vPipe: Piped I/O Offloading for Efficient Data Movement in Virtualized Clouds

Sahan Gamage[†] [*], Cong Xu[‡], Ramana Rao Kompella[‡], Dongyan Xu[‡]

[†]VMware Inc. [‡]Purdue University

Abstract

Virtualization introduces a significant amount of overhead for I/O intensive applications running inside virtual machines (VMs). Such overhead is caused by two main sources: (1) device virtualization and (2) VM scheduling. Device virtualization causes significant CPU overhead as I/O data need to be moved across several protection boundaries. VM scheduling introduces delays to the overall I/O processing path due to the wait time of VMs' virtual CPUs in the run queue. We observe that such overhead particularly affects many applications involving *piped I/O data movements*, such as web servers, streaming servers, big data analytics, and storage, because the data has to be transferred first into the application from the source I/O device and then back to the sink I/O device, incurring the virtualization overhead twice. In this paper, we propose vPipe, a programmable framework to mitigate this problem for a wide range of applications running in virtualized clouds. vPipe enables direct "piping" of application I/O data from source to sink devices, either files or TCP sockets, at virtual machine monitor (VMM) level. By doing so, vPipe can avoid both device virtualization overhead and VM scheduling delays, resulting in improved I/O throughput and application performance as well as significant CPU savings.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management—Input/Output

General Terms Design, Measurement, Performance

Keywords Virtualization, Cloud Computing, I/O

1. Introduction

Increasingly enterprises are moving their applications hosted in traditional infrastructures to private cloud environments or public cloud platforms such as Amazon EC2. A key technology that drives cloud computing is virtualization. In addition to enabling multi-tenancy in cloud environments, virtualizing hosts in the cloud environment has made hardware resource management increasingly flexible, resulting in significant savings in operational costs.

Many applications hosted in the cloud are I/O intensive: Ever increasing usage of distributed data processing frameworks (e.g., Hadoop) contributes to a significant share of I/O intensive workload in the cloud. Organizations and companies use public cloud infrastructures to host their multi-tier web services. Their data or contents are stored either in traditional databases (e.g., MySQL) or in NoSQL databases (e.g., MongoDB, CouchDB). In addition, many online services in the cloud (e.g., cloud-based storage) store user-generated contents either in databases or file systems. Although these applications are diverse in their semantics, many of them share one common characteristic: *they involve moving data from one I/O device to another*. For example, a web server may move data *from disk to network* when serving client requests; a storage server may move data *from network to disk* when uploading user data; a web proxy may move data *from one network socket to another* when dispatching incoming traffic; and a local file backup agent may copy data *from one disk to another*. The performance of such "I/O data piping" may affect the overall performance of the corresponding applications and, in some cases, become their bottleneck.

The performance of I/O data piping becomes more critical in a virtualized environment. A known consequence of virtualization is its negative impact on the I/O performance of applications running in the virtual machines (VMs) [6, 9, 12, 19, 21, 25, 27]. Such performance penalty on I/O processing stems from two main sources: (1) VM consolidation [9, 19, 27], and (2) device virtualization [6, 12, 21]. Recent efforts [9, 19] have shown that by offloading protocol processing functionality to the VMM layer it is possible to reduce the negative impact of VM consolidation on TCP performance and improve TCP throughput for VMs. However, these approaches still incur device virtualization overhead as they copy data to the VMs. A plethora of projects have been focusing on mitigating the performance penalty imposed by

[*] This work was conducted, in part, while Sahan Gamage was a graduate student at Purdue University.

SoCC '14, November 03 - 05 2014, Seattle, WA, USA.
ACM Copyright 2014 978-1-4503-3252-1/14/11 $15.00.
http://dx.doi.org/10.1145/2670979.2671006

the device virtualization layer [20, 22, 23]. These optimizations aim at reducing CPU cycles on the I/O data path and are mostly oblivious to VM scheduling latency. It is possible that a combination of these approaches could achieve better performance. Yet none of these methods can completely eliminate both device virtualization overhead and performance penalty incurred by VM consolidation.

We mainly focus on cloud applications that involve significant *inter-device data movement*, either partially or completely, in their workflows. More specifically, we target an application's movement of data from one device to another – without performing transformation on the data. Such data movement is common in many applications such as file servers, distributed file systems (e.g., HDFS), intermediate data stores of large data analytics applications (e.g., Hadoop), web servers serving static files, file backup services, load balancing servers, and data sharing services. These applications typically read data from one device (using the *read()* system call or one of its variants) and write to the same or another device (using the *write()* system call or one of its variants). Most of the time, the application does not need to touch the data and hence there is no fundamental need to bring the data into its memory address space. But due to the current separation between OS and applications, the data will have to be copied to the application's address space *inside* the VM, crossing multiple protection domain boundaries (VMM-VM and VM kernel space-VM user space). Clearly, such "piped" data movement will incur significant virtualization overhead from the two sources mentioned above.

In this paper, we propose to alleviate this overhead by offloading the entire piped I/O operation to the VMM layer. By doing so, we would be able to avoid (1) VM scheduling delays – because the operation will be performed in the VMM (or the driver domain) which gets scheduled more often; and (2) device virtualization overhead such as data copying, page table and grant table modifications, and switching CPU among different protection domains – because the data now reside outside the VM.

Offloading piped I/O processing to the VMM layer is, however, not straightforward. Since most of the semantic knowledge about the data resides in the VM context, for the VMM to perform the offloaded operation, we need to obtain such knowledge from the VM layer and convert it to a form that the VMM can interpret. As an example, even though the data blocks belonging to a file in the VM reside in the physical disk and can be read by the VMM layer without involving the VM, to map the file to physical disk blocks that the VMM can understand, we need knowledge about the file system running inside the VM. A similar justification can be made for the TCP packets arriving at the network interface card (NIC).

To realize the idea of offloading piped I/O operations to the VMM layer, we develop a system called vPipe, where data from a source device can be "piped" to a destination device with minimal involvement of the VM's kernel and the application running inside the VM. vPipe installs a module in the guest kernel – with a programming interface – so that it would be able to obtain the required semantic information from the VM without transferring the actual data to the VM. Once the semantic information or mapping is obtained from the VM, the VMM component of vPipe will perform the actual data transfer between the source and destination devices.

To summarize, our contributions in this paper are:

1. We propose a new programming interface for a wide range of cloud applications with piped I/O, which allows those applications to offload the piped data movement operation to the VMM layer in the same host.
2. We develop the vPipe system, which performs the I/O shortcutting at the VMM level via components in the VM's kernel and in the VMM. The in-VM component collects semantic information about the data source and data sink, while the VMM component performs the actual data movement.
3. We present evaluation results from a vPipe prototype implemented on Linux running on the Xen hypervisor. Our microbenchmark results show that vPipe achieves throughout improvements for all types of supported source-to-destination combinations. Results from Apache HDFS, web server systems, and video streaming servers also show significant application-level performance improvements with vPipe.

2. Motivation

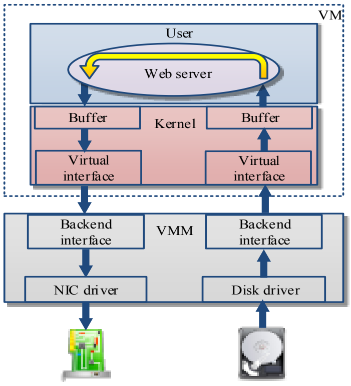


Figure 1. I/O data flow in the web server example.

Let us focus on a concrete example where a web server hosted in a VM is sending a static file located in the disk to a client. In this scenario, the web server process receives an HTTP request from the client via a TCP connection. The web server parses this request and locates the file requested by the client. If the file is a legitimate one, the web server process sends an HTTP header and then engages in a loop where the file content is read into a buffer in the process' address space and then written to the socket corresponding to the client's TCP connection. The flow of I/O data in this example is shown in Figure 1. To perform this data transfer, the web server process and the OS running in the VM only need to examine some meta-data of the file – such as the file's length, type, and permissions – to verify its validity and to construct the HTTP header. Yet content of the entire file is copied all the way to the web server process' memory.

Such a data flow model has two main inefficiencies in virtualized clouds: (1) When data from disk cross multiple protection boundaries (i.e., VMM-VM boundary and kernel-user boundary) to reach the process and then head back to the NIC, a significant amount of CPU overhead is incurred. Although there exist optimizations (such as *splice()/sendfile()* in Linux) to avoid kernel-user boundary crossing overhead, much of the overhead is caused by the data's crossing the VMM-VM boundary, which is also known as the device virtualization overhead. (2) When the VM running the web server is sharing CPUs with other VMs in the host (i.e., VM consolidation), this VM may not always be scheduled, resulting in delays in processing events posted by the VMM regarding the availability of data and completion of I/O operations hence causing degraded I/O performance.

Device Virtualization Overhead In a virtualized system, either the VMM or a privileged domain, called driver domain, uses its native device drivers to access I/O devices directly, and performs I/O operations on behalf of other guest VMs. The guest VMs use virtual I/O devices controlled by paravirtualized drivers to ask the VMM for device access. VMs that need to read from or write to a device have to follow a number of steps involving data copying, page sharing (which requires page table and grant table modifications), virtual interrupt processing, and protection domain switching (VMM to VM mode) in order to make the VMM access the device on behalf of the VM. As noted by many existing efforts [16, 20, 23], this I/O model causes significant performance penalty for I/O-intensive workloads in VMs. VMM-bypassing I/O devices such as SR-IOV-enabled NICs can eliminate this overhead by directly assigning devices to VMs. However such devices inhibit the interposition of the VMM on I/O paths to perform QoS and security tasks [8, 15]. Hence paravirtualized I/O is still widely used in virtualized clouds.

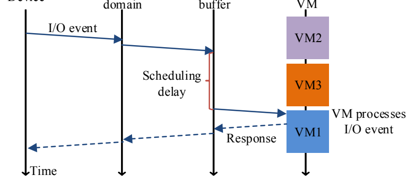


Figure 2. VM scheduling delay affecting I/O event processing.

VM Scheduling Latency While device virtualization causes high CPU overhead for I/O processing and results in I/O performance degradation, research in [9, 19, 26, 27] suggests that the main cause of *latency* in I/O event processing under virtualization is VM scheduling, not device virtualization. In an OS directly running on a physical machine, pending I/O events get processed almost instantaneously because I/O processing holds higher priority in most modern OSes. However, for a VM sharing CPU cores with other VMs, I/O processing may be significantly delayed because this VM may not be scheduled when its I/O event (e.g., network packet arrival) occurs. An example in Figure 2 shows a device sending an event signaling the availability of I/O data for VM1, when VM1 is sharing a CPU with two other VMs (VM2 and VM3). The VMM layer will first process the event and then place the event in a shared bus connecting to VM1. However, at this moment VM1 is in the CPU runqueue waiting for its turn for the CPU. Once VM1 gets scheduled, the event can be consumed and a response will be sent to the device. As shown in the figure, the VM scheduling delay is a major factor in the overall I/O processing latency. Since each VM scheduling slice is typically in the order of tens of milliseconds, the VM will experience equally high latency for I/O event processing, which would otherwise be in the range of sub-milliseconds.

2.1 Key Observation

Based on the above examples and analysis, we make a critical observation: If there were an API via which the application could request the VM's kernel, which in turn would instruct the VMM, to transfer data directly from disk to NIC on behalf of the application, both of the problems with I/O processing discussed earlier could be avoided. First, it would avoid device virtualization overhead because the data movement operation happens inside the VMM layer and hence data copying, page sharing, virtual interrupts processing, and protection domain switching would be unnecessary. Second, it would avoid delays caused by VM scheduling because the VMM component performing the data movement is scheduled more often than the guest VMs to perform interrupt processing and other management tasks. In fact, an earlier, primitive version of vPipe, which could only perform file-to-socket transfers proved to be effective in improving I/O throughput while reducing CPU resource consumed [10].

3. Design

vPipe essentially offloads the actual data transfer segment of the data movement operation – originally performed by the application in a VM – to the VMM. The application instructs the VM's kernel to construct a "pipe" between designated source and destination devices. The VM's kernel gathers necessary meta-data about the source and the destination which are needed by the VMM (or the driver domain) to interpret the data on disk as well as data coming from or going towards the NIC. The driver domain, which has access to the physical devices and the drivers controlling them, can then construct a pipe between them using the meta-data. For

API Function	Input Parameters	Return Value	Description
vpipe_create()	in_fd, out_fd	vPipe descriptor	Creates a vPipe descriptor instance for a given source and a destination descriptors
vpipe_socket_to_file()	vPipe descriptor, file offset, length	Bytes written to out_fd	Perform a vPipe operation with a socket as the source and a file as the destination
vpipe_socket_to_socket()	vPipe descriptor, length	Bytes written to out_fd	Perform a vPipe operation with a socket as the source and another socket as the destination
vpipe_file_to_socket()	vPipe descriptor, file offset, length	Bytes written to out_fd	Perform a vPipe operation with a file as the source and a socket as the destination
vpipe_file_to_file()	vPipe descriptor, in file offset, length, out file offset	Bytes written to out_fd	Perform a vPipe operation with a file as the source and another file as the destination
vpipe_destroy()	vPipe descriptor	0 on success, failure code otherwise	Deregisters the vPipe descriptor and deallocates memory associated with it

Table 1. vPipe API.

applications that insert new data into the data stream, such as a web server sending dynamic content along with static files, there needs to be sufficient flexibility in vPipe to let the VM assume control of the two ends of the pipe when required.

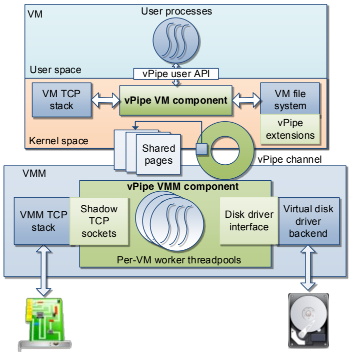


Figure 3. vPipe system design.

We concretize the above conceptual design with two main components of vPipe as shown in Figure 3: (1) a vPipe VM component that resides in the VM's kernel space, whose responsibility is to interact with the application, collect metadata regarding the source and destination, and call the driver domain to carry out the requested piped I/O; and (2) a vPipe VMM/driver domain component which accesses the physical devices and performs the actual data movement operation on behalf of the VM. The vPipe VM component interacts with applications by providing a special device and a set of library calls which wrap the access to this device to provide easy access to the vPipe interface. The vPipe VM component and VMM/driver domain component communicate with each other via a standard inter-domain communication channel used by paravirtualized device drivers. We will discuss each of these components and their interfaces in the remainder of the section.

3.1 vPipe API

First we outline the vPipe API for cloud application developers. The vPipe API is a set of functions provided in a user-level library (*libvpipe*) which hides the complexity of interacting with the vPipe VM component from applications. This library provides six main functions shown in Table 1 that correspond to each type of piped I/O operation. An application can associate a handle for vPipe operations by calling the *vpipe_create()* function with the input file descriptor and the output file descriptor. By doing this ahead of time, the application can save time for socket or file descriptor lookup by the vPipe VM module, especially for processes that perform vPipe operations on the same set of file descriptors periodically (e.g.,, a proxy server supporting persistent connections). The set of *vpipe_source_to_destination()* calls instruct the VM component to perform the four types of supported vPipe operations. Currently these functions are blocking calls even if the corresponding file descriptors are working under non-blocking mode. Implementing non-blocking functionality is left for future work, although conceptually it is quite similar to the blocking call. Finally, the *vpipe_destroy()* function deregisters the vPipe descriptor and deallocates any memory associated with the descriptor. In our current implementation we require the application to explicitly call *vpipe_destroy()*, as we do not have a timeout mechanism to detect applications which are not responding. However, destroying the application (hence the process) would trigger destroying the vPipe handles registered to that process. We have another set of similar *vpipe_source_to_destination()* functions which do not require a handle – they will create a handle on the fly and destroy it once the vPipe operation is finished. We realize that this API is somewhat similar to Linux *sendfile()* API (from Linux 2.6.33 onward) and adapting *libvpipe* to be compatible with sendfile() API is left as future work.

We also develop a Java user API of vPipe to be used with applications such as Apache HDFS. This Java API is a wrapper on top of *libvpipe*, developed using Java native interface (JNI). This library locates the socket and file descriptors associated with Java *Socket* and *FileInputStream* / *FileOutputStream* objects and calls the corresponding *libvpipe* function to perform the offloaded operation.

3.2 vPipe VM Component

The vPipe VM component resides in the VM's kernel context and acts as the "glue" between the application which requests the vPipe operation (via *libvpipe*) and the driver domain component which carries out the operation. It also performs the essential task of communicating with the VM's I/O subsystem to locate the meta-data about the source and destination of an offloaded operation. In the current design of vPipe, we focus on files on disk and TCP sockets. Despite the design details below, we point out that the vPipe VM component is *modular and lightweight*, involving just one loadable module plus 200 lines of modified kernel code, which we believe is acceptable in practice, given the significant performance gains achieved by vPipe.

Offloading File I/O In most file systems, files are represented by a set of descriptors (*inodes*) which contain information about the file such as size, permissions, block device the file is associated with, pointer to the descriptor describing the file system information and a set of identifiers pointing to the physical blocks which contain the file's content. Even though an access to the file requires accessing most of the information contained in these inodes, accessing raw data can be done by just reading/writing the data blocks in the physical device, which can be performed by the driver domain even though it does not have access to the inode information located in the VM's kernel. (It is possible however to access the VM's kernel memory using VM introspection techniques [11] – but the overhead of such techniques is usually very high for throughput/latency-sensitive tasks.)

File read: When a file read operation is requested at one end of the vPipe data movement pipe, the VM module communicates with the VM's file system requesting physical block identifiers of the corresponding file. In our Linux-based implementation, this is done by invoking the bmap() function of the file object. Most modern file systems use block sizes around 4KB. Given that cloud applications such as Hadoop often involve file size of GBs or larger, we expect the size of this identifier data structure to be large. Fortunately, most file systems store files in a few consecutive ranges of blocks, known as file extents. Hence by locating the first block and the range size we can access the entire range without requiring the identifiers of individual blocks. Once all the extents are retrieved from the VM's file system, the vPipe VM module can pass this information along with the block device identifier to the driver domain for reading the file blocks.

File write: Writing a file requires handling a few more details. In particular, if the write operation is an append operation (writing to a new file can be viewed as creating an empty file and then appending to it), the VM's file system requires allocating new empty data blocks and send their identifiers to the vPipe VM module. This allocation involves operations such as locating free disk blocks, modifying file system-specific data structures, and performing journaling and other consistency measures, which are specific to each file system type. Hence, higher-level file system APIs do not provide methods to create such empty data blocks. vPipe modifies the lower-level file system layer as well as the high-level generic file system layer as shown in Figure 3. Porting vPipe to a specific file system directly depends on how difficult it is to implement this functionality of that file system. Specifically, we require to add methods to allocate empty disk blocks without the data associated with those blocks, to perform necessary housekeeping required by the specific file system such as adding/removing orphan nodes, and starting and committing journal transactions. Since vPipe is unable to know how much space is needed when allocating the empty blocks in advance, we currently let the application decide the number of blocks to allocate at once. (In our experience, larger batch size leads to lower CPU utilization.)

For file overwrite operations, the vPipe VM module has to flush dirty pages associated with the file and mark them as invalid, so that the VM's file system will not overwrite the blocks written by vPipe's driver domain component later while flushing the disk cache. Once the necessary block identifiers are created with the help of the file system, the vPipe VM module can pass these identifiers to the driver domain which will perform the actual file write operation.

An optimization to the file reading operation is to use the disk cache at the VM level when possible. Reading blocks from the disk cache is much faster than reading from the disk drive. Hence, even with virtualization overhead, it is better to use these cached blocks if there exist a sizable range of consecutive blocks. However, if the number of cached blocks is small and they are not the first blocks to be transferred, we would have to first perform a vPipe operation to transfer data up to the cached data blocks, send the cached blocks from the VM, and finally perform another vPipe operation to transfer the rest. Communicating with the driver domain and transferring control to it multiple times introduces overhead and hence would defeat the savings obtained by the usage of disk cache. The same reasoning applies even when the number of cached blocks are high, but scattered in the range to be transferred. In our current design we use the disk cache only if the continuous cached page range exceeds $max\{1MB, 10\% \times file\_size\}$. Reading contents from the disk cache is also required in cases where the content of the disk cache is marked dirty. In both read and write operations, we mark the file inode as read/write locked correspondingly (e.g.,, by performing *down* on *i_alloc_sem* field of the inode) to prevent race conditions.

Offloading TCP Sockets If either end of the offloaded piped I/O operation is an established TCP socket, vPipe will offload the entire TCP processing functionality to the driver domain by establishing a shadow socket in the driver domain using the driver domain's TCP stack. This approach is somewhat different from the approach to offloading disk file access, in which we perform file system-specific tasks at the VM layer and restrict the driver domain to just read and write raw physical blocks. A less drastic solution along that line would be letting the guest VM fabricate placeholder TCP packets ahead of time that are then filled with data in the driver domain. However, that would not work here as TCP packets coming from or going to the NIC have metadata associated with them (e.g., sequence numbers and window sizes) and proper handling of a TCP connection requires some processing of the meta-data before sending or receiving data. Even with this partial offloading of TCP processing, important functions such as congestion control have to be performed by the driver domain anyway, otherwise the benefit of offloading would be completely lost. So we decide to handover the entire TCP socket to the driver domain during the vPipe operation and then pass it back to the VM along with the updated information once the vPipe operation is over. An application performing "piping" operations and sending *generated* data in between can invoke a sequence of {vPipe, *send()*} operations.

In order to offload a TCP socket as one end of a vPipe operation, the application first needs to have a valid TCP socket by establishing the connection via either *connect()* or *accept()*. When getting this socket file descriptor from the application, the vPipe VM module, with the help of VFS and the VM's TCP stack, can convert this descriptor to a socket structure with the necessary information to establish the shadow socket in the driver domain. Such information includes the source and destination IP addresses, source and destination ports, last sent and received sequence numbers, and window size. We also collect the congestion control information from the VM's TCP stack, such as the congestion window, ssthreshold (slow start threshold) and the current mode of the connection (either slow start or congestion avoidance), so that we can continue with the same congestion information at the driver domain instead of beginning from slow start to build up the connection. However, before collecting this information from the acquired socket structure, we have to make sure that all the packets sent by the VM before the vPipe operation are acknowledged by the other end. This is because if there was a loss among those packets, we would not be able to recover the loss once the socket was offloaded to the driver domain – with the socket now residing in the driver domain while the data to be retransmitted is in the VM. So we perform a busy wait on the socket until all the sent data packets are acknowledged.

There is another subtle issue when an offloaded TCP socket is the source end of the pipe. In this case, as soon as the TCP connection is established by the application and just before establishing the vPipe offload operation, the other end of the connection might start sending data while the VM's TCP socket is still active. This would make the offloading complex if the other end keeps sending data to the VM, while the shadow socket is being created. To prevent this from happening, when a TCP socket is passed as the input of the pipe, we will install a packet filter rule at the IP layer of the VM to drop packets coming from that connection. (Once the shadow socket is established, the sender's TCP retransmission mechanism will resend those packets). Then we check the VM's TCP buffers to see if there are any received data from the other end, starting when the vPipe operation is invoked by the application and ending when the vPipe VM module installs the packet-dropping rule. If there are any data segments in the buffer, we will move them first to the destination of the pipe and adjust parameters such as the length of the operation, starting block of the file (if the destination is a file) and so on.

3.3 vPipe VMM/Driver Domain Component

The vPipe VMM/driver domain component (or driver domain component) is responsible for carrying out vPipe operations offloaded by guest VMs in the host. The component interacts with the TCP stack of the driver domain and with the virtual disk driver backend which exposes VM disk images to guest VMs. At system initialization, the driver domain component enumerates all the active virtual disks and initializes a device descriptor for each of them. This allows the driver domain component to interact with these virtual disks immediately for vPipe operations. It also creates a hook to the VM management subsystem to notify the addition or removal of virtual disks when VMs are created or destroyed. We allocate a pool of pages as the buffers associated with each of these descriptors to perform vPipe operations – we use a statically allocated pool of pages so that the page allocation time is minimal when a vPipe operation is offloaded to the driver domain component.

The vPipe operations are encapsulated in vPipe descriptors. A thread from the VM-specific thread pool (Figure 3) is assigned to carry out each vPipe operation. The thread is responsible for carrying out one vPipe operation at a given time – we adopt this model instead of an event-driven model for simplicity.

The driver domain component receives offloading requests from the vPipe VM component via an inter-domain communication channel, which is similar to those used in implementing paravirtual device drivers (e.g.,, Xen's ring buffer mechanism). This channel uses a ring buffer to hold the descriptors for shared pages which describe the vPipe operation with information such as the type of operation (i.e., socket to socket, socket to file, file to socket, or file to file), the meta-data about the source and the destination, and the length of the operation (in bytes). A virtual interrupt mechanism (such as Xen's inter-domain events) is used to convey the availability of a new vPipe operation in the ring buffer from the VM, or the completion of an operation from the driver domain.

File I/O When the driver domain component receives an offloading request with a file as one end of the pipe, it first verifies the validity of the request by checking whether the supplied block identifier range falls within the VM's virtual disk image. The requests passing this check are assigned a set of pages from the page buffer to carry out the vPipe operation. If the file is the source end of the pipe, disk read requests are created with the block identifiers and memory pages and submitted to the disk driver in batches. The batch size is determined by the queue length of the virtual disk driver. When the read requests complete, the write function of the pipe descriptor is called along with the associated pages. If the file is the destination of a pipe, the write function receives a set of pages filled with data to be written. These pages are used to create a set of disk write requests along with the disk block identifiers received from the vPipe VM component and submitted to the virtual disk driver in batches. Both read and write operations are carried out in a loop until the supplied length of the vPipe operation is exhausted.

TCP Sockets The driver domain component uses the driver domain's TCP stack to establish "shadow sockets", which are replicas of the original sockets in the VM contexts. During the vPipe operation, the original socket at the VM level is disabled so that the user process will not be able to use it to send or receive data.

There are three high-level steps to create a shadow socket: First, a socket structure is created by the driver domain component in the driver domain's TCP stack using the standard socket kernel interface. This socket however, is not a complete socket structure since we have not connected it to a peer using either *connect()* or *accept()* functions. It is merely a template socket created in the driver domain's kernel space.

Second, the driver domain component associates the meta-data received from the VM with this empty socket structure. Since the kernel socket interface at the driver domain does not have APIs to populate sockets directly with information such as source and destination IP addresses, source and destination ports, and window information, we slightly modify the driver domain's TCP stack, which allows us to directly manipulate the sockets and supply relevant information when a shadow socket is created.

Finally, forwarding entries are added to the driver domain's IP routing subsystem and to the bridging subsystem so that the packets destined to an offloaded socket are routed to the driver domain's TCP stack rather than to the VM. Marking the shadow socket as *ESTABLISHED* at the end of these three steps allows the shadow socket to be fully functional. The driver domain component, depending on whether the socket is the source or destination of a pipe, can perform *recv()* or *send()* operations on this socket.

3.4 Sharing the Driver Domain

We must ensure that the vPipe worker threads working for different VMs (Figure 3) are sharing the driver domain's resources fairly, as the VM's I/O workloads are now carried out by the driver domain. Specifically, we should "charge" the work done by these threads to the VMs requesting vPipe operations. Lack of such accounting and control will lead to unfairness in carrying out vPipe operations and may allow some VMs to gain unfair advantage in the shared system. The native process scheduler in the driver domain alone cannot handle this task because it is not aware of the user-set priorities of VMs which vPipe has to honor.

We address this challenge by using a simple credit-based scheme. Each per-VM thread pool of the vPipe driver domain component is allocated a certain amount of credits based on the priority (weight) of the corresponding VM. These credits are consumed by the threads as they perform I/O operations, based on the number of bytes transferred. When the threads run out of credits, they wait for a function that runs periodically to add more credits to them. The running frequency of that function is tunable to make fine grained adjustments for fairness. In situations where a work conserving property is desirable, we assign a very low priority to the threads that have exhausted their credits, until the credits are replenished. The total amount of credits divided among the thread pools is calculated by running a calibration task during vPipe's initialization. This calibration process takes into account the maximum I/O capacity of the virtualized host. One limitation of this scheme is that, it cannot perform accounting for the VMs' regular I/O operations. We believe that, for these I/O operations, the existing I/O scheduler at the driver domain is sufficient.

4. Implementation

We have implemented a prototype of vPipe with Xen 4.1 as the VMM and Linux 3.2 as the kernel of the VMs and the driver domain. The vPipe VM component is implemented as a loadable Linux kernel module plus minor changes to the kernel. The module uses Linux VFS functions to manipulate file descriptors and to locate kernel socket data structures when a process makes vPipe calls using file and socket descriptors. Since most of the VFS functions, data structures, and kernel socket structures are exposed to kernel modules, we do not have to make any changes to guest kernels for implementing pipes involving file read and socket read/write operations.

For file write operations, we slightly modify both the *ext3* file system and the generic VFS layer. Two functions are added to the ext3 inode operations. The first function, *ext3_vpipe_dio_start()*, allows the creation of new block identifiers for append operations without any data pages. It also takes care of creating an ext3 journal transaction and adding the file's inode to the list of orphan nodes so the file system could be rolled back to a previous state in case of an abandoned write operation. The second function, *ext3_vpipe_dio_end()*, which gets called once the actual data write is done by the driver domain component, performs clean up operations on the temporary state created by the write operation, such as committing the journal transaction, extending the file size (in case of a successful write), truncating the extra data blocks (if the driver domain returns an error or returns with less number of blocks than allocated), syncing inode to the disk, and removing the inode from the orphan list. However, our current implementation of these functions does not support *data journaling mode* of ext3. The changes made in the generic file system (VFS) layer involve adding pointers to the file's address space operations structure so that the above functions can be called from the VFS layer and adding functionality to flush and invalidate pages which have dirty content in case of a file overwrite.

Similar to the vPipe VM component, the vPipe driver domain component is also implemented as a loadable Linux kernel module. We make changes to the driver domain kernel – this time for creating offloaded sockets. Our modifications include changes to the Linux generic sockets layer (*sock_create_shadow()*) – to create shadow sockets with meta-data supplied by the VM; to the inet_stream layer (*inet_stream_shadow_socket()*) – to perform IP-specific initialization for the shadow socket; to the TCP v4 subsystem (*tcp_v4_shadow_socket()*) – to perform TCP-specific initialization for the shadow socket such as setting sequence numbers and congestion information, initializing buffers etc.; to the IP routing layer (*ip_rt_add_shadow_flow()*); and to the Ethernet bridging subsystem (*br_input_add_shadow_flow()*) – to forward packets destined to an offloaded socket to the driver domain's TCP stack. The vPipe driver domain module interacts with the Linux block I/O subsystem using the *bio_*()* API. Each vPipe operation is carried out by a Linux kernel thread (*kthread*) picked from the per-VM thread pool.

The application interface of the vPipe VM module is a special device in the VM's device directory (*/dev/vpipe*). A user-level process may call *ioctl()* on this special device to interact with the vPipe VM module. However, as discussed in Section 3.1, we wrap this interface with a set of C function calls and Java methods for application programming convenience.

5. Evaluation

This section presents our evaluation of vPipe using both microbenchmarks and cloud applications.

Evaluation Setup Our testbed consists of multiple servers, each with a 3GHz Intel Xeon quad-core CPU and 16GB of memory. These servers run Xen 4.1.2 as the VMM and Linux 3.2 as the OS for all guest VMs and the driver domain. To make sure that the driver domain gets enough CPU time to serve I/O, we pin the driver domain to one physical core, following the recommended setup for Xen-based servers. We also pin the guest VMs to another physical core to create VM scheduling effects. We use lookbusy [3] to generate deterministic CPU workloads.

5.1 Microbenchmark Performance

File Send Throughput We use a simple program that reads from a file on the disk and writes to the socket connected to a client running in another host. The application uses two modes to transfer the file to the client: (1) via *sendfile()* system call and (2) via *vpipe_file_to_socket()* library function.

Figure 4(a) shows the throughput improvements achieved by vPipe transferring a 512MB file when the VM running the application is co-located with 0, 1, 2, and 3 other VMs. When only the application VM is running on the core, both *sendfile()* and vPipe modes can reach the full available bandwidth of the 1Gbps link. As the number of VMs sharing the core increases, throughput drops for the *sendfile()* mode. However, since vPipe offloads the processing of the entire I/O operation to the driver domain, the throughput remains the same regardless of the number of co-located VMs.

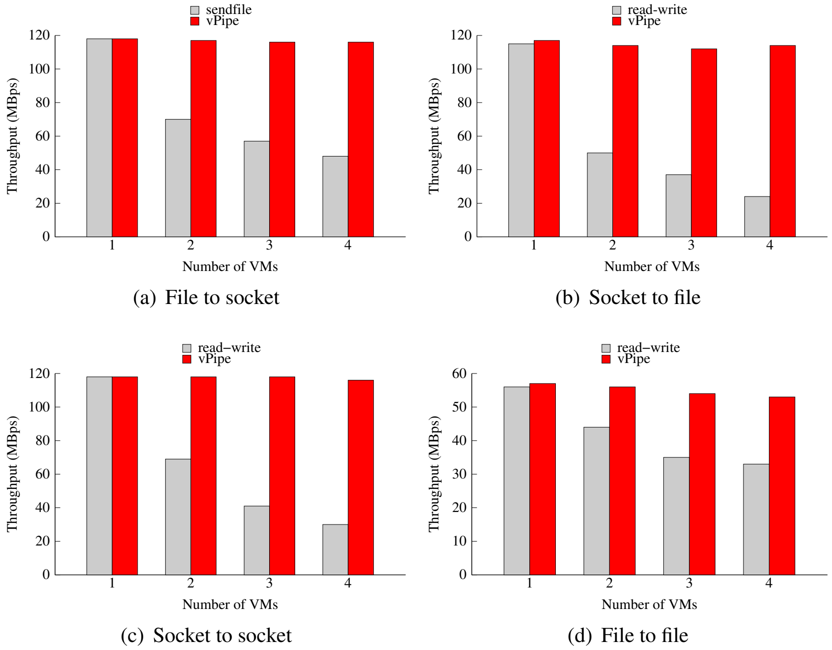
Figure 5(a) shows the results when we vary the file size being transferred while the application VM is sharing the CPU with two other busy VMs. The improvements are quite stable over the file sizes from 64MB to 1GB, with the throughput under the *sendfile()* mode slightly improved for the larger file sizes.

File Receive Throughput For this experiment, we use a simple application that receives a file sent from a client running in a different host and writes it to the disk. The file is opened with *O_SYNC* flag and hence the writes are directly flushed to the disk, avoiding any effects of Linux disk buffer cache. Figure 4(b) shows the throughput improvements achieved by vPipe when the number of VMs sharing the core with the application VM varies from 0 to 3, while a 512MB file is transferred from the client to the application VM. The results in Figure 5(b) show that the throughput improvements are stable over various file sizes (from 64MB to 1GB), when the application VM is sharing the CPU with two other VMs.

Connecting Two Sockets This experiment emulates a proxy server which performs connection forwarding. In this experiment, we have an application consisting of three components – a server component which accepts a connection from a client and writes a file from the disk to the connection, a client which connects to a server and receives the file, and a proxy which forwards the connection from the client to the server. We run both the server and client components on physical machines while running the proxy in a VM, which uses (1) read-write mode and (2) vPipe to perform the connection forwarding. Figure 4(c) shows the throughput improvements achieved by vPipe, when the number of co-located VMs varies from 0 to 3, while a 512MB file is transferred from the server to the client via the proxy.

Figure 5(c) shows the results of the experiment when we vary the file size while the proxy VM is sharing the CPU with two other busy VMs. The results show that the improvements are independent of the file size and mainly depend on the number of VMs sharing the same CPU core.

File Copying Throughput We modify the Linux *copy (cp)* utility to use vPipe for this experiment. We also create two disk images for the VM so that we can emulate an application copying a file from one disk to another during a backup operation. Figure 4(d) shows the throughput improvements achieved by vPipe when copying a 512MB file, with the number of co-located VMs varying from 0 to 3. In this case, even though vPipe shows improvements over the vanilla Xen configuration, the improvements are not as significant as in



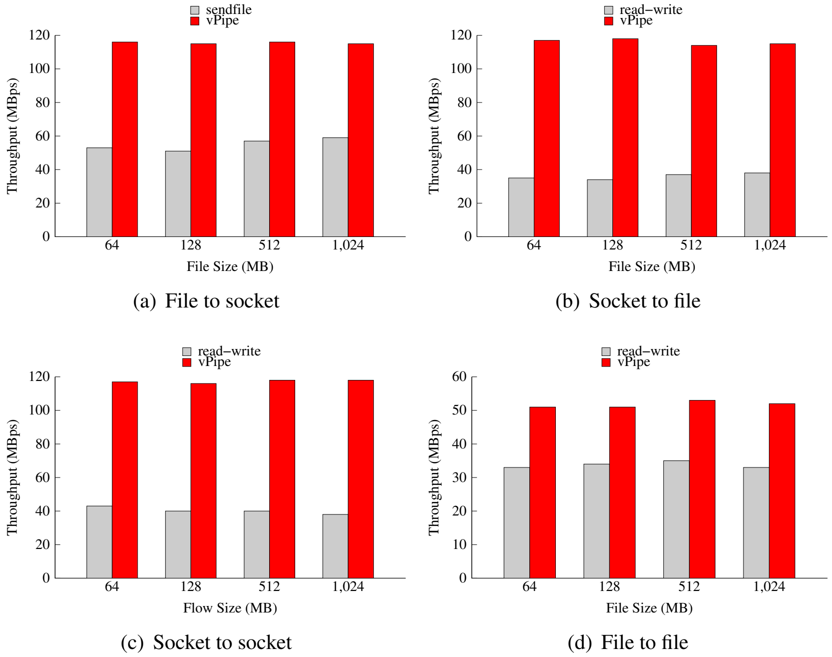
(a) File to socket

(b) Socket to file

(c) Socket to socket

(d) File to file

Figure 4. Throughput improvement with different number of guest VMs per core.



(a) File to socket

(b) Socket to file

(c) Socket to socket

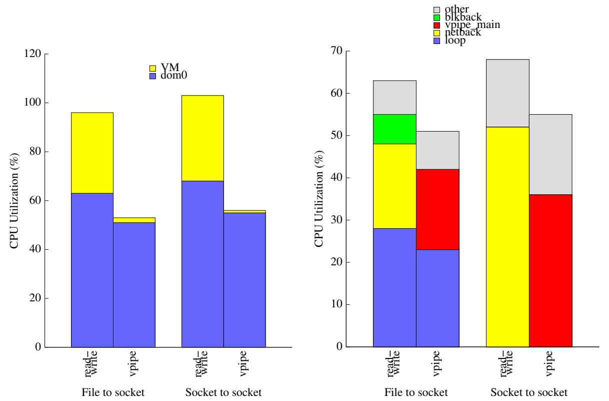
(d) File to file

Figure 5. Throughput improvement with varying data size.

the earlier three experiments. The main reason is that, during the file copy operation, a high volume of read and write requests are issued to the same physical disk, making the physical disk itself the bottleneck on the data path.

Figure 5(d) shows the throughput improvements when the file size varies from 64MB to 1GB, with two other co-located VMs. The results show improvements similar to the earlier experiments, and the trend shows that the gains are independent of the file size.

CPU Savings Figure 6(a) shows the average CPU utilization when performing (1) a file to socket operation and (2) a socket to socket operation for 1GB of data. As expected, the VM's CPU utilization under the read-write mode is higher since it requires copying data across all layers. vPipe incurs almost no CPU cost at the VM level because 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 6. CPU utilization.

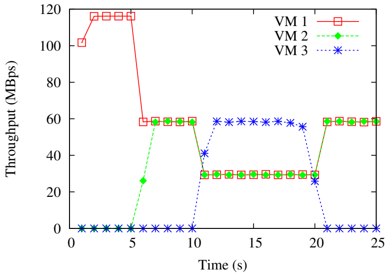


Figure 7. I/O intensive VMs sharing the driver domain.

vPipe saves CPU cycles at the driver domain as well. Figure 6(b) shows the breakdown of CPU utilization of the driver domain kernel threads involved in I/O processing for the VM. In "file to socket" mode, the *loop* thread, which is responsible for reading blocks from the VM image (block device emulation), is similar in both the vPipe and vanilla cases. The slight difference is due to the following: When the VM performs read-write/sendfile, the amount of disk requests issued per one VM-to-dom0 switch is small, hence leading to many small disk requests. However, vPipe batches as many requests as the disk allows to cause less overhead while performing disk requests.

The *netback* thread, which is responsible for emulating the network device for the VM takes up about 20% utilization for the read-write mode. The *blkback* thread, which emulates the disk for the VM, takes about 7-9% utilization in both modes. These device emulation threads are not a factor in vPipe since the data do not cross the VMM-VM boundary. In "socket to socket" transfer mode there are two *netback* threads involved in the vanilla case to carry out data transfer for the two virtual NICs. We see that these backend drivers use even more CPU than the vPipe components in the driver domain. Backend drivers use data copying between dom0 and the VM and they use an interdomain event channel to signal. Both of these operations are CPU costly. Also with vPipe we perform much more data batching compared to the read-write mode.

Sharing the Driver Domain We now evaluate our credit-based scheme for sharing the driver domain fairly among multiple VMs requesting vPipe I/O operations. In this experiment, we have three VMs performing file-to-socket vPipe operations on the same physical machine. The weight ratio among VM1, VM2 and VM3 is set to 1:1:2. We start with VM1 sending a file on disk to a client using vPipe. After 5 seconds, VM2 starts sending another file on disk to another client. After another 5 seconds, VM3 starts sending a file to a third client using vPipe.

Figure 7 shows the throughput (1-second average) achieved by each of the three operations. Initially VM1 gets about 117MBps (close to the wire speed of the 1Gbps network) with no other VM to compete with. When VM2 starts the vPipe operation at the 5th second, both throughputs settle at about 58MBps because of their 1:1 weight ratio. When VM3 starts its vPipe operation, it gets about 58MBps while VM1 and VM2 get about 29MBps each, consistenting with the 1:1:2 weight ratio. At the 21st second, VM3 finishes its I/O operation and the throughputs of VM1 and VM2 go back up to 58MBps each.

5.2 Application Performance

Web Server System with a Load Balancer In this experiment, we create a load balanced web server system consisting of two backend web severs running Lighttpd [2] and a load balancing server running Pound load balancer [5] in front of them. Pound distributes the requests to the backend web servers using a round robin algorithm while the Lighttpd servers serve various sized files located in the disk. We modify the Lighttpd web server to use the *vpipe_file_to_socket()* function instead of *sendfile()* when serving static files. We also modify the Pound load balancer to use the *vpipe_socket_to_socket()* function instead of the *send()-receive()* pair when forwarding connections between the backend servers and the clients.

We populate both web servers with exactly the same set of static files. We use httperf to generate repeated requests for files of size 1MB, 10MB, 100MB, and 1GB to stress test the web server system. Figure 8 shows the results of our experiment under four configurations. The "Xen" scenario involves the vanilla Xen running with no optimization. In "vPipe-LB" scenario, we install vPipe only on the host running the load balancer. In the "vPipe-web" case we enable vPipe on hosts running the two web servers, but not on the host running the load balancer. Finally, in "vPipe-both" configuration, we enable vPipe on all the hosts.

As shown in Figure 8, all vPipe-* configurations outperform the Xen configuration for all file sizes. The vPipe-both configuration gives the best throughput improvement (up to 2.9 ×) as it uses file-to-socket pipe to improve the web servers' throughput and the socket-to-socket pipe to improve the load balancer's throughput. vPipe-LB configuration outperforms vPipe-web, because the load balancer is the bottle-

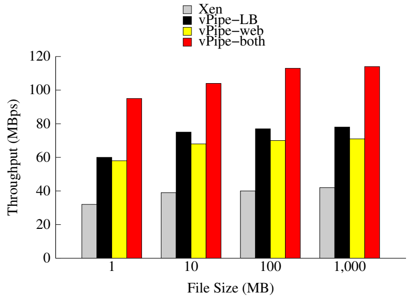
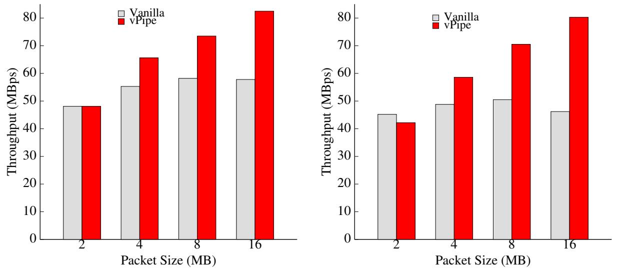


Figure 8. Throughput improvements for web server system.

neck of the system hence optimizing the load balancer better improves the overall system's throughput.



(a) HDFS node no extra CPU load (b) HDFS node 30% extra CPU load

Figure 9. HDFS throughput improvements by vPipe.

Apache HDFS In this experiment, we apply vPipe to the widely used cloud file system HDFS. We measure the read performance of the HDFS server as it is more common in HDFS' write-once-read-many model.

HDFS stores data in different datanodes to provide high fault tolerance and throughput. When one client reads a file, the HDFS server reads the actual data from the corresponding datanodes and send them to the client via socket. This pattern matches vPipe's file-to-socket mode, so we use the vPipe file_to_socket API in the HDFS datanode using our vPipe Java library. More specifically, we modify HDFS' *sendChunks()* function to use the vPipe API.

The read/write unit of each datanode is an HDFS packet. We vary the packet size from 2MB to 16MB and measure the read throughput for each size. For simplicity, we only run one HDFS datanode and one client. The HDFS client and server are running on two VMs located in two different physical hosts. Each VM shares one core with 2 other VMs which only generate 35% CPU load. In our experiment, the client reads 1GB of data from the HDFS datanode each time. The results are shown in Figure 9(a). When the packet size is larger than 2MB, vPipe effectively improves the read throughput (up to 42%). For the smaller packet size (i.e. 2MB), the overhead of offloading the operation offsets the performance gain from vPipe. Figure 9(b) shows the HDFS read throughput when we add 30% CPU load to the datanode itself (e.g., running a mapper/reducer in the datanode).

Video Streaming We modify GNUMP3d [1], an open source streaming server for MP3s, OGG vorbis files, movies, and other media formats, to use vPipe when serving video files. We run GNUMP3d in a VM hosted by a server along with 2 or 3 other core-sharing VMs. We use real video files (size exceeding 500MB) of MPEG-4 format in our experiments.

First we evaluate the number of simultaneous clients that can be supported by GNUMP3d while running with the default Xen configuration and with vPipe. We use MPlayer [4] in command line mode running in 3 other physical machines, with video and audio output directed to the *null* device. We measure the number of simultaneous MPlayer instances which can be supported by GNUMP3d without causing interruption in playback (i.e., "buffer empty" events). Columns 2 and 4 of Table 2 show the results when the VM running GNUMP3d shares the same core with 2 and 3 other VMs, respectively.

Next, we evaluate the throughput improvement of GNUMP3d by vPipe by measuring the time to buffer a complete movie of 750MB. Columns 3 and 5 show the time to buffer the movie file by a media player, when the VM running GNUMP3d shares a core with 2 and 3 other busy VMs, respectively.

	3 CPU sharing VMs		4 CPU sharing VMs	
	Max. no. of clients	Buffer time (s)	Max. no. of clients	Buffer time (s)
Xen	910	11.7	780	13.6
vPipe	1303	8.1	1216	8.3
% Improvement /(Reduction)	**43.1**	**(31.6)**	**55.8**	**(38.9)**

Table 2. Performance improvement for GNUMP3d.

Compared with default Xen configuration, GNUMP3d with vPipe achieves 43% and 56% improvements for the number of simultaneous clients for the 3-VM and 4-VM configurations, respectively. From the client's perspective, GNUMP3d with vPipe achieves 32% and 39% reduction of movie buffering time for an individual client.

6. Discussion

Offloading Code with vPipe Operations Our application case studies above have demonstrated vPipe's applicability. However, vPipe can be further enhanced to support data processing on the piped data I/O path. For example, a proxy server may encode/decode data coming from the backend before sending them to the clients. Such encoding/decoding function could be executed by the driver domain as part of the vPipe operation, reducing a significant amount of data crossing the VMM-VM boundary. Challenges in offloading such a data processing function include the recreation of proper execution context for its execution in the VMM (as if it were executed in the VM), as well as the isolation of its impact from the rest of the VMM (for security). We leave this as our future work.

Interplay with Modern Hardware Modern hardware techniques such as SR-IOV can eliminate most of the device virtualization overhead by directly transferring data to the VM's memory without crossing the driver domain. However, the VM will still be subject to VM scheduling delays and hence there is still room for vPipe to optimize data movement performance. The new challenge is that these devices bypass the driver domain and hence it would be difficult to route packets towards the shadow socket once a vPipe operation has been initiated. We envision the capability of programming the hardware to forward packets from a matching connection to the driver domain – but such capability is not yet supported. [16] also discusses the disadvantages of such direct access devices when driver domain interposition is necessary to perform functions such as firewalling and rate limiting.

7. Related Work

Reducing Device Virtualization Overhead There exist many previous efforts that focus on reducing the overhead associated with device virtualization along the data path of I/O processing. In [20] Menon *et al.* propose several optimizations such as scatter/gather I/O, checksum offload and TCP segmentation offload (TSO) to improve TCP performance in Xen VMs. In [23] they propose packet coalescing to reduce the overhead of TCP per-packet processing cost in VMs and hence improve TCP receive performance. [22] proposes performing part of the network device's functionality at the hypervisor level to reduce CPU overhead incurred by network packet processing.

Similarly, Gordon *et al.* propose exit-less interrupt delivery mechanisms to alleviate interrupt handling overhead [12, 16] 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 [7] based on the number of in-flight commands to the disk controller. Virt-FS [17] presents a para-virtualized file system as an alternative to NFS or CIFS which allows sharing the driver domain's (or host's) file system with VMs for 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 piped I/O at the VMM layer.

Reducing VM Scheduling Latency Since VM scheduling delay can significantly affect a VM's I/O processing throughput as well as application-perceived latency in VM consolidation environments, many previous projects have focused on minimizing VM scheduling delay for I/O-intensive applications. vSlicer [27] categorizes VMs into latency-sensitive VMs and non-latency-sensitive VMs and schedules the latency-sensitive VMs more frequently for shorter period of time in each scheduling period, thereby reducing the application-perceived latency. vTurbo [26] offloads VMs' I/O processing to a dedicated core with very small time slices, so that the scheduling delay for virtual CPUs in this core is extremely small. The work in [18] exploits the homogeneous nature of VMs running MapReduce tasks and proposes grouping these VMs and sorting the runqueue based on such grouping. It also involves batching I/O requests to reduce context switches between the VM and VMM. Govindan *et al.* propose [14] preferential scheduling of communication-oriented applications over their CPU-intensive counterparts to reduce network receive latency and anticipatory scheduling to reduce network transmit latency. The primary focus of all these efforts is to reduce the scheduling delay of I/O-intensive VMs and thereby to either reduce the latency or increase the throughput of I/O processing. However, they cannot reduce the overhead introduced by device virtualization.

Offloading Functionality to VMM Offloading I/O operations for performance improvement is a well studied approach. In [13] the authors discuss the idea of offloading common middleware functionality to the hypervisor layer to reduce guest/hypervisor switches. In contrast, vPipe introduces shortcutting at the I/O level and is hence applicable to a broader range of cloud applications. In [24] the authors propose offloading the TCP/IP stack to a separate core. vSnoop [19] and vFlood [9] mitigate the impact of VMs' CPU access latency on TCP by offloading acknowledgement generation and congestion control to the driver domain, respectively. They however focus only on improving TCP throughput but not on improving the performance of more general I/O.

8. Conclusion

We have presented vPipe, a system that offloads piped data I/O operations of a VM to the VMM layer for more efficient data movement and consequently better application performance. We observe that traditional virtualized systems would first move the data in a piped I/O operation to the application's memory space and then back to the VMM layer, incurring I/O overhead arising from virtual device emulation and CPU scheduling latency among VMs. vPipe mitigates such performance penalty by shortcutting piped I/O operations at the VMM layer. Our evaluation of a vPipe prototype shows that vPipe can improve the throughput of file-to-file, file-to-socket, socket-to-file, socket-to-socket data movements, which are common in cloud applications. Our application case studies demonstrate vPipe's applicability and effectiveness.

9. Acknowledgments

We thank our shepherd, Ashvin Goel and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by NSF grants 0855141, 1054788, and 1219004. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] GNUMP3d [GNU MP3/Media Streamer]. http://www.gnu.org/software/gnump3d/.

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

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

[4] MPlayer - The movie player. http://www.mplayerhq.hu/.

[5] Pound: Reverse-Proxy and Load-Balancer, 2013. http://www.apsis.ch/pound.

[6] AGESEN, O., MATTSON, J., RUGINA, R., AND SHELDON, J. Software techniques for avoiding hardware virtualization exits. In *USENIX ATC* (2012).

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

[8] BILLAUD, J.-P., AND GULATI, A. hClock: Hierarchical QoS for packet scheduling in a hypervisor. In *EuroSys* (2013).

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

[10] GAMAGE, S., KOMPELLA, R. R., AND XU, D. vPipe: one pipe to connect them all. In *USENIX HotCloud* (2013).

[11] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (2003).

[12] 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).

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

[14] GOVINDAN, S., NATH, A. R., DAS, A., URGAONKAR, B., AND SIVASUBRAMANIAM, A. Xen and Co.: communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *ACM VEE* (2007).

[15] GULATI, A., MERCHANT, A., AND VARMAN, P. mClock: Handling throughput variability for hypervisor IO scheduling. In *USENIX OSDI'10* (2010).

[16] HAR'EL, N., GORDON, A., LANDAU, A., BEN-YEHUDA, M., TRAEGER, A., AND LADELSKY, R. Efficient and scalable paravirtual I/O system. In *USENIX ATC* (2013).

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

[18] KANG, H., CHEN, Y., WONG, J. L., SION, R., AND WU, J. Enhancement of Xen's scheduler for MapReduce workloads. In *ACM HPDC'11* (2011).

[19] 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).

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

[21] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM VEE* (2005).

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

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

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

[25] WALDSPURGER, C., AND ROSENBLUM, M. I/O virtulization. In *Communications of the ACM* (2012).

[26] XU, C., GAMAGE, S., LU, H., KOMPELLA, R. R., AND XU, D. vTurbo: Accelerating virtual machine I/O processing using designated turbo-sliced core. In *USENIX ATC* (2013).

[27] XU, C., GAMAGE, S., RAO, P. N., KANGARLOU, A., KOMPELLA, R. R., AND XU, D. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *HPDC* (2012).