. NetKernel enforces another level of indirection in order to achieve flexibility which does not cause performance degradation in most cases as we will show in §7, and part of it can run on hardware for more efficiency (see §8). Lastly, one may have security concerns with using the NSM to handle tenant traffic. Most of the security protocols such as HTTPS/TLS work at the application layer and are not affected. One exception is IPSec. Due to the certificate exchange issue, IPSec does not work in our approach. However, in practice IPSec is implemented at dedicated gateways instead of end-hosts [62]. Thus we believe the impact is not serious. More discussion on security can be found in §8.

2.2 Alternative Solutions

We now discuss several alternative architectures depending on where the network stack resides, and why they are inadequate compared to NetKernel as summarized in Table 1. Note that none of them provides all four key benefits as NetKernel does.

Host-based. The first alternative is a host-based paradigm where the network stack runs on the host machine. This corresponds to the container scenario in the cloud. A container is essentially a process with namespace isolation: it shares the host's network stack in the hypervisor. Therefore containers can achieve some of NetKernel's benefits, i.e multiplexing and performance optimization with infrastructure, since the operator can access the hypervisor. However, container has tight coupling with the host OS which makes the stack deployment difficult. A Windows application in a container cannot use the Linux-based mTCP, unless the operator ports mTCP to Windows. With NetKernel no such porting is needed: mTCP can run in a Linux-based NSM and serve a Windows user because the only coupling is the BSD socket APIs.

We also note that currently containers have performance isolation problems [38] and as a result are usually constrained to be deployed inside VMs in production settings. In fact we find that all major public clouds [1, 2, 5] require users to launch containers inside VMs. Thus, our work is centered around VMs that cover the majority of usage scenarios in a cloud. NetKernel readily benefits containers running inside VMs as well.

Application-based. Another alternative is to move the network stack upwards by taking an application-based paradigm. A representative scenario is library OS including unikernels [22, 44] and microkernels [26], where many OS services including the network stack are packaged as libraries and compiled with the application in userspace. Similar to the guest-based paradigm, users have to deploy the network stack by themselves though the I/O performance can be improved with unikernels [46] and microkernels. In addition, application-based paradigm is a clean-slate approach and requires radical changes to both the virtualization software and user applications. NetKernel can flexibly decouple the network stack from the guest without re-writing existing applications or hypervisor.

3 Design Philosophy

NetKernel imposes three fundamental design questions around the separation of network stack from 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 a 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 [33, 34, 42].

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 §7.4 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 [52]. 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.

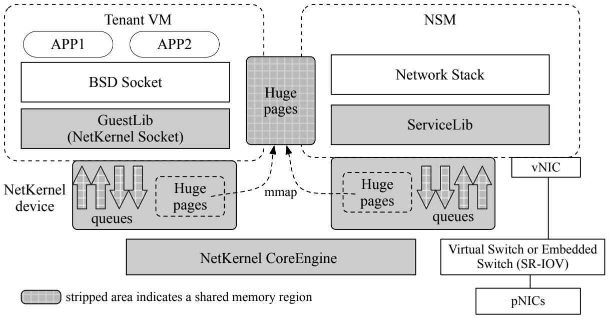


Figure 2: NetKernel design overview.

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 operator 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 in Figure 2

(§4.5). Note that as the socket API that copies data is preserved, misbehaving applications cannot pose security risks on NetKernel, this is the same as original kernel design. We discuss additional security implications of NetKernel in §8.

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.

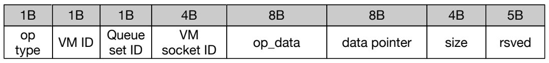
Kernel space API redirection. There are essentially two approaches to redirect BSD socket calls to NSM, each with its unique tradeoffs. One is to implement it in userspace using `LD_PRELOAD` for example. The advantages are: (1) It is efficient without syscall overheads and performance is high [34]; (2) 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 operators are already doing nowadays. Note that kernel space redirection follows the asynchronous syscall model [61] to get better performance.

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 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()`.

4.2 A Lightweight 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.

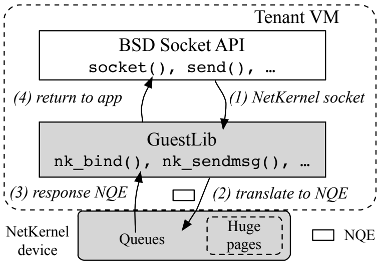


Figure 4: NetKernel socket implementation in GuestLib redirects socket API calls. GuestLib translates socket API calls to NQEs and ServiceLib translates results into NQEs as well (not shown here).

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. Figure 5 depicts the details. 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

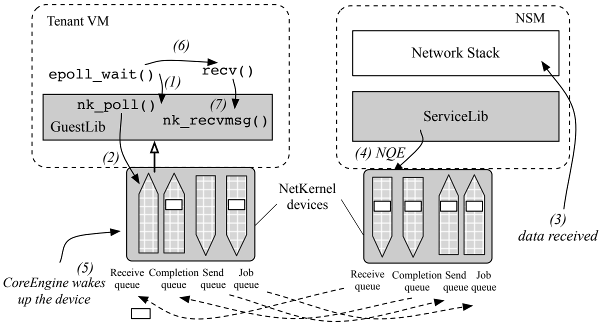


Figure 5: The socket semantics channel with epoll as an example. GuestLib and ServiceLib translate semantics to NQEs, and queues in the NK devices perform NQE transmission. Job and completion queues are for socket operations and execution results, send queues are for socket operations with data, and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

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.

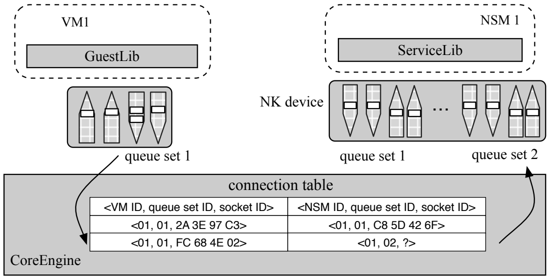
4.3 NQE Switching across Lockless Queues

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 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 VM1 as instructed in the NQE. Later NQEs for this VM connection can be processed by the correct NSM connection and vice versa. ServiceLib pins its connections to its vCPUs and queue sets, so processing the NQE and sending the response NQE are done on the same CPU.



<VM ID, queue set ID, socket ID>	<NSM ID, queue set ID, socket ID>
<01, 01, 2A 3E 97 C3>	<01, 01, C8 5D 42 6F>
<01, 01, FC 68 4E 02>	<01, 02, ?>

Figure 6: NQE switching with CoreEngine.

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 polls 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 sets 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. In our design CoreEngine polls each queue set in a round-robin fashion to ensure the basic fair sharing. Operator can implement other isolation mechanisms to rate limit a VM in terms of bandwidth or the number of NQEs (i.e. operations) per second, which we show in §7.3. Note that CoreEngine isolation happens for egress; ingress isolation at the NSM is more challenging and may have to use physical NIC queues [21].

Busy-polling. The busy-polling design of CoreEngine requires a dedicated core per machine which is an inherent

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

overhead of our design. We resort to this simple design as we focus on showing feasibility and potential of NetKernel in this work, and prior work also used dedicated cores for software polling [40]. One can explore hardware offloading using FPGAs for example to eliminate this overhead [23, 24].

4.5 Processing Application Data

We now discuss the last missing piece of NetKernel design: how application data are actually processed in the system.

Sending data. Data is 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 by the kernel. The `send()` now returns to application. ServiceLib invokes `tcp_sendmsg()` provided by the kernel stack upon receiving the send NQE. Data are obtained from hugepages, processed by the network stack, and sent to the vNIC. A new NQE is generated with the result of send by the NSM and sent to GuestLib, who then decreases the send buffer usage.

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 hugepages 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. ServiceLib busy-polls its queues for maximum performance.

4.6 Optimization

We present several optimizations employed in NetKernel.

Pipelining. NetKernel applies pipelining 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 `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 [13]; 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. This is similar in spirit to busy-polling sockets in Linux kernel [3, 10]. 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 use busy polling to maximize performance.

Batching. 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.

5 Implementation

Our implementation is based on QEMU KVM 2.5.0 and Linux kernel 4.9 for both host and guest, with over 11K LoC.

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 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 to 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 [12] as a userspace stack NSM. It uses DPDK 17.08 for packet I/O. For simplicity, we maintain its two-thread model and per-core data structure. We implement the NSM in mTCP's application thread at each core. The ServiceLib is essentially an mTCP application: once receiving a NQE from its send queue, it accesses data from the shared hugepage by the data pointer in the NQE and sends it using mTCP with DPDK. For receiving, the received data is copied into the hugepage, and ServiceLib encapsulates the data pointer into a NQE of the receive queue. The per-core application 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, 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 hugepages. The hugepages 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 ⟨ce_op, ce_data⟩. 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 Evaluation: New Use Cases

In the first part of evaluation, we present some new use cases that are realized using our prototype to demonstrate the potential of NetKernel. Details of the performance and overhead microbenchmarks are presented in §7.

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.

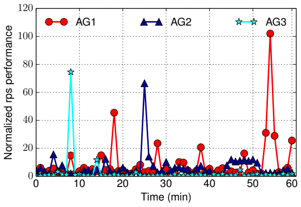


Figure 7: Traffic of three most utilized application gateways (AGs) in our trace. They are deployed as VMs.

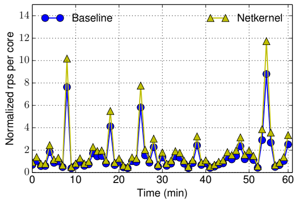


Figure 8: Per-core rps comparison. Baseline uses 12 cores for 3 AGs, while NetKernel with multiplexing only needs 9 cores.

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.

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

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

6.2 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 [4, 11, 34]. For example, in mTCP an application has to use `mtcp_epoll_wait()` to fetch events [34]. The semantics of these APIs are also different from socket APIs [34]. These factors lead to expensive code changes and make it difficult to use the stack in practice. The lack of modern APIs also makes it difficult to support complex web servers like nginx. mTCP also lacks some modern kernel TCP features such as advanced loss recovery, small queue, DSACK, *etc*.

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.

# 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 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.

NetKernel also mitigates the maintenance efforts required from users. We provide another piece of evidence with mTCP here. When compiling DPDK 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 the DPDK mlx5 driver. We have submitted our fix to mTCP community. Without NetKernel users 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.

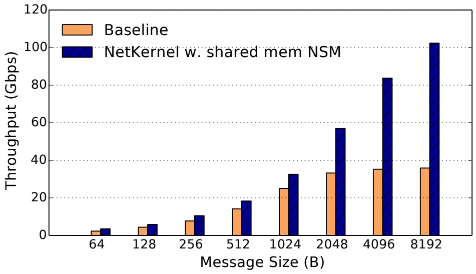


Figure 9: Using shared memory NSM for NetKernel for traffic between two colocated VMs of the same user. NetKernel uses 2 cores for each VM, 2 cores for the NSM, and 1 core for CoreEngine. Baseline uses 2 core for the sending VM, 5 cores for receiving VM, and runs TCP Cubic. Both schemes use 8 TCP connections.

6.3 Shared Memory Networking

Inter-VM communication is well-known to suffer from high overheads [58]. A VM's traffic goes through its network stack, then the vNIC and the vSwitch, even when the other VM is on the same host. It is difficult for users and operator to optimize for this case, because a 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 9, with 7 cores in total, NetKernel with shared memory NSM can achieve 100Gbps, which is ∼2x of Baseline using TCP Cubic and same number of cores.

7 Evaluation: Microbenchmarks

We now present microbenchmarks of crucial aspects of NetKernel: performance and multicore scalability in §7.2; isolation of multiple VMs in §7.3; and system overhead in §7.4.

7.1 Setup

Each of our testbed servers has 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 results. The same TCP parameter settings are used for both systems. The NSM uses the same number of vCPUs as Baseline since CPU is used almost entirely by the network stack in Baseline. NetKernel allocates 1 more vCPU for the VM to run the application and ServiceLib throughout the evaluation. Its CPU utilization is usually low: we report the actual CPU overheads of NetKernel in §7.4. The throughput results are measured by `iperf` and the rps results are measured by `ab`, unless stated otherwise. The throughput results are averaged over 5 runs each lasting 30 seconds.

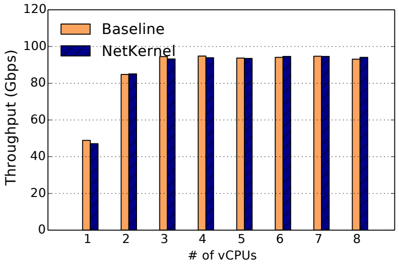


Figure 10: Send throughput of 8 TCP streams with varying numbers of vCPUs, 8KB messages.

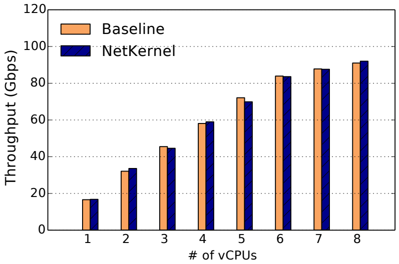


Figure 11: Recv throughput of 8 TCP streams with varying numbers of vCPUs, 8KB messages.

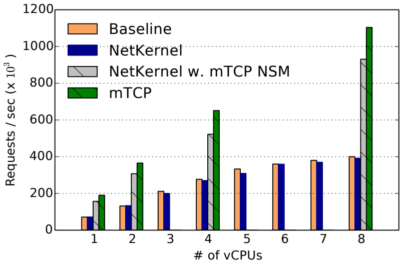


Figure 12: Performance of TCP short connections with multiple vCPUs. Message size 64B.

7.2 Performance and Scalability

We now look at NetKernel's basic performance.

NQE switching and memory copy. NQEs are transmitted by CoreEngine as a software switch. It is important that CoreEngine offers enough horsepower to ensure performance at 100G and higher. We measure CoreEngine throughput which is defined as the number of 32-byte NQEs copied from GuestLib's NK device queues to the ServiceLib's NK device queues. Table 4 shows the results with varying batch sizes. CoreEngine achieves ~8M NQEs/s 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.

We also measure the memory copy throughput between GuestLib and ServiceLib via hugepages. A memory copy in this experiment includes the following: (1) application in the VM issues a `send()` with data; (2) GuestLib gets a pointer from the hugepages; (3) copies the message to hugepages; (4) prepares a NQE with the data pointer; (5) CoreEngine copies the NQE to ServiceLib; and (6) 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.

We observe from Table 5 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 100G deployment in production.

Batch Size (B)	1	2	4	8	16	32	64	128	256
NQEs per second ($\times 10^6$)	8.0	14.4	22.3	41.4	65.9	100.2	119.6	178.2	198.5

Table 4: CoreEngine switching throughput using a single core with different batch sizes.

Message Size (B)	64	128	256	512	1024	2048	4096	8192
Throughput (Gbps)	4.9	8.3	14.7	25.8	45.9	80.3	118.0	144.2

Table 5: Message copy throughput via hugepages with different message sizes.

Throughput. We examine throughput performance using the kernel stack NSM and 8 TCP streams with 8KB messages. Figures 10 and 11 show respectively the send and receive throughput with varying number of vCPUs. NetKernel achieves the same throughput performance and scalability with Baseline. The single-core send and receive throughput reaches 48Gbps and 17Gbps, respectively. 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. Both systems achieve the line rate of 100G using at least 3 vCPUs for send throughput as in Figure 10. For receive, both achieve 91Gbps using 8 vCPUs as in Figure 11.

Short TCP connections. We also benchmark NetKernel's performance in handling short TCP connections using a custom server sending a short message as a response. The server runs multiple worker threads that share the same listening port. Each thread runs an epoll event loop. Our workload generates 10 million requests in total with a concurrency of 1000. The connections are non-keepalive. The message size is 64B. Socket option `SO_REUSEPORT` is always used for the kernel stack. Figure 12 shows that NetKernel has the same multicore scalability as Baseline: performance increases from ~71Krps with 1 vCPU to ~400Krps with 8 vCPUs, i.e. 5.6x the single core performance. To demonstrate NetKernel's full capability, we also run the mTCP NSM with 1, 2, 4, and 8 vCPUs.[2] NetKernel with mTCP offers 167Krps, 313Krps, 562Krps, and 979Krps respectively, and shows better scalability than the kernel stack.

The results here show that NetKernel preserves the performance and scalability of network stacks, including high performance stacks like mTCP since our scalable queue design can ensure NetKernel is not the bottleneck and the contention is not severe in this situation.

7.3 Isolation

Isolation is important to ensure co-located users do not interfere with each other, especially in a public cloud. It is different from fair sharing: Isolation ensures a VM's performance guarantee is met despite network dynamics, while fairness ensures a VM obtains a fair share of the bottleneck capacity which varies dynamically. We conduct an experiment to verify NetKernel's isolation guarantees. As discussed in §4.4, CoreEngine

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

uses round-robin to poll each VM's NK device for basic fairness. In addition, to achieve isolation 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 13 shows the time series of each VM's throughput, measured by our epoll server 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. We can 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 clouds with bandwidth caps.

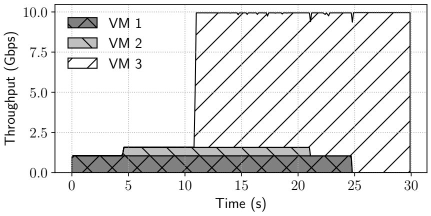


Figure 13: VM 1 is capped at 1Gbps, VM2 at 500Mbps, and VM3 uncapped. All VMs use the same kernel stack NSM. The NSM is assigned 10Gbps bandwidth. NetKernel isolates VM1 and VM2 successfully while allowing VM3 to obtain the remaining capacity.

7.4 Overhead

Latency. One may wonder if NetKernel with the NQE transmission would add delay to TCP processing, especially in handling short connections. Table 6 shows the latency statistics when we run `ab` to generate 1K concurrent connections to our epoll server for 64B messages. A total of 5 million requests are used. NetKernel achieves 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 [34]. The standard deviation of mTCP latency is much smaller, implying that NetKernel itself provides stable performance to the network stacks. We also investigate the latency without the effect of connection concurrency. To measure microsecond granularity latency, we use a custom HTTP client instead of `ab`, which reports application-level latency from the transmission of a request to the reception of the response. The experiments show the latency of Baseline and the NetKernel is 61.14μs and 89.53μs, respectively. The latency overhead is mostly introduced by CoreEngine in NetKernel.

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

CPU. Now to quantify NetKernel's CPU overhead, we use the epoll server 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 ensure the CPU usage is almost zero without running epoll servers. During the experiments, we measure the total number of cycles spent by the VM in Baseline, and that spent by the VM and NSM together in NetKernel. We then report NetKernel's CPU usage normalized over Baseline's for the same performance level in Tables 7 and 8.

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

Table 7: 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 8: Overhead for short TCP connections. The NSM runs the 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. To put things into perspective, we also measure CPU usage when the client runs in a docker container with the bridge networking mode. Docker incurs 13% CPU overhead compared to Baseline to achieve 40 Gbps throughout whereas NetKernel's is 28%. The overhead here is due to the extra memory copy from the hugepages to the NSM. It can be optimized away by implementing zerocopy between the hugepages and the NSM, which we are working on currently.

Table 8 shows NetKernel's overhead with short TCP connections. We can observe that the overhead ranges from 5% to 9% in all cases and is mild. As the message is only 64B here, the results verify that the NQE transmission overhead in NK devices is small.

8 Discussion

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 users 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.

What about troubleshooting performance issues? In current virtualized environment, operators cannot easily determine whether a performance issue is caused by the guest network stack or the underlying infrastructure. With NetKernel operators gain much visibility of the guest network stack, which potentially facilitates debugging the performance issues. For example operators can closely monitor their NSMs to detect problems with the network stack; they can also deploy additional mechanisms in the NSMs to monitor their datacenter network [29,49], all without disrupting users at all.

Does NetKernel increase the attack surface? It is well-known that shared memory design might suffer from side-channel attacks where malicious tenants could temper with other tenants' data on the hugepages. In this regard, NetKernel limits the visibility of NK devices into the hugepage for guest VMs: each device can only access its own address space. This is guaranteed by enforcing the address allocation and isolation control at CoreEngine.

How about supporting stacks with non-socket API? There are many fast network stacks with non-socket API such as PASTE [32], Seastar [11], and IX [18]. As NetKernel keeps the socket API, the central challenge to support these stacks (as NSMs) is how to resolve the semantic differences. While this requires case-by-case porting efforts, in general the ServiceLib should take care of the semantic transformation between the APIs.

Future directions. We outline a few future directions that require immediate attention with high potential: (1) Performance isolation. When multiple guest VMs share the same NSM, fine-grained performance isolation is imperative. In addition, it is necessary and interesting to design charging policies that promote fair use of the NSM and CoreEngine; (2) Resource efficiency. Various aspects of NetKernel's design can be optimized for efficiency and practicality. The CPU overhead of CoreEngine, mostly to poll the shared memory queues for NQE transmission, can be optimized by offloading to hardware like FPGA and SoC.

9 Related Work

We discuss related work besides those mentioned in §2.2.

There are many novel network stack designs to improve performance. The kernel stack continues to receive optimization in various aspects [42,53,64]. Userspace stacks based on fast packet I/O are also gaining momentum [7,11,34,40,45,48,55,65]. Beyond transport layer, novel flow scheduling [16] and end-host based load balancing schemes [30,37] are developed to reduce flow completion times. These proposals are targeting specific problems of the stack, and can be potentially deployed as NSMs in NetKernel. This paper takes on a broader and more fundamental issue: how can we properly re-factor the network stack, so that new designs can be easily deployed, and operating them in cloud can be more efficient?

Snap [47] is a microkernel networking framework that implements a range of network functions in userspace motivated by the need of rapid development and high performance packet processing in a private cloud. As NetKernel's design space and design choice are significantly different, it achieves many advantages that Snap does not target, such as multiplexing, porting a network stack across OSes or from kernel to user space, enforcing different network stack for different VMs, etc.

Lastly, our earlier position paper [51] presents the vision of network stack as a service. Here we provide the complete design, implementation, and evaluation of a working system in addition to several new use cases compared to [51].

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 focused on efficiency benefits of NetKernel in this paper since they seem most immediate. The idea of separating network stack from the guest VM applies to public and private clouds as well, and 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 [39,49] can be integrated into the network stack implementation. It opens up new design space to more freely exploit end-point coordination [25,54], software-hardware co-design, and programmable data planes [15,43]. We encourage the community to fully explore these opportunities in the future.

Acknowledgment

We thank the anonymous ATC reviewers and our shepherd Michio Honda for their valuable comments. The project was supported in part by the Hong Kong RGC GRF (CityU Project #11210818). Dongsu was supported by MSRA Collaborative Research 2016 Grant Award. Keith was supported by a Sloan Research Fellowship and by Google, Huawei, VMware, Dropbox, Amazon, and Facebook.

References

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

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

[3] Busy Polling: Past, Present, Future. https://netdevconf.info/2.1/papers/BusyPollingNextGen.pdf.

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

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

[6] 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.

[7] 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.

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

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

[10] Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll. http://caxapa.ru/thumbs/793343/Open_Source_Kernel_Enhancements_for_Low-.pdf.

[11] Seastar. http://www.seastar-project.org/.

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

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

[14] 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.

[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] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *Proc. USENIX NSDI*, 2018.

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

[34] 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.

[35] 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.

[36] G. Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *Proc. USENIX NSDI*, 2015.

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

[46] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI*, 2014.

[47] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A microkernel approach to host networking. In *Proc. ACM SOSP*, 2019.

[48] 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.

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

[50] 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.

[51] Z. Niu, H. Xu, D. Han, P. Wang, and L. Liu. Netkernel: Network stack as a service in the cloud. In *Proc. ACM HotNets*, 2017.

[52] 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.

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

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

[55] 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.

[56] 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.

[57] 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.

[58] L. Rizzo, G. Lettieri, and V. Maffione. Speeding Up Packet I/O in Virtual Machines. In *Architectures for Networking and Communications Systems*, 2013.

[59] 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.

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

[61] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. USENIX OSDI*, 2010.

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

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

[64] 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.

[65] 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.

[66] 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.