A Review of Research in System Software Communications Since 2010

2.1 Overview

Network interface cards (NICs) supporting speeds above 10 Gbps became commonplace in the early 2010s and are now widely used in data-center and other applications. With the performance of NICs rising, the efficiency of the system software that controls this hardware, particularly its data communications, has become increasingly important, and the research community has pursued many avenues to improve this performance.

In Section 2.2, I start by looking at the behavior of system software when processing communications. Section 2.3 then summarizes past research aimed at speeding this up. With that background in place, Section 2.4 then looks at IIJ Research Laboratory's efforts in this area in recent years.

I hasten to add, however, that the efforts described in Section 2.4 are still in the research stages and not yet part of IIJ's service infrastructure.

2.2 Main Communications-related Program Behaviors

In Section 2.2.1, I start by walking through communications processing in general-purpose OSes, and then in Section 2.2.2 I discuss communications on virtual machines (VMs) commonly used in data centers.

2.2.1 Communications-related Processing in General-purpose OSes

Let's look at communications processing in a general-purpose OS environment, with reference to Figure 1.

■ Basic system structure

The three main components are (from top to bottom in Figure 1):

(1) Application running in user space
(2) The kernel, which implements the network stack and device drivers
(3) A NIC, which sends and receives packets.

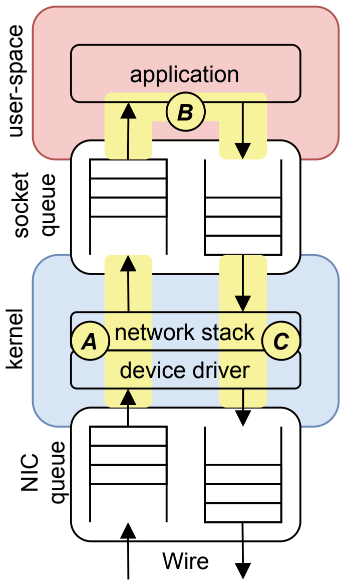

Figure 1: Structure of Communications Software in a General-purpose OS

Typical loop

Programs, called servers, that respond to client requests typically run the following loop: (A) process incoming packets in kernel space, (B) perform application-specific processing in user space, and (C) process outgoing packets in kernel space.

Packet receipt and transmission processing is summarized in Figures 2 and 3 by execution context.

A: Incoming packet processing in kernel space

STEP 1: Hardware (NIC) notifies software

When a NIC receives a new packet, it issues a hardware interrupt to the CPU to notify the software. This interrupts the program that was running on the CPU and switches to the hardware interrupt handler set up by the kernel in advance. Hardware interrupt handlers are implementation-dependent, but it's fairly common for them to start a kernel thread to process incoming packets.

STEP 2: Process incoming packets

The kernel thread started to process the incoming packets in Step 1 reads the incoming packet headers and processes the packets accordingly. For example, if a TCP packet is received, it checks the TCP ACK number for the corresponding connection, and adds the packet to the queue of the socket associated with that connection.

STEP 3: Notify user-space process

In Step 2, when data or a new connection is added to a socket queue, if the user-space process/thread associated with that socket is waiting (blocking state) for new input per the select, poll, epoll_wait, or read family of system calls (e.g., read or recvmsg), then the process/thread started (unblocked).

B: Program processing in user space

STEP 1: Awaiting and detecting input events

Many server programs that run in user space stop execution (remain in a blocking state) when using select, poll, epoll_wait, or read system calls to wait for new input to sockets (file descriptors) they are listening on. If input is received on a socket, this standby (blocking) state is released in Step 3 of A above (incoming packet processing in kernel space). Also, when a system call like select, poll, or epoll_wait unblock execution and return a value, the kernel passes on information about which socket (file descriptor) the input event occurred on.

STEP 2: Data passed from kernel to user space

The user-space program issues a read system call to the socket (file descriptor) on which the input event in Step 1 was detected, and then copies the data added to the socket queue in Step 2 of A above (incoming packet processing in kernel space) from the kernel to user space.

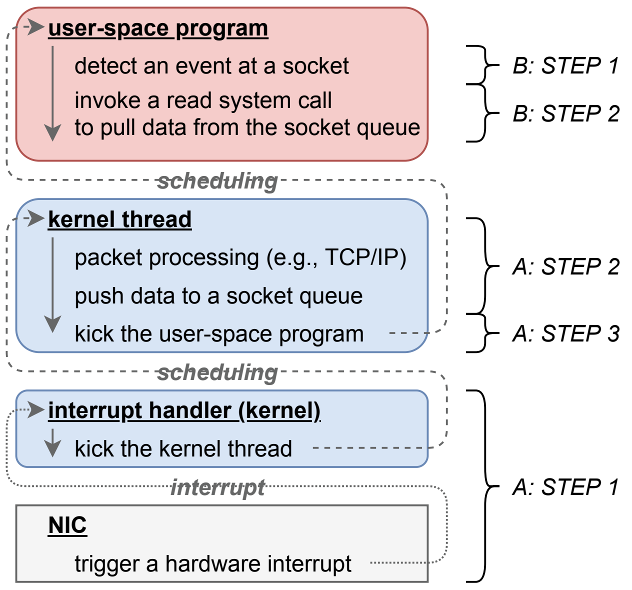
user-space program
detect an event at a socket
invoke a read system call to pull data from the socket queue
B: STEP 1
B: STEP 2
scheduling
kernel thread
packet processing (e.g., TCP/IP)
push data to a socket queue
kick the user-space program
A: STEP 2
A: STEP 3
scheduling
interrupt handler (kernel)
kick the kernel thread
interrupt
A: STEP 1
NIC
trigger a hardware interrupt

Figure 2: Processing of Incoming Packets on a General-purpose OS —Part and Step Indicated at Right

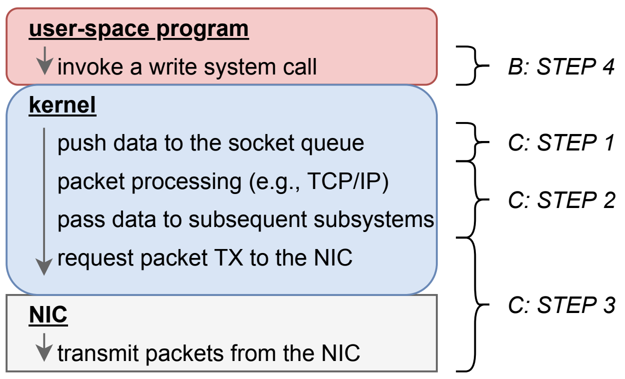
user-space program
invoke a write system call
B: STEP 4
kernel
push data to the socket queue
packet processing (e.g., TCP/IP)
pass data to subsequent subsystems
request packet TX to the NIC
C: STEP 1
C: STEP 2
C: STEP 3
NIC
transmit packets from the NIC

Figure 3: Processing of Outgoing Packets on a General-purpose OS— Part and Step Indicated at Right

☐ **STEP 3: Application-specific processing**

The program performs its application-specific processing on the data received in Step 2. For example, a web server would parse the received data, determine the content of the request, and then generate response data.

☐ **STEP 4: Tell the kernel to send the data**

The program issues a write system call (e.g., write or sendmsg) to the socket (file descriptor) that tells the kernel to send the data generated in Step 3.

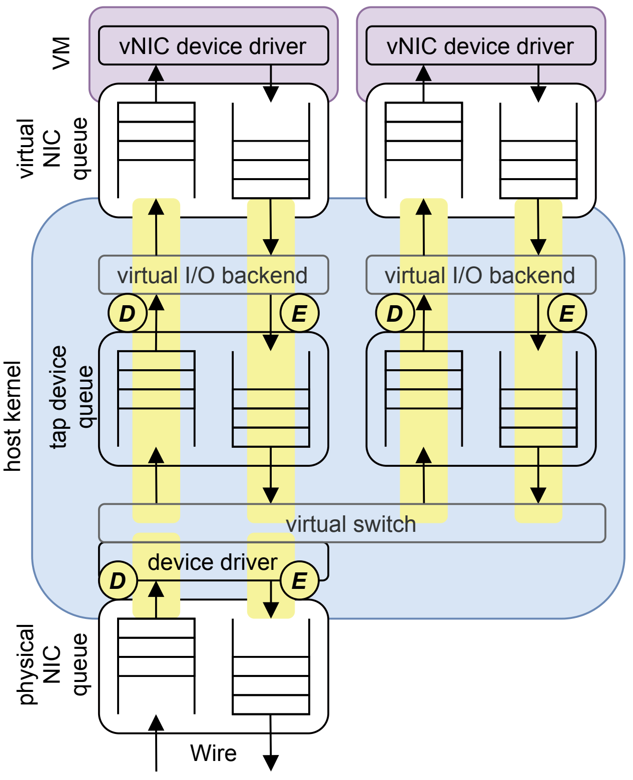

Figure 4: Virtual Machine Communications Mechanism

C: Outgoing packet processing in kernel space

☐ **STEP 1: Data passed from user space to kernel**

The write system call issued in Step 4 of B switches processing to kernel space. The kernel then copies the data generated by the user-space program into kernel space and adds it to the send queue associated with the socket specified by the user-space program.

☐ **STEP 2: Header processing based on protocol**

In the same kernel context as in Step 1, packet headers are added to the data to be transmitted if necessary. Once the packet is ready and the kernel subsystem determines it is okay to transfer the data, it is passed on to the next subsystem[*1].

☐ **STEP 3: Data transmitted from the NIC**

The data (with header added) is ultimately passed to the NIC device driver, and the device driver tells the NIC to send the data.

2.2.2 VM Network I/Os

Now let's look at how communications processing works in a VM environment, with reference to Figure 4.

Basic systm structure

The four components are (from top to bottom in Figure 4):

(1) Virtual machine (VM)
(2) Virtual NICs assigned to the VM
(3) A host kernel that implements a virtual I/O backend, tap devices, virtual switches, and device drivers
(4) A physical NIC.

VM communication functions can be implemented in a number of ways, but here I consider a format similar to Linux vhost-net, in which threads inside the host kernel function as the virtual I/O backend.

*1 In some cases, a write system call for a TCP socket (file descriptor) may not immediately result in the data specified by the user-space program being transmitted by the NIC. Possible reasons for this include the TCP implementation's congestion control, Nagle's algorithm waiting for the outgoing buffer to reach a certain size as a means of improving performance, and the delay of data transmissions by subsystems such as qdisc, which handles NIC bandwidth control.

■ Typical loop

On VMs, programs that respond to requests, as discussed above, typically run the following loop: (D) process incoming packets for the VM in the host kernel, Parts (A)–(C) as described above for general-purpose OSes[*2], (E) process the VM's outgoing packets in the host kernel.

Packet receipt and transmission processing is summarized in Figures 5 and 6 by execution context.

■ D: Processing incoming packets for the VM

□ STEP 1: Notification from hardware (physical NIC)

The initial processing performed when a packet arrives at the physical NIC is the same as in Step 1 of A above. A hardware interrupt handler is started in the host kernel, and a kernel thread is started to process incoming packets.

□ STEP 2: Incoming packets passed to virtual switch

As in Step 2 of A above, the kernel thread started in Step 1 processes the received packet, but the processing performed is different from in A above. First, the received packet is passed to the virtual switch. The virtual switch reads the Ethernet header of the received packet, finds the appropriate destination interface for the packet, and adds the packet to that interface's receive queue. Here, if the destination interface is a tap device, it starts the backend kernel thread that is responsible for virtual I/O and associated with that tap device.

□ STEP 3: Pass received data to virtual NIC

The virtual I/O backend kernel thread started in Step 2 pulls data from a tap device and pushes it to the virtual NIC's receive queue. It then sends an interrupt to the VM to notify it that packets were received on the virtual NIC.

□ STEP 4: Process incoming packets within the VM

The VM receives the interrupt sent by the host in Step 3, and processing switches to the interrupt handler set up by the kernel within the VM. From this point on, processing within the VM follows the process starting from Step 1 of A above.

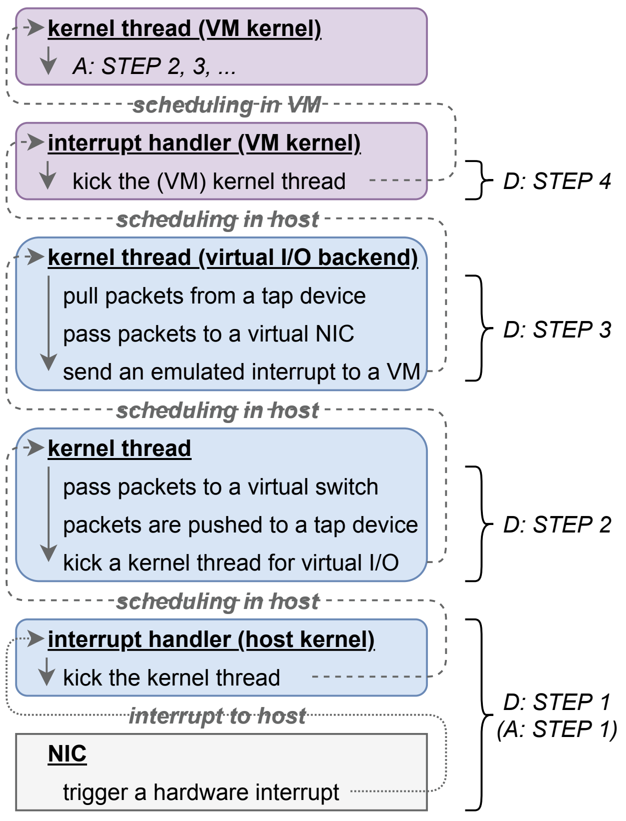
kernel thread (VM kernel)
A: STEP 2, 3, ...
scheduling in VM
interrupt handler (VM kernel)
kick the (VM) kernel thread
D: STEP 4
scheduling in host
kernel thread (virtual I/O backend)
pull packets from a tap device
pass packets to a virtual NIC
send an emulated interrupt to a VM
D: STEP 3
scheduling in host
kernel thread
pass packets to a virtual switch
packets are pushed to a tap device
kick a kernel thread for virtual I/O
D: STEP 2
scheduling in host
interrupt handler (host kernel)
kick the kernel thread
interrupt to host
NIC
trigger a hardware interrupt
D: STEP 1
(A: STEP 1)

Figure 5: Processing of Incoming Packets on a Virtual Machine —Part and Step Indicated at Right

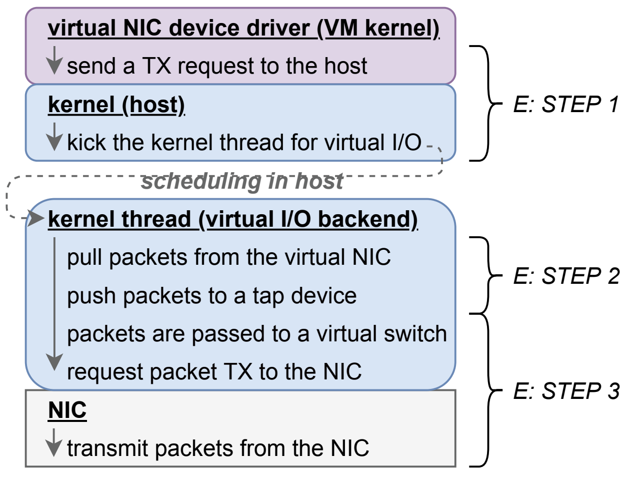
virtual NIC device driver (VM kernel)
send a TX request to the host
kernel (host)
kick the kernel thread for virtual I/O
E: STEP 1
scheduling in host
kernel thread (virtual I/O backend)
pull packets from the virtual NIC
push packets to a tap device
packets are passed to a virtual switch
request packet TX to the NIC
E: STEP 2
NIC
transmit packets from the NIC
E: STEP 3

Figure 6: Processing of Outgoing Packets on a Virtual Machine —Part and Step Indicated at Right

*2 When a general-purpose OS is running on a VM, the behavior of communication programs within the VM is basically the same as described in Section 2.2.1.

■ **E: Process the VM's outgoing packets**

□ **STEP 1: VM sends a transmission request**

At this point, the VM has executed Step 3 of C above and added the outgoing data to the virtual NIC's send queue via the virtual NIC device driver. Now when the virtual NIC is asked to send packets, execution context switches from the VM to the host kernel, and the kernel thread for virtual I/O is started.

□ **STEP 2: Data passed from virtual NIC to tap device**

The kernel thread for virtual I/O started in Step 1 above pulls data from the virtual NIC's send queue and pushes it to a tap device.

□ **STEP 3: Transfer data from tap device to virtual switch**

After Step 2, packets are passed through the tap device to the virtual switch and transmitted from the interface corresponding to the packet's destination.

2.3 Research Community Efforts

Here, I go over efforts by the research community to speed up the workloads discussed in the previous sections.

2.3.1 Reducing System Call Costs

The faster a NIC's I/O operations, the more frequently (to the extent allowed by CPU resources) the user-space program described in Part B above runs its loop. The point to note here is that the system calls involve switching the user and kernel contexts and are thus CPU-intensive. In specific terms, the workloads discussed in Part B involve frequent system call invocations: select, poll, and epoll _ wait in Step 1, read system calls in Step 2, and write system calls in Step 4. The issue is that this increases the amount of time spent on context switching as a proportion of the overall program execution time.

■ **Issuing multiple system calls at once**

In 2010, researchers presented a system called FlexSC[*3] designed to allow multiple processing requests to be sent to the kernel at once (batching). To achieve this, the system creates a set of memory pages that is shared among user and kernel space. To execute a system call, user-space threads write the system call and its arguments to the shared memory area, and a kernel thread asynchronously executes these calls and returns the results. This mechanism eliminates the need for context switching on a call-by-call basis. Implementation methods differ in their details, but this approach[*4] came to be widely adopted in efforts to optimize network stack implementations, as described in Section 2.3.3.

*3 Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). (https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls).

*4 The idea of reducing context switching by issuing multiple requests in batches had been explored before the advent of FlexFC via a technique called multi-calling in compilers[*5] and hypervisors[*]6.

*5 Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. 2003. Cassyopia: Compiler Assisted System Optimization. In Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HotOS '03), 18. (https://www.usenix.org/conference/hotos-ix/cassyopia-compiler-assisted-system-optimization).

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

■ Eliminating the boundary between apps and the kernel

Another approach is to run applications and the OS kernel in the same address space, thus eliminating the boundary between applications and the kernel. This makes it possible for application programs to use features implemented by the kernel via ordinary function calls rather than system calls. This can be done via unikernels[*7], which runs all programs—including applications and the kernel—in the same address space, and library OSes, which implement kernel functions as libraries that can run in user space. In addition to improved performance due to reduced system call context switching costs, unikernels and library OSes also offer other notable advantages such as shorter startup times for high-demand OS functions in data-center environments, reduced memory usage, and improved security. This approach has yielded much research and a range of implementations, including unikernel systems like OSv[*8], IncludeOS[*9], LightVM[*10], HermiTux[*11], Lupin Linux[*12], Unikraft[*13], and Unikernel Linux[*14], as well as library OSes like VirtuOS[*15], Graphene[*16], EbbRT[*17], KylinX[*18], and Demikernel[*19].

*7 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13), 461–472. (https://doi.org/10.1145/2451116.2451167).

*8 Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv - Optimizing the Operating System for Virtual Machines. In 2014 USENIX Annual Technical Conference (USENIX ATC 14), 61–72. (https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity).

*9 Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), 250– 257. (https://doi.org/10.1109/CloudCom.2015.89).

*10 Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My Vm Is Lighter (and Safer) Than Your Container. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), 218–233. (https://doi.org/10.1145/3132747.3132763).

*11 Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019), 59–73. (https://doi.org/10.1145/3313808.3313817).

*12 Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys ' 20). (https://doi.org/10.1145/3342195.3387526).

*13 Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21), 376–394. (https://doi.org/10.1145/3447786.3456248).

*14 Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. 2023. Unikernel Linux (UKL). In Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23), 590–605. (https://doi.org/10.1145/3552326.3587458).

*15 Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13), 116–132. (https://doi.org/10.1145/2517349.2522719).

*16 Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14). (https://doi.org/10.1145/2592798.2592812).

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

*18 Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), 173–186. (https://www.usenix.org/conference/atc18/presentation/zhang-yiming).

*19 Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In Proceedings of the ACM Sigops 28th Symposium on Operating Systems Principles (SOSP '21), 195–211. (https://doi.org/10.1145/3477132.3483569).

2.3.2 More Efficient Packet Passing Between User Space and NICs

With 10Gbps NICs now widespread, it has become difficult to achieve wire-rate performance, particularly with small packet sizes, with configurations like that illustrated in Figure 1.

■ Program behavior

To address this issue, in the early 2010s researchers presented packet I/O frameworks like Data Plane Development Kit (DPDK)[*20] and netmap[*21] to enable the efficient transfer of data between user space and NICs. Packet I/O frameworks have two basic functions:

(1) Paste the NIC's packet buffer directly into the user-space program.
(2) Provide a lightweight interface to allow the user-space program to
  a) detect new packets received by the NIC, and
  b) request the NIC to transmit packets.

■ Program behavior

When a user-space program uses these basic packet I/O framework functions to perform processing in the manner described in Part B above (receiving data and then generating and sending a response), the behavior is as follows.

□ STEP 1: Detect received packets

The program uses the interface provided by the packet I/O framework to detect new packets received by the NIC[*22].

□ STEP 2: Application-specific processing

As in Step 3 of Part B, the program performs application-specific processing based on the data received.

□ STEP 3: Transmit data from the NIC

If the application-specific processing requires data to be transmitted, the program first populates the outgoing packet buffer that was pasted into user space with the data it wants to send, and then uses the interface provided by the packet I/O framework to ask the NIC to transmit the packets.

□ Caveat

The overall program behavior described above replaces all of the processing done in Parts A, B, and C in the previous sections and greatly simplifies things by making it possible to pass data between the user-space program and the NIC extremely quickly. But it must be noted that because this does not include protocol-related processing as described in Step 2 of A and Step 2 of C, it is not possible to run a web server that delivers data via TCP connections with this setup as is.

□ Available cost reductions

The details depend on the packet I/O framework implementation, but with DPDK[*20], for example, not only is there no intervening protocol-related processing (as discussed in the caveat above), other costs that can be reduced relative to the general-purpose OS environment discussed in Section 2.2.1 include the kernel thread

*20 Intel. 2010. Data Plane Development Kit. (https://www.dpdk.org/).

*21 Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In 2012 USENIX Annual Technical Conference (USENIX ATC 12), 101–112. (https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo).

*22 When a newly received packet is detected, the data will already be in the packet buffer pasted into the user space, so there is no need to perform the processing described in Step 2 of B.

scheduling that starts in Step 1 of A, the scheduling associated with user-space program startup originating in Step 3 of A, and the system calls and associated copying of memory between user space and the kernel included in Steps 1, 2, and 4 of B.

☐ Main use cases

As mentioned in the above caveat, protocol-related processing—such as for TCP—is not performed on data that the user-space program receives from the NIC. This is actually quite useful when network functions such as a router are implemented in software, and so packet I/O frameworks are widely used in contexts like Network Function Virtualization (NFV)[*23]. The research community, for example, has developed packet I/O framework-based NFV platforms such as FastClick[*24], E2[*25], NetBricks[*26], and Metron[*27]. Also, as described in the next section, protocol stacks that run on packet I/O frameworks have been developed to enable applications like web servers to be used in combination with packet I/O frameworks. Packet I/O frameworks are also being used to speed up VM communications, as discussed in Section 2.3.4.

2.3.3 Rethinking Network Stack Design

■ Scaling in multicore environments

Many NICs let you create multiple packet queues to scale performance in multicore environments, and dividing them up for use by separate CPU cores makes it possible to avoid lock contention when attempting to access the queues. Many high-performance NICs also implement a feature called Receive Side Scaling (RSS) in hardware. RSS allows processing to be distributed by steering received packets to specific queues according to TCP connection or IP address. It is not enough to separate the packet queues, however. There is only one queue per socket for newly established TCP connections, and performance does not scale if accept system calls are issued to the same socket in parallel in a multicore environment. To address this, systems such as Affinity-Accept[*28], MegaPipe[*29], and Fastsocket[*30] offer a means of setting up TCP connection queues for each core, and it has been shown that this makes it possible to scale the performance of accept processing in multicore environments. MegaPipe[*29] also enables batch processing inspired by FlexSC[*3], which I covered in Section 2.3.1.

■ Use of packet I/O frameworks

Researchers have studied ways of using packet I/O frameworks to speed up programs like web servers, as mentioned under “Main use cases” in Section 2.3.2. Specifically, protocols like TCP/IP have been implemented that can be incorporated into Step 2 under “Program behavior” in Section 2.3.2, and this makes it possible to eliminate processing costs as mentioned in

*23 NFV makes it possible to implement network functions in software on commodity hardware, whereas previously you needed to purchase expensive custom hardware appliances for each network function. NFV allows a single computer to be used in multiple applications, and It is considered easier to add/change functions with NFV than with custom hardware. The availability of high-speed NICs at low prices, in particular, has likely been a tailwind for the uptake of NFV.

*24 Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 5–16. (https://doi.org/10.1109/ANCS.2015.7110116).

*25 Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP ’15), 121–136. (https://doi.org/10.1145/2815400.2815423).

*26 Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V Out of NFV. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 203–216. (https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda).

*27 Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), 171–186. (https://www.usenix.org/conference/nsdi18/presentation/katsikas).

*28 Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys ’12), 337–350. (https://doi.org/10.1145/2168836.2168870).

*29 Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 135–148. (https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han).

*30 Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’16), 339–352. (https://doi.org/10.1145/2872362.2872391).

"Available cost reductions" in that section, resulting in significant speed increases. In 2014, researchers presented user-space network stacks called Sandstorm[*31] and mTCP[*32]. mTCP[*32] offers a number of optimizations. In addition to request batching as proposed in FlexSC[*3], it also divides TCP connection queues among CPU cores as in Affinity-Accept[*28] and MegaPipe[*29], which I mentioned in Section 3.3.1. Also in 2014, researchers presented new OSes called Arrakis[*33] and IX[*34] designed to make it faster to use devices. Both of these allow a network stack based on a TCP/IP implementation called lwIP[*35] to deliver I/O directly to the NIC. In 2016, researchers presented a system called StackMap[*36] that uses a packet I/O framework to adopt the program behavior described in Section 2.3.2 for packet sending/receiving while also offering the benefits of the full-featured TCP/IP implementation in the kernel by using the kernel implementation for TCP/IP protocol-related processing like that in Step 2 of A and Step 2 of C. In 2019, researchers announced a TCP stack implementation called TAS[*37], which also uses DPDK[*20] and operates in user space. In 2022, researchers presented a system called zIO[*39] that extends TAS[*37] and the Strata[*38] file system and makes it possible to eliminate I/O-related memory copying without making changes to existing applications. Researchers have also looked at ways of optimizing the allocation of CPU cores to tasks in order to achieve the low levels of communications latency required in data-center settings, as demonstrated by systems like ZygOS[*40], Shenango[*41], Shinjuku[*42], and Caladan[*43]. These systems also employ a TCP/IP stack running on top of DPDK[*20].

*31 Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14), 175–186. (https://doi.org/10.1145/2619239.2626311).

*32 EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 489–502. (https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong).

*33 Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System Is the Control Plane. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 1–16. (https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter).

*34 Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 49–65. (https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay).

*35 Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science 2, 77.

*36 Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), 43–56. (https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata).

*37 Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19). (https://doi.org/10.1145/3302424.3303985).

*38 Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), 460–477. (https://doi.org/10.1145/3132747.3132770).

*39 Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 431–445. (https://www.usenix.org/conference/osdi22/presentation/stamler).

*40 George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), 325–341. (https://doi.org/10.1145/3132747.3132780).

*41 Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 361–378. (https://www.usenix.org/conference/nsdi19/presentation/ousterhout).

*42 Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 345–360. (https://www.usenix.org/conference/nsdi19/presentation/kaffes).

*43 Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 281–297. (https://www.usenix.org/conference/osdi20/presentation/fried).

■ Offloading processing to hardware

The sort of processing involved in TCP, such as connection state management, is relatively complex, putting high loads on the CPU, so researchers have also explored an approach known as the TCP Offload Engine (TOE) for offloading such processing to hardware devices like NICs. In 2020, researchers presented a system called Accell TCP[*44], which allows processing related to certain states—such as establishing TCP connections—to be offloaded to the NIC, making it possible to perform connection splicing and other such processing at high speed. The researchers showed that this can mainly help improve the performance of L7 load balancers. Also in 2020, researchers presented Tonic[*45], a hardware design that enables the implementation of transport layer protocols in the NIC. In 2022, researchers unveiled FlexTOE[*46], a TOE implementation that runs on smart NICs, and 2023 saw researchers present IO-TCP[*47], a system in which the NIC, in addition to performing TCP processing, is given direct access to storage hardware to streamline content delivery workloads.

2.3.4 Speeding up VM Communications

As Figure 4 shows, the main software components in VM communications are virtual switches that multiplex packet input/output on physical NICs, and a backend that handles virtual NIC emulation. In this section, I go over efforts to optimize these two components.

■ Speeding up virtual switches

Packet I/O frameworks, which I mentioned in Section 2.3.2, have had a huge impact in speeding up virtual switches, and research has shown that applying packet I/O frameworks around virtual switches can improve performance significantly relative to conventional approaches. In the case of VM I/O as discussed in Section 2.2.2, virtual switches run in Step 2 of D and Step 3 of E, so improving virtual switch performance can be a huge help in enhancing VM communications performance. On the research front, in 2012 researchers presented a virtual switch called VALE[*48] that can run on the netmap[*21] API mentioned in Section 2.3.2, and 2013 saw the unveiling of CuckooSwitch[*49], which uses DPDK[*20]. And in

*44 YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 77–92. (https://www.usenix.org/conference/nsdi20/presentation/moon).

*45 Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 93–109. (https://www.usenix.org/conference/nsdi20/presentation/arashloo).

*46 Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 87–102. (https://www.usenix.org/conference/nsdi22/presentation/shashidhara).

*47 Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. 2023. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), 275–292. (https://www.usenix.org/conference/nsdi23/presentation/kim-taehyun).

*48 Luigi Rizzo and Giuseppe Lettieri. 2012. VALE, a Switched Ethernet for Virtual Machines. In Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12), 61–72. (https://doi.org/10.1145/2413176.2413185).

*49 Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13), 97–108. (https://doi.org/10.1145/2535372.2535379).

2015, researchers presented mSwitch[*50] as an extension of VALE[*48]. Work to add support for DPDK[*20] to Open vSwitch[*51], a widely used virtual switch implementation, was also underway around this time.

■ Improvements to the virtual I/O backend

Around 2013, researchers made attempts to use virtual switches like that described above under "Speeding up virtual switches" in the VM communications backend. One of these attempts uses VALE[*48] as the network backend on QEMU[*52]. Specifically, it replaces the existing virtual switch implementation in the OS kernel shown in Figure 2 with the VALE[*48] switch, and the researchers showed that this can improve VM I/O performance[*53]. With this optimization, however, the virtual NIC assigned to the VM was of the existing type and thus not all that compatible with VALE[*48]. The drawback here was that, in Step 2 of D and Step 2 of E, it was unable to eliminate the copying of packet data memory between the virtual switch and the virtual NIC, so room to improve performance remained. To fill the gap, in 2015 researchers implemented ptnetmap[*54*55], which assigns netmap[*21] interfaces directly to the VM, for QEMU[*52]/KVM[*56], and showed that on ptnetmap, VMs can achieve the 10Gbps wire rate of 14.88Mpps with the smallest possible packet size. In 2014, researchers also presented ClickOS[*57], which uses VMs and replaces netfront/netback, the VM communications mechanism used in Xen[*6], with a communications mechanism based on VALE[*48] and the netmap[*21] API in order to speed up communications throughput on NFV platforms. Also in 2014, researchers presented NetVM[*58], an NFV platform based on VMs, which uses DPDK[*20] and QEMU[*52]/KVM[*56] to speed up VM communications throughput. In 2017, researchers presented a framework called HyperNF[*59], which, even in VM environments that use VALE[*48], addresses the problem of suboptimal CPU utilization due to the separation of the kernel threads running on the host side as described in Step 2 of E and the threads run on the VM's virtual CPU. It does this by performing the sort of virtual switch processing that happens on VALE[*48] within hypercalls that place it inside the virtual CPU's execution context. The researchers showed that the increased efficiency of CPU utilization resulted in high VM communications throughput.

*50 Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. 2015. mSwitch: A Highly-Scalable, Modular Software Switch. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15). (https://doi.org/10.1145/2774993.2775065).

*51 Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 117–130. (https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff).

*52 Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In 2005 USENIX Annual Technical Conference (USENIX ATC 05), 41–46. (https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator).

*53 Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. 2013. Speeding up Packet I/O in Virtual Machines. In Architectures for Networking and Communications Systems, 47–58. (https://doi.org/10.1109/ANCS.2013.6665175).

*54 Stefano Garzarella, Giuseppe Lettieri, and Luigi Rizzo. 2015. Virtual Device Passthrough for High Speed VM Networking. In 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 99–110. (https://doi.org/10.1109/ANCS.2015.7110124).

*55 Vincenzo Maffione, Luigi Rizzo, and Giuseppe Lettieri. 2016. Flexible Virtual Machine Networking Using Netmap Passthrough. In 2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), 1–6. (https://doi.org/10.1109/LANMAN.2016.7548852).

*56 Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In Proceedings of the 2007 Ottawa Linux Symposium (OLS '07).

*57 Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 459–473. (https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins).

*58 Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 445–458. (https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang).

*59 Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and Michio Honda. 2017. HyperNF: Building a High Performance, High Utilization and Fair NFV Platform. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17), 157–169. (https://doi.org/10.1145/3127479.3127489).

■ Offloading processing to hardware

Many NICs implement a hardware packet switching function like Single Root I/O Virtualization (SR-IOV)[*60], and using this can often produce better performance than a virtual switch implementation in software. Yet SR-IOV[*60] only provides limited behavioral control of packet forwarding between physical and virtual interfaces from software, which can impede its usefulness in settings where fine-grained control is needed, such as data centers. To address this, in 2018 researchers published a paper on AccelNet[*61], a system that uses smart NICs to enable more flexible network control (the deployment of AccelNet in commercial environments had apparently begun in 2015).

2.4 Recent Work at IIJ Research Laboratory

In this section, I explain what sort of efforts IIJ Research Laboratory is undertaking based on the past research covered above.

2.4.1 Integrating New OS Features and Existing Programs

As the preceding section illustrates, for over a decade now the research community has been proposing designs and implementations of new OS features that would replace existing mechanisms.

■ Problem

System call hooking is commonly used to apply new OS features transparently to existing application programs.

*60 PCI-SIG. 2010. Single Root I/O Virtualization and Sharing Specification. (https://pcisig.com/specifications/iov/single_root/).

*61 Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), 51–66. (https://www.usenix.org/conference/nsdi18/presentation/firestone).

But the existing system call hook mechanism does have its drawbacks: it can cause significant application performance degradation, and some system call hooks can fail. These shortcomings limit the applicability of unikernels, library OSes, and new network stack implementations like those discussed in previous sections, which, as a result, prevents many people from enjoying the benefits of the research work that has been done. This problem makes it difficult to use existing technologies that could greatly improve software execution efficiency, which would reduce the number of servers needed and cut power consumption.

Solution

To solve this problem, we devised a new system call hook mechanism called zpoline[*62][*63] that addresses the existing mechanism's drawbacks. zpoline[*62][*63] replaces the `syscall` and `sysenter` instructions, which are two bytes and used to issue system calls, with `callq *%rax` call instructions (also two bytes) and sets up trampoline code at virtual memory address 0, thus replacing `syscall`/`sysenter` calls in the program with jumps to specific hook processing routines. We found this mechanism to produce loads 28–700x lower than conventional mechanisms, such as binary rewriting techniques that use int3 instructions, ptrace, and Syscall User Dispatch[*64]. Also, when we used these techniques to apply lwIP[*35] and DPDK[*20] with Redis[*65], a widely used key-value store, we found that, relative to when there is almost no load from system hooks, the conventional mechanisms caused a 72.3–98.8% load degradation vs. only a 5.2% degradation with our proposed method.

2.4.2 Speeding up VM I/O

The communications performance of VMs has improved significantly over the last decade. Yet challenges remain.

Problem

The cost of exiting a VM context is considered to be a cause of performance degradation in VM environments

*62 Kenichi Yasukata. 2021. A Method for Rapidly Hooking System Calls with Zero Call Drops. IIJ Engineers Blog. (https://eng-blog.iij.ad.jp/archives/11169, in Japanese).
*63 Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. zpoline: a system call hook mechanism based on binary rewriting. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), 293–300. (https://www.usenix.org/conference/atc23/presentation/yasukata).
*64 Gabriel Krisman Bertazi. 2021. Syscall User Dispatch. (https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html).
*65 Salvatore Sanfilippo. 2009. Redis - Remote Dictionary Server. (https://redis.io/).

that has yet to be eliminated. In specific terms, the exit generated in Step 2 of E in Section 2.2.2 hinders I/O performance, which limits the performance of workloads running on the VMs. If the I/O performance of the VM itself is low, this limits the maximum achievable performance even when using the mechanisms described in Sections 2.3.1 and 2.3.3 to streamline processing related to the network running on the VM. And this is a serious issue given that a lot of computational work currently runs on VMs within data centers.

■ Solution

To address this, we developed a mechanism dubbed Exit-Less, Isolated, and Shared Access (ELISA)[*66][*67], which allows the VM to access NICs shared between VMs without exiting the VM context. With our proposed method, the VMFUNC CPU instruction is used to create a new context within the VM in which only behavior permitted by the host can happen, and the NIC can only be accessed within this new context. This prevents VMs from performing malicious behavior through the NIC. Our method also implements methods for VMs to access devices in software, so it can provide greater behavioral flexibility than SR-IOV[*60]. We showed that implementing VM communication functions with our method can yield up to 163% performance gains compared with a mechanism that is similar to HyperNF[*59] in that it exits from the VM context with every VM I/O request.

2.5 Conclusion

I first took a brief look at the general behavior of system software communications functions, and then examined how past research in system software communications since the early 2010s has improved these functions, and then rounded out the discussion with a look at IIJ Research Laboratory's recent work in this area.

Kenichi Yasukata
Researcher, Research Laboratory, IIJ

*66 Kenichi Yasukata. 2023. The Path to Getting a Paper Accepted for ASPLOS 2023—Challenges and Solutions in the Sharing of Memory Between VMs. IIJ Engineers Blog. (https://eng-blog.iij.ad.jp/archives/18819, in Japanese).

*67 Kenichi Yasukata, Hajime Tazaki, and Pierre-Louis Aublin. 2023. Exit-Less, Isolated, and Shared Access for Virtual Machines. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023), 224–237. (https://doi.org/10.1145/3582016.3582042).