IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 30, 2021, accepted June 10, 2021, date of publication June 17, 2021, date of current version June 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3090308

Block-Level Storage Caching for Hypervisor-Based Cloud Nodes

BYUNGCHUL TAK[1], (Member, IEEE), CHUNQIANG TANG[2], (Senior Member, IEEE), RONG N. CHANG[3], (Senior Member, IEEE), AND EUISEONG SEO[4], (Member, IEEE)

[1]Department of Computer Science and Department of Data Convergence Computing, Kyungpook National University, Daegu 41566, Republic of Korea
[2]IBM Research, Yorktown Heights, NY 10598, USA
[3]IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA
[4]Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea

Corresponding author: Euiseong Seo (euiseong@skku.edu)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government [Ministry of Science and ICT (MSIT)] under Grant NRF-2019R1C1C1006990 and Grant NRF-2021R1A5A1021944.

ABSTRACT Virtual block devices are heavily used to fulfill the block storage needs of hypervisor-based *virtual machine* (VM) instances through either local or remote storage spaces. However, a high degree of VM co-location makes it increasingly difficult to physically provision all the necessary block devices using only local storage space. Also, the local storage performance degrades rapidly as workloads interleave. On the other hand, when block devices are acquired through remote storage services, the aggregated network traffic may consume too much cluster-wide network bandwidth in a cloud data center. In order to solve these challenges, we propose a caching scheme for virtual block devices within the hypervisor. The scheme utilizes the physical node's finite local storage space as a block-level cache for the remote storage blocks to reduce the network traffic bound to the storage servers. This allows hypervisor-based compute nodes to serve the hosted VMs' I/O (Input/Output) requests from its local storage as much as possible while enabling VMs to exercise large storage space beyond the capacity of local disks for new virtual disks. Caching virtual disks at block-level in a cloud data center poses several challenges in maintaining high performance while adhering to the virtual disk semantics. We have realized the proposed scheme, called **vStore**, on Xen hypervisor nodes with factual assessment on its design effectiveness and implementation efficiency. Our comprehensive experimental evaluations show that the proposed scheme substantially reduces the network traffic (49% on average), and incurs less than 12% overheads on the storage I/O performance.

INDEX TERMS Virtual block device, storage cache, virtualization, network storage.

I. INTRODUCTION

Modern cloud service infrastructures have reached an unprecedented level of performance and scale spanning multiple geographies to meet rapidly growing demands of the user base [1]–[6]. Such a massive scale is due, in part, to the proliferation of data-driven application workloads in the areas of big data and deep learning. Thus, the requirements for delivering sufficient networked storage and network bandwidth have become a critical design focus for a cloud infrastructure under development or improvement.

Let us consider a cloud application scenario in which machine learning experts try to build optimal deep learning models through repeated training on massive amounts of data stored in the cloud. If data happens to be located in a separate network unit in the hierarchy, be it rack-level, pod-level, plane, region, or even data center level, the data has to travel over the network *en masse* repeatedly to where the training is conducted [7], [8]. This will exhaust an excessive portion of the data center network bandwidth. On the other hand, it is too costly and infeasible to have such a huge amount of data replicated at multiple levels of the network hierarchy just to mask the latency of remote accesses. Some form of data caching scheme needs to be employed in the storage system on which the computation occurs.

In this work, we aim to utilize the local storage space of cloud nodes as the place to build such caching scheme. The current trend of the local storage medium for cloud nodes is the increasing use of the Solid State Drives (SSDs) as compared to traditional Hard Disk Drives (HDDs) [9]–[11].

The associate editor coordinating the review of this manuscript and approving it for publication was Muhamamd Aleem.

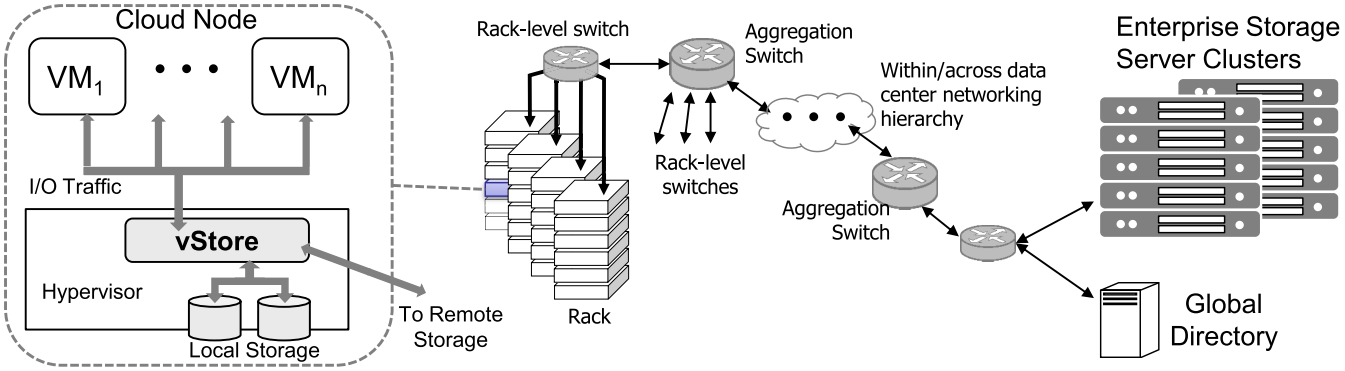


FIGURE 1. A sample model of cloud storage environment and the location of our technique. The enterprise storage servers may reside within the same data center, or they could be even farther away in geographically distant data centers.

However, although I/O performance of SSD is an order of magnitude higher than HDD, it is expected that HDD will continue to remain as the major local storage medium. It is because the improvement rate of byte-per-dollar of SSD is far behind the growth rate of data size in general. It is not expected that the byte-per-dollar of SSD will exceed that of the HDD in the near future. Thus, we target HDD-based local storage as the place to build our technique.

As a proving ground of our concept, we assume an environment where there are a large number of cloud nodes equipped with HDDs relying on the use of the *virtual block device* mechanism. We aim to build a storage caching solution at the individual cloud node level where the hypervisor hosts multiple virtual machines (VMs). There are a few factors that may hinder the VMs from achieving high performance and scalability. First, the current practice of a high degree of VM co-location makes it difficult to provision all the virtual block devices only on the local storage space. A physical cloud node hosting VMs needs to provide virtual disks to many VMs on the node, and the local storage space may not be enough to hold the sum of the storage spaces requested by all VMs. Also, it is difficult to anticipate the upcoming demands since VMs may dynamically migrate between the set of cloud nodes. Second, although the use of remote network storage as the source of virtual block devices alleviates such issues, heavy I/O traffics may overload the cluster-based networks in use. Under the hierarchical network topology commonly employed in the data center network, aggregated heavy I/O traffic may lead to serious congestion at the bottleneck and, thus, degrade the I/O performance of VMs.

We propose an approach to mitigating those issues by utilizing the VM-hosting node's local storage as a *block-level* cache for the remote network storage in use (e.g, those deployed in a cloud data center). This cache allows a specific hypervisor-based node to serve its VMs' disk I/O requests using the node's local storage most of the time while providing the illusion of larger storage space for holding new virtual disks. Although caching has been studied in many different contexts, there are limited investigations on the feasibility of block-level (storage) caching for VMs in cloud data centers and on the performance impact under various workloads.

Hypervisor-side caching of VM I/Os at the block-level requires addressing several challenges. First, the on-disk cache block layout must be carefully designed to minimize the performance impact. The cache block placement policy that gives us higher sequentiality is preferred. Second, the cache integrity should survive in the event of host failures or crashes. Write I/O operations issued from within the VMs and returned with confirmation should not violate the write semantics caused by losing the written data (even on sudden failures such as the power failure). Third, we need to have an adaptive destaging mechanism that gradually sends modified blocks to the remote storage without causing performance interference to the normal I/O operations.

For systematic evaluations, our block-level caching mechanism has been implemented in the Xen virtualization platform. According to our evaluation, it shows that the block-level caching is effective in handling various workloads without incurring high overheads. The reduction of network traffic is more than 49% on average while the overall performance overhead is kept less than 12%.

The rest of this paper is organized as follows. In Section II, we elaborate on the addressed issues. In Section III, we describe the architecture and design of our caching scheme. Section IV describes the implementation details. In Section V, we empirically evaluate the efficacy of our system under various workloads. We give related work in Section VI, and conclude in Section VII.

II. MOTIVATION AND CHALLENGES

We adopt a simplified storage model of the virtualized cloud environment in which there are several network storage server clusters and a large number of virtualization nodes as depicted in Figure 1. Major components of our storage model are described below.

- **Cloud Nodes:** Physical cloud nodes host a large number of VMs. We want to use the local storage space of the cloud nodes as a block-level cache and effectively

provide much greater storage space to the VMs. The hypervisor running in the host attaches one or more virtual block devices to the VMs upon requests. Virtual block devices are seen as raw block devices to the VMs. VMs need to install file systems on the block devices. The hypervisor can only see the block I/O requests generated from the VMs.

- **Remote Storage Servers:** Remote storage servers provide networked access to the storage services for cloud nodes. The remote storage typically offers either the block-level interface such as iSCSI or file-level interfaces. In the case of a file-level interface, the hypervisor directly mounts a directory in the storage server to the local file system. A file is created on this space to hold the virtual block device for the VMs. Independent of the interface to the remote storage servers, the VMs and the hypervisor operate at the block level.
- **Directory Server:** These servers are used by the cloud nodes to look up the network location of the available storage services. Hypervisors query the directory server to obtain the access point information and then initiate the connection to the storage servers.
- **Network Infrastructure:** Network bandwidth between servers within the rack is sufficient for high-speed communications. However, the network bandwidth between racks is limited by the capacity of the rack-level switch which is about 10 times less than within-rack bandwidth [12]. There are higher-level aggregation switches that connect several rack-level switches.

A. MOTIVATIONS

Provisioning virtual block devices to VMs can be done using either the local storage devices of the cloud node in use or via remote storage systems. The current trend is to rely more on remote storage systems rather than to use local storage. Using the local storage to meet all local VMs' demands for virtual block devices can be challenging. Let us suppose an unfavorable scenario where a cloud node is busy with VMs running I/O-intensive workloads. Then, it may not be viable to deliver the desired degree of I/O performances due to performance degradations from heavily interleaved workloads. Nowadays order of tens of VMs per cloud node is common, and, thus, local storage's I/O bandwidth is severely limited to provide acceptable I/O performances for the co-located VMs. In addition, there are equally serious problems with the use of local storage such as complications to the VM live migration and the subtlety with data persistence on host failures.

On the other hand, using remote storage requires handling different issues. It may incur excessive network traffic if a large number of VMs happen to run I/O-bound applications concurrently. Let us assume that the disk access workloads of a VM can mildly reach about 10-20 MB/s with a provisioned network speed of 10 Gbps (i.e. 1250 MB/s bandwidth available). Since there are usually about four uplinks to the rack-level switch, 62-125 VMs are sufficient to saturate the network which is not a large number of VMs to be hosted in four racks. It gets worse if some VMs generate sequential disk accesses of 50 MB/s or higher. The limitation of bandwidth inherent to hierarchical structures of the data center networks is actively being researched from the network perspective [8], [13]. We believe our system-oriented approach will also help mitigate the limited bandwidth problem.

B. CHALLENGES OF APPLYING BLOCK-LEVEL CACHE

In leveraging the local storage space as a block-level cache for the virtual volumes of VM instances, two main challenges must be addressed: performance and data integrity.

1) PERFORMANCE ISSUE

Several factors affect the performance of the block-level cache scheme. Maintaining the reasonable level of performance of the cache requires handling the following challenges.

- **The loss of sequentiality of block layout:** The blocks adjacent in a virtual disk may not be adjacent anymore within the cache. This renders otherwise sequential access to become random access impacting the sequential I/O performance. Ideally, we want to duplicate the block layout of the original virtual disk blocks (whether local or in NAS) in the local cache as much as possible. Also, local cache handling operations themselves must be efficient so that the performance of a warmed-up cache is comparable to that of normal direct-attached disks.
- **Metadata overheads:** If not designed carefully, the on-disk metadata required for the block-level cache may incur additional disk seeks.
- **Premature eviction:** There can be dependencies among outstanding requests. It is ideal to avoid cache eviction when there are requests on that entry in the queue.

2) DATA INTEGRITY ISSUE

The disk write operations from the VMs should maintain the data integrity even under the sudden failures of the physical host. If the host crashes while updating either the metadata or the data blocks in the block cache, we need a mechanism for detecting such accidents upon restart. Otherwise, an incorrect state may persist in the cache and become the cause of silent data corruption. In other cases, disk write operations that returned with success must have been persisted. If the caching mechanism keeps the data of the already returned write operations only in memory even for a short while, the data may get lost on crash. This is a violation of the write semantics that VM expects. Addressing these challenges involves making careful trade-offs between the performance and data integrity.

III. SYSTEM DESIGN

A. ARCHITECTURE OVERVIEW

Figure 2 illustrates the components of our hypervisor-side caching scheme, called **vStore**. VMs are assumed to issue block I/O requests of the form (address, count) in the unit of

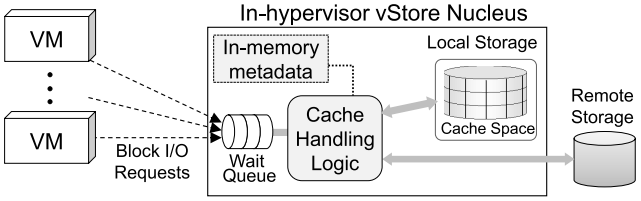


FIGURE 2. Logical architecture of vStore caching system.

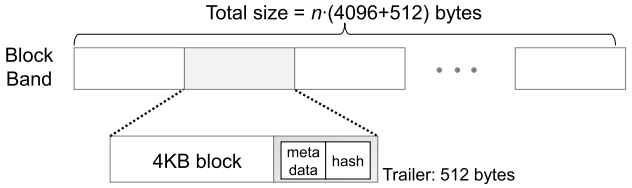


FIGURE 3. The structure of a block band. A block band contains *n* consecutive 4096 + 512 byte blocks.

sectors. They are all received by the hypervisor, where our cache handling logic intercepts them for processing prior to the normal I/O handling. The **vStore** nucleus runs within the hypervisor and implements the block I/O handling functionality. In the Xen hypervisor environment where the split driver model is used, it is on the back-end device driver.

The **vStore** nucleus contains a queue for holding pending I/O requests, an in-memory metadata structure, and a cache handling logic. Incoming requests are all first received by the queue. It checks for any conflicting I/O requests where request blocks are under update or eviction. If no such conflicts exist, it is passed to the cache logic. The logic looks up the metadata structure to gather various metadata information needed for the cache handling logic. Lastly, the logic decides whether to access the remote network storage or the local cache for processing the request.

1) SIZE OF I/O OPERATION UNIT

We allow **vStore** to be configured to operate at arbitrary block size (other than 4KB) while interacting with the remote storage servers. This is because the state-of-the-art storage servers offer better throughput under a larger size of I/O unit [14]. In addition, a larger block size is beneficial to the **vStore** as well since it reduces the number of entries of the in-memory metadata. A larger I/O unit size may consume more network bandwidth, but the benefit from caching turns out to be greater as can be seen in Figure 8 (b) of Section V. The *block band* is defined to be this default unit size in **vStore**.

2) METADATA

vStore keeps various cache information as a metadata structure on the disk. Metadata fields we define are:

- Virtual disk ID (2B): Used by **vStore** to identify the remote storage servers. This field allows **vStore** to recognize the remote storage servers in the events of detach and reconnect. The scope of the ID is within a hypervisor.
- Sector address (4B): Remote address of the sector.
- Read count, Write count (2B each)
- Accessed time (8B)
- Dirty bit, Valid bit, Lock bit (1 bit each): Lock bit is set if the cache entry is being updated.
- Bitmap (variable size): Single bit maps to one 4KB within the *block band*. It allows **vStore** to update only those requested 4KB blocks within the block band. Otherwise, **vStore** has to fetch from the remote storage the entire block band before applying the update.

For quick access during the cache handling, **vStore** also keeps the copy of the metadata in the memory. When it is modified, the on-disk part is updated to maintain the metadata consistency. Metadata is used during the crash recovery process. Upon reboot, **vStore** goes through the metadata entries to identify any modified blocks not yet flushed to the remote storage.

The use of on-disk metadata can be the source of performance degradation if designed without care. Normally one write request takes two accesses, one for updating the metadata, another for the actual write of the data. In the design of **vStore**, we put the metadata and the data adjacent to each other so that they can be updated together with only one write operation. As mentioned above, a copy of the metadata is kept in memory to expedite the read when needed. The default *block band* size of **vStore** is chosen to be 256KB to keep the metadata size small enough to be loaded into the memory.

3) CACHE STRUCTURE

The cache in **vStore** is a set-associative cache with the write-back policy. We view our cache structure as a table. Each row in this table contains multiple *block band*s. Block bands contain contiguous blocks from one remote storage server, but adjacent block bands may be from different storage servers. The set associativity and cache row length are configurable.

As shown in Figure 3, one block band contains a sequence of 4KB blocks and a trailer. The size of the trailer is 512 bytes and it contains the hash field as well as the metadata. The hash field records the hash value of the block and it is written upon the write operations. In order to avoid additional disk I/Os due to separate metadata accesses, we build this one block band entry made of multiple 4KB blocks and the trailer pair, and write to the cache in a single I/O operation. The hash value is intended to be used to check whether the crash has occurred during the write operation. If the crash happens before the write is completed, one entry of the block band will have the hash value in the hash field of the trailer that does not match the hashed value of the data block. Any entry that has such unmatched hash can be ignored during the recovery process because the write is not completed and has not been returned to the VM.

For read operations, **vStore** reads the entire entry that has the data block and the trailer. Although the read size is increased by 512 bytes per single 4KB block, we find that the

overheads are small since the read will be sequential. In the worst case, the read is expected to show the slow-down of no more than $512/4096 = 12.5\%$.

4) CACHE REPLACEMENT

In designing the cache replacement algorithm for **vStore**, we take into account the sequentiality of the blocks within the cache space. Conventional replacement policies such as LRU (Least recently used) or LFU (Least frequently used) policies are potentially detrimental to the I/O performance since they may end up separating blocks that should be adjacent. VM may have issued consecutive reads or writes, but cache replacement may turn them into random accesses. In **vStore**, we want to maintain the sequential I/Os to be also sequential in the cache space.

Cache space on the local storage is a two-dimensional space with rows and columns of blocks. Since this cache space shares blocks from multiple virtual disks, we want to separate the cached blocks of different virtual disks to different regions of the cache space as much as possible. If blocks of two or more virtual disks compete for the overlapping cache space starting from the first row, some active blocks may have to be evicted and brought back unnecessarily. In order to separate them apart, we make use of the *virtual offset row*. The *virtual offset row* for a specific virtual disk represents the first row within the cache space that the block address of 0 is located. The idea is to apply different offsets to different virtual disks so that their starting address in the cache space differs. For example, let us assume two virtual disks and the cache associativity of 4. There will be four rows within a set. The virtual disk VD_1 is assigned the *virtual offset row* of 0 and the VD_2 is assigned 2. It is possible that too many blocks from one virtual disk may be cached that it may start to overlap with the rows that other virtual disk's cached blocks reside. Instead of partitioning the cache space, we allow cached block regions of different virtual disks to overlap to increase the utilization of the cache space.

Newly arrived blocks are assigned the cache location in **vStore** as follows. The cache address is first computed from the sum of the *virtual offset row* and the block address. If the computed cache address holds a dirty valid block, it looks for other available blocks within the set. In selecting the victim, **vStore** uses the following 6 factors - *(i) how recent the blocks are?, (ii) is block dirty?, (iii) current block sequentiality, (iv) new block sequentiality, (v) current distance and (vi) new distance*. The sequentiality factor considers how sequential the blocks are within the cache before the eviction and after the eviction. Our intention is to discourage the sequential blocks from losing the sequentiality by eviction. We also want to select victims in a way that increases the sequentiality. The distance factors for (v) and (vi) refers to the row differences between the *virtual offset row* and the new row. Preference is given to the rows that are closer to the *virtual offset row*. Combining all these 6 factors, we form the following equation to come up with the score. Let us denote x_n as representing six factors.

$$Y = \sum_{n=0}^{5} \alpha_n \cdot x_n \tag{1}$$

The α coefficients indicate the weights assigned on factors. The weights are configurable. In current **vStore** implementation, we have assigned equal weights to all factors. This score Y is computed for all entries within the set and the one with the smallest value is chosen as a victim. The weights of factors can be varied to fine-tune the victim selection. These weights are left as configurations to the system administrators who employ **vStore** so that they can be adjusted to the workloads and settings.

5) CACHE HANDLING OPERATIONS

The primary focus of designing the cache handling operations in **vStore** is the performance. Since there will be unavoidably additional disk accesses due to cache handling, it is expected that there will be overheads. Our goal is to minimize performance degradation even under such disadvantages. The secondary focus is the integrity of data. Our design choice is, as described earlier, to add a trailer to each data block so that consecutive reads or writes remain consecutive. The cost of this strategy is the increased data size from trailers. But, reading or writing increased data size has a much smaller impact on the performance than breaking the sequentiality. The existence of a trailer helps protect cache data integrity from sudden failures of the host.

Read and write handling of the cache data is designed in a way that data integrity is maintained upon sudden failures. In read handling, **vStore** returns the data to the VM as soon as it receives the block from the remote storage. Housekeeping tasks such as metadata updates and cache updates are carried out next so that the perceived I/O read request is fast. This strategy is safe because there will be no loss of data even if the host encounters failures before completing the cache and metadata updates.

It is not possible to do the same for the write operations. Metadata and cache updates must be completed before they return to the VM. Otherwise, unexpected host failures can result in the violation of write semantics for VMs. If write operation returns, it should be guaranteed that the data is persisted in the **vStore** cache space. Thus, write handling can be the cause of most of the overheads. If the cache flush is required, the disk I/Os can be as many as four times in the worst case as described below.

- Local read of a (target) block to be evicted.
- Remote read of blocks in the block band: This may be necessary if there are any invalid blocks within the block band containing the block to be evicted. Target block will be merged to this to form one complete block band.
- Remote write of one complete block band.
- Local write of a new block to the vStore cache.

B. DESTAGING

Our **vStore** design includes the destaging capability in order to flush out the modified blocks in the cache to the remote storage in a controlled manner. Destaging functionality aims to maintain lower than designated proportion of dirty blocks in the cache space whenever possible. Since the flushing action of dirty blocks at the time of high normal I/O requests may impact the response time significantly, **vStore** tries to use only the idle times to flush parts of dirty blocks without affecting the normal I/O performances. Successful destaging allows us quick detachment of virtual disks for VM migrations since there are fewer number of blocks to flush.

1) DESTAGING MECHANISM DESIGN

In designing the destaging mechanism, we have the following goals. First, the performance overheads from destaging activities over **vStore**'s normal operations should be controllable. Since destaging activity increases the number of I/Os, it will incur some performance penalty. The mechanism should allow us to specify how much overheads we are willing to take. Second, destaging behaviors should automatically adjust to the changing workloads. When the system is busy with heavy I/Os, destaging should stand by. Only when the system becomes idle, the destaging should activate.

Destaging action is triggered based on the configured parameter, called LOD (Level of Dirtiness). Our approach is to determine the amount of destaging, called *window data size*, based on both the remote and local I/O latencies. Upon the start of destaging, **vStore** first determines the number of blocks to transfer within a given time window t. We introduce a parameter s_t, which is the bytes-per-msec (BPMS) **vStore** is allowed to send over the network. The parameter s_t is dynamically adjusted over time as I/Os and network conditions change. Here, the data includes both the normal network traffic as well as the traffic induced by the destaging mechanism.

In order to compute s_t, we combine the flow control techniques from FAST TCP [15] and PARDA [16]. Let $\mathcal{M}$ be the network latency we want to maintain. We compute and keep the exponentially weighted moving average of the actual latency as in $\mathcal{M}_t = \alpha\mathcal{M}_{t-1} + (1-\alpha)\mathcal{M}$, with the α smoothing parameter. Then, we compute s_t as follows using another smoothing parameter γ. It shows that the remote latency $\mathcal{M}_t$ and s_t forms an inverse relationship.

$$s_t = (1-\gamma)s_{t-1} + \gamma\frac{\mathcal{M}}{\mathcal{M}_t}s_{t-1} \tag{2}$$

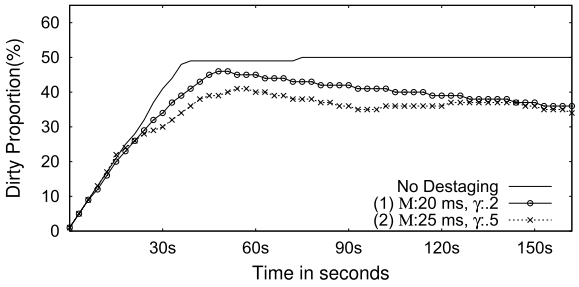
Let $\mathcal{N}$ be the local I/O latency we target. Also, let us denote r_t as the number of bytes per msec for local disk I/O. We can compute r_t in a similar way with $\mathcal{N}_t = \alpha\mathcal{N}_{t-1} + (1-\alpha)\mathcal{N}$.

$$r_t = (1-\gamma)r_{t-1} + \gamma\frac{\mathcal{N}}{\mathcal{N}_t}r_{t-1} \tag{3}$$

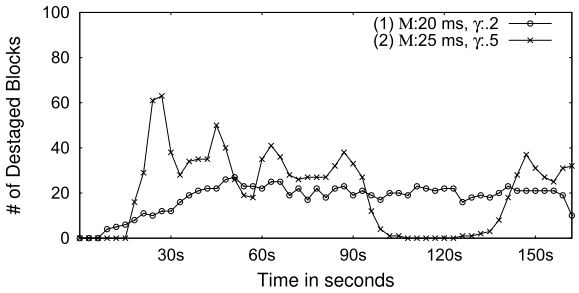
Minimum of s_t and r_t is set to be the *window data size*. The number of I/Os for destaging, u_t at time t is calculated as follows.

$$u_t = (min(s_t, r_t) \times \tau_t - P_t)/B \tag{4}$$

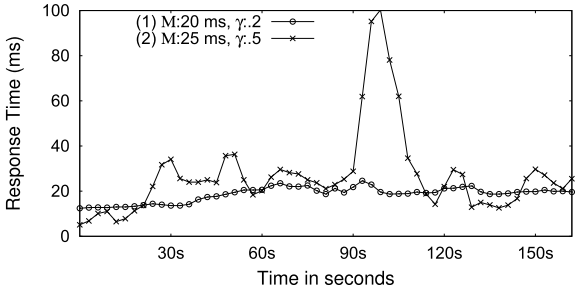
The τ_t is the time (in msec) duration between t and $t-1$. Variable B is the block band size. Variable P_t is the pending I/O requests generated from the normal remote accesses, not the **vStore** operations. Destaging starts when u_t becomes greater than zero.



(a) Dirty proportion change over time.



(b) I/O operations triggered by destaging over time.



(c) Response time change over time.

FIGURE 4. Destaging operation over time. Sudden burst of response time in (c) is due to our intentionally inserted interference to observe destaging adjustment capability to the latency changes for the configuration (2).

Figure 4 illustrates an example of destaging in action using two different set of configurations operated under **Varmail** workload in the **Filebench** [17] benchmark. The first configuration of ($\mathcal{M} = 20$ms, $\gamma = .2$) models a mild destaging policy. The second one, ($\mathcal{M} = 25$ms, $\gamma = .5$) is more aggressive towards destaging. The destaging trigger point is set to be 10% of dirty blocks in the cache space. As the workload progresses, the dirty proportion rises at the early stage, and as soon as it crosses the 10% threshold, the destaging kicks in at about 20 seconds. Configuration 2 shows a faster reduction of

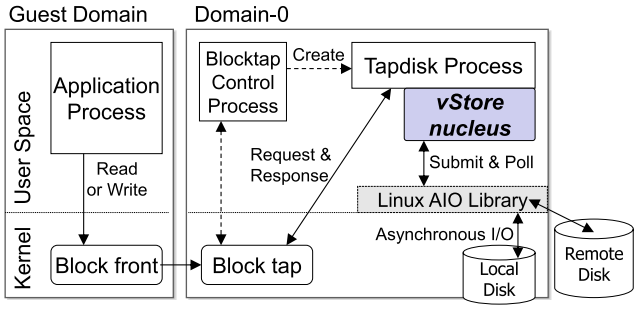


FIGURE 5. Implementation in Xen environment.

dirty blocks in Figure 4 (a). In order to observe the effectiveness of destaging adjustment capability to the network loads, we have intentionally created an interference at 90 seconds for configuration 2. The workload was a heavy copying of large files over the network for the duration of 30 seconds. As can be seen from Figure 4 (b) and (c), the response time rapidly rises and the destaging rate drops immediately in order to prevent the network latency from deteriorating due to destaging. As soon as the interference disappears, it resumes to the previous level of destaging showing that our mechanism works effectively.

C. IMPLICATION TO VM MIGRATIONS

Live migration of VMs is an important functionality for the flexible management of VMs within the data center. However, the client-side caching mechanisms such as **vStore** may potentially add overheads to live migration operations. The migration technique typically depends on the use of remote storage so that the file system state transfer becomes simple volume detaching and attaching actions to the target host. With this support, the live migration only needs to handle the transfer of memory states. However, the existence of cached data on the local host by **vStore** implies that the volume cannot be immediately detached. Our current solution is to flush locally cached blocks to the remote storage first as a preparation for the live migration. Although this adds delays, our destaging mechanism will help reduce the number of blocks to flush to a low number. Efficient integration of **vStore** with the live migration is our future research topic.

IV. IMPLEMENTATION

The **vStore** is implemented in Xen environment using the **blktap** mechanisms [18] as depicted in Figure 5. Xen follows the split driver architecture for I/Os where the front-half of the driver (**blkfront**) is placed within the guest VM and the back-end (**blkback**) in the Dom0. Xen's **blktap** mechanism replaces the back-end part of the split driver into the block tap device. Then, all the guest VM's disk I/Os, once they leave the front-end, are redirected to the user process, called **tapdisk** in Dom0. This mechanism allows us to capture all the block requests of the guest VM and implement the desired function (caching in this work) in the user space conveniently.

The tapdisk process opens a block device by the specified types. These types include a block device that performs synchronous I/Os and asynchronous I/Os. If it is opened under the synchronous mode, all block requests are processed as normal via usual read, write, open, close system calls. However, if it is opened in asynchronous mode, tapdisk will invoke Linux AIO library functions to handle the block requests. In **vStore**, we create a new type of tapdisk mode and register the set of callback functions. Internally, we build on top of the Linux AIO library.

Most of our code is contained within the new type of tapdisk driver, named as **block-cache**, located under `blktap/drivers` directory within the Xen source tree. The tapdisk driver is a group of following callback functions: `open`, `queue_read`, `queue_write`, `submit`, `close`, `do_callbacks`, `get_parent_id` and `validate_parent`. Among these callback functions, we modified `open`, `queue_read`, `queue_write`, and `close` based on the `block-aio` driver. The *In-memory metadata* component of Figure 2 is declared within this driver code as a global integer array. The cache space is a file (`cache_file.img`) at a predefined location and it is opened at the `open` callback function with `O_DIRECT` switch when this driver initializes. The *Cache Handling Logic* component depicted in the Figure 2 is implemented within the `queue_read`, and `queue_write` callback functions. Finally, this tapdisk type is defined in the `tapdisk.h` header file as a `tap_disk` structure.

V. EVALUATION

A. EXPERIMENTAL SETTINGS

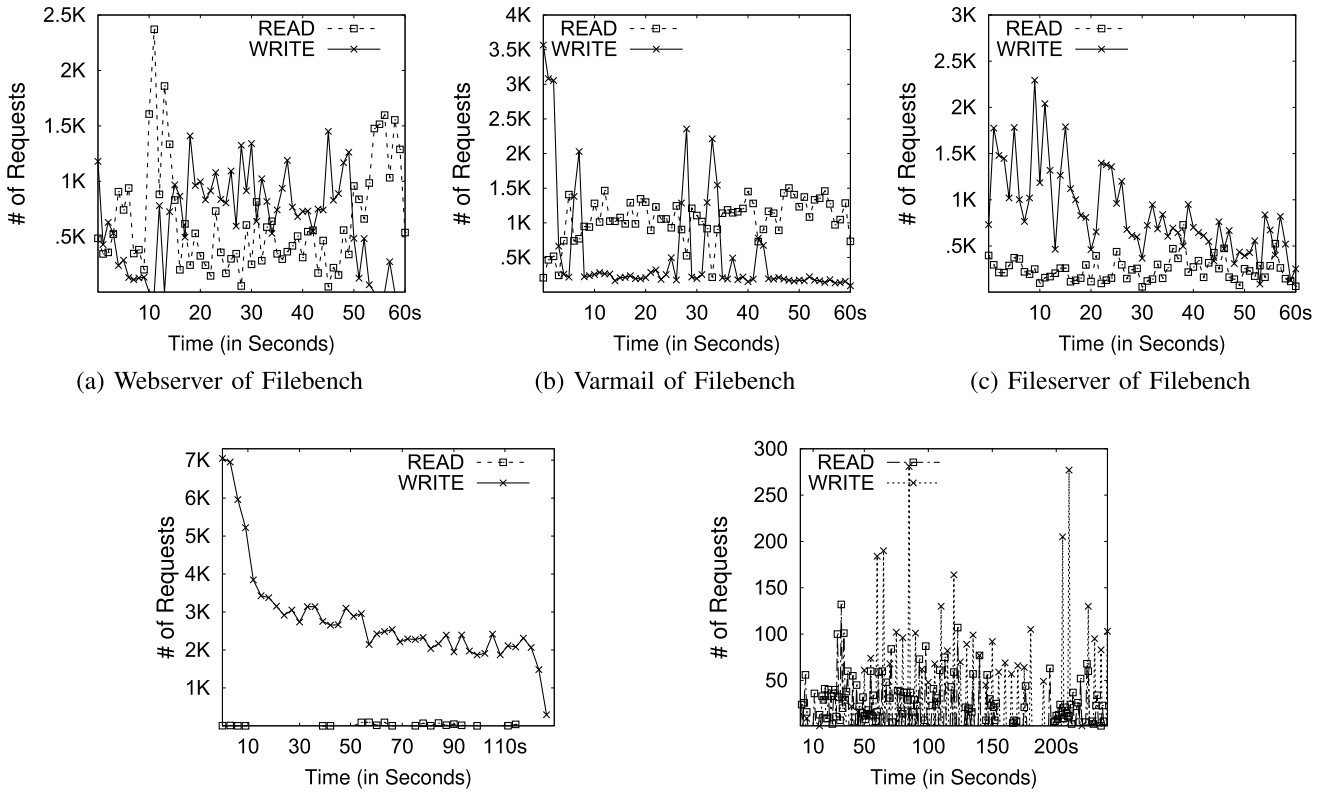
Our evaluation environment consists of host machines with dual Intel(R) Xeon(TM) CPU of 3.40 GHz. Hosts are placed within a rack of 20 physical machines and they communicate at the speed of 1Gbps. Virtual machine images are provided to the host on the NFS (Network File System) volume. This NFS storage space is for storing the VM images and does not count as a remote storage server in our scenario. We have attached this separate storage volume to the VM and run the experiments on them so that we can leave the local storage entirely to be used as a cache space. Our VMs are made lightweight with 512MB memory running on Linux.

The role of the remote storage server is played by the separate host in a separate rack and subnet. We have installed TCP version of **NBD** (Network Block Device) [19] on them to make it function as the remote storage server. Our test VM becomes the nbd-client. NBD server space can be accessed via regular open, read and/or write calls onto the `/dev/nbd0` device.

B. BENCHMARKS

These three workloads are used for our evaluations. They exhibit distinct workload characteristics as described below.

- **Filebench** [17]: A flexible file system benchmark where it allows users to specify desired behaviors



(a) Webserver of Filebench (b) Varmail of Filebench (c) Fileserver of Filebench

(d) Postmark (e) Gcc

FIGURE 6. Workload characteristics of five workloads used in the evaluation. We can observe variety of read-write block request ratios from the hypervisor side.

TABLE 1. Read-write ratio comparison of workloads from the application and the hypervisor level.

	Filebench			Postmark	Gcc
	Webserver	Varmail	Fileserver		
At the application level					
IO/s	155.8	261.9	354.5	131.0	N/A
MB/s	3.1	5.1	4.2	26.6	N/A
Read-Write Ratio	10:1	1:1	1:2	N/A	N/A
At the hypervisor level					
Req per sec	1300	1635	1114	2828	46.4
Read	39K	64K	15K	0.6K	3.6K
Write	38K	34K	52K	121K	10K
Read-Write Ratio	1:1	1.9:1	1:3.5	1:205	1:2.8

with custom workload model language. There are several pre-built workloads in the **Filebench** benchmark. Among these, we have selected **Webserver**, **Varmail**, and **Fileserver** workloads.

- **Postmark** [20]: It is a file system benchmark initially created by NetApps to correctly model the pattern of small ephemeral files commonly found in the Internet software such as email, news and e-commerce.
- **Gcc** build: A build workload of gcc-core-3.0 source package. This workload is known to generate and delete many small temporary files [21].

Figure 6 depicts the read and write patterns of all the workloads used in the evaluation over time. The **Filebench** workloads are run for 60 seconds, **Postmark** 120 seconds and the **Gcc** workload about 3 minutes, respectively. We can observe that each workload has its unique read-to-write ratio. The **Filebench Webserver** shows about the same ratio of reads and writes whereas the **Filebench Varmail** shows read-heaviness. The **Filebench Fileserver** and the **Postmark** workloads exhibit strong write-dominant workloads, but the **Postmark** is more write-oriented. The **Postmark** benchmark has the highest workload intensity among the workloads we use. The **Gcc** workload intensity is much smaller compared to others (See the y-axis unit in Figure 6).

Table 1 compares the read vs. write ratio observed from the application and from the hypervisor. The workload observed from the hypervisor side is very different from the application level. This is due to the high locality of data blocks.

C. EXECUTION TIME OVERHEAD

Our base of comparison is against the local storage's performance as opposed to the remote storage's performance. This is because the performance of the remote storage server can vary from a high-end enterprise-grade storage server to a low-end storage server built from commodity hardware components. Comparing with the performance of the high-end storage server will always put **vStore** at a disadvantage. The

TABLE 2. Measurement of the execution time overhead.

	Filebench Benchmark									Postmark Benchmark				Gcc	
	Webserver			Varmail			Fileserver								
	op/s	MB/s	Overhead	op/s	MB/s	Overhead	op/s	MB/s	Overhead	Elapsed Time(sec)	Throughput Read(MB/s)	Throughput Write(MB/s)	Overhead	Elapsed Time(sec)	Overhead
AIO	233.42	4.66		974.58	3.36		393.56	4.7		125	9.7	11.8		214	
AIO-512	213.44	4.26	**8.6%**	925.42	3.2	**5.0%**	374.58	4.5	**4.8%**	133.4	9.1	11.1	**6.7%**	214	**0%**
vStore	205.92	4.08	**11.8%**	911	3.18	**5.4%**	363.72	4.32	**7.6%**	135.8	8.9	10.9	**8.6%**	214	**0%**

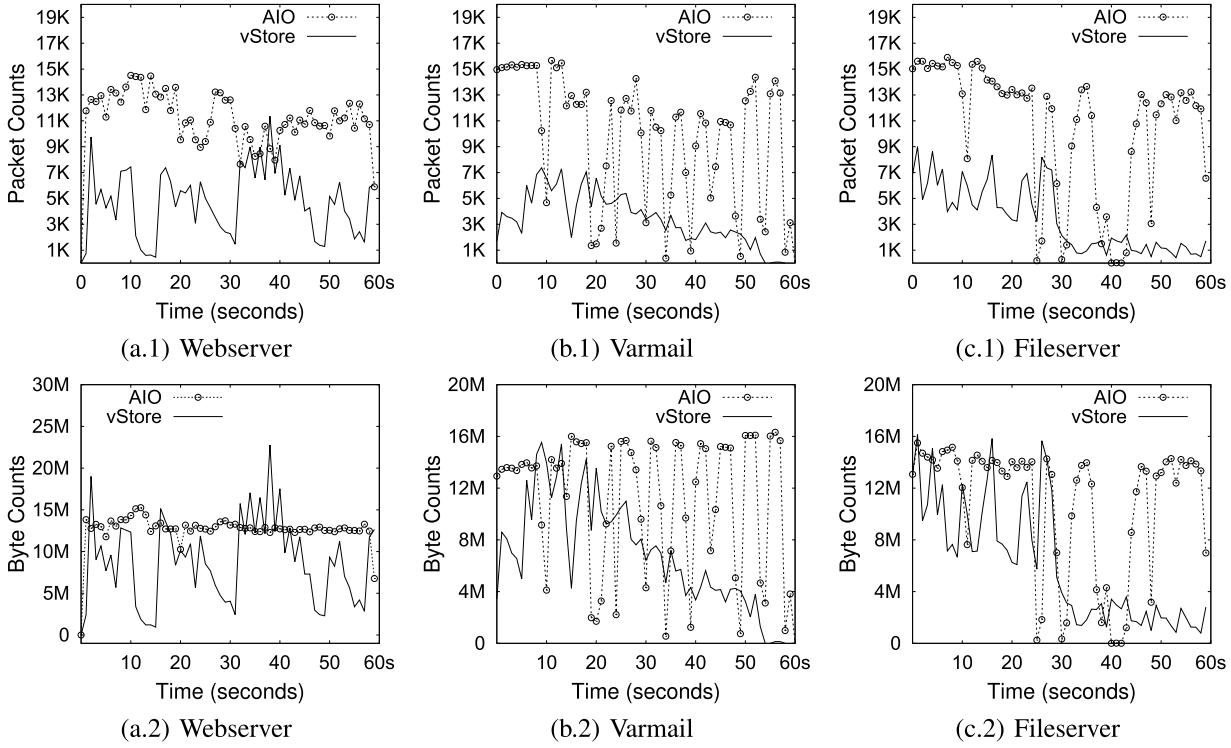


FIGURE 7. Network packets and bytes produced by AIO and vStore.

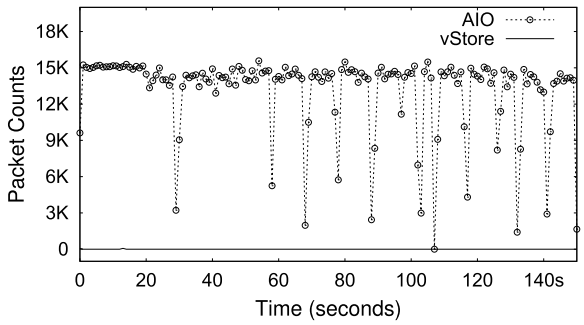
opposite scenario is also possible. Thus, comparing vStore's performance against local storage allows us to see how much true overheads **vStore** incurs. It is expected that the execution time overheads will be at least 12.5% because of the 512 byte trailer at the end of each block. We are interested in investigating how much additional overheads exist in our design.

We have prepared a virtual disk for the VMs under test using Xen's AIO tapdisk on the vStore cache file. This VM image file resides on the local disk so that all the VM's I/O requests become the I/O on the local disk (i.e. **vStore** cache space). Using the local disk and the same disk area on this local disk makes the comparison fair. The performance comparison is between the local disk accesses via AIO tapdisk with and without **vStore** caching.

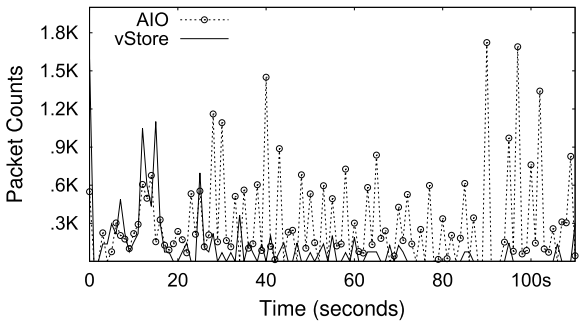
The execution time performance overheads are shown in Table 2. Native disk performance is shown with the label AIO. It is using Xen's AIO tapdisk via Linux Asynchronous library [22]. The label *AIO-512* is a specially modified AIO where all 4KB blocks are extended to have additional 512 bytes to simulate **vStore**'s trailer size. It is intended to reveal how much just the addition of a trailer affects the performance of the AIO mechanism. The goal is to find out how much of the **vStore**'s performance is due to the trailer and how much due to the cache handling logic. The performance number of *AIO-512* can also be considered as the best-case performance **vStore** can achieve.

Table 2 shows that **vStore** has less than 12% execution time overheads in all workloads. It can be seen that the performance overheads of *AIO-512* from AIO are significant so that the pure **vStore** overheads without the trailer part are less than 2%. The overheads difference between *AIO-512* and the **vStore** is high when the workload is heavy as in **Filebench Webserver** and **Filebench Fileserver** workloads. Overheads difference is minimal when the workload is moderate (i.e. **Filebench Varmail** and **Gcc** workloads).

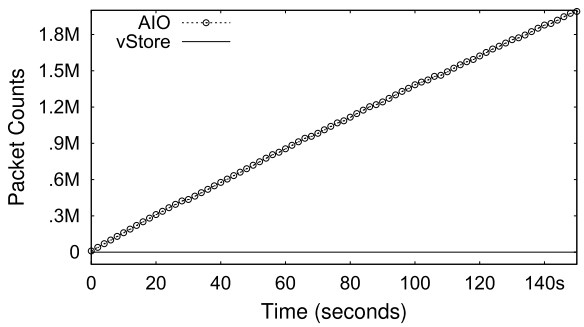
The previous five workloads are a diverse mixture of reads and writes with changing sequentiality. It is useful to observe the overheads by separating the mixture of different types of workloads and applying them independently so that we can see what type of workloads affect the overall performance more than the others. To this end, we have evaluated **vStore** on synthetic workloads to better understand the overheads



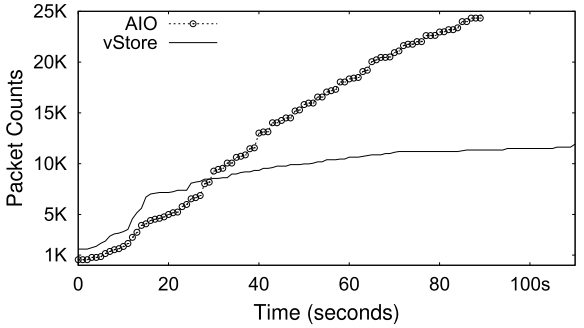
(a.1) Packets generated by Postmark over time



(b.1) Packets generated by Gcc over time



(a.2) Cumulative series of packets by Postmark



(b.2) Cumulative series of packets by Gcc

FIGURE 8. Network packets (normal and cumulative) produced by AIO and vStore. In the Postmark workload, almost all packets are absorbed by the vStore cache.

TABLE 3. Overheads measured on various synthetic workloads.

	AIO	AIO-512	vStore
	op/s	op/s (overhead)	op/s (overhead)
Sequential Read (Single threaded)	38.8	34.1 (12.2 %)	33.9 (12.8 %)
Sequential Read (Multi-threaded)	13.5	12.6 (6.6 %)	12.4 (8.6 %)
Sequential Write (Single threaded)	39.5	35.3 (10.5 %)	35.3 (10.7 %)
Sequential Write (Multi-threaded)	25.3	23.6 (6.5 %)	23.6 (6.5 %)
Random read	125.3	118.9 (5.1 %)	121.2 (3.3 %)
Random write	1242.6	1143.7 (8.0 %)	1168.9 (5.9 %)

incurred under controlled workloads as shown in Table 3. There are six types of workloads generated by varying the read vs. write, sequentiality, and the number of threads. It shows that the multi-threaded workloads generate the I/O operations by only about half of the single-threaded one since there is more chance of page cache hit when multiple workloads are interleaved within the VM. Most of the **vStore**'s overheads are from handling of the trailer size as we can see from the small difference between *AIO-512* and the **vStore** column. Among the workloads, the largest difference between them is from the multi-threaded sequential reads. In the case of random workloads, the op/s of normal AIO is high compared to other workload types since cache miss becomes more likely. But, **vStore** is still able to absorb some of the block requests to the cache and shows reduced overheads than the *AIO-512* case.

D. NETWORK BANDWIDTH REDUCTION

Next, we demonstrate the efficacy of **vStore** on reducing the network traffic. Remote storage service is provided by the **NBD** server residing on a different subnet. Our evaluation point is to observe how many packets as well as bytes are reduced by **vStore** when compared to the AIO. The results are obtained by running the **vStore** with empty blocks in the cache to make it unfavorable. In the actual runs, we expect that the network savings would be generally higher than these results since the cache will be warm most of the time except for the rare restart time of the system. Also, the creation of data files needed for running the workloads is done while **vStore** is turned off so that those files are not being cached. This also adds to the unfavorable condition for **vStore**.

In this evaluation, we compare the network packets and bytes generated from two cases: (i) Remote volume attached via AIO, and (ii) using **vStore**. The results are shown in Figure 7 and Figure 8 for all five workloads. They show that **vStore** is effective in reducing the network traffic significantly. As Table 4 presents, the overall savings can be from 28% to 43% for three **Filebench** workloads. In the case of **Varmail** and **Fileserver** workloads, the network traffic of **vStore** diminishes over time, clearly showing the caching effect of popular blocks. The reason why **Webserver** is

TABLE 4. Comparison of network traffic savings.

	Number of Packets			Number of Bytes		
	AIO	vStore	Saving	AIO	vStore	Saving
Webserver	667K	291K	56.3 %	756MB	538MB	28.9 %
Varmail	593K	204K	65.5 %	670MB	412MB	38.6 %
Fileserver	636K	206K	67.6 %	650MB	368MB	43.4 %
Postmark	1991K	138	99.9 %	1743MB	3MB	99.9 %
Gcc	35K	12K	65.4 %	35MB	22MB	37.4 %

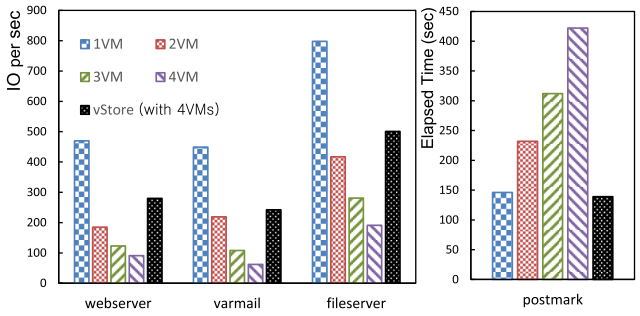


FIGURE 9. Performance degradation due to network saturation as we increase the number of VMs, and the mitigation effect of vStore applied to the 4 VM setting. For Webserver, Varmail, Fileserver workloads, higher value implies better performance. For Postmark, smaller value implies better performance since the graph is in the elapsed time.

not showing the diminishing network traffic by **vStore** is that it is requesting new data blocks from the remote disk being a read-dominant workload. For the **Postmark** and **Gcc** workloads that are shown to have write-heavy I/O properties (Table 1), they benefit from **vStore** because a large portion of writes can be served by the cache. Even among the three **Filebench** workloads, the one with a higher write proportion exhibits more savings. It is hypothesized that **vStore** is more effective in absorbing the network traffic for the write-dominant workloads.

In the case of the **Postmark** with the extremely write-dominant workload, the network traffic saving is as high as 99.9% supporting the claim that writes are absorbed by the **vStore** cache. **Postmark** workload consists of mostly the write I/Os by repeatedly creating, appending, reading and deleting a large number of files consecutively. Although there are read activities, read I/O requests are covered by the **vStore** since they are immediately after the write and **vStore** already has the block in the cache. The overall effect is huge on **vStore**. As can be seen in Figure 8 (a.1) and (a.2), almost all the network traffic is masked by the **vStore** cache.

Network saving behavior for the **Gcc** is shown in Figure 8 (b.1) and (b.2). Interestingly, the cumulative number of packets for the **vStore** is more than the **AIO** at the early stage. This is because **vStore** brings in a larger amount of data from the remote storage at the *block band* granularity. However, the tide is quickly turned as the cached block starts to take effect. At the end, **vStore** ends up saving about 65% of network traffic.

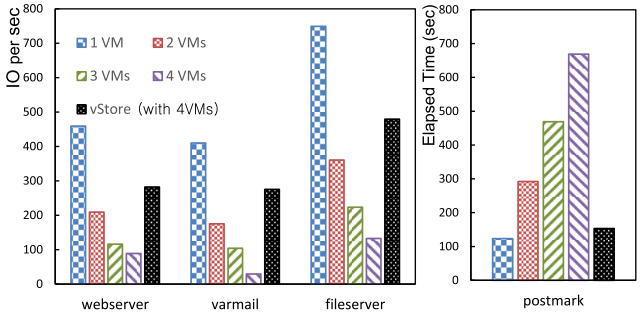


FIGURE 10. Performance degradation caused by the remote storage server being the bottleneck as we increase the number of VMs, and the mitigation effect of vStore applied to the 4 VM setting.

E. MULTIPLE VM RUNS

This section reports on the benefit of **vStore** on improving the performance of multiple VMs when sharing the remote storage server. These two scenarios are evaluated.

- Network saturation: Multiple VMs have attached virtual disks from the same remote storage server and are generating high network traffic causing the network bandwidth saturation. Our goal in this evaluation is to make the network bandwidth a bottleneck. To achieve this, we have placed four virtual disks on physically different disks in the storage server.
- Storage I/O saturation: This scenario simulates the case in which the disk I/O becomes saturated due to a large number of I/O requests on the virtual disk. In order to make the disk bandwidth a bottleneck instead of the network, we have placed four virtual disks on the same physical disk in the storage server.

Figure 9 and Figure 10 shows the result of both cases. In both cases, the IO-per-second drops significantly as we increase the number of VMs sharing the remote storage server. As the number of VMs reach 4, the performance degrades to almost 20% of the single VM case. However, the use of **vStore** on 4 VMs helps reduce the performance degradation significantly. Although **vStore** does not prevent the performance loss altogether, it is able to keep to performance at or higher than the performance level of 2 VM case. Especially in case of the postmark workload, vStore is able to offer the performance level comparable to the 1 VM case. From these evaluations, we can see that **vStore** can be used to prevent significant performance slow-down due to network and storage I/O saturation on the remote storage side.

VI. RELATED WORK

The approach of utilizing local disks as a block-level cache for remote storage was first introduced by Disk Caching Disks (DCDs) [23], [24]. Their design is to employ the *cache-disk*, a small log disk as a disk cache for the main *data-disk*. The *cache-disk* is a physically identical disk as the *data-disk*, but they utilize the speed difference by using the *cache-disk* as an append-only log style (similar to the log-structured file system [25]) along with a small NVRAM buffer for collecting small writes. One weakness of DCD is that it optimizes only

for the writes, not reads. In **vStore**, we do not consider the log-structured architecture since we want to support reasonable performances for both reads and writes.

Parallax [26] is a storage system for the virtualized environment focusing on the fine-grained snapshot capability and the virtual disk provisioning for the VMs. According to the authors, local disk caching is not fully implemented and evaluated. But, they mention the advantages of using local cache to reduce the network traffic. Their caching technique primarily aims to handle the write I/O burst by appending the writes. Similar to DCD, it differs from our goal in that we want to address both the read and write performance, instead of optimizing for the writes.

There had been several work in the direction of building an efficient storage server using distributed storage such as FAB [27], petal [28] and Data ONTAP GX [29]. Their main goal is to build a unified view at the block-level. Although they also operate at the block-level, **vStore** focuses on building the client-side caching solution whereas they are solutions on the server-side. **vStore** can run using these block-level storage systems seamlessly regardless of whether they are a collection of distributed storage servers unified as a single view.

Lithium [30] is a distributed block storage system that uses local storage similar to ours. But, their goal differs from ours in that Lithium tries to replace or reduce the dependency on the centralized shared storage by replicating data to multiple participating nodes' local storage. Data is stored in the local storage as the log-structured manner. Also, it includes a user-level component that interacts with the Lithium kernel module.

Netco [31] provides cache services near compute nodes from disaggregated storage devices in cloud systems. It prefetches data into the cache (based on workload predictability), and appropriately divides the cache space and network bandwidth between the prefetches and serving ongoing jobs. Different from Netco, which is an independent cache service, vStore is designed as an internal component of a hypervisor and provides a transparent block storage cache. Therefore, vStore provides lower latency and system set-up overhead. However, the prefetch mechanism and network bandwidth control of Netco can be applied to vStore to earn more benefits and predictable storage performance.

Several management policies for remote block storage in cloud systems have been proposed [32]–[34]. OSCA [32] is a cache allocation scheme for shared cache servers among cloud block storage devices, and can search for a near-optimal configuration scheme at a very low complexity based on a novel online cache model leveraging re-access ratio. The lazy eviction cache algorithm (LEA) [33] aims at efficient management of SSD-based cloud block storage caches. It remedies the cache inefficiencies caused by cache blocks with large reuse distances. The machine-learning-based write-policy (ML-WP) [34] avoids writing write-only data to the caches to reduce the cache traffic and inefficient cache space usage. The authors used machine learning techniques to filter out write-only data from write-then-read data. These intelligent cache management policies can be applied to our approach to further improve cache efficiency.

Tang have proposed FVD [35], a new virtual disk format for VM images that supports Copy-on-Read, and adaptive prefetching to achieve fast boot time of VM images. Although FVD also handles block-level data in the image format, our goal is to design a hypervisor functionality that can further utilize various live information to adapt the behaviors. We also analyze and quantify the effect of hypervisor-side caching using various workloads.

VII. CONCLUSION

In modern virtualized cloud infrastructure, remote storage system plays a critical role in delivering the scalability. However, an increasing number of VMs and surging access to the network storage from them can become a serious bottleneck by saturating limited network bandwidth in the cloud data center. We have designed and implemented the **vStore** system that uses local disks of virtualization hosts as a block-level cache to mitigate the problem. Our evaluation shows that the use of a local disk cache can reduce the network traffic due to the remote storage I/Os to about 49% with less than 12% performance overhead.

From the experience and lessons gained in this work, we can enhance the resiliency in using the local disk cache and do that with better performance. Performance can be improved by transferring blocks between hosts for rapid VM migration. We can also consider replicating blocks across multiple hosts so that failed hosts can quickly recover the state at the moment of failure and restart.

REFERENCES

[1] V. Dukic, G. Khanna, C. Gkantsidis, T. Karagiannis, F. Parmigiani, A. Singla, M. Filer, J. L. Cox, A. Ptasznik, N. Harland, W. Saunders, and C. Belady, "Beyond the mega-data center: Networking multi-data center regions," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, New York, NY, USA, 2020, pp. 765–781, 10.1145/3387514.3406220.

[2] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, New York, NY, USA, 2015, pp. 123–137, doi: 10.1145/2785956.2787472.

[3] G. Brataas, E. Stav, S. Lehrig, S. Becker, G. Kopčak, and D. Huljenic, "CloudScale: Scalability management for cloud systems," in *Proc. ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA, 2013, pp. 335–338, doi: 10.1145/2479871.2479920.

[4] C. K. Filelis-Papadopoulos, G. A. Gravvanis, and P. E. Kyziropoulos, "A framework for simulating large scale cloud infrastructures," *Future Gener. Comput. Syst.*, vol. 79, pp. 703–714, Feb. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17303230

[5] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Commun. Surveys Tuts.*, vol. 13, no. 3, pp. 311–336, 3rd Quart., 2011.

[6] P. Garraghan, I. S. Moreno, P. Townend, and J. Xu, "An analysis of failure-related energy waste in a large-scale cloud environment," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 2, pp. 166–180, Jun. 2014.

[7] N. Farrington and A. Andreyev, "Facebook's data center network architecture," in *Proc. Opt. Interconnects Conf.*, May 2013, pp. 49–50.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun. (SIGCOMM)*, New York, NY, USA, 2008, pp. 63–74, doi: 10.1145/1402958.1402967.

[9] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized SSD cache for primary storage," in *Proc. USENIX Annu. Tech. Conf.* Berkeley, CA, USA: USENIX Association, 2014, pp. 501–512.
[10] S. He, X.-H. Sun, and B. Feng, "S4D-cache: Smart selective SSD cache for parallel I/O systems," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, Jun. 2014, pp. 514–523.
[11] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proc. 10th USENIX Conf. File Storage Technol.* Berkeley, CA, USA: USENIX Association, 2012, p. 25.
[12] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, vol. 6, M. D. Hill, Ed. San Rafael, CA, USA: Morgan & Claypool, doi: 10.2200/S00193ED1V01Y200905CAC006.
[13] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: A scalable and fault-tolerant network structure for data centers," in *Proc. ACM SIGCOMM Conf. Data Commun. (SIGCOMM)*, New York, NY, USA, 2008, pp. 75–86, doi: 10.1145/1402958.1402968.
[14] E. Kim, "An introduction to solid state drive performance, evaluation and test," Storage Netw. Ind. Assoc., San Francisco, CA, USA, White Paper, 2013.
[15] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006.
[16] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional allocation of resources for distributed storage access," in *Proc. 7th Conf. File Storage Technol. (FAST)*. Berkeley, CA, USA: USENIX Association, 2009, pp. 85–98.
[17] *Filebench*. Accessed: Jun. 17, 2021. [Online]. Available: https://github.com/filebench/filebench/wiki
[18] *Xen Blktap Overview*. Accessed: Jun. 17, 2021. [Online]. Available: https://wiki.xenproject.org/wiki/biktap
[19] *Network Block Device*. Accessed: Jun. 17, 2021. [Online]. Available: http://nbd.sourceforge.net/
[20] *Postmark*. Accessed: Jun. 17, 2021. [Online]. Available: https://www.vi4io.org/tools/benchmarks/postmark
[21] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, "Understanding manycore scalability of file systems," in *Proc. USENIX Annu. Tech. Conf.* Berkeley, CA, USA: USENIX Association, 2016, pp. 71–85.
[22] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O support in Linux 2.5," in *Proc. Ottawa Linux Symp.*, 2003, pp. 351–530.
[23] Y. Hu and Q. Yang, "DCD–disk caching disk: A new approach for boosting I/O performance," in *Proc. 23rd Annu. Int. Symp. Comput. Architecture (ISCA)*, New York, NY, USA, 1996, pp. 169–178.
[24] T. Nightingale, Y. Hu, and Q. Yang, "The design and implementation of a dcd device driver for unix," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATEC)*. Berkeley, CA, USA: USENIX Association, 1999, p. 22.
[25] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[26] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Parallax: Virtual disks for virtual machines," in *Proc. 3rd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst. (Eurosys)*, New York, NY, USA, 2008, pp. 41–54.
[27] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence, "FAB: Building distributed enterprise disk arrays from commodity components," in *Proc. 11th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS-XI)*, New York, NY, USA, 2004, pp. 48–58.
[28] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proc. 7th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS-VII)*, New York, NY, USA, 1996, pp. 84–92.
[29] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and C. Wagner, "Data ONTAP GX: A scalable storage cluster," in *Proc. 5th USENIX Conf. File Storage Technol. (FAST)*. Berkeley, CA, USA: USENIX Association, 2007, p. 23.
[30] J. G. Hansen and E. Jul, "Lithium: Virtual machine storage for the cloud," in *Proc. 1st ACM Symp. Cloud Comput.*, New York, NY, USA, 2010, pp. 15–26, doi: 10.1145/1807128.1807134.
[31] V. Jalaparti, C. Douglas, M. Ghosh, A. Agrawal, A. Floratou, S. Kandula, I. Menache, J. S. Naor, and S. Rao, "Netco: Cache and I/O management for analytics over disaggregated stores," in *Proc. ACM Symp. Cloud Comput.*, ser. SoCC '18. New York, NY, USA, 2018, pp. 186–198.
[32] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, and B. Cheng, "OSCA: An online-model based cache allocation scheme in cloud block storage systems," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2020, pp. 785–798. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/zhang-yu
[33] K. Zhou, Y. Zhang, P. Huang, H. Wang, Y. Ji, B. Cheng, and Y. Liu, "Efficient SSD cache for cloud block storage via leveraging block reuse distances," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2496–2509, Nov. 2020.
[34] Y. Zhang, K. Zhou, P. Huang, H. Wang, J. Hu, Y. Wang, Y. Ji, and B. Cheng, "A machine learning based write policy for SSD cache in cloud block storage," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1279–1282.
[35] C. Tang, "FVD: A high-performance virtual machine image format for cloud," in *Proc. USENIX Annu. Tech. Conf.* Berkeley, CA, USA: USENIX Association, 2011, p. 18.

BYUNGCHUL TAK (Member, IEEE) received the B.S. degree from Yonsei University, South Korea, in 2000, the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2003, and the Ph.D. degree in computer science from Pennsylvania State University, in 2012. He is currently an Assistant Professor with Kyungpook National University, Daegu, South Korea. Prior to joining KNU, he was a Research Staff Member of IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. His research interests include distributed systems, virtualization, operating systems, cloud computing, and fog/edge computing.

CHUNQIANG TANG (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Rochester. He joined IBM Research and works on large-scale distributed systems. His research interests include distributed systems, cloud computing, services computing, large-scale platform, mass storage, computer networks, operating systems, and information retrieval.

RONG N. CHANG (Senior Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1990. He is with IBM Thomas J. Watson Research Center and is leading an in-market research and development effort on composable enterprise microservices fabric in support of intelligent computing. He holds more than 30 patents and has published more than 50 articles. He is a member of IBM Academy of Technology, an ACM Distinguished Engineer, the Chair of IEEE Computer Society Technical Committee on Services Computing, and an Associate Editor-in-Chief of the IEEE TRANSACTIONS ON SERVICES COMPUTING.

EUISEONG SEO (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from KAIST, in 2000, 2002, and 2007, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Sungkyunkwan University, South Korea. Before joining Sungkyunkwan University, in 2012, he had been an Assistant Professor with UNIST, South Korea, from 2009 to 2012, and a Research Associate with Pennsylvania State University, from 2007 to 2009. His research interests include system software, embedded systems, and cloud computing.