vPipe: One Pipe to Connect Them All

SAHAN GAMAGE, RAMANA KOMPELLA, AND DONGYAN XU

Sahan Gamage is a Ph.D. candidate in the Computer Science Department at Purdue University and is advised by Professors Dongyan Xu and Ramana Kompella. His research focuses on improving I/O performance in virtual machines in cloud environments. Sahan received an M.S. in Computer Science from Purdue University and a B.S. in Computer Science from University of Moratuwa, Sri Lanka. sgamage@purdue.edu

Ramana Kompella is an Associate Professor in the Computer Science Department at Purdue University. He directs the Systems and Networking (SYN) Lab at Purdue, conducting research on various networking research problems in cloud computing, virtualization, datacenter networking, and software-defined networking. Before coming to Purdue, he obtained his Ph.D. from UCSD. rkompella@purdue.edu

Dongyan Xu is a Professor and University Faculty Scholar in the Computer Science Department at Purdue University. He leads the FRIENDS Lab at Purdue, conducting research in virtualization technologies, cloud computing, and computer systems security and forensics. He received a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2001. dxu@cs.purdue.edu

Many enterprises use the cloud to host applications such as Web services, big data analytics, and storage, which 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, which introduces a significant overhead for I/O activity as data needs to be moved across several protection boundaries. CPU sharing among virtual machines (VMs) introduces further delays into the overall I/O processing data flow. In this article, we present an 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.

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 on the EC2 cloud. Many enterprises (e.g., Foursquare) also use the cloud for running analytics and big data applications using the MapReduce framework. Companies such as Dropbox also use the cloud for storing customers' files. While these applications are quite diverse in their functionality and the services they offer, they share one common characteristic: they all involve a significant number 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 (see Table 1). Although an application may sometimes process or modify 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 compute units, respectively, with each EC2 unit roughly equivalent to a 1 GHz core [1]. Because modern commodity cores run at 2–3 GHz, a core may be shared by more than one instance.

Now, imagine running the above I/O-intensive applications in such CPU-sharing instances in the cloud. As an example, let us focus on a simple Web application that receives an HTTP request from a client that 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 (provisions more CPU for hypervisor) as well as the tenant (costs more for the job).

Application	Data Source	Data Sink
Web server hosting static files	Disk	TCP socket
User uploading a file to cloud storage	TCP socket	Disk
File backup service	Disk	Disk
Web proxy server or a load balancer	TCP socket	TCP socket

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

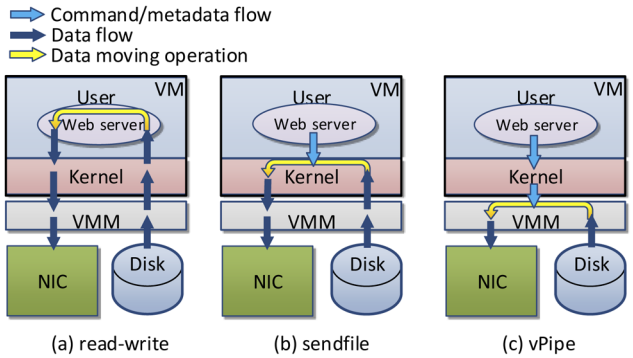

Figure 1: I/O data flow for a Web server

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, because the major portion of the overhead (e.g., virtual 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. For more information about how VM scheduling affects TCP, refer to [5] and [3].

With vPipe, we propose a new abstraction to address both of these problems—i.e., eliminate CPU overhead and reduce I/O processing delay—in virtualized clouds with CPU sharing. The key idea of vPipe is to empower the VMM to "pipe" data directly from the source to the sink without involving the guest VM. As shown in Figure 1(c), 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 I/O performance.

Although our idea of vPipe makes intuitive sense, realizing it is not that straightforward because 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 identify the physical block identifiers of the file 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 also must be sufficient flexibility in vPipe to allow the VMM and VM to take control of the pipe. For example, HTTP responses are typically preceded by an HTTP response header; so the Web server first needs to write the HTTP header to the sink (i.e., TCP socket), 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 describe how vPipe works and show the effectiveness of vPipe using a proof-of-concept implementation of a simple disk-to-network vPipe in Xen/Linux with the example of a Web server serving static files to clients.

Creating an I/O Shortcut at VMM

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 set of new library calls (e.g., vpipe_file() similar to the UNIX sendfile()) to enable applications to create and manage this I/O shortcut. Implementing these new calls (shown in Figure 2) requires support at the guest kernel and the VMM layer, 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 a standard inter-domain channel (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 process and the guest kernel via ioctl() function. This step is designed to prevent introducing a new system call, which would in turn require modifications to the guest kernel.

There are four main steps involved in vPipe-enabled I/O. First, the application running inside the VM invokes the corresponding vPipe call with source and sink file/socket descriptors and blocks (we can also implement a non-blocking version of this) until it is completed. Second, the vPipe-vm component validates the file/socket descriptors and dereferences them to obtain the corresponding information about them (e.g., block IDs, socket structures) that is then passed on to the driver domain. Third, the vPipe-drv component uses this information and performs

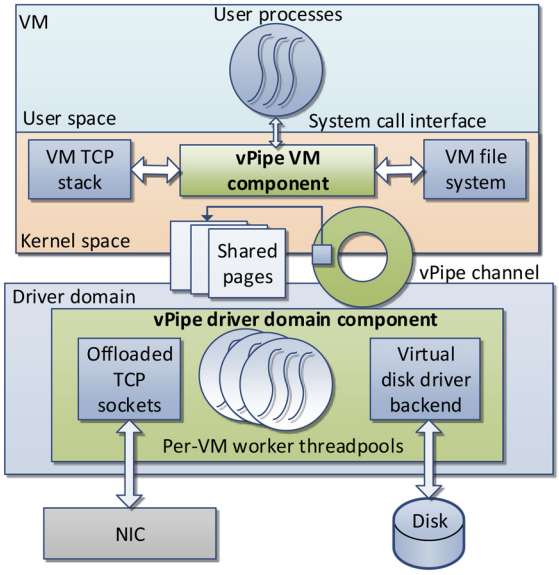

Figure 2: vPipe architecture

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.

Offloading a 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. Most VMMs have a fully functional TCP stack to carry out management tasks, and we use this for our offloaded TCP sockets.

When a TCP socket is used as either end of a vPipe, we first use the guest OS virtual file system (VFS) to translate the file descriptor to the kernel socket structure and collect the socket information (TCP 4-tuple, sequence numbers, and congestion control information). We reuse congestion control information from the VM's socket to initialize the vPipe socket at the driver domain, instead of restarting 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 metadata from vPipe-vm. There is an additional issue we need to address: the need to add a static route entry in the driver domain's IP routing table to route the packets to the 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-drv can then perform standard socket operations such as send() and recv() on this socket.

Offloading a File I/O Operation

If the vPipe source/sink is a file, similar to the socket, we use VM's file system to obtain metadata 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 metadata is stored separately from the actual data, in the form of separate disk blocks (e.g., inode blocks). Once the metadata is passed on from the VM level, for the driver domain to access the corresponding file by simply using the physical block identifiers is straightforward.

When the source of a vPipe is a file, vPipe-vm will first locate the file's inode using the file descriptor. Then vPipe-vm uses file system-specific functions and device information from the inode to obtain the file's physical 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, it prepares a set of block I/O operation descriptors using a preallocated set of pages and the block identifiers supplied by vPipe-vm and submits them to the emulated block device.

Writing to a file involves either creating a new file, appending to an existing file, or overwriting an existing file. When overwriting a file, we can use the same method as reading the file to get the file block identifiers. But when we are creating a new file or appending to an existing file, we need to request the guest's file system to create new block identifiers for new data. 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 that will be transferred to vPipe-drv, where the actual writing of the data blocks will be performed.

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 metadata describing each source/sink and two functions: a read function that implements one of the above read strategies, and a write function that 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 function using the source's metadata. As data returns from the source, the thread will call the write function to output the data using the metadata of the sink.

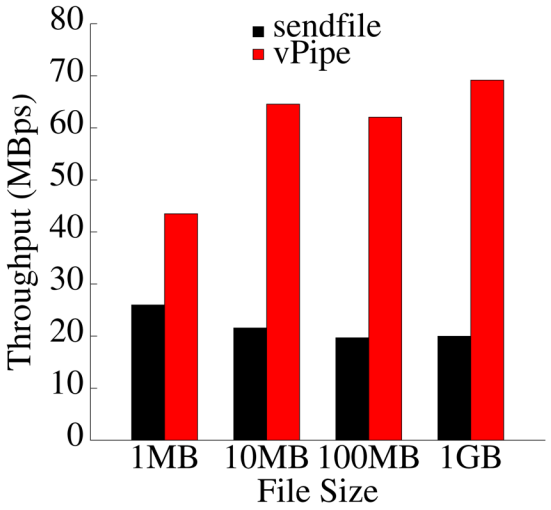

Figure 3: lighttpd throughput improvement

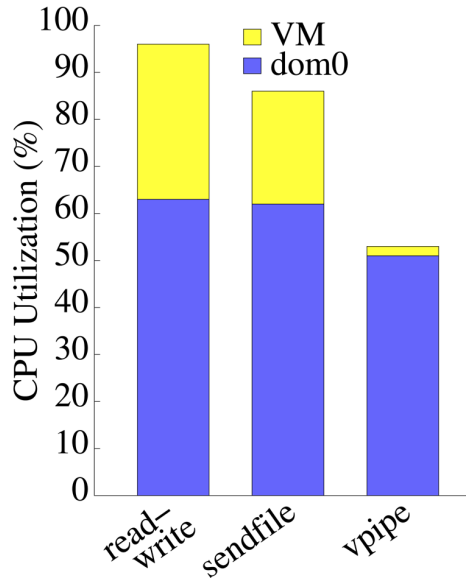

Figure 4: CPU savings by vPipe

Sharing the Driver Domain

vPipe poses one more challenge: because 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 number of bytes transferred. When the credits run out, the corresponding worker threads will block until a timer task adds more credits to them.

vPipe on Xen/Linux

We implemented a prototype of vPipe on Xen 4.1 as the virtualization platform and Linux 3.2 as the kernel of VMs and the driver domain. vPipe-vm is implemented as a loadable kernel module. Because it uses standard Linux VFS functions already exposed to kernel modules to manipulate file descriptors and sockets, vPipe-vm requires no changes to the guest kernel. This makes vPipe-vm attractive for customers, because 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 must make a few small changes in the main kernel code, such as adding special functions to create offloaded sockets and adding static routes.

We implement the driver domain-VM communication channel as a standard Xen device with a ring buffer and an event channel.

Improved lighttpd Throughput

lighttpd [2] is a highly scalable lightweight Web server that we adapt to vPipe. To do so, we just replaced "sendfile()" with "vpipe_file()" in the lighttpd source code and recompiled it. Figure 3 shows the average I/O throughput reported by httperf for different file sizes, when the VM running lighttpd is co-located with two other VMs. Whereas lighttpd using vPipe shows throughput improvement for all file sizes tested, improvement for larger files tends to be greater (up to 3.4×). 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.

CPU Savings by vPipe

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

With vPipe offloading the I/O processing task to the driver domain, we would expect that the driver domain CPU utilization for vPipe mode would be the highest. (Somewhat) surprisingly, this is not the case, as shown in Figure 4. This is because, with vPipe, we eliminate the data processing by the device emulation

layer at the driver domain, which is required to transfer disk blocks and network packets to and from the VM in the other two modes.

Wrapping Up

vPipe is a new I/O interface for applications in virtualized clouds, which mitigates virtualization-related performance penalties by shortcutting I/O operations at the VMM layer. Our experiments with the vPipe prototype shows that vPipe can improve lighttpd I/O throughput while reducing CPU utilization. vPipe also requires minimal modifications to existing applications, such as Web servers, and facilitates a simple deployment. You can find more information about vPipe in [4].

References

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

[2] lighttpd Web server: http://www.lighttpd.net/.

[3] S. Gamage, A. Kangarlou, R. R. Kompella, and D. Xu, "Opportunistic Flooding to Improve TCP Transmit Performance in Virtualized Clouds," ACM SOCC (2011).

[4] S. Gamage, R. R. Kompella, and D. Xu, "vPipe: One Pipe to Connect Them All!" USENIX HotCloud (2013): https://www.usenix.org/conference/hotcloud13/vpipe-one-pipe-connect-them-all.

[5] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu, "vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload," ACM/IEEE SC (2010).