vRead: Efficient Data Access for Hadoop in Virtualized Clouds

Cong Xu
Purdue University
xu172@purdue.edu

Brendan Saltaformaggio
Purdue University
bsaltafo@purdue.edu

Sahan Gamage
VMware Inc.
sahans@gmail.com

Ramana Rao Kompella *
Google Inc.
rkompella@gmail.com

Dongyan Xu
Purdue University
dxu@purdue.edu

ABSTRACT

With its unlimited scalability and on-demand access to computation and storage, a virtualized cloud platform is the perfect match for big data systems such as Hadoop. However, virtualization introduces a significant amount of overhead to I/O intensive applications due to device virtualization and VMs or I/O threads scheduling delay. In particular, device virtualization causes significant CPU overhead as I/O data needs to be moved across several protection boundaries. We observe that such overhead especially affects the I/O performance of the Hadoop distributed file system (HDFS). In fact, data read from an HDFS datanode VM must go through virtual devices multiple times — incurring non-negligible virtualization overhead — even though both client VM and datanode VM may be running on the same machine. In this paper, we propose vRead, a programmable framework which connects I/O flows from HDFS applications directly to their data. vRead enables direct "reads" to the disk images of datanode VMs from the hypervisor. By doing so, vRead can significantly avoid device virtualization overhead, resulting in improved I/O throughput as well as CPU savings for Hadoop workloads and other applications relying on HDFS.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management—*Input/Output*

General Terms

Design, Measurement, Performance

1. INTRODUCTION

Many enterprises are increasingly moving their applications from traditional infrastructures to private/public cloud platforms in order to reduce application running costs, both in terms of capital as well as operational expenditure. Cloud providers generate revenue by keeping their operational costs low while providing good performance for their "tenants". The key technology which drives cloud computing is virtualization. In addition to enabling multi-tenancy in cloud environments, virtualizing hosts in the cloud environment makes resource management increasingly flexible, resulting in significant savings in operational costs.

Similar to other cloud applications, Hadoop [16] applications can also benefit from cloud deployment by taking advantage of the agility to help deploy, run, and manage these clusters while maintaining reasonable performance on par with physical deployments. Compared with running Hadoop on physical machines, virtualized Hadoop allows clusters to be scaled dynamically — separating data and computation in different virtual machines (VMs) while keeping data safe and persistent. Several public and private cloud platforms already embrace this concept. For instance, the Amazon EC2 [1] provides an Elastic Map/Reduce (EMR) service [7] for hosting data processing applications. Similarly, Openstack [39] is developing the Sahara [11] platform with similar goals as EMR. VMware's Hadoop Virtualization Extention (HVE) [12] goes one step further to enhance Hadoop's topology awareness on virtualized platforms (upstreamed into Apache Hadoop release 1.2.0+).

However, running Hadoop inside VMs can lead to sub-optimal performance due to virtualization and data movement overheads. Specifically, the performance of Hadoop inside VMs is heavily dependent on the I/O efficiency of the Hadoop distributed file system (HDFS) [41], because all consumed data by big data applications is first loaded from HDFS. In general, when the client application requests the HDFS datanode to read a file, it reads that file from the local disk and sends its content back to the client over a TCP socket. Depending on the location of the datanode in relation to the client, this performance can vary drastically. For instance, if there is a co-located datanode, standard Hadoop implementations prefer a *local read* from the co-located datanode over other replicas elsewhere. While the local read is efficient when Hadoop is run in non-virtualized environments, its performance can suffer when the client and datanode are co-located on the same physical host but in different VMs (recommended deployments by Docker [6] and VMware's HVE [12, 13]), due to device virtualization overheads and data movement through protection boundaries (hypervisor, OS, application). *Remote reads* are even slower because of the additional network data transfer overheads.

In particular, the data flow for each file read on HDFS in a virtualized cloud causes data to pass through the virtual devices (e.g., virtual block and virtual NIC) multiple times — causing excessive CPU consumption and performance degradation compared to running Hadoop in physical machines. Further, even if high speed stor-

*Contributed to the work while at Purdue University.

Middleware '15 December 07-11, 2015, Vancouver, BC, Canada

DOI: http://dx.doi.org/10.1145/2814576.2814735

age hardware (e.g. SSD) is used in the virtualized hosts, the HDFS performance, in terms of throughput and latency, will still be degraded due to a lack of CPU cycles to copy the data. In addition, if low-power processors (Atom, ARM, etc.) are used (as is the trend in some data centers to obtain better per-watt performance), this degradation is even more serious. Therefore, if we can provide an efficient data movement channel between datanode VMs and client VMs, then we can mitigate the negative impact caused by device virtualization overhead and achieve better data read performance.

In this paper, we focus on improving the I/O performance of virtualized Hadoop applications or other big data applications which rely on HDFS that involve significant data reads, either partially or completely, in their work flows. More specifically, we target data movement between datanode VMs and client VMs — without performing transformation over virtual network and virtual disk. We propose to alleviate the involved device virtualization overheads by enabling HDFS client VMs to directly read data from the co-located datanode VM's virtual disk or utilizing RDMA [37] over converged Ethernet (RoCE) [42] to transfer data directly from the remote disk to the memory of client VMs via its zero-copy networking behavior. By doing so, we are able to reduce 1) device virtualization overhead such as copying data through the virtual disk, virtual network, and the network stack in both the datanode VM and client VM, 2) data copy overhead between the guest kernel/application memory and the data buffers in the host kernel for *remote reads*, and 3) I/O threads scheduling and synchronization overheads caused by "indirect" reads unnecessarily involving the virtual network between VMs.

To realize the idea of an efficient data movement channel in the hypervisor layer, we have developed a system called vRead, where data needed by the HDFS client VM is directly read from the virtual disk of a datanode VM — avoiding unnecessary data copies involved in virtual I/O behaviors. vRead installs a kernel module and a library in the guest providing the file operations interface and a daemon in the host to read data owned by datanode VMs from local and remote physical disks (via RDMA) then map it into the guest memory for the application's use. vRead is transparent to user level applications (such as Hadoop MapReduce, Hbase, and Hive) using HDFS. Therefore, it is able to support all existing applications storing data in HDFS.

To summarize, our contributions in this paper are:

1. We propose a new file operation interface for HDFS client VMs which allows Hadoop applications to read data from HDFS more efficiently.
2. We develop the vRead system, which provides I/O shortcuts at the hypervisor level via components in the guest and in the hypervisor. vRead works for both *virtual local read* (read from co-located datanode VMs) and *remote read*.
3. We present evaluation results from a vRead prototype implemented on KVM. Our microbenchmark results show that vRead achieves higher read throughput, lower latency, and less CPU cycle consumption compared to standard HDFS running on VMs. For example, Hadoop's throughput can be improved by up to 60% for read and 150% for re-read. Results from a number of Hadoop benchmarks also show significant application-level performance improvements with vRead.

We next explain our motivation in detail in Section 2 followed by the design and implementation of vRead in Section 3 and Section 4 respectively. We then present the results of our evaluations of vRead in Section 5. We discuss future work in Section 6, related work in Section 7, and conclude the paper in Section 8.

2. MOTIVATION

In this section, we motivate the problem by demonstrating the impact of virtualization overheads on Hadoop I/O efficiency. We then discuss the inadequacy of existing solutions.

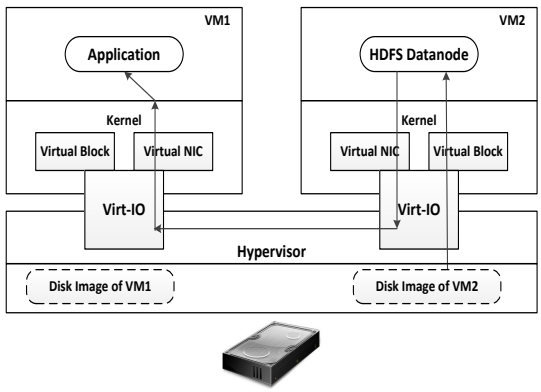


Figure 1: I/O flow in Hadoop for co-located VMs.

2.1 Problem Analysis

Virtualization-based overheads (e.g., device virtualization and VM or I/O thread scheduling) cause serious performance degradation to HDFS, and this prevents Hadoop and other applications which rely on HDFS (e.g. Hbase, Hive, and generic Java applications storing data in HDFS) from achieving their expected performance. To illustrate this problem, Figure 1 presents a concrete example of a Hadoop application hosted in a VM reading a file from a co-located VM hosting the HDFS datanode[1]. In this scenario, the Hadoop application first creates a TCP socket, connects to the HDFS datanode, and sends the file read request via this socket. Then the HDFS datanode reads the requested file from disk and sends its content back over the same TCP connection.

Even though the virtual disk and virtual network between co-located VMs are very fast (mainly due to inter-VM para-virtual I/O techniques such as virt-io [38] and vhost), this single I/O flow involves *at least 5 data copies*: 3 data copies caused by virt-io, 1 inter-VM data copy, and 1 copy between the kernel buffer and application buffer in VM1 (which may also happen in VM2). Note that each data copy consumes non-negligible CPU cycles and the whole data transfer incurs overhead from both VMs' network stacks. Further, if the file being read was located on a remote datanode VM running on another physical machine, then we would need to also consider the physical networking overhead and additional delays in the host kernels' network stacks on each physical machine. Intuitively, such high I/O costs mean *less CPU cycles for the real Hadoop workload*, which negatively impacts the performance of Hadoop applications.

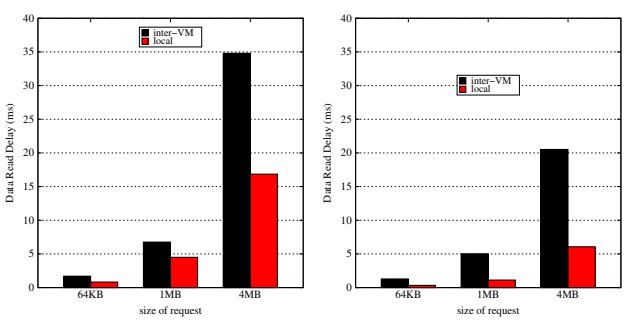
To illustrate this performance degradation, Figure 2 compares the observed read delays from HDFS versus the local file system in a virtualized host. In this experiment, we ran a Java application in one VM that reads a file from the local file system and an HDFS co-located datanode VM. The local file read (i.e., the baseline reading performance) only involves 2 data copies: 1) from disk to guest kernel buffer and 2) from guest kernel buffer to guest user space. We varied the request size (application buffer size) from 64KB to 4MB and used two different read patterns. "Read without cache" means reading data after clearing the disk memory buffer in the guest kernel (virtual disk cache in the hypervisor is disabled).

[1]Such virtual local reads from co-located VMs are more common than remote reads due to existing virtual Hadoop optimizations.

"Read with cache" (or re-read) means reading data without clearing the cache. Figure 2 shows that the delay of HDFS hosted in a co-located VM is significantly higher than the baseline read for all cases. The root cause of this result is that inter-VM reads involve *more data copies* and suffer *more device virtualization overhead.*

However, besides the additional data copies, there exists a second, more systemic cause of this performance degradation: *I/O thread synchronization* in the virtualized host. Most existing hypervisors perform I/O (network or disk) in a dedicated per-VM thread that is optimized for I/O performance. For instance, Xen uses a netback thread to process I/O requests for the virtual network, and similarly KVM uses a vhost-net thread for the same task. Therefore, to get good I/O performance, the VM and the corresponding I/O thread *must run cooperatively on different cores* so that the synchronization delay between them is short. If not, context switches between them will cause slow-down at the hypervisor level.

In addition to the synchronization between a VM and its I/O thread, data movement between 2 co-located VMs requires the I/O threads of *both VMs* to synchronize as well. Therefore, we would need 4 free cores to allow 2 I/O VMs to communicate with each other unimpededly. As more VMs run in the same host, the data transfer between VMs is further degraded because the VM scheduler cannot find enough free cores to run the cooperating threads. Figure 3 highlights this problem. In this experiment, we ran 2 co-located VMs hosting a netperf [10] server and client respectively in a quad-core machine. When there are no other active VMs running, we can get high transaction rates, even with varying request sizes. However, if an additional 2 VMs are running CPU-intensive workloads (85% lookbusy [9]) in the same host, then the TCP transaction rate drops by 20%. Since the total CPU utilization of the vCPU thread and I/O thread of each VM hosting netperf is *less than 75%* , we know the host is not overloaded for the 4 VMs scenario. Thus, the only reason for the drop in transaction rate is the synchronization delay of VMs and I/O threads. Again, because virtualized Hadoop requires many such cross-VM data movements, this same scenario causes a loss in I/O performance in virtualized Hadoop as well.



(a) Access delay without cache. (b) Access delay with cache.

Figure 2: Virtual HDFS data access delay caused by device virtualization overhead.

2.2 Alternative Solutions

We now examine several alternative solutions and their shortcomings when used with virtual Hadoop.

HDFS Short-Circuit Local Reads *HDFS Short-Circuit Local Reads* (HDFS-2246 and HDFS-347) [8] allow a read to bypass the datanode process — so that the client to read each file directly. This approach is only possible when the client process and the datanode process execute in the same *operating system* (OS). However, virtual Hadoop separates HDFS clients and datanodes in different VMs to obtain a more scalable Hadoop cluster and better on-demand resource allocation. Further, locating some clients and datanodes in the same VM to utilize *HDFS Short-Circuit Local Reads* would cause a significant penalty to *virtual remote reads* (inter-VM data reads). Thus, traditional *HDFS Short-Circuit Local Reads* does not work for virtual Hadoop.

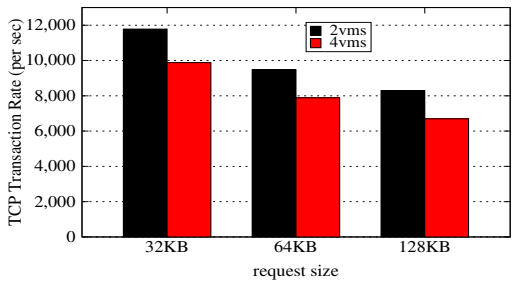


Figure 3: I/O Threads synchronization overhead.

VirtFS *VirtFS* [26] is a para-virtualized file system interface designed to improve pass-through technologies which rely on the virtio framework and the 9P protocol. With VirtFS, a guest can easily exchange data with the host. However, there are several aspects of VirtFS which impedes its useful application to HDFS. First, the 9P protocol used by VirtFS is not efficient, resulting in unacceptable disk I/O performance. Second, the explicit shared directory assignment of VirtFS makes the virtual Hadoop cluster setup more complex and inflexible. Third, it is not applicable when datanodes and clients are in different physical machines, which is a typical pattern in distributed Hadoop systems.

Hadoop Virtualization Extensions VMware's HVE enables virtual Hadoop to know the location of data files in order to co-locate inter-VM data reads. However, it does not optimize the data read path in the hypervisor (causing excessive data copies). This data flow is similar to the scenario shown in Figure 1, and is thus susceptible to the same aforementioned issues.

Inter-VM Shared Memory Inter-VM Shared Memory [44, 31, 35] is a popular technology used to boost the performance of inter-VM communication. However, this zero-copy communication between VMs can only reduce one data copy in the data flow of virtual Hadoop (Figure 1). The involvement of datanode VMs still imposes additional overheads (i.e., device virtualization and I/O threads synchronization) on each data read performed by applications running in client VMs. Again, this approach only works for co-located VMs running in the same machine.

3. DESIGN

Based on the above discussion, it is evident that if we provide an efficient file read mechanism for HDFS (i.e. if the HDFS clients can read the disk blocks owned by datanode VMs directly regardless of their location), then we can improve virtual Hadoop's performance significantly. vRead achieves this by letting the HDFS client VMs independently perform data reads directly from the disk instead of channeling it through the datanode VM.

First, vRead enables a shortcut in the I/O path that reduces the data access delay. Second, a reduction of data copies translates to the reduction of CPU consumption, thus more CPU resources are available for the actual Hadoop CPU-bound tasks. Third, a shortened inter-VM data transfer path significantly reduces the I/O processing delay caused by synchronization of the VMs' I/O threads.

To enable an HDFS client VM to read data directly from a datanode VM's virtual drive, vRead needs to provide three features: 1)

API Function	Input Parameters	Return Value	Description
vRead_open()	blk_name, datanodeID	vRead descriptor	Open the file for an HDFS block stored in a specified datanode and get the corresponding vRead descriptor.
vRead_read()	vRead descriptor, buffer offset, length	Number of bytes read into buffer	Attempt to read up to *length* bytes from the file pointed to by vRead descriptor.
vRead_seek()	vRead descriptor, offset	Resulting offset as measured in bytes	Set the file offset for an opened file pointed to by the given vRead descriptor.
vRead_close()	vRead descriptor	Successful (0) or not (-1)	Close the file for a given HDFS block indicated by the vRead descriptor.

Table 1: vRead API.

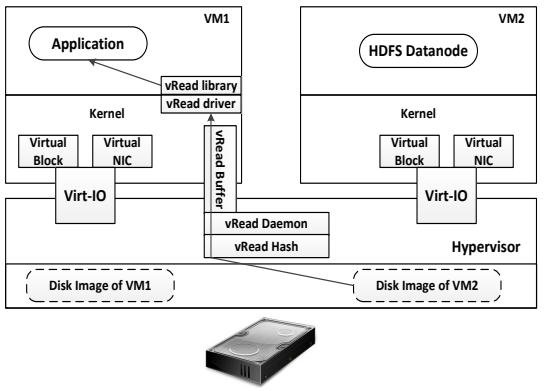


Figure 4: I/O flow in Hadoop for co-located VMs with vRead.

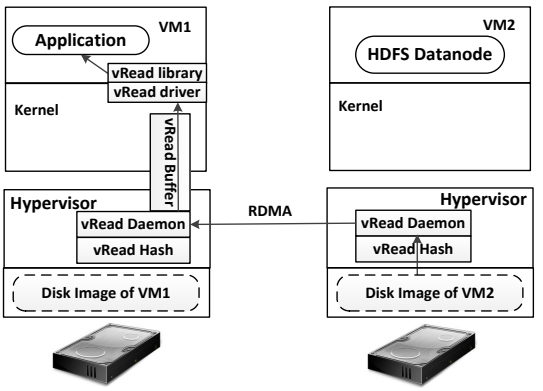


Figure 5: I/O flow in Hadoop for remote VMs with vRead.

a new user-level block read interface for inter-VM block reads, 2) a mechanism to directly read from the virtual drive of the datanode (an active VM) without channeling through the datanode and, 3) an efficient data sharing (zero-copy) and communication channel between the guest OS and hypervisor.

The architecture of vRead is shown by Figure 4 and Figure 5. The above requirements are realized with the three main components of vRead shown in these figures. In the following subsections we will discuss each of these sub-components of vRead in detail.

In this paper, vRead targets HDFS but this framework is able to be generalized to other similar distributed file systems such as QFS [36] and GFS [21].

3.1 vRead User-level API

vRead provides a set of user-level library functions for HDFS to use. It should be noted that these API calls will only be used by the HDFS components. Hadoop applications are unaware of the existence of the vRead system and continue to use the original HDFS interfaces to read data — hence requiring no modifications to them. vRead's implementation is independent of the guest OS so that this user-level library does not need to be adjusted or changed whenever a different OS is used. We outline the vRead user-level APIs in Table 1. The vRead APIs are a set of functions provided in a user-level library (*libvread*) which hides the complexity of interacting with the underlying vRead components. This library provides 4 main functions. To read a file stored in a datanode VM's virtual disk, we first need to call *vRead_open* to initialize a set of data structures inside both the guest OS as well as inside the hypervisor. This will return a vRead descriptor which is used as a parameter in the rest of the functions. HDFS only understands block names, hence the vRead descriptor is invisible to it. Thus, each obtained descriptor is stored in a hash table in the user-level library, which maps the block names to vRead descriptors, until the *vRead_close* function is called. This lets HDFS reuse the descriptor for subsequent read/seek operations on the same block file.

HDFS is designed to support very large files in a distributed environment. Thus, each file stored in HDFS is divided into chunks called data blocks (some smaller files on disk, 64MB each by default), and if possible, all blocks will reside on different datanodes. As a distributed file system, all actual data (HDFS blocks) are stored in the same path in each datanode. To read a file from HDFS, the Hadoop client needs the assistance of the namenode which stores the metadata of each file (block mappings, destination data node ID, etc.). In this work, we preserve all logic between clients and the namenode, and only modify the file read logic in the HDFS client interface. We re-implement the HDFS interface with the vRead file operation interface. When an HDFS client plans to read a specific file, it first gets the block list from a namenode then uses the vRead interface wrapped by the HDFS interface to send target block information (block name and datanode ID) and operations to the vRead per-VM daemon running in the hypervisor. The daemon then reads the data from the destination virtual disk and returns to the client. We discuss the operation of this daemon in the following subsection.

3.2 Reading from a Datanode's VM Disk Image

We use a daemon running in the hypervisor to aid in reading from the virtual disk images of datanodes. This daemon receives the requests from a guest and uses a hash table to store the mappings between HDFS datanode IDs and the corresponding virtual disk (which can be a local image file, NFS, or iSCSI) information for each datanode VM. This hash table is initialized when Hadoop is started by accepting information from the namenode and looking up the VM's configuration. For the datanode VMs running on other machines, we only store the IP address of the destination host machine. This table is dynamically updated once there are any datanode VMs created, deleted, or migrated.

Reading from a Local Datanode To read data from the datanode VM's virtual disk (i.e., an image file) in the hypervisor using existing POSIX APIs, vRead has to meet the following two requirements. First, the vRead hypervisor daemon should be able to understand the file system in the virtual disk. This is because the HDFS blocks are stored as regular files in the datanode's file system and accessing these files requires the hypervisor layer to understand the file system layout of the datanode VM. In KVM environments, a Linux kernel functions as the hypervisor and this kernel can in-

terpret most file systems used by guest OSs (typically guests also use a similar if not the same Linux version). To access the content of the datanode's file system, all datanode VMs' virtual disks are mounted in read only mode to a specific directory in the hypervisor (e.g. /mnt/datanode1) as loop devices with the assistance of *losetup* (qemu-nbd module is needed if the disk image file is qcow format). Since each virtual disk is installed independently of the file systems (maintained by the guest OS), we need to read the partition tables on these virtual devices and create device maps over the partitions' segments (we use the existing Linux tool *kpartx* for this).

Second, we need to synchronize accesses to this file system by the datanode and HDFS clients. The datanode's access is read-write while the HDFS clients' accesses are read-only. Since the file system within the guest OS is independent of the file system in the hypervisor (file system of the VM is in the VM's address space and hence opaque to the hypervisor unless we use VM introspection tools), new HDFS blocks generated by the datanode are invisible to the vRead daemon. Thus, we need to refresh the directory entry and inode cache information of the hypervisor mount point of the datanode's disk partition if any HDFS blocks are created, deleted, or renamed. However, HDFS mostly operates in "append-only" mode, hence we do not need to refresh all information for the mount point. Only the added inodes (i.e., new files representing the HDFS blocks) need to be updated.

The synchronization is achieved through the Hadoop namenode. When a datanode writes new blocks to the file system, it notifies the namenode about the availability of the new blocks so the readers (HDFS clients) can access these blocks. We use this notification as a trigger to refresh the mount point corresponding to that datanode. The new block information is obtained from the *vfsmount* structure and *superblock* of the corresponding virtual disk. The whole process is similar to a remount. This process is sufficient to guarantee that there is no read/write conflict issues since HDFS follows the *write-once and read-many* approach for its data blocks. This approach assumes that a file in HDFS will not be modified once it is written. All new written data will generate new blocks. So we do not need to be concerned with read/write conflicts when issuing a direct read on a virtual disk without notifying the owner VM.

Reading from a Remote Datanode To read data from a remote datanode, the local vRead daemon contacts the remote host's vRead daemon using RDMA and sends the request to the remote daemon. The remote daemon then performs the read operation on the local disk (as discussed above) and returns the data via RDMA. For vRead, RDMA is not necessary (traditional TCP/IP also works), but RDMA helps vRead consume less CPU cycles and ensure lower latency for remote data reads.

RDMA allows an application to communicate directly with another application via remote memory read/write. This means that an application does not need to rely on the operating system to transfer messages. To communicate with the remote end, we (1) Register a Memory Region (MR). This Memory Region is similar to the packets buffer, but it is shared by the remote party which can directly access it via the RDMA device. (2) Create a Send and a Receive Completion Queue (CQ). (3) Create a Queue Pair (QP) that is a Send/Receive Queue Pair. To send or receive messages, Work Requests (WRs) are placed onto a QP. When processing is completed, a Work Completion (WC) entry is optionally placed onto a CQ associated with the work queue. To enable the RDMA, we use the support of two libraries (rdmacm and libverbs) in userspace and a physical NIC with RDMA support. Traditional RDMA requires an infiniband network which is very expensive, so we use RoCE (RDMA over Converged Ethernet) instead of infiniband.

3.3 Data Sharing and Communication Channel

The communication between the guest OS and local vRead daemon is accomplished using a shared memory channel and an event mechanism. This channel is a memory ring buffer shared by the VM and hypervisor. For each HDFS read request, the vRead driver in the guest places a request in the shared memory and fires an event to notify the vRead daemon in the hypervisor. In turn, the vRead daemon in the hypervisor reads the data from the datanode VM's disk image, writes the data to this channel, and then sends an event to the guest OS so the data can be consumed by client applications. We cannot use KVM virt-io's ring buffer for this purpose since the HDFS client VM needs to read data from the datanode VM's virtual disk and such an I/O operation is not the intended use of virt-io channels. Instead we have to use a shared memory mechanism between the local vRead daemon and guest OS's vRead driver.

To achieve zero-copy between the hypervisor and guest OS in the VM, we use a POSIX SHM object as the shared buffer that is assigned to each VM as a character device. For each guest, this shared memory object appears as a virtual PCI device inside the guest OS. This POSIX SHM object is divided into multiple chunks (default 1024) to comprise a ring buffer. The vRead daemon uses SysV APIs to read from/write to this shared buffer. The guest maps the virtual PCI device's address space to it's own address space and then performs read/write operations. The synchronization between the vRead daemon and guest OS is guaranteed by a read/write lock on each chunk.

To send notifications between the vRead daemon in the hypervisor and guest OS, vRead uses interrupts between them that are implemented by assigned *eventfds*. Each VM listens on its own eventfd, and uses its corresponding vRead daemon's eventfd to send an event (and vice versa). The only difference is that the vRead daemon operates the event directly, whereas the event received by the VM has to be translated into a virtual interrupt which can be recognized by the guest OS. This translation is done by the vRead driver in the guest kernel. With this channel, applications in the guest OS can send requests (e.g., read a HDFS block) to the vRead daemon and get the result from the shared memory buffer.

4. IMPLEMENTATION

We have implemented a prototype of vRead for the KVM [29] hypervisor. We used Linux 3.12 as the kernel of the VMs and the KVM host. The Hadoop version is 1.2.1.

vRead includes new implementations of the read interfaces for an HDFS client (in the *DFSClient* class). These interfaces mainly contain *read*, *seek*, and *skip* functions located in the *DFSInputStream* class (a subclass of *DFSClient*). Of these functions, the *read* function is the most important as it is frequently called during HDFS reads. The *DFSInputStream* class has 2 different *read* functions that vRead overrides: called *read1* and *read2* in this paper. *read1* reads a large file from the beginning and its request size is smaller than one HDFS block (e.g., for use by applications performing sequential reads). Its vRead implementation is shown in Algorithm 1. Before reading an HDFS block, vRead checks whether the corresponding file has been opened previously (and thus has a corresponding *vfd* in hash) or not. If the file has not been opened previously, a new vRead descriptor *vfd* is created by calling *vRead_open()* and added to a hash table for future use. Upon subsequent calls to *read1*, vRead checks if the input descriptor is a valid vRead descriptor (i.e., in the hash table). If so, it is used to read data via *vRead_read()*; if not, the original HDFS function *read_buffer* is called to perform the read from the datanode. *read2* reads data from a specific position in a file (e.g., for use in asyn-

chronous/random reads). The implementation of *read2* is outlined by Algorithm 2. Generally, *read2* is similar to *read1* except that it is allowed to read across multiple blocks so vRead has to collect all involved block information from the namenode and perform the *vRead_read* on them one by one.

Additionally, we slightly modify the write interfaces of HDFS to update the dentry/inode of the mount point for new blocks generated by the datanode. Specifically, we call the *vRead_update* function at the end of the standard *append* function (in the *DFSOutputStream* class) once a full block is written to the datanode VM. Likewise, the same thing happens for a block delete or rename. Note that we do not have to call *vRead_update* for each *append* operation before a new block is completely created. Since all vRead functions in *libvread* are written in C, but HDFS is implemented in Java, all vRead functions have to be called via a Java native interface (JNI). After adding the vRead extensions to the *DFSClient* class, the Hadoop source code was re-compiled and we replaced the hadoop-core-1.2.1.jar required by the Hadoop running environment with our new one.

Algorithm 1 DFSInputStream read1 with vRead interface

```
1: vfd is the vRead descriptor for a given HDFS block
2: vfd_hash is the hashtable storing the mappings of HDFS block and vfd
3: datanode_id indicates the target datanode
4: blk is the instance of an HDFS block to read
5: buf is the application buffer
6: len is the number of bytes to read
7: off is the offset of the data block
8: procedure READ(buf, off, len)
9:     blk = getCurrentBlock();
10:    if vfd_hash.containsKey(blk.name) == null then
11:        /* call vRead_open() to get the vRead descriptor */
12:        vfd = vRead_open(blk.name, datanode_id);
13:        vfd_hash.put(blk.name, vfd);
14:    else
15:        vfd = vfd_hash.get(blk.name);
16:    end if
17:    /* read the data with vRead descriptor */
18:    if vfd != null then
19:        result = vRead_read(vfd, buf, off, len);
20:    else
21:        /* original method of HDFS */
22:        result = read_buffer(blk, buf, 0, len);
23:    end if
24:    if result > 0 then
25:        position += result;
26:        if position == blk.size then
27:            vRead_close(vfd);
28:        end if
29:    end if
30: end procedure
```

To interact with the vRead buffer, we implemented a guest kernel driver that: 1) helps the guest OS recognize the assigned POSIX SHM object as a virtual PCI device and 2) translates the eventfd signals to virtual interrupts and vice versa. This driver is a loadable kernel module whose implementation is based on the ivshmem [31] VM driver. The address of the virtual PCI device representing the vRead buffer is mapped to the address space via *mmap()* — so that applications in the guest can read from/write to this ring buffer by calling the vRead series functions in *libvread*. The vRead ring buffer is divided into 1024 slots (the size is configurable, with a default of 4KB) comprising the critical area between the application thread in the guest OS and the vRead daemon in the hypervisor. A spinlock (*pthread_spinlock_t*) is used on each slot to guarantee synchronization safety.

The vRead daemon is a generic thread granted read privilege to the entire local physical disk of the hypervisor. In the KVM platform, each VM is a process/thread in the host. Therefore, the vRead daemon can communicate with the process representing a VM via an eventfd and a read/write on the shared POSIX SHM object (vRead buffer).

Algorithm 2 DFSInputStream read2 with vRead interface

```
1: vfd is the vRead descriptor for a given HDFS block
2: vfd_hash is the hashtable storing the mappings of HDFS block and vfd
3: datanode_id indicates the target datanode
4: blk is the instance of an HDFS block to read
5: position is the absolute start position of the target file stored in HDFS
6: buf is the application buffer
7: len is the number of bytes to read
8: off is the offset of the data block
9: procedure READ(position, buf, off, len)
10:    blk_list = getRangeBlock(position, len);
11:    remaining = len;
12:    for each blk in blk_list do
13:        start = position - blk.getStartOffset();
14:        bytesToRead = min(remaining, blk.size − start);
15:        if vfd_hash.containsKey(blk.name) == null then
16:            /* call vRead_open() to get the vRead descriptor */
17:            vfd = vRead_open(blk.name, datanode_id);
18:            vfd_hash.put(blk.name, vfd);
19:        else
20:            vfd = vfd_hash.get(blk.name);
21:        end if
22:        /* read the data with vRead descriptor */
23:        if vfd != null then
24:            result = vRead_read(vfd, buf, start, bytesToRead);
25:        else
26:            /* original method of HDFS */
27:            result = fetchBlocks(blk, start, bytesToRead, buf);
28:        end if
29:        remaining -= bytesToRead;
30:        position += bytesToRead;
31:    end for
32: end procedure
```

To connect to remote vRead daemons on other machines with low latency and low CPU cost, we use RDMA interfaces (declared in rdma/rdma_cma.h and infiniband/arch.h) instead of TCP/IP APIs to exchange data[2]. Specifically, we call a few standard infiniband verbs such as *ibv_reg_mr* (register memory regions), *ibv_post_send* and *ibv_post_recv* (send and receive requests) on the Ethernet links via RoCE techniques to directly map the working set address of request/response to the remote memory.

To update the file system for new blocks added in a mounted virtual disk, vRead needs to refresh the dentry/inode of the mount point if the *vRead_update* function is called in the guest OS. This is done by calling a function extended from *attach_recursive_mnt()* (in the source code of the mount command) which is responsible for updating the *vfsmount* structure of the host file system.

5. EVALUATION

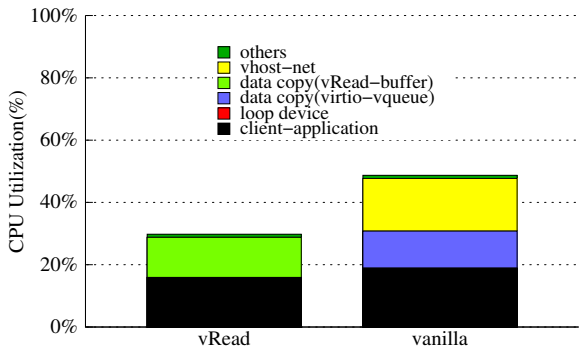
This section presents our evaluation of vRead using both microbenchmarks and real world Hadoop applications.

Evaluation Setup Our testbed consists of multiple servers, each with a 3.2 GHz Intel Xeon quad-core CPU and 16GB of memory. An SSD and 10Gbps RoCE NIC are installed in each server. All physical servers are connected by 10Gbps network in a LAN. These servers run KVM as the hypervisor and Linux 3.12 as the OS for all guest VMs and the hosts. The Hadoop version is 1.2.1. To emulate the different CPUs (low power and high frequency), the frequency

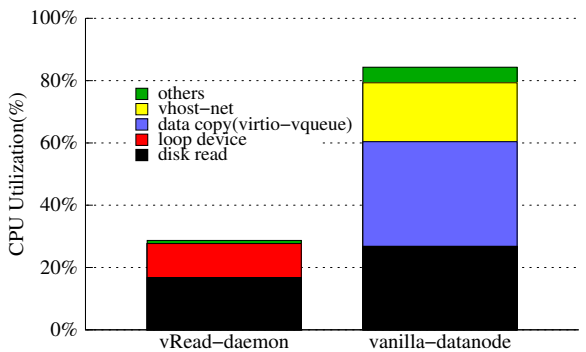
[2]We also implemented a TCP/IP version prototype, but note that it consumes more CPU cycles for remote reads

of our Xeon processor is set to different values (1.6 GHZ, 2.0 GHZ and 3.2 GHZ) via the *cpufreq-set* command [5].

All VMs in our experiments are assigned 1 vCPU and 2GB RAM each. We do not set the CPU affinity for the VMs. KVM vhost-net is enabled to boost the virtual network performance. vhost-blk is disabled because it is still the test version in the latest KVM release. The virtual disk image of each VM is a raw image file located in the local SSD.



(a) Client CPU utilization.



(b) Datanode CPU utilization.

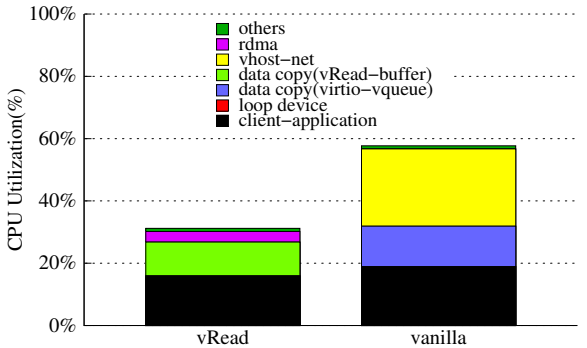
Figure 6: CPU utilization for co-located read.

5.1 Microbenchmark Performance

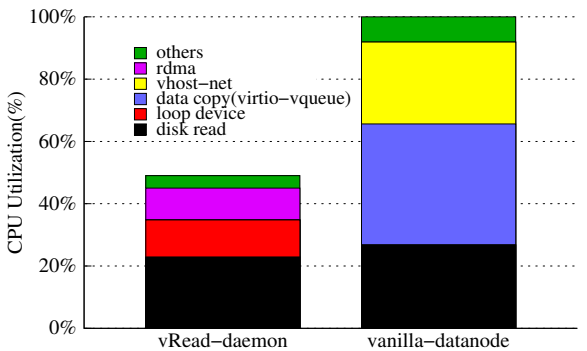
CPU Savings To verify whether vRead's shortcut to file reading can reduce overall CPU cost or not, we compare the CPU utilization of reading a 1GB file from the HDFS with vRead and without. The request size (application buffer on the client-side) of each read is 1MB. There are three scenarios: 1) the client VM and HDFS datanode VM are running on the same machine (i.e., co-located scenario), 2) the client VM and datanode VM are running on two different machines (i.e., remote scenario) and the vRead daemons use RDMA to exchange data, and 3) still the remote scenario but the vRead daemons use TCP instead of RDMA to exchange data.

Figure 6 shows the average CPU utilization when the client reads from the co-located datanode VM. As expected, the VMs' CPU utilization with vRead is much lower than the vanilla case. Since there is no virtual network involved in vRead for this case, vRead saves a significant number of CPU cycles both in the guest and host. The direct data read from disk also avoids any unnecessary data copies between the 2 VMS, between the host and datanode VM, and between the guest kernel buffer and application buffer in the datanode VM. In total, we save around 40% of the CPU cycles on the client side and around 65% on the datanode side with vRead.

The results of the remote-read scenario with RDMA enabled are presented in Figure 7. vRead still beats the vanilla case on both the client and datanode sides. Thanks to RDMA, the inter-host



(a) Client CPU utilization for remote read with RDMA.



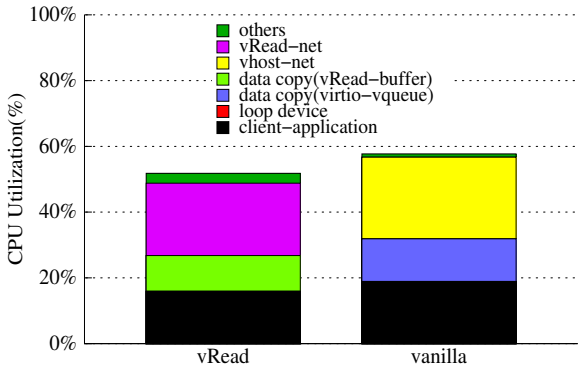
(b) Datanode CPU utilization for remote read with RDMA.

Figure 7: CPU utilization for remote read with RDMA.

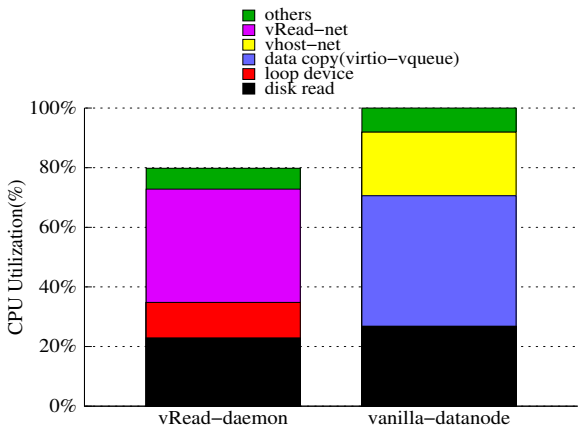
network cost of vRead (shown by the *rdma* bar) is far lower than the vanilla (shown by the *vhost-net* bar). Since our prototype uses an *active model* for RDMA data exchange on the datanode side (actively pushing data into the client's memory), the RDMA cost of the host running the datanode VM is higher than that of the host holding the client VM. In this case, we save around 45% of the CPU cycles on client side and more than 50% on datanode side.

We also evaluate the TCP version of the data exchange for remote reads. In this setup, the vRead daemons running on different machines use TCP/IP interfaces instead of RDMA verbs to exchange data. The results of these tests are shown in Figure 8. Compared with the RDMA version, the number of CPU cycles spent in sending/receiving data with TCP is significantly higher. Note that the total CPU utilization is still slightly lower than the vanilla case, which also uses TCP/IP, because it avoids copying data from the host to the datanode VM. Nonetheless, the network processing of the vanilla setup (*vhost-net*) is even more efficient than our TCP component ("vRead-net"). This is because all operations of vhost-net are completely done in kernel space, while our TCP version of vRead is a user-level thread in the host — which has to switch between kernel space and user space and thus consumes more CPU cycles. Therefore, we prefer the RDMA version utilizing the RoCE because it helps achieve encouraging performance with low cost.

Data Read Delay Reduction vRead allows the client VM to read files (HDFS blocks) from a datanode VM's disk image directly. Theoretically it would achieve performance close to that of reading data from the local file system. So we repeat the data access delay experiment (shown in Figure 2) described in Section 2. However, now we replace the *local reads* by HDFS reads with vRead; the baseline is still vanilla HDFS reads. Figure 9 shows the average data read delay when performing a 1GB file read from a co-located HDFS datanode VM. The request size varies from 64KB to 4MB. In the first scenario, only 2 VMs (client and datanode VMs) are



(a) Client CPU utilization for remote read with TCP.



(b) Datanode CPU utilization for remote read with TCP.

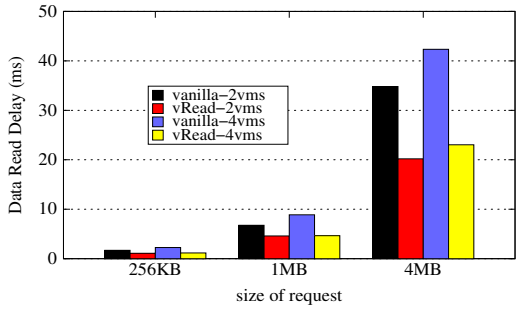
Figure 8: CPU utilization fore remote read with TCP.

running in a quad-core machine. The CPU frequency is set to 2.0 GHZ. We issue two kinds of reads for this case. Figure 9(a) shows the results after clearing the memory cache in the datanode VM and host. Figure 9(b) shows the results without clearing the memory cache, that is, all data are read from the memory cache and not the disk (called re-read). Our results show that for any request size, vRead beats the vanilla case in both read and re-read evaluations, because it cuts 3 data copies for each read.

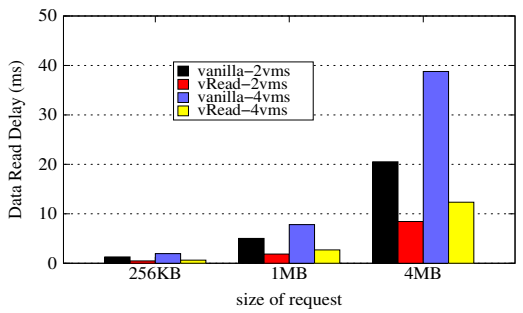
Further, recall that I/O thread synchronization may negatively impact inter-VM communication — resulting in HDFS read degradation when CPU competition happens among VMs and I/O threads. To measure vRead's effectiveness in this scenario, we ran an additional 2 VMs in the same quad-core machine so that all vCPU threads and I/O threads *cannot* always find a free core to run on. Hence, the HDFS data read delay in the 4 VMs scenario is higher than the 2 VMs case. vRead's performance is also affected, but its degradation is lower than the vanilla case. Therefore, the gap between vRead and the vanilla case is larger in the 4 VMs scenario. Overall, vRead can reduce the data access delay of the co-located HDFS reads by up to 40% for the 2 VMs scenario and up to 50% for the 4 VMs scenario compared with the vanilla environment.

5.2 Application Performance

Hadoop Performance In this experiment, we set up a simple Hadoop cluster containing one client VM and two datanode VMs. The namenode resides in the same VM as the client. More specifically, one datanode VM shares the same host (Host1) with the client VM, the other datanode is hosted by another physical machine (Host2) in the same LAN. Each physical machine hosts up to 4 VMs, and the rest of the VMs are background VMs running an



(a) Data access delay without cache.



(b) Data access delay with cache.

Figure 9: Data access delay for virtual HDFS.

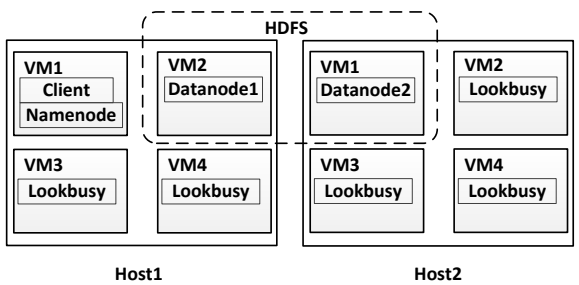
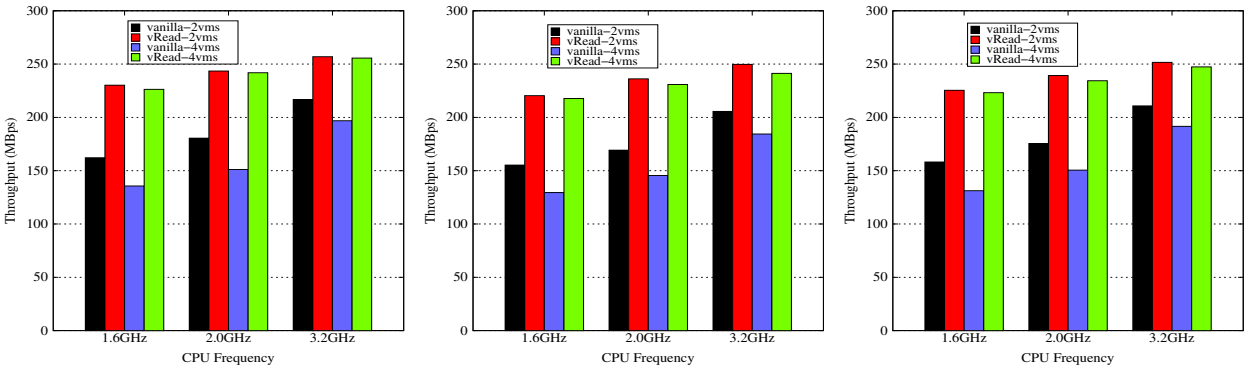


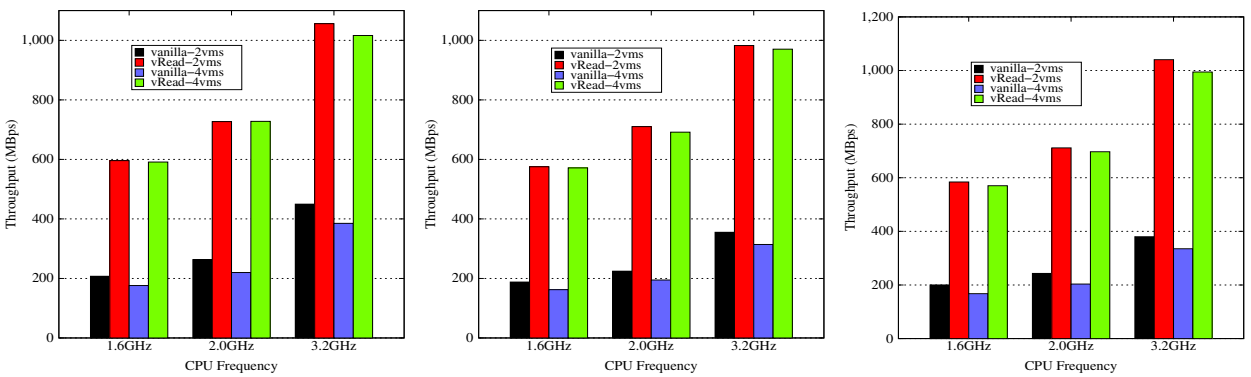
Figure 10: Hadoop setup.

85% lookbusy [9] workload. The setup is shown in Figure 10.

In the *virtual local read* scenario, the client reads data from only the co-located datanode VM. In the *remote* scenario, only the data stored in the datanode VM located on Host2 is read. *hybrid* means that the client read data from both the co-located datanode VM and remote datanode VM, which is a more generic scenario in the real world. A widely used HDFS benchmark TestDFSIO is chosen to measure the read throughput of HDFS and the CPU running time. Unlike the simple Java application used in our data access delay experiment, TestDFSIO is a real Hadoop workload utilizing the Map/Reduce framework. In our experiment, the client reads/re-reads 5GB of data from the HDFS each time with the default 1MB memory buffer. To measure the performance on different processors, we vary the CPU frequency from 1.6 GHZ to 3.2 GHZ to emulate low-power processors and powerful processors. The results shown in Figure 11 indicate that if only the client VM and datanode VM are running (2 VMs scenario) vRead obtains around 20% throughput improvement over the vanilla Hadoop on powerful processors (3.2 GHZ). While, on the low-power processors (1.6 GHZ), the throughput improvement increases to around 41%. The CPU bottleneck on low-power processors becomes more severe for the vanilla case, but its impact on vRead is slight because vRead requires far fewer CPU cycles to perform a read from the HDFS



(a) DFSIO throughput for co-located read. (b) DFSIO throughput for remote read. (c) DFSIO throughput for hybrid read.



(d) DFSIO throughput for co-located re-read. (e) DFSIO throughput for remote re-read. (f) DFSIO throughput for hybrid re-read.

Figure 11: HDFS read throughput.

which is verified by our CPU saving experiments.

When 2 or 3 additional VMs hosting 85% *lookbusy* are running on the same hosts (4 VMs in total), each VM and its I/O thread cannot be assured a free core to run on. Thus the synchronization among VMs and their I/O threads are delayed by the CPU fair-share scheduler. This is why the vanilla case's throughput drops by up to 22% for the 4 VMs scenario. Whereas, vRead's performance just drops slightly due to less work being done by the I/O threads (i.e., no inter-VM communication). Therefore, vRead has up to 65% improvement over the vanilla case in the 4 VMs scenario. Figure 12 shows the actual CPU running time (not the task completion time) spent by the TestDFSIO benchmark when performing the 5GB reads from the HDFS. This shows that vRead still saves significant CPU cycles along with gaining better throughput, which is helpful to reduce the electric power cost for data centers while obtaining encouraging performance.

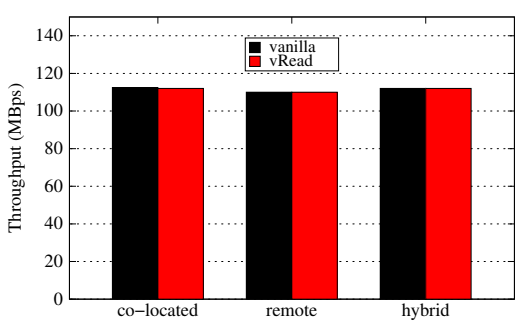


Figure 13: HDFS write throughput.

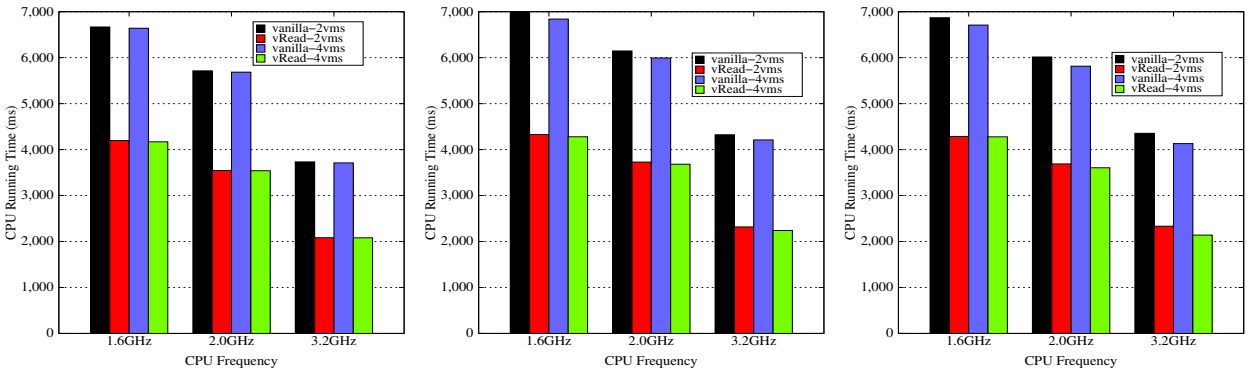
To enable the new HDFS blocks written into the datanode VM to be visible to the vRead daemon in the hypervisor, we need to update the mount point information once a new file is generated in the file system of the virtual disk image. We verify that this will not hurt the HDFS write performance by running TestDFSIO-write with the same setup as TestDFSIO-read. Figure 13 shows the results of 3 different scenarios (CPU is set to 2.0 GHz) for the vanilla case and vRead. From this figure we can see that the overhead of updating the information of the mount directory is negligible.

Big Data Analysis Tools with vRead There are a number of powerful data query tools in the Apache Hadoop ecosystem helping people store and analyze big data efficiently and safely. In this subsection, we will evaluate the performance of vRead on some of these tools (Hbase [2], Hive [3], and Sqoop [4]).

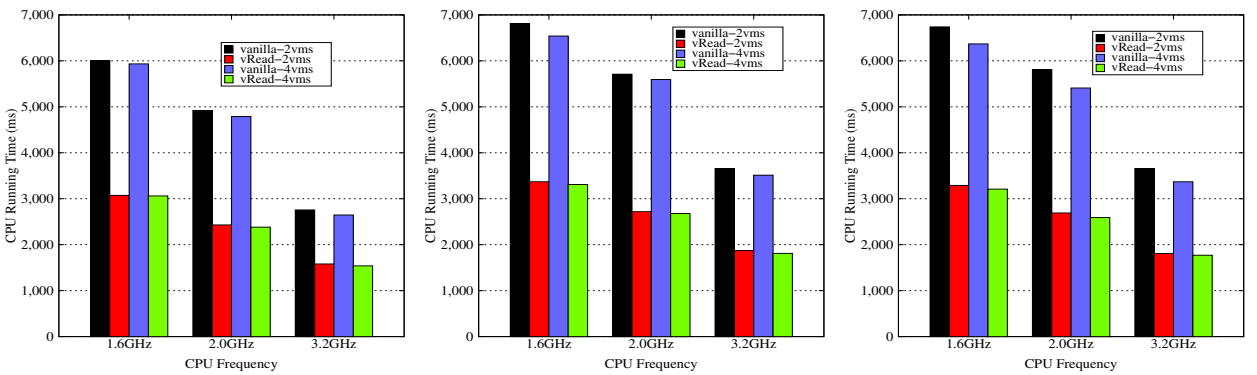
	Scan	SequentialRead	RandomRead
Vanilla	6.26MB/s	3.01MB/s	2.48MB/s
vRead	7.97MB/s	3.72MB/s	2.91MB/s
% Improvement	**27.3**	**23.6**	**17.3**

Table 2: Performance improvement for Hbase.

HBase Apache HBase is a Hadoop database: a distributed, scalable, big data store. It is capable of hosting very large tables — millions, or even billions, of rows on top of commodity hardware. Each read/write operation is split into Map/Reduce jobs running on the underlying Hadoop clusters. For this experiment, we installed HBase-0.94 on top of our Hadoop deployment (same as the *hybrid* 4 VMs setup in last subsection). The CPU frequency is set to 2.0 GHZ and the frequency scaling is disabled. In order to use the extended HDFS with vRead, we replace the Hadoop-core-1.2.1.jar under the *hbase-0.94/lib* directory with our new jar package with vRead. We use the built-in HBase benchmark *Perfor-*



(a) DFSIO CPU time for co-located read. (b) DFSIO CPU time for remote read. (c) DFSIO CPU time for hybrid read.



(d) DFSIO CPU time for co-located re-read. (e) DFSIO CPU time for remote re-read. (f) DFSIO CPU time for hybrid re-read.

Figure 12: HDFS read CPU time.

manceEvaluation to measure the performance of scan, sequential read, and random read in HBase. We inserted 5 million records via *PerformanceEvaluation-SequentialWrite* to HBase as testing data. The results are shown in Table 2. Compared with the vanilla case, vRead can improve the throughput of the 3 operations by 27.3%, 23.6% and 17.3%, respectively.

Hive Apache Hive is a data warehouse software that facilitates querying and managing large datasets residing in distributed storage. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language. Similar to the HBase test, we installed Hive-1.1 on top of our Hadoop deployment (same as the *hybrid* 4 VMs setup in TestDFSIO test). Following the evaluation approach from the UC Berkeley AMP Lab, we first created a test table in Hive storing some user information (*id, name, birthday, etc.*) and loaded 30 million rows into this table. Then we ran a sql query (*select * from test where id* $\geq x$ *and id* $\leq y$) to select the rows meeting the query conditions. The query completion time is shown in the second column of Table 3. From this we can see that a 21.3% time reduction was achieved by vRead.

	Select Sql for Hive	Sqoop Export
Vanilla	17.945s	385.136s
vRead	14.117s	342.508s
% Improvement /(Reduction)	**21.3**	**11.3**

Table 3: Performance improvement for Hive and Sqoop.

Sqoop Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop data store and structured datastores such as relational databases (MySQL, Oracle, MSSQL etc.). Here, we measure the performance of exporting data from Hive to MySQL. The *export* operation, in fact, is a process of reading data from HDFS and inserting them to a relational database. In this experiment, we move the test table storing 30 million rows used in our Hive test to a MySQL database running in another physical machine on the same LAN through *Sqoop-export*. The job completion time is shown in the third column of Table 3. For these results, we can see that vRead can reduce the time by around 11.3%. The reduction is lower than our other test case, because the *export* performance is limited by both the read efficiency of HDFS and the insert (write) efficiency of MySQL which vRead cannot optimize.

6. DISCUSSION

Interplay with Modern Hardware SR-IOV [18] devices and IOMMUs such as Intel VT-d [25] enable the hypervisor to directly assign devices to the guests. This allows the guests to directly interact with the physical devices and eliminates virtualization overhead. However, it does not work for inter-VM data movement, which is common with virtual Hadoop. Additionally, with these hardware, delays caused by synchronization between VMs and I/O threads will still impact the I/O performance of the communicating parties. Actually, vRead is compatible with SR-IOV and IOMMUs because vRead does not modify the networking routines for packets on the outgoing host. Therefore, vRead and those modern hardware are complimentary and could mutually benefit from each other.

Compatibility with VM Migration VM live migration [17] is helpful for maintaining overall load balance among physical servers in a data center. Live migration requires storing the VM disk images in a centralized storage. Hypervisors (or the VM host) access the VM images via NFS or iSCSI. vRead still works in this setup. All image files deployed by NFS or iSCSI can be mounted in the hypervisor's file system. The reads/writes on these virtual disk im-

ages are the same as that on image files located on a local disk drive. Once a VM is migrated to another host, the vRead hash tables in both hosts just need to be updated.

Direct Read Bypassing the File System in the Host Since the vRead daemon has the privilege to access all local devices, it can directly read a datanode VM's virtual disk and bypass the file system in the host. This method can avoid mounting the virtual image files in the host and updating the mount point's dentry/inode for any new blocks. However, the main drawback of this method is that it cannot benefit from the file system cache, that is, all reads have to load data from the physical disk drive. Also, this approach needs to manually translate the address of each file several times (guest logical to guest physical, guest physical to host logical, host logical to host physical) for each read. This is much more complex than mounting the virtual disk image to the host's file system — which allows vRead to use Linux POSIX APIs to read/write files.

7. RELATED WORK

We have introduced some alternative solutions for optimizing data movement for virtualized systems in Section 2. Here, we discuss other related work in the same area. These efforts can be divided into three categories: reducing device virtualization overhead, VM scheduling optimization, and functionality offloading.

Reducing Virtual Device Overhead In recent years, many efforts have focused on reducing device virtualization overhead to improve VM I/O performance or capacity of VM hosts. vPipe [19] enables direct "piping" of application I/O data from source to sink devices, either files or TCP sockets, at the hypervisor level. By doing so, vPipe can avoid both device virtualization overhead and VM scheduling delays, resulting in better VM I/O performance. vPipe focuses on reducing the virtualization overhead between the virtual devices in the same VM, while vRead targets reducing the redundant data copies between VMs. Menon [32] proposes several optimizations such as offloading datagram checksum and TCP segmentation (TSO) to the Xen virtual machine monitor (VMM) [15] to improve TCP performance in Xen VMs. [34] aimed to reduce the TCP per-packet processing cost in VMs by packet coalescing to achieve better TCP receive performance. [33] proposes offloading part of the network device's functionality to the hypervisor to reduce CPU cycles consumed by network packet processing. These three work focus on optimizing some functionalities of TCP/IP in virtual environments, whereas vRead focuses on optimizing the data movement path between VMs communicating with each other and mainly targets the applications relying on HDFS.

Similarly, Ahmed *et al.* propose virtual interrupt coalescing for virtual SCSI controllers [14] to reduce disk I/O processing overhead in virtualized hosts. In [22, 24], Gordon *et al.* propose exitless interrupt delivery mechanisms to mitigate the overhead of virtual interrupt processing in KVM so that the incoming I/O events are sent to the destination VM without switching to the hypervisor by a VM-Exit. These two works can reduce the virtual interrupts' overhead incurred by processing disk or network I/O requests in a VM, but they cannot eliminate the unnecessary I/O flow between VMs which is targeted by vRead.

VM Scheduling Optimization Since VM scheduling delay can significantly affect a VM's I/O performance in terms of throughput as well as application-perceived latency in virtual systems, many previous efforts have focused on reducing VM scheduling delay for I/O-intensive applications. [30] proposes a soft-realtime VM scheduler to reduce the response time of I/O requests thus improving the performance of soft-realtime applications such as media servers. However, its preemption-based policy may violate CPU fair-share if a VM is I/O-intensive. vSlicer [43] minimizes CPU scheduling delay and hence the application-perceived latency — to a certain degree by setting a smaller time-slice for latency-sensitive VMs. However, such a time-slice is not small enough to improve TCP/UDP throughput in LAN/datacenter environments. These two efforts both assume multiple VMs are running on the same CPU core. If there is no CPU sharing among VMs, they are less helpful. As new CPUs increasingly have more cores in each socket, the CPU sharing scenario is less common. vRead does not have any CPU sharing assumption, it also works no matter the VMs have dedicated cores or not. Besides, vRead reduces the I/O processing delay by avoiding redundant data copies between VMs thus eliminating the scheduling delay of I/O threads. MRG [27] proposes a VM scheduler specifically for Map/Reduce jobs. This scheduler keeps Map/Reduce job fairness by introducing a two-level group credit-based scheduling policy. The efficiency of map and reduce tasks can be improved by batching I/O requests within a group, hence superfluous context switches are eliminated. But, this work can not improve the I/O performance between Map/Reduce jobs and HDFS.

Functionality Offloading to the Hypervisor Offloading partial I/O operations to reduce virtualization overhead and improve I/O performance is a well studied approach. [23] proposes the idea of offloading common middle-ware functionality to the hypervisor layer to reduce context switches between the guest OS and hypervisor. Differently, vRead introduces shortcutting at the inter-VM I/O level and is applicable to efficiently read files from other VMs' virtual disks. In [40], the whole TCP/IP stack is offloaded to a separate core to reduce the I/O response time of VMs sharing the same core. vSnoop [28] and vFlood [20] mitigated the negative impact of CPU access latency on TCP by offloading acknowledgement generation and congestion control to the driver domain of the Xen VMM. However, they all focus on the CPU sharing scenarios, but vRead is applicable no matter the VMs have dedicated cores or not. Besides, they are hardly applicable to inter-VM communication on the same host, which vRead addresses.

8. CONCLUSION

We have presented vRead, a system that directly improves the performance of HDFS. We observe that traditional virtual Hadoop systems frequently move data from a disk to a datanode VM which then sends the data to a client VM via the virtual network — regardless of if the two VMs are co-located or not. Thus, each HDFS read requires at least 5 data copies which incurs I/O overhead arising from device virtualization and CPU scheduling latency among VMs and I/O threads. vRead mitigates such penalty by shortcutting the HDFS reads at the hypervisor layer. Our evaluation of a vRead prototype shows that vRead can improve I/O throughput and reduce the CPU cost of HDFS. This benefits all applications (not limited to Hadoop) storing data in HDFS. Our application case studies demonstrate vRead's applicability and effectiveness.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF under Awards 0855141 and 1219004.

10. REFERENCES

[1] Amazon EC2. http://aws.amazon.com/ec2.
[2] Apache HBase. http://hbase.apache.org/.
[3] Apache Hive. https://hive.apache.org/.

[4] Apache Sqoop. http://sqoop.apache.org/.

[5] CPU Frequency Utils. http://mirrors.dotsrc.org/linux/utils/kernel/cpufreq/cpufrequtils.html.

[6] Docker. http://www.docker.com/.

[7] Elastic Map/Reduce (EMR). http://aws.amazon.com/elasticmapreduce/.

[8] HDFS Short-Circuit Local Reads. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[9] lookbusy – load generator. http://www.devin.com/lookbusy/.

[10] Netperf Benchmark. http://www.netperf.org/.

[11] Sahara. https://wiki.openstack.org/wiki/Sahara.

[12] Hadoop Virtualization Extensions on VMware vSphere5. In *VMware technical white paper* (2012).

[13] A Benchmarking Case study of Virtualized Hadoop Performance on VMware vSphere5. In *VMware technical white paper* (2013).

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

[15] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM SOSP* (2003).

[16] BORTHAKUR, D. The hadoop distributed file system: Architecture and design. In *Hadoop Project Website* (2007), vol. 11, p. 21.

[17] CLARK, CHRISTOPHER, KEIR FRASER, S. H., JACOB GORM HANSEN, E. J., CHRISTIAN LIMPACH, I. P., AND WARFIELD., A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation* (2005), vol. 2, pp. 273–286.

[18] DONG, Y., YU, Z., AND ROSE, G. SR-IOV networking in Xen: architecture, design and implementation. In *WIOV* (2008).

[19] GAMAGE, S., CONG, X., KOMPELLA, R. R., AND XU, D. vPipe: piped i/o offloading for efficient data movement in virtualized clouds. In *ACM SOCC* (2014).

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

[21] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 29–43.

[22] 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).

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

[24] 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).

[25] HIREMANE, R. Intel virtualization technology for directed I/O (Intel VT-d). *Technology@ Intel Magazine 4*, 10 (2007).

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

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

[28] 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).

[29] KIVITY, A., YANIV KAMAY, D. L., LUBLIN, U., AND LIGUORI., A. KVM: the Linux virtual machine monitor. In *In Proceedings of the Linux Symposium* (2007).

[30] LEE, M., KRISHNAKUMAR, A. S., KRISHNAN, P., SINGH, N., AND YAJNIK, S. Supporting soft real-time tasks in the Xen hypervisor. In *ACM VEE* (2010).

[31] MACDONELL, CAM, XIAODI KE, A. W. G., AND LU, P. Low-Latency, High-Bandwidth Use Cases for Nahanni/ivshmem. In *KVM Forum* (2011).

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

[33] 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).

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

[35] MOHEBBI, H. R., KASHEFI, O., AND SHARIFI, M. Zivm: A zero-copy inter-vm communication mechanism for cloud computing. *Computer and Information Science 4*, 6 (2011).

[36] OVSIANNIKOV, M., RUS, S., REEVES, D., SUTTER, P., RAO, S., AND KELLY, J. The quantcast file system. *Proceedings of the VLDB Endowment 6*, 11 (2013), 1092–1101.

[37] RECIO, R., CULLEY, P., GARCIA, D., HILLAND, J., AND METZLER, B. An rdma protocol specification. In *IETF Internet-draft draft-ietf-rddp-rdmap-03* (2005).

[38] RUSSELL, R. Virtio – towards a de-facto standard for virtual i/o devices. In *ACM SIGOPS Operating Systems Review* (2008).

[39] SEFRAOUI, O., AND MOHAMMED AISSAOUI, M. E. Openstack: toward an open-source solution for cloud computing. In *International Journal of Computer Applications* (2012), vol. 55.

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

[41] SHVACHKO, K., HAIRONG KUANG, S. R., AND CHANSLER., R. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 2010)* (2010).

[42] SUBRAMONI, H., PING LAI, M. L., AND PANDA., D. K. RDMA over Ethernet – A preliminary study. In *In Cluster Computing and Workshops (CLUSTER)* (2009).

[43] 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).

[44] ZHANG, X., SUZANNE MCINTOSH, P. R., AND GRIFFIN, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *Middleware* (2007).