vPipe: One Pipe to Connect Them All!

Sahan Gamage, Ramana Kompella, Dongyan Xu
Department of Computer Science, Purdue University
{*sgamage,kompella,dxu*}*@cs.purdue.edu*

Abstract

Many enterprises use the cloud to host applications such as web services, big data analytics and storage. One common characteristic among these applications is that, they involve significant I/O activities, moving data from a source to a sink, often without even any intermediate processing. However, cloud environments tend to be virtualized in nature with tenants obtaining virtual machines (VMs) that often share CPU. Virtualization introduces a significant overhead for I/O activity as data needs to be moved across several protection boundaries. CPU sharing introduces further delays into the overall I/O processing data flow. In this paper, we propose a simple abstraction called vPipe to mitigate these problems. vPipe introduces a simple "pipe" that can connect data sources and sinks, which can be either files or TCP sockets, at the virtual machine monitor (VMM) layer. Shortcutting the I/O at the VMM layer achieves significant CPU savings and avoids scheduling latencies that degrade I/O throughput. Our evaluation of vPipe prototype on Xen shows that vPipe can improve file transfer throughput significantly while reducing overall CPU utilization.

1 Introduction

Cloud computing platforms such as Amazon EC2 support a large number of real businesses hosting a wide variety of applications. For instance, several popular companies (*e.g.*, Pinterest, Yelp, NetFlix) host large-scale web services and video streaming services on the EC2 cloud. Many enterprises (*e.g.*, Foursquare) also use the cloud for running analytics and big data applications using MapReduce framework. Companies such as DropBox also use the cloud for storing customers' files. While these applications are quite diverse in their semantics, they share one common characteristic: They all involve significant amount of I/O activities, moving data from one I/O device (*source*) to another (*sink*). The source or sink can be either the network or the disk, and typically varies across applications as shown in Table 1. Although an application may sometimes process or modify the data after it reads from the source and before it writes to the sink, in many cases, it may merely relay the data without any processing.

Meanwhile, cloud environments use virtualization to achieve high resource utilization and strong tenant isolation. Thus, cloud applications/services are executed in virtual machines that are multiplexed over multiple cores of physical machines. Further, there is a lot of variety in the CPU resources offered to individual VMs. For instance, Amazon EC2 supports small, medium, large and extra large instances, which are assigned 1, 2, 4 and 8 EC2 units respectively, with each EC2 unit roughly equivalent to a 1GHz core [1]. Since modern commodity cores run at 2-3 GHz, a core may be shared by more than one instance.

Application	Data Source	Data Sink
Web server hosting static files	Disk	TCP socket
User uploading a file to a service	TCP socket	Disk
Local file backup service	Disk	Disk
Web proxy server	TCP socket	TCP socket

Table 1: I/O sources and sinks for typical cloud applications.

Now, imagine running the aforementioned I/O intensive applications in such CPU-sharing instances in the cloud. For concreteness, let us focus on a simple web application that receives an HTTP request from a client which results in reading a file from the disk and then writing it to a network socket. The flow of data, as shown in Figure 1(a), involves reading the file's data blocks into the application after they cross the VMM and the guest kernel boundaries, and then writing them into the TCP socket causing the data to pass again through the same protection boundaries before reaching the physical NIC.

There are two main problems with this simple data flow model. First, transferring data across all the protection layers incurs significant CPU overhead, which affects the cloud provider (provision more CPU for hypervisor) as well as the tenant (costs more for the job). Using zero-copy system calls such as `mmap()` and `sendfile()` in the guest VM, as shown in Figure 1(b), would clearly reduce the copy overhead to some extent, but not by much, since the major portion of the overhead (*e.g.*, interrupts, protection domain switching) is actually incurred when data crosses the VMM-VM boundary. Second, and perhaps *more importantly*, because of CPU sharing with other VMs, this VM may not always be scheduled, which will introduce delays in the data flow resulting in significant degradation of performance.

In this paper, we propose a new abstraction called vPipe to address both problems, *i.e.*, eliminate CPU overhead and reduce I/O processing delay, in virtualized clouds with CPU sharing. The *key idea* of vPipe

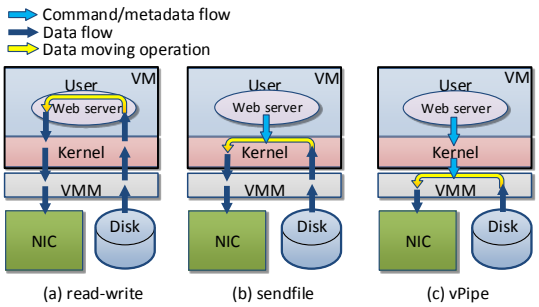


Figure 1: I/O data flow for the web server example.

is to empower the VMM to directly "pipe" data from the source to the sink *without* involving the guest VM, as shown in Figure 1(c). As can be observed from the figure, vPipe incurs fewer copies across protection boundaries, and completely eliminates the more costly VMM-VM data transfer overhead, thereby reducing CPU usage significantly which in turn saves money for both the cloud provider and the tenant. Furthermore, because the VMM is often running in a dedicated core, any scheduling latencies experienced by the guest VM due to CPU sharing have virtually no impact on the I/O performance.

While the idea of vPipe makes intuitive sense, realizing it is not that straightforward. More specifically, the meta information regarding the source and sink of a "vPipe" resides in the VM context; we need to create a new interface to enable the application to, with support of the guest kernel, pass this information to the VMM and instruct it to create the source-sink pipe. For example, the VM needs to de-reference the file blocks and establish the TCP socket which can then be passed down to the VMM layer for establishing the pipe. For applications that insert new data into the data stream, there needs to be sufficient flexibility in vPipe to allow the VMM and VM to assume control of the pipe. For example, HTTP responses are typically framed within an HTTP response header; thus the web server first needs to write the HTTP response to the sink and then call vPipe to transfer control to the VMM layer to pipe the file to the TCP socket, and then transfer the control back (*e.g.*, to keep the connection alive for persistent HTTP).

We address these challenges in a proof-of-concept implementation of a simple disk-to-network vPipe in Xen/Linux (though we have not yet addressed challenges in developing other three types of pipes—disk-to-disk, network-to-disk and network-to-network). Our micro-benchmark evaluation, which involves transferring a 1GB file from a server (VM) to a client, shows that vPipe achieves up to 2.4× throughput improvement compared with the default file transfer program not leveraging vPipe. Our evaluations also show that vPipe reduces the overall CPU utilization by 38%. A vPipe-enabled version of Lighttpd [2] improves throughput by 3.9× over the baseline implementation when sending static files to a client.

2 Design

The key idea behind vPipe is to create an I/O data "shortcut" at the VMM layer when an application needs to move data from one I/O device to another. We essentially expose a *new* library call `vpipe_file()` similar to the UNIX `sendfile()` to enable applications to create and manage this I/O shortcut. Implementing this new system call (shown in Figure 2) requires support at the guest kernel and the VMM layers, which are provided by two main components: (1) *vPipe-vm* for support in the guest kernel; and (2) *vPipe-drv* for support in the driver domain (VMM layer). Coordination across the driver domain-VM boundary is achieved with the help of the standard inter-domain channels (*e.g.*, Xen uses ring buffers and event channels) that exist in any virtualized host.

Initially, when we activate vPipe from inside the VM, the vPipe-vm module registers a special device in the system, `/dev/vpdev`, that facilitates communication between the user-level process and the guest kernel via `ioctl()`. This step is designed to prevent introducing a new system call which would in turn require modifications to the guest kernel. In the driver domain, we initialize vPipe-drv by interacting with the block device emulation layer (*e.g.*, loop device) in the driver domain to locate the VM image files (virtual disks) and then allocate a set of block device descriptors corresponding to them. It also pre-allocates a set of pages for each VM that are going to be used as a page cache for vPipe operations. A per-VM kernel thread pool (shown in Figure 2) is created to carry out different operations concurrently.

The main steps involved in vPipe-enabled I/O are as follows: First, the application running inside the VM invokes the `vpipe_file()` call with source and sink file/socket descriptors and blocks[1] until it is completed. Second, the vPipe-vm component validates the file/socket descriptors and de-references them to obtain the corresponding semantic information (*e.g.*, block ids, socket structures) that is then passed on to the driver domain. Third, the vPipe-drv component uses this semantic information and performs the actual "piping" operation. Finally, upon completion, the driver domain component notifies the guest OS with information about the data transfer through the inter-domain channel. The guest OS then passes the notification back to the application unblocking the call.

While vPipe exposes a single unified system call to connect between various types of sources and sinks (just like the UNIX interface), we currently support only TCP sockets and files on the disk as data sources/sinks.

[1] We can also implement a non-blocking version of this.

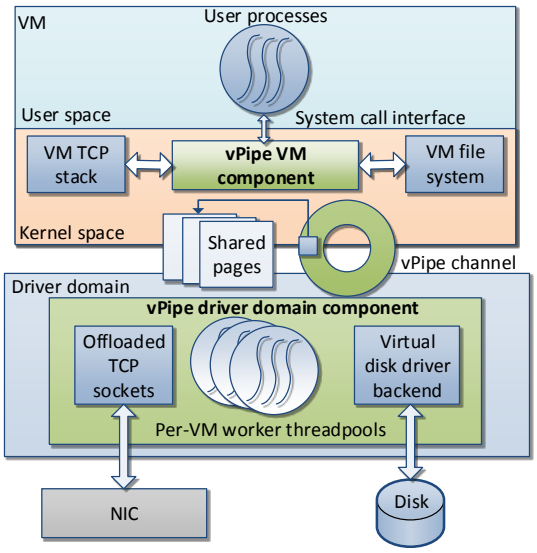


Figure 2: vPipe architecture.

2.1 TCP Connection

If the vPipe source/sink is an established TCP connection, we offload the entire TCP connection to the driver domain by supplying essential socket details (such as IP addresses, ports and sequence numbers) and letting the TCP stack at the driver domain perform TCP processing as long as vPipe exists. Note that less drastic solutions such as the guest VM fabricating placeholder TCP packets ahead of time that are filled with data in the driver domain are not that advantageous, since congestion control and other TCP processing needs to happen at the driver domain anyway, otherwise the benefit of shortcutting is completely lost. Thus, it is better to pass down the entire TCP socket to the driver domain, and then pass it back up once the vPipe operation is done.

Read/Write on a TCP socket: In preparation for vPipe-enabled network I/O, we first use the guest OS virtual file system (VFS) to translate the file descriptor to the socket structure. We then check whether the VM has any earlier sent packets that have not been acknowledged, and wait until all of them are acknowledged. This is very important because, if there is a packet loss, we will not be able to recover the lost packet once the socket is taken over by the driver domain. We then collect the socket information (TCP 4-tuple, sequence numbers and congestion control information) from the corresponding kernel socket data structure. We reuse congestion control information from the VM's socket to initialize the vPipe socket at the driver domain, instead of re-starting it from slow start. This information is then passed on to vPipe-drv.

Upon receiving this information, vPipe-drv creates a TCP socket using the driver domain's TCP stack. However, we do not use system calls such as `connect()` on this socket; we instead instantiate the kernel socket structure of the new socket using the original connection's meta-data from vPipe-vm. We then add it to the TCP hash table to allow look-ups using the TCP 4-tuple of incoming packets. There is an additional issue we need to address: We need to add a static route entry in the driver domain's IP routing table to route the packets to local TCP/IP stack if the packets match the 4-tuple described above, otherwise they will go directly to the guest VM. Finally, we mark the socket as "established", which informs the driver domain's TCP stack that the socket is ready to receive packets. vPipe will then perform standard socket operations and use pages pre-allocated as buffers to perform socket read/write.

2.2 File on Disk

If the vPipe source/sink is a file, similar to the socket, we use VM's file system to obtain meta-data about the file data blocks and transfer this information to the driver domain where either the reading or writing of the data blocks is carried out. Unlike TCP packets, file meta-data is stored separately from the actual data, in the form of separate disk blocks (*e.g.*, inode blocks). Once the meta-data is passed on from the VM level, it is straightforward for the driver domain to access the corresponding file by simply following the mapping between the files and their disk blocks.

Reading from a file When the source is a file, vPipe-vm will first locate the file's inode using the file descriptor. It then uses file system-specific functions and device information from the inode to obtain file data block identifiers. This information is then encapsulated in a vPipe custom data structure–along with number of bytes to read and offset of the first byte to transfer–and passed to the driver domain via the communication channel.

Once vPipe-drv receives this information and the block device identifier (in case the VM uses several image files), it uses this block device identifier to locate the block device descriptor it has already opened during the initialization phase. vPipe-drv then prepares a set of block I/O operation descriptors using a pre-allocated set of pages and the block identifiers supplied by vPipe-vm and submit them to the emulated block device. The batch size of the block I/O operations depends on the queue length of the emulated block device. We also supply a call-back function in this block I/O descriptor, so that we can send out the data blocks as soon as the emulated block device completes the read.

Writing to a file This operation involves either creating a new file or appending to an existing file. In both cases, the file system needs to manipulate the file mapping information. This is done by vPipe-vm requesting the guest file system to create or update the inode for the

new data blocks, with an empty set of data blocks. This will generate a new set of block identifiers which will be transferred to vPipe-drv where the actual writing of the data blocks will be performed.

2.3 Connecting the Dots

When vPipe-drv receives a vPipe request from the VM, it creates a "pipe descriptor" associated with that operation. This descriptor contains meta-data describing each source/sink and two functions: a read actor which implements one of the above read strategies and a write actor which implements one of the write strategies depending on the source and the sink. A free thread is picked up from the thread pool and this thread will call the read actor using the source's meta-data. As data returns from the source, the thread will call the write actor to output the data using the meta-data of the sink.

2.4 Fair Access to Driver Domain

vPipe poses one more challenge: Since the actual I/O operations are performed by vPipe-drv, we should "charge" the work done by the worker threads in the driver domain (Figure 2) to the VMs requesting vPipe-enabled I/O. Lack of driver domain access accounting and control will lead to unfairness among the requesting VMs. To address this problem we propose a simple credit-based system. Each VM-specific thread pool in the driver domain is allocated a certain amount of credits based on the priority (weight) of the VM. As the threads execute, they consume the allocated credits based on the amount of bytes transferred. When the credits run out, the corresponding worker threads will wait for a timer task to add more credits to them.

3 Implementation

We have implemented a partial prototype of vPipe on Xen 4.1 as the virtualization platform and Linux 3.2 as the kernel of VMs and the driver domain. Our prototype currently does not support file write operations.

vPipe-vm is implemented as a loadable kernel module. Since it uses standard Linux VFS functions already exposed to kernel modules to manipulate file descriptors and sockets, it requires no changes to the guest kernel. This makes it very attractive for customers, since no kernel recompilation is required for using vPipe.

We add a similar loadable kernel module in Xen's driver domain to implement vPipe-drv. However we have to make a few small changes in the main kernel code such as adding special functions to create offloaded sockets and adding static routes (discussed in Section 2). We implement the driver domain-VM communication channel as a standard Xen device with a ring buffer and an event channel.

4 Preliminary Evaluation

Testbed setup We use two identical hosts—one as a server equipped with vPipe and the other as client in our experiments. Each host has a 3GHz Intel Xeon quad-core CPU, 16GB of memory and runs Xen 4.1.2 as the VMM and Linux 3.2 as the OS for all of the VMs and driver domain. In the server, we pin the driver domain to one of the cores. During experiments involving multiple VMs, we pin all the VMs to another core to demonstrate CPU sharing. Also, we use `lookbusy` tool [3] to keep the CPU utilization of the VMs at specific levels.

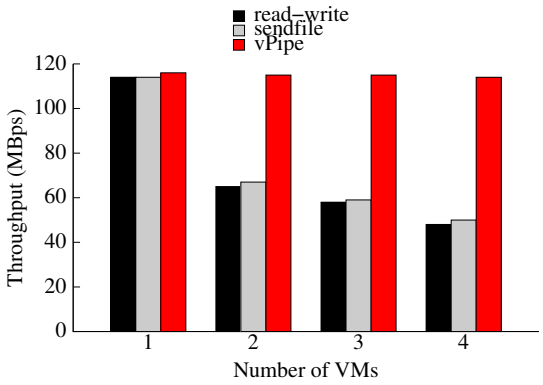
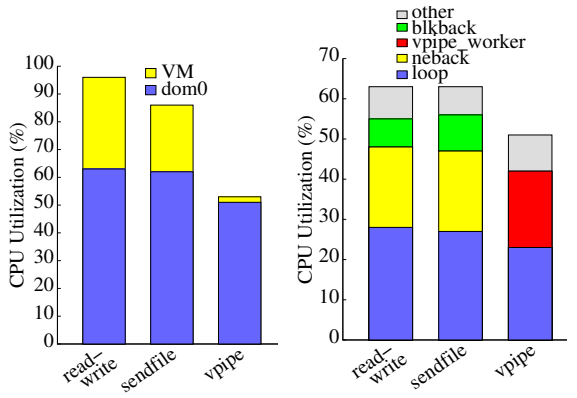


Figure 3: File transfer throughput improvement.

File transfer throughput We use a simple application that reads from a file on the disk and writes it to a socket connection with a client. The file transfer is performed in the three I/O modes shown in Figure 1. Figure 3 shows the throughput improvements achieved by vPipe transferring a 1GB file when the VM running the application is co-located with 0, 1, 2, and 3 other VMs. When only the application's VM is running on the core, all three modes can reach the full available bandwidth of the 1Gbps link. As the number of VMs sharing the core increases, throughput drops for both read-write and sendfile modes. However, since vPipe offloads the processing of the entire I/O operation to the driver domain, throughput remains the same regardless of the number of VMs sharing the core.

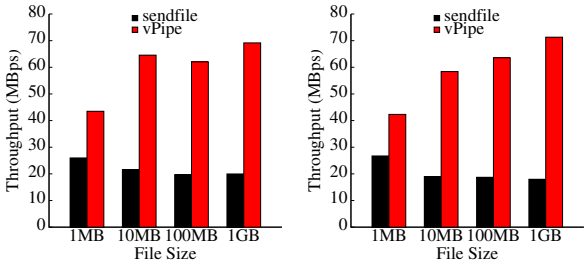
CPU utilization Figure 4(a) shows the overall average CPU utilization of both the driver domain and the VM when sending the 1GB file. As expected, the VM's CPU utilization for read-write mode is the highest since it requires copying data across all layers. The `sendfile()` system call eliminates the kernel to user-land copying and hence, the VM CPU utilization is less compared to the read-write mode. Finally, vPipe incurs the least CPU utilization at VM level since there is no work to be done in the VM context once the operation is offloaded to the driver domain.



(a) Overall CPU utilization (b) Driver domain CPU utilization

Figure 4: CPU utilization.

With vPipe offloading I/O processing task to the driver domain, we would expect that the driver domain CPU utilization for vPipe mode be the highest. (Somewhat) surprisingly, this is not the case as shown in Figure 4(a). Figure 4(b) shows the breakdown of CPU utilization of the driver domain kernel threads involved in I/O processing for the VM. We see that the CPU utilization for *loop* thread, which is responsible for reading blocks from the VM image (block device emulation), is similar for all three modes. The *netback* thread, which is responsible for emulating the network device for the VM takes up about 20% utilization for read-write and sendfile modes. Also the *blkback* thread, which emulates the disk for the VM, takes about 7-9% utilization in these two modes. Those threads are not a factor in vPipe mode, because neither the disk data related to the file transfer is read into the VM nor network packets are sent out from the VM, thus saving a total of around 29% of CPU. However, the vPipe worker thread incurs about 19% CPU utilization because it has to read data blocks from the disk and send them out using the offloaded socket; the net CPU saving using vPipe is therefore about 10-12%.



(a) 3VMs sharing the core (b) 4VMs sharing the core

Figure 5: Lighttpd throughput.

Lighttpd performance Lighttpd is a highly scalable lightweight web server that we adapt to use vPipe. To do so, we just replace "`sendfile()`" with "`vpipe_file()`" in the Lighttpd source code. Figure 5(a) and Figure 5(b) show the average I/O throughput reported by httperf for different file sizes, when the VM running Lighttpd is co-located with 2 and 3 other VMs, respectively. While Lighttpd using vPipe shows throughput improvement for all file sizes tested, improvement for larger files tends to be higher (up to 3.4× in 3-VM configuration and up to 3.9× in 4-VM configuration). For smaller files, the overhead of offloading the connection and the file block information to the driver domain affects the overall time, and hence the throughput improvement is comparatively less than that for large files.

5 Related Work

Several prior works focus on reducing the overhead associated with device virtualization. For instance, Menon *et al.* propose several optimizations such as packet coalescing, scatter/gather I/O checksum offload, and offloading device driver functionality [11, 13, 12]. Similarly, Gordon *et al.* propose exit-less interrupt delivery mechanisms to alleviate interrupt handling overhead [6, 8] in virtualized systems, where I/O events are passed to the VM without exiting to the hypervisor. Ahmed *et al.* propose virtual interrupt coalescing for virtual SCSI controllers [4], based on the number of in-flight commands to the disk controller. Virt-FS [9] presents a paravirtulized file system, which allows sharing driver domain's (or hosts) file system with VMs causing minimal overhead. While these techniques have been proved quite effective in reducing the virtualization overhead, they cannot fundamentally eliminate it. Instead, vPipe aims at avoiding the overhead by performing I/O at the VMM layer.

Offloading I/O operations to improve performance is a very well studied area. In [7] the authors discuss the idea of offloading common middleware functionality to the hypervisor layer to reduce the guest/hypervisor switches. In contrast, vPipe introduces shortcutting at the I/O level and hence applies to a broader class of cloud applications. In [14] the authors discuss offloading TCP/IP stack to a separate core. vSnoop [10] and vFlood [5] mitigate the impact of VMs' CPU access delay on TCP by offloading acknowledgement generation and congestion control to the driver domain respectively. They however only focus on improving TCP send and receive throughput but not on improving the performance of more general I/O.

6 Conclusions and Future Work

We presented vPipe, a new I/O interface for applications in virtualized clouds. vPipe mitigates virtualization-

related performance penalties by shortcutting I/O operations at the VMM layer. Our evaluation of a partial vPipe prototype shows that vPipe can improve file-to-network I/O throughput while reducing CPU utilization. vPipe also requires minimal modifications to existing applications such as web servers and facilitates a simple deployment.

Our current prototype does not support all four source/sink combinations shown in Table 1. We are addressing new challenges in developing a full-fledged vPipe. Another area vPipe can be improved is to make use of cached disk pages at the VM-level instead of re-reading them from the disk when the source is a file. We are also improving the accounting accuracy of vPipe operations on behalf of hosted VMs. Finally, We will port more applications such as Hadoop, NFS to show the ease of using vPipe and to evaluate the performance gain for those applications.

7 Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported in part by US NSF under Awards 0855141, 1054788, and 1219004. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Amazon EC2 instance types. `http://aws.amazon.com/ec2/instance-types/`.

[2] Lighttpd web server. `http://www.lighttpd.net/`.

[3] lookbusy – a synthetic load generator. `http://www.devin.com/lookbusy/`.

[4] AHMAD, I., GULATI, A., AND MASHTIZADEH, A. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX ATC* (2011).

[5] GAMAGE, S., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. Opportunistic flooding to improve TCP transmit performance in virtualized clouds. In *ACM SOCC* (2011).

[6] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS* (2012).

[7] GORDON, A., BEN-YEHUDA, M., FILIMONOV, D., AND DAHAN, M. VAMOS: virtualization aware middleware. In *WIOV* (2011).

[8] GORDON, A., HAR'EL, N., LANDAU, A., BEN-YEHUDA, M., AND TRAEGER, A. Towards exitless and efficient paravirtual I/O. In *SYSTOR* (2012).

[9] JUJJURI, V., HENSBERGEN, E. V., AND LIGUORI, A. VirtFSa virtualization aware file system pass-through. In *OLS* (2010).

[10] KANGARLOU, A., GAMAGE, S., KOMPELLA, R. R., AND XU, D. vSnoop: Improving TCP throughput in virtualized environments via acknowledgement offload. In *ACM/IEEE SC* (2010).

[11] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *USENIX ATC* (2006).

[12] MENON, A., SCHUBERT, S., AND ZWAENEPOEL, W. TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ACM ASPLOS* (2009).

[13] MENON, A., AND ZWAENEPOEL, W. Optimizing TCP receive performance. In *USENIX ATC* (2008).

[14] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly efficient network processing on dedicated cores. In *USENIX ATC* (2010).