BASS: Improving I/O Performance for Cloud Block Storage via Byte-Addressable Storage Stack

Hui Lu[†], Brendan Saltaformaggio[†], Cong Xu[‡] [*], Umesh Bellur[+] [†], Dongyan Xu[†]

[†]Purdue University, [‡]IBM Research Austin, [+]IIT Bombay

[†]{lu220,bsaltafo,dxu}@cs.purdue.edu, [‡]xucong@us.ibm.com, [+]umesh@cse.iitb.ac.in

Abstract

In an Infrastructure-as-a-Service cloud, cloud block storage offers conventional, block-level storage resources via a storage area network. However, compared to local storage, this multilayered cloud storage model imposes considerable I/O overheads due to much longer I/O path in the virtualized cloud. In this paper, we propose a novel byte-addressable storage stack, BASS, to bridge the *addressability gap between the storage and network stacks* in cloud, and in return boost I/O performance for cloud block storage. Equipped with byte-addressability, BASS not only avails the benefits of using variable-length I/O requests that avoid unnecessary data transfer, but also enables a highly efficient non-blocking approach that eliminates the blocking of write processes. We have developed a generic prototype of BASS based on Linux storage stack, which is applicable to traditional VMs, lightweight containers and physical machines. Our extensive evaluation with micro-benchmarks, I/O traces and real-world applications demonstrates the effectiveness of BASS, with significantly improved I/O performance and reduced storage network usage.

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

General Terms Design, Measurement, Performance

Keywords Virtualization, Cloud Computing, Cloud Block Storage

[*] Contributed to the work while at Purdue University.

[†] Was a visiting faculty at Purdue University during 2015-2016.

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987557

1. Introduction

Cloud block storage (hereinafter referred to as block storage), such as Amazon EBS, Google Persistent Disk and Azure Premium Storage, offers reliable, high-performance and on-demand block-level storage volumes for applications hosted on virtual machines (VMs) or containers in today's cloud data centers. Block storage provides a powerful separation of administrative domains – VMs/containers are free to organize data on volumes with individual file systems just like using local drives, while the underlying hypervisor chooses suitable storage protocols to actually store the data in remote physical media.

However, compared to local storage, this multilayered model brings considerable I/O overheads and latency due to much longer I/O path in the virtualized cloud environments [10, 14]. To mitigate or hide such I/O overheads, block storage systems rely heavily on the VM/container-side page cache layer [17, 21, 27] to buffer volume data in memory pages: I/O accesses to cached pages can be quickly returned, while accesses to non-cached pages trigger a slow path – to fetch data from the backing store. To avoid data loss, dirty pages are periodically flushed to the backing store.

Unfortunately, we observe that two critical complications arise from cloud block storage which make the traditional caching mechanism less effective. First, *partial writes* (i.e., unaligned with the cache page size) to *non-cached* pages cause VMs/containers to suffer extremely long I/O blocking time. This is because, the *block-level* read-modify-write (RMW) restriction [20] requires a slow page read before a partial write. Further, unlike local storage, this long I/O latency cannot be offset by state-of-the-art fast storage technologies (e.g., PCM- or STTM-based devices [9, 13]), as most I/O time in block storage is spent on data transmission within the multiple software layers (as we will see in Section 2). Second, *cached* pages, despite being only *partially* dirtied, must be fully flushed to the remote backing store via the network. This *block-level* flush granularity works well with local storage, as one data page – regardless of how partially dirtied – will trigger only one I/O access with similar cost. However, it becomes sub-optimal for block storage, because one partially dirtied page will require *far more network bandwidth than the dirtied data requires*, and thus longer data transfer and I/O completion time.

Recent studies have confirmed that there exists a significant fraction of partial writes (>65%) in a variety of production file system workloads [12, 18, 25, 33]. Our investigation in Section 2 further demonstrates that the performance of these workloads in block storage will be negatively impacted due to the above-mentioned complications. For instance, the performance of partial writes to non-cached pages can decrease by *4 times* in block storage than that in local storage. The total data transferred via the network for a 512-byte write request can be *18 times* the size of the actually accessed data.

To overcome these challenges, we present **BASS**, a **B**yte-**A**ddressable **S**torage **S**tack with new designs and approaches to accelerate I/O operations for cloud block storage. BASS builds upon the often overlooked fact that: In block storage the lower layer of the VM/container-side storage stack is actually the *byte-addressable* network stack, while the *block-addressable* storage devices (e.g., HDD and SSD) are hosted on a remote storage server.

This inherent layered design makes two ideas feasible in BASS: First, unlike storage devices, the byte-addressable network stack is able to take *variable-length* I/O requests from the VM/container file system, which eliminates the restrictions caused by block-granularity addressing, such as in RMW. Instead, the *originally synchronous* blocking I/O can be decomposed into a *fast-and-small* synchronous portion (putting only the dirtied bytes into the VM/container-side cache) and a *slow-and-large* asynchronous portion (fetching or flushing data from/to the backing store). This division (detailed in Section 3.3) allows BASS to hide any latency of the large portion from VMs/containers, which only wait for the completion of the local caching. Second, leveraging the byte-addressable network stack, the VM/container-side file systems are free to flush *only the dirtied portions* of data pages to the network layer, avoiding unnecessary data transmission and thus accelerating data write-back speed and saving network bandwidth. Consequently, the block-level operations (e.g., data page alignment before a write) are taken care of by the remote storage server, which interacts with the block devices directly.

BASS realizes the above ideas by re-designing the block-based storage stack, common to all block-based file systems, to seamlessly expose the byte-addressability of the network layer. Specifically, BASS enhances the storage I/O stacks on both the hypervisor/container host and storage server, to make the key data structures and functions aware of variable-length I/O data rather than fixed block-based data. This enables the lower-level driver layer to transfer only the needed I/O data (of arbitrary length) between the VM/container and storage server over the network. Meanwhile, BASS still maintains block-level attributes for variable-length I/O requests to allow existing block-layer optimizations (e.g., I/O merging, scheduling and DMA) to interoperate seamlessly with BASS. Further, with the help of byte-addressability, BASS invents a new *non-blocking* approach for non-cached writes by decoupling original blocking (synchronous) operations into fast caching and fetch/flush portions (as mentioned above).

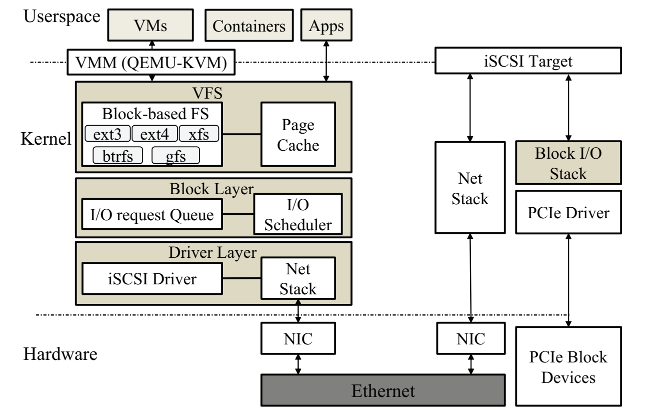


Figure 1. Cloud block storage architecture using iSCSI.

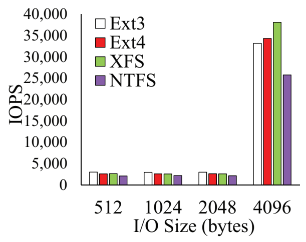
To the best of our knowledge, BASS is among the first to explicitly consider the *addressability gap* between the storage stack and the network stack. We have implemented BASS within the Linux storage stack, which works seamlessly with existing block-based file systems and is applicable to traditional VMs, lightweight containers and physical machines. Our evaluation with a wide range of real-world workloads shows that BASS both shortens applications' I/O latency (e.g., up to 50% for Facebook and Twitter I/O traces), and saves network bandwidth (e.g., up to 60% for YCSB workloads).

2. Motivation

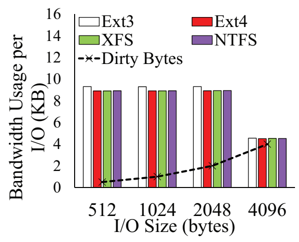
2.1 Block Storage Overview

Cloud block storage is typically built upon SAN architectures. Existing SAN systems use either software-based (e.g., iSCSI), hardware-based (e.g., Fiber Channel) or hybrid (e.g., FCoE) protocol solutions. They have similar stack-based organizations to orchestrate multiple software and hardware layers and provide the block device interfaces. In this paper, we mainly focus on software-based solutions, however, BASS's design is equally applicable to other similar architectures.

Figure 1 shows a software-based implementation using the iSCSI protocol to provide block transport. The storage server (i.e., iSCSI target) turns its local block storage devices into virtual volumes. An initiator (i.e., iSCSI drivers), which serves the same purpose as a conventional SCSI bus adapter (but replying on IP network instead of physical cables), accesses these virtual volumes with TCP/IP encapsulated SCSI commands. Conventional block-based storage stacks run above the driver layer. The networked virtual block volumes can be freely configured with various block-based file systems by VMs/containers and native applications.



(a) Throughput.



(b) Bandwidth.

Figure 2. IOPS and network bandwidth usage over sized write requests on different block-based file systems.

2.2 I/O Overhead Illustration

To illustrate the I/O overhead of block storage incurred in a multilayered architecture (Figure 1), we compare the throughput (in terms of IOPS) and network bandwidth usage (consumed per I/O request) using various sized write requests issued by applications running inside VMs/containers. Specifically, on the host machine we run a Linux container [1], which attaches an SSD device (in the form of a virtual volume) from a remote storage server. On top of the SSD device, various popular block-based file systems are investigated, including Ext3/4 (the Linux file systems), XFS (a journaling file system), and NTFS (an open source implementation of the Windows NTFS).

Fio [1] is used as the workload to generate random writes to a large file (e.g., 50 GB) created in advance on the container's file system. The I/O request size ranges from 512 bytes to 4 KB. Note that, except for the I/O requests which happen to be 4 KB, all others are *unaligned* with the cache page size of the investigated file systems (i.e., 4 KB). We run one thread in Fio to generate I/O requests with an I/O depth of one – the next I/O request is issued only after the current one completes – to measure the worst case I/O latency. The page caches on both the container and storage sides are cleared before each run to simulate the non-cached case.

Figure 2(a) shows that the throughput of *unaligned* I/O requests (i.e., partial writes of 512, 1024 or 2048 bytes) is much lower (e.g., *10 times* for Ext3) than that of *well-aligned* I/O requests (i.e., 4 KB in size). This indicates that the file systems running on block storage are unfavorable to partial, non-cached writes. Though the local case (i.e., using direct attached storage) can also suffer from this issue, these results show that this condition becomes much more severe in cloud block storage (e.g., *4 times* worse). Further, unlike in the local case, the long I/O latency of block storage cannot be offset by state-of-the-art fast storage technologies [2]. We term this problem *partial-write blocking*.

[1] We investigated both VMs and containers, which yielded similar results.

[2] For example, the storage-side cache could load the data in advance (to simulate a fast storage device with DRAM-like speed), but the throughput of unaligned I/O requests in the block storage case remains 6 times lower than that of the well-aligned I/O requests, despite there being no performance gap between these two types of I/O requests in the local case.

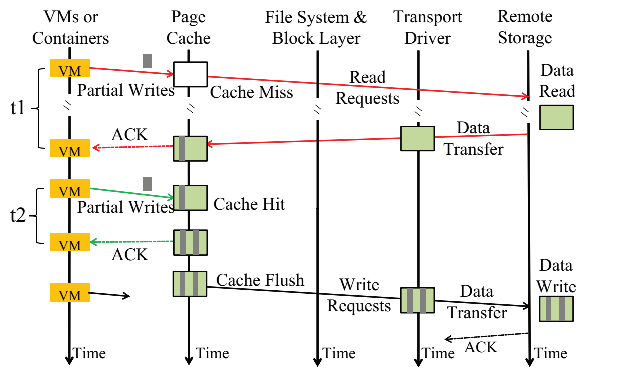


Figure 3. I/O bottleneck root cause analysis.

Using the same setup as above, Figure 2(b) depicts the average network bandwidth usage of one I/O request in each case. For instance, one 512-byte modification on Ext3 needs more than 9-KB of network traffic (packets) to complete, while a 4-KB modification only consumes ∼4.5 KB of network traffic. This indicates that block storage network usage is quite inefficient for partial I/O requests. Note that the TCP/IP and iSCSI headers only contribute ∼7% of the data in one iSCSI network packet, while the actual network bandwidth used for one 512-byte write I/O can be up to *18 times* the size of the actual data modification. We term this the *data transmission amplification problem*.

2.3 Problem Root Cause Analysis

Let us take a deeper look into the causes of the problems presented in Section 2.2. Figure 3 shows the I/O flow of control from the application layer to the backing store. We assume one application (running in the VM/container) issues I/O requests to the file systems, and the operating system caches file data in the page cache layer [3].

Once a partial write is issued to a page that is not in the cache (i.e., the partial, non-cached write case), the page cache layer will trigger an atomic RMW operation, which fetches the original data copy from the backing store at the granularity of one page. During this page fetch, the application's I/O thread is blocked to wait for the acknowledgement of the write completion. Because the I/O path of block storage consists of multiple layers, the blocking time, $t1$, could be significant. In contrast, once a partial write is issued to a page that is in the cache (or a full-page write [4]), the cache layer simply puts the dirty data into the cache page, and the application's I/O thread returns. As this process mainly involves the virtual file system layer and the cache layer, which are fast, the blocking time, $t2$, will be much smaller than $t1$ (several orders of magnitude). This explains the cause of the *partial-write blocking* problem.

Further, the page cache layer needs to periodically (e.g., 5 seconds) flush dirty pages to the backing store to avoid

[3] We assume a write-back cache – the best performance cache solution. BASS is compatible with other cache policies, like write-through.

[4] A full-page write does not require RMW.

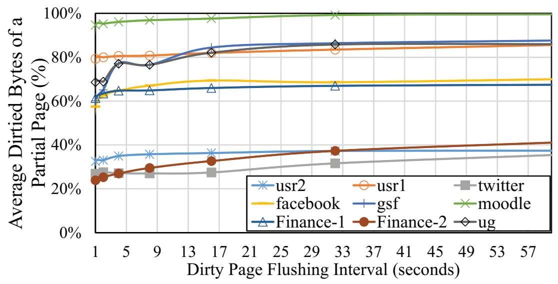


Figure 4. The average number of dirty bytes as a percentage of the 4-KB page size with varying cache flush interval.

I/O traces	Partial Writes(%)	Partial, non-cached Writes(%)	Description
usr1	52.77	5.69	Desktop1
usr2	72.21	22.02	Desktop2
moodle	55.89	40.92	Web & Database
gsf-filesrv	32.72	6.81	CIFS server1
ug-filesrv	31.38	7.49	CIFS server2
facebook	87.81	9.64	Facebook app
twitter	85.93	9.66	Twitter app
Finance-1	53.86	23.96	OLTP app1
Finance-1	65.67	7.18	OLTP app2

Table 1. Partial write percentage of the total writes.

data loss due to system crashes. Conventionally, a dirty page is written back in terms of a whole page, regardless of how many bytes on the page have been modified. For file systems running over local storage, this block-based treatment is reasonable, as one page access will only trigger one I/O request – the page data is usually placed in an allocation unit on disks consisting of contiguous groups of physical sectors. However, for file systems over networked storage, this mechanism causes the network layer to transfer both modified and unmodified content of a dirty page, resulting in the *data transmission amplification* problem. This problem becomes particularly severe for partial, non-cached writes – for one such write, the network layer needs to transfer a read page and a write page due to RMW.

Essentially, the above mechanisms are based on the "block" attribute of the storage stack – most modern file systems are built on blocks, and data is organized and accessed at block granularity (e.g., 4 KB). However, in cloud block storage, the lower layer of the VM/container-side file systems becomes the byte-addressable network stack, while the block-based storage devices move to the remote storage server. This new layer orchestration breaks down the "block" restrictions, and provides opportunities to address the two problems above (detailed in Section 3).

2.4 Prevalence of I/O Bottlenecks

Section 2.2 indicates that the performance of applications with *partial and/or non-cached* writes could be negatively impacted. To quantitatively investigate this problem in a realistic environment, we analyzed several I/O traces from the user desktop, web and database servers, CIFS file server [5], Facebook and twitter applications [3], and OLTP applications [7] as listed in Table 1. We analyzed the last 2-hours of write operations from each trace, except for Facebook and Twitter's traces which only contain 120-second's worth of I/O operations.

First, we analyzed the percentage of the total pages that are partially dirtied (i.e., partial writes). Table 1 shows that theses traces have a significant number of partial writes, ranging from 30% to 90% of the total writes. Moreover, 5% to 40% of the *total writes* [5] will encounter cache misses (i.e., the partial, non-cached writes) and suffer from the *partial-write blocking* issue. Further, our evaluation in Section 5.2 shows that, for example, the ∼10% partial, non-cached writes of Facebook's trace lead to ∼50% of the overall I/O latency.

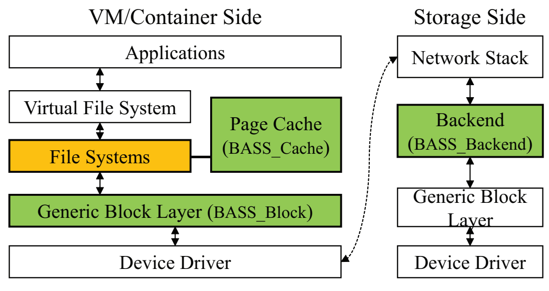
Next, to investigate the *data transmission amplification* problem, we analyzed the dirty-byte percentage of a 4-KB page size – the actual bytes that need to be transferred between VMs/containers and storage servers. Note that, the degree to which a page has been dirtied depends on the *cache flush interval* – the longer the interval, the more the dirty bytes aggregate during the interval. To account for such influence, we use a simple cache simulator: We assume the cache size is unlimited (i.e., no pages will be evicted); the flush interval is configured from 0 (i.e., dirty pages are flushed immediately once dirtied) to more than 60 seconds. Figure 4 shows the average number of dirty bytes as a percentage of the 4-KB page size. These results show that most page dirtying happens early – the number of dirty bytes does not change much as the cache flush interval increases. Further, when we keep the flush interval sufficiently long (e.g., 60 seconds), the dirty-byte percentages of three traces are still very low – less than 40%. Five others vary between 60% and 80%. Only one I/O trace's dirty-byte percentage is ∼100%.

To summarize, on average ∼60% of the total writes involve partial writes. ∼20% of the total writes are partial, non-cached writes. Besides, a significant number of bytes (>40%) in a partial write page are *not* dirtied. In light of this, such workloads on block storage will be negatively impacted due to partial-write blocking and data transfer amplification – which BASS aims to overcome.

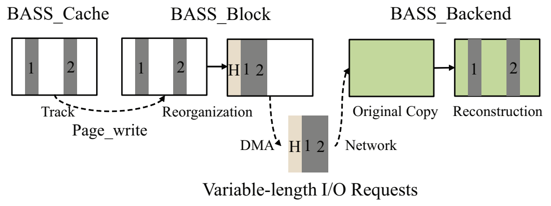
3. Design

To overcome the problems illustrated in Section 2, BASS adapts the existing block-based cloud storage stack to make it byte-addressable. First, BASS enables the generation of variable-length, instead of blockwise, I/O requests within the existing storage stack (Section 3.1). Meanwhile, BASS carefully maintains the block-level attributes to allow block-layer optimizations to work for the variable-length I/O

[5] We consider writes which access pages that have not been accessed for more than one hour to be non-cached writes. For the Facebook and Twitter traces, we only assume that no data is cached *before* the traces begins.



(a) Architecture.



(b) Workflow.

Figure 5. BASS architecture overview and workflow.

requests as well (in Section 3.2). Last, with the help of byte-addressability, BASS involves a highly efficient non-blocking approach for partial, non-cached writes by decoupling the synchronous blocking operation into two (performance optimized) halves (in Section 3.3).

3.1 Byte-addressable Storage Stack

While many different block-based file systems are designed with various ways of organizing stored files, they all share a generic block-based storage interface (e.g., the Linux storage stack), where the fundamental (low-level) data read and write operations are provided. To enable byte-addressability for this storage stack (and thus for file systems implemented upon it), BASS incorporates new components into the entire storage I/O stacks of both the VM/container and storage server: including a BASS_Cache in the cache layer, BASS_Block in the generic block layer, and BASS_Backend in the storage backend. Figure 5 shows these components alongside the original storage stack and their workflow.

BASS_Cache While applications can issue arbitrary length I/O requests to file systems, this "length" information is ignored by the cache layer, and resolves into a page-level operation – to fetch a *full* non-cached page or flush an *entire* dirty page. However, to realize byte-addressability, such length information must be recorded.

BASS_Cache is designed to keep track of application-level accesses at a byte granularity in the page cache layer. These byte-grained accesses will be used for the generation of variable-length I/O requests to achieve byte-addressability. Although it is possible for BASS to support variable-length read requests, a full-page fetch for a partial read will benefit subsequent accesses to the same page (analogous to the "readahead" feature commonly used by the Linux kernel). To keep this optimization, BASS performs reads at the block level – to read a full page regardless of the I/O request length – with one exception (detailed in Section 3.3). On the contrary, it is imperative that partial writes be byte-addressable, since a full-page flush of a partially dirtied page will result in *data transmission amplification* (as stated in Section 2.2). To this end, BASS_Cache maintains a linked list for each data page to store the non-overlapped dirty segments. This linked list will be updated by every write operation and used for constructing *variable-length write I/O requests* when the page is flushed.

BASS_Block Periodically or based on trigger events, file systems transform dirty pages into block I/O requests and then deliver such I/O requests to the generic block layer – an abstraction for block devices. Because the block layer processes I/O requests at the granularity of the physical sector size (e.g., 512 bytes or 4 KB), conventionally, I/O requests must be generated with the length being an integer multiple of the physical sector size. However, this limits the realization of variable-length I/O requests, desired by byte-addressability. To overcome this, BASS_Block provides a new I/O generation function, `submit_io`.

A straightforward way for `submit_io` to create variable-length I/O requests from a dirty page may be to transform each dirty segment into a single I/O request. However, this method tends to produce many I/O requests, because multiple dirty segments may exist in a dirty page. Excessive I/O requests would increase the overhead of the storage system. Instead, `submit_io` chooses to pack multiple dirty segments and generates one I/O request for each dirty page. Specifically, `submit_io` reorganizes the data layout of a dirty page by placing all dirty segments into one contiguous segment. An I/O header (i.e., metadata) is then created to store the pairs of position and length of dirty segments. The I/O header and the contiguous merged dirty segments construct the *effective data chunk*, used as the I/O request content. Note that, a temporary page is used to store the effective data chunk, as the original page will be accessed by subsequent requests. The temporary page will be shortly released after the I/O request completes.

Yet, an issue arises during I/O request generation: the size of the effective data chunk may be larger than the page size, such as the *full-page* (well-aligned) writes – each has one dirty segment with the length being the page size [6]. Simply using the above generation method will lead to the transfer of more than *one-page-size* of data, which is inefficient. In order *not* to bring additional overhead to these full-page writes, `submit_io` selectively falls back on the original I/O request generation scheme (i.e., without data layout reorganization). Consequently, two types of I/O requests will coexist in the storage system – ones with I/O headers (i.e., partial I/O requests), and ones without (i.e., full-page I/O requests). To distinguish between these two types of I/O requests, an

[6] BASS treats any highly-dirtied page writes, whose effective data chunk size is larger than one page size, as full-page writes.

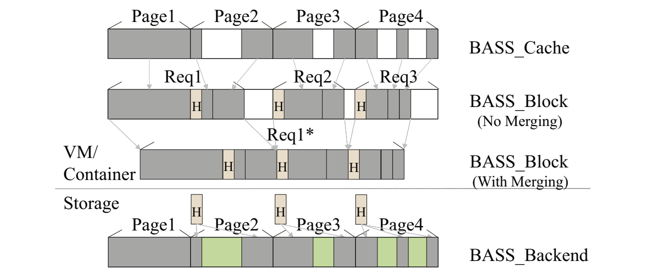


Figure 6. BASS supports I/O merging for partial writes.

identifier field (with a distinctive unique value) is added to the header of partial I/O requests[7].

Last, new data structures are required for variable-length I/O requests (similar to the Linux bio structure) to keep track of the position/length information for the effective data chunk as well as the data pointer to the temporary page. Further, the functions in the block layer (that were only aware of sector-level information) should be enhanced to account for variable-length I/O requests. By doing so, the effective data chunk will be correctly encapsulated into I/O data packets by the low-level device drivers and direct memory mapping (DMA).

BASS_Backend The storage backend decomposes received I/O requests and performs the actual storage operations. As BASS_Block changes the format of a partial-page write into packed dirty segments, the storage backend also needs new processing logic to unpack the I/O operations.

BASS_Backend first checks whether the *identifier field* exists or not in the header of the received I/O data (we will discuss merging requests in Section 3.2). If the identifier does not exist, a full-page write is indicated and BASS_Backend processes it along the original I/O path – data is written to either the storage-side cache or the storage devices. Otherwise, BASS_Backed switches to the partial-page write path. The I/O header is used to locate the position and length of dirty segments, and BASS_Backend locates the original data page that the dirty segments belong to. If this page is cached, the page will be updated to include the dirty segments directly; otherwise, the data page needs to be fetched from the backing store and then updated.

This allows BASS to move the block-level processing logic from the VMs/containers to storage servers, while remaining fully compatible with the layered setup of cloud-based block storage. Further, this design not only avails the benefits of using variable-length I/O requests (avoiding unnecessary data transfer), but also enables our non-blocking optimization detailed in Section 3.3.

3.2 BASS Support for I/O Merging

Residing in the generic block layer, I/O merging functions and schedulers play an important role in maximizing block

[7] To avoid collisions with our unique identifier, the BASS_Backend also performs sanity checks on the header's dirty segment layout data structure and payload size (i.e., variable-length requests are not page aligned). We ensure the probability of a collision is greatly less than an undetectable data corruption event.

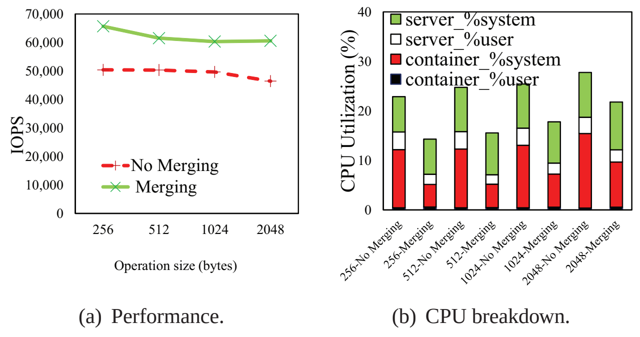


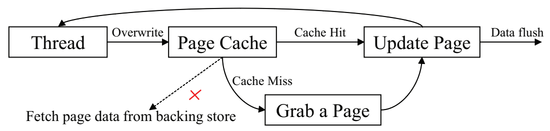
Figure 7. Performance comparison with and without I/O merging support.

device performance. However, the involvement of BASS may change the default behaviors of these functions, which BASS must proactively account for.

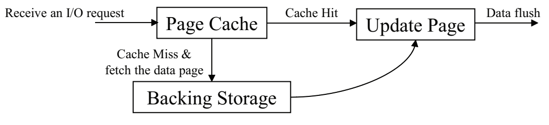
Let us consider the example in Figure 6. Four pages, Page1 to Page4, contain data of contiguous groups of physical sectors. By default, these four pages will first convert to four block-based write I/O requests (if dirtied). Later, they would merge into one single I/O request at one of several merging points along the I/O path. These merging functions decide to merge I/O requests based on their physical locations (i.e., sector location) and the length of the I/O requests (i.e., total sector number). However, with BASS some originally *mergeable* I/O requests become *unmergeable*. For instance, the write I/O requests of Page2 and Page3 in Figure 6 will not merge anymore, because the merging functions consider these two I/O requests to be *physically inconsecutive*, according to their location and (unaligned) length information. Further, the four pages will in total produce three I/O requests, incurring high overheads.

To allow variable-length I/O requests to benefit from these optimizations, BASS carefully maintains the block attributes for these requests as well. First, BASS keeps the sector related information (i.e., sector position and total sector number), when generating I/O requests. This information will be exposed to the merging functions and I/O schedulers for merging and sorting purposes. Additionally, the actual I/O length information is kept and will be exposed to the data encapsulation related functions, such as the scatter/gather processes, which perform DMA operations on pages that are scattered throughout physical memory. Based on this, the I/O requests can be merged and sorted in the same way as before; meanwhile, only the *effective data chunk* will be put into the I/O request data packets.

On the storage side, BASS_Backend must be aware that one "large" I/O request may contain several small (merged) I/O requests – probably a mixture of full-page and partial I/O requests. To distinguish each separate request (per page), BASS_Backend first screens out the partial I/O requests by locating their *identifier fields*, and separates them from the full-page I/O requests. After this step, BASS_Backend pro-



(a) The VM/container-side function.



(b) The storage-side function.

Figure 8. The non-blocking write approach.

cesses these requests separately by applying the appropriate logic described in Section 3.1.

To demonstrate the benefits of I/O merging, we compare the performance of BASS with and without the merging support using a representative example: We run one Fio thread in a container to generate the sequential write I/O requests but with "holes". For example, when the I/O request size is 1 KB, we skip 3 KB after each write request. With this configuration, the Fio thread accesses contiguous blocks in the backing store. But for each 4 KB block, the thread only modifies the leading portion of the block data with a certain size (256 bytes to 2 KB). Without the merging approach, BASS simply generates one I/O request for each 4 KB block write, while with the merging approach, BASS enables the I/O schedulers to merge as many requests as possible to create a large I/O request. To ensure no performance bottlenecks are due to disks, we cache data in the cache of the storage server, while data is not cached on the container side.

Figure 7(a) shows that, compared to the non-merging case, BASS with merging improves the performance of sequential writes by up to 30% (at 256 bytes). Further, BASS with merging reduces the overall CPU utilization (including both the container and storage server) by up to 40% (at 256 bytes) as shown in Figure 7(b). Most of the reduction is contributed by the decreased CPU utilization of the container-side system (kernel), indicating merging makes the container-side storage system more efficient. In the remainder of this paper, we will only consider BASS with merging support.

3.3 Non-blocking Approach

Prior to BASS, all partial, non-cached writes followed the RMW convention, which blocks the write thread and thus impacts application-level performance greatly. With byte-addressability however, BASS is able to break the originally synchronous, blocking write process into two optimized halves: the fast first half on the VM/container side and the slower second half on the storage side.

Non-blocking Writes Figure 8 shows the workflow of the non-blocking write approach. Once a partial, non-cached write encounters a cache miss, instead of fetching the page from the remote backing store, BASS allows the write to occur directly by grabbing a free page from the page cache layer, storing the write in this page, and then allowing the writing thread to return. This makes the first half of the non-blocking write extremely fast, because all operations occur in main memory. Later, during the period of dirty page flushing, the written portion of a dirty page is flushed to the storage side via variable-length I/O requests. The actual RMW operation is then performed by the storage server, which aligns the partial writes to the underlying physical blocks. Specifically, the original page is fetched from the backing store (or memory if buffered) and updated to include the dirty portion. Notice that, this effectively hides the RMW processing latency from the applications, which only wait for the in-memory caching to complete.

The reasons that the two halves can be separated are that: (1) Like well-aligned writes, the dirty pages of partial writes do not need to flush to the backing store immediately. (2) With byte-addressability, it is possible to only deliver the dirty portion of a page to the storage side (i.e., without the need to fetch the original data page).

Handling Read "Holes" Because of the non-blocking write approach, the dirty pages on the VM/container side may contain some "holes" – regions that are neither written by applications nor fetched from the backing store. If the data being read is completely contained in the page (by examing the linked list), then the page can be served and the read can return immediately, namely *non-blocking reads*. However, future reads may be blocked, if any part of the data being requested is not contained in the page (i.e., it falls in a hole). For such blocking reads, ideally, only the missing data needs to be fetched from the storage side. However, the missing data could be spread across noncontiguous regions and thus require multiple separate read requests. To avoid such overhead, BASS divides a page into small sub-blocks – if the missing data covers part of any sub-block, that sub-block will be fetched[8]. In this way, BASS limits the request number of fetching one partially dirtied page.

On the surface, BASS appears to transfer the blocking time from writes to reads – the reads after writes may be blocked due to holes. However, the amortized blocking time of BASS is much smaller than the conventional approach (e.g., RMW) because: (1) There is a high possibility that future reads can be served by existing pages. (2) A significant number of partial writes are not followed by any future reads of the same page, indicating the page fetch operation can be completely eliminated – recall that 62% $\sim$ 98% of the writes are not followed by any future reads of the same pages (obtained from the traces in Section 2). (3) Even in the worst case – each partial write is followed by a read that cannot be served by the current page – our evaluation shows the non-blocking approach of BASS leads to better overall performance compared to RMW, as BASS fetches less data.

[8] In practice, a 512-byte sub-block (i.e., aligned with the size of a physical sector) works well. To support "readahead", all sub-blocks with holes will be fetched, once the fetching condition is triggered.

4. Implementation

We have implemented BASS_Cache and BASS_Block in the Linux kernel 3.2.10 (∼400 LOC), and leveraged Linux's SCSI target framework [2] to realize BASS_Backend as a loadable module (∼600 LOC).

File System Integration Integrating a file system into BASS is relatively easy: To enable byte-addressability, BASS provides a new `submit_io` function for file systems to call when page reads or writes are issued. To enable non-blocking writes, BASS replaces the file systems' handling of partial writes with the *first part* of BASS's non-blocking approach. Lastly, BASS adds an additional branch to read processing to check whether pages are partially dirtied or not. We have integrated and tested Linux Ext2/3/4 with BASS.

Consistency BASS supports all existing file system journaling modes (e.g., write-back, ordered and full data journaling). In fact, BASS is only involved in the processes of data caching and I/O request generation. It does not alter any semantic level activities, allowing such file system operations to remain unchanged. Our implementation follows the Linux page locking protocol to return only *up-to-date* data to applications: A page is locked once updated (e.g., during data layout reorganization and fetching).

Error Handling Traditionally, I/O errors (e.g., due to the disks) will be reported directly to the application processes. With BASS, application processes receive reports from the *first part* of the non-blocking writes. Instead, like existing well-aligned writes, any I/O errors of partial writes during page flushing will be reported to and handled by the VM/container-side iSCSI initiator.

Cache Policy There are three main cache policies: write-back, write-through and write-around (i.e., no cache). The write-back policy exposes BASS's full potential, including both byte-addressability and the non-blocking write approach. To ensure strong durability, write-through cache strictly directs writes into both cache and underlying permanent storage before confirming I/O completion. Therefore, a write-through cache can take advantage of BASS's byte-addressability when flushing partial writes. BASS does not impact the write-around policy – the direct I/O requests follow the original I/O path.

5. Evaluation

This section presents our comprehensive evaluation of the benefits and costs of BASS using micro-benchmarks, practical I/O traces, and real world applications.

Evaluation Setup: Our testbed consisted of six physical machines hosting VMs/containers and a storage server, connected via a 10 Gigabit Ethernet switch. Each host machine was equipped with a quad-core 3.2 GHz Intel Xeon CPU and 16 GB of RAM. The storage server was equipped with an 8-core 2.5 GHz Intel Xeon CPU and 32 GB of RAM. A Linux SCSI target ("tgt") ran inside the storage server with BASS_Backend support. One 500G solid-state drive (SSD) was installed in the storage server as the backing store.

We evaluated BASS with two virtualization techniques: containers and virtual machines. Specifically, we ran micro-benchmarks and replayed I/O traces using containers and ran application workloads inside VMs. To enable BASS for containers, we installed BASS_Cache and BASS_Block in the host machine's Linux 3.2 kernel, while to enable BASS for VMs, we installed these two components inside the VM's Linux 3.2 kernel. Both containers and VMs were assigned sufficient resources to ensure that neither CPU nor memory was a bottleneck. All experiments were performed with the Linux Ext4 file system (with the *writeback* journal, i.e., only the metadata is journaled) installed in the containers or VMs. Each data-point in the experiments was calculated using the average of three executions. The standard error of measurement was less than 5% of the average for more than 90% of the results, and was less than 10% for the rest.

5.1 Micro-benchmark

In this section, we evaluate the effectiveness of BASS, using the micro-benchmark tool Fio. We focus on three aspects, impacts on writes, impacts on reads, and benefits of using high speed storage devices.

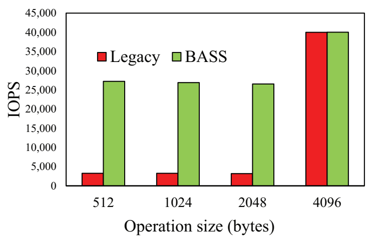
5.1.1 Impacts on Writes

We evaluated two representative write scenarios: *random-write* and *sequential-write* (with "holes" as that in Figure 7). On a host machine, we ran a Linux *container* that attached a virtual storage volume from the remote storage server. A 50 GB file was created and stored in the container's Ext4 file system. One thread of Fio generated write requests to this large file, with the I/O depth being one and the sizes ranging from 512 bytes to 4 KB. The cache of the container and storage server were cleared before each experiment, which lasted 60 seconds. All data was collected until all dirty pages were flushed to the backing store.

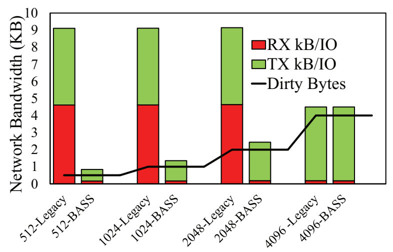
Performance Benefits To evaluate how BASS's non-blocking approach improves overall I/O performance, we compared the throughput (i.e., IOPS) with BASS and without (which we term "Legacy").

In Figure 9(a), we observe that the average throughput of partial random-writes (i.e., with the size being 512, 1024 and 2048 bytes) under BASS is ∼8X higher than that under Legacy. Specifically, under Legacy, the average throughput is only ∼3,000 IOPS, while with BASS it reaches ∼27,000 IOPS. A similar trend is observed with sequential-write in Figure 10(a): The average throughput of partial sequential-write under Legacy is about 5,000 IOPS, while the throughput using BASS reaches the same level as that in the random-write case, 27,000 IOPS. Notice that, sequential writes tend to produce higher performance than random ones because of I/O merging. Further, we identified that the reason both write cases under BASS achieve the same I/O throughput was because the storage device reached 100% utilization.

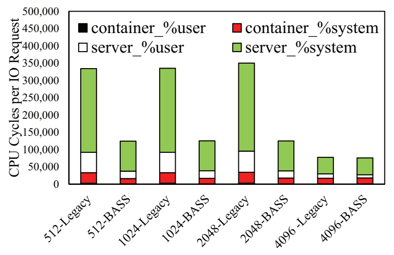
For well-aligned writes (i.e., 4 KB), the *same* throughput is observed with and without BASS. Moreover, we find well-



(a) Performance.

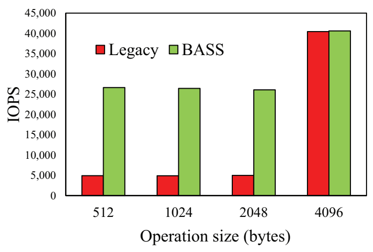


(b) Network bandwidth.

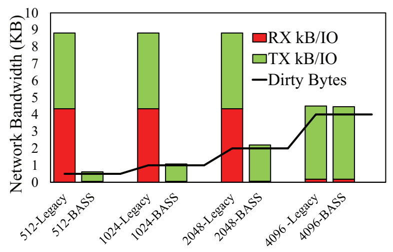


(c) CPU utilization.

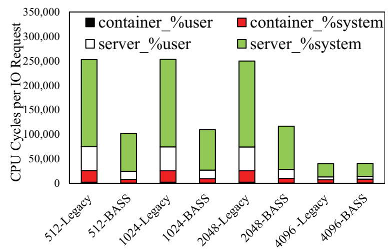
Figure 9. Random write performance, network bandwith and CPU utilization with various I/O sizes.



(a) Performance.

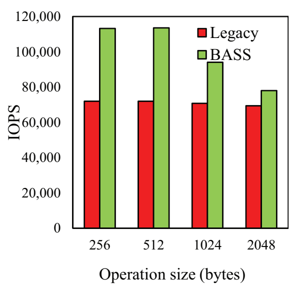


(b) Network bandwidth.

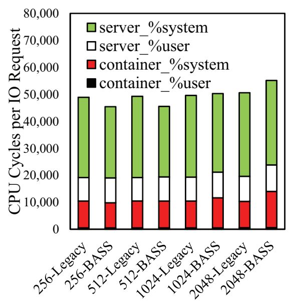


(c) CPU utilization.

Figure 10. Sequential write performance, network bandwith and CPU utilization with various I/O sizes.



(a) Performance.



(b) CPU utilization.

Figure 11. The scenario of cached random writes.

aligned writes achieve higher throughput, ∼40,000 IOPS, compared with the partial ones. This is because, as explained in Section 2, one partial write triggers two data accesses to the disk (i.e., *fetch-before-write* due to RMW), while one well-aligned write only involves one access (i.e., write-back). Note that, the slow I/O path of partial writes on the storage side has been hidden from applications by BASS's non-blocking approach. When the storage device is not fully utilized, BASS can bring more performance improvement on partial writes than Legacy (in Section 5.1.3).

These results indicate that, compared with Legacy, BASS significantly improves the throughput of both partial random-write and sequential-write. In addition, BASS does not impose any overheads on well-aligned writes.

Bandwidth Saving Recall that the byte-addressability of BASS not only enables the non-blocking approach, but also reduces the amount of data to be transferred via the network. To measure the bandwidth saving, Figure 9(b) and Figure 10(b) show the average network bandwidth usage – including both ingress (i.e., RX) and egress (i.e., TX) on the host machine – by one I/O request. The solid line depicts the number of bytes dirtied per I/O request.

As these two figures show, without BASS each partial write requires almost 9 KB of network bandwidth, while with BASS the bandwidth is only slightly larger than the size of the bytes being written. This is because (1) with the non-blocking approach, BASS avoids the page fetching of a partial write, and (2) with byte-addressability, BASS only transfers the dirty bytes of a partial write via the network. The slight overhead with BASS is caused by TCP and iSCSI headers. Moreover, the network bandwidth usage of one partial sequential-write is slightly smaller than that of one random-write. This is because, multiple sequential writes were merged into one large I/O request, resulting in fewer numbers of network packets and thus reduced overheads.

The I/O performance benefits from the bandwidth saving are not obviously demonstrated in Figure 9(a) and Figure 10(a) because, as mentioned above, the storage device is 100% utilized. To demonstrate such benefits, we used both container- and storage-side cache to buffer the accessed data, which released the storage device's load. In such a condition, Figure 11(a) shows that BASS improves the throughput of random-write by up to 57% (at 256 bytes), indicating that the saved network bandwidth further accelerates I/O performance.

Overhead As presented in Section 3, BASS introduces slight CPU and memory utilization overheads. To assess the time cost, we compare the CPU utilization with BASS and without. As BASS improves I/O performance, the overall CPU utilization also increases. For ease of comparison,

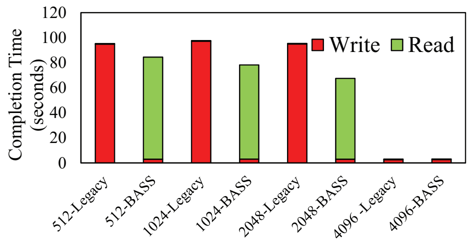
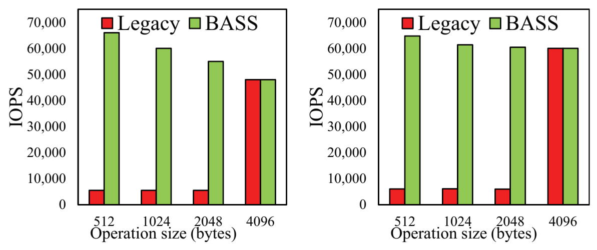


Figure 12. A worst-case scenario: each read after a partially dirtied page.



(a) Random-write. (b) Sequential-write.

Figure 13. Performance of random and sequential writes with storage-side cache loaded.

we convert CPU utilization to cycles per I/O request (CPI). First, when writes are partial and non-cached as shown in Figure 9(c) and Figure 10(c), BASS reduces the CPI of random-write and sequential-write by ∼70% and ∼55% respectively. This is because BASS optimizes the I/O path for these writes: (1) BASS eliminates page fetching, and (2) BASS avoids container-side I/O thread blocking (thus, no overheads on thread suspending/resuming). Second, when writes are partial and cached, Figure 11(b) shows that BASS yields lower or similar CPI when I/O sizes are small (from 256 bytes to 1 KB), but slightly higher CPI when I/O sizes become large (e.g., 2 KB). This indicates that as the number of dirty bytes increases, the CPU cost for data reorganization increases. However, this cost is relatively small – less than 5%. Last, when writes are well-aligned, from Figure 9(c) and Figure 10(c), we observe that BASS results in the similar cost as Legacy.

In addition to the CPU cost, BASS needs some memory space for storing the linked list data structure and reorganized I/O data. Such memory space will be released and become reusable once an I/O request completes. Our experiments show that BASS consumed at most 28 MB (i.e., 0.17% of total memory) more container-side kernel space (excluding the cached data) compared with Legacy.

5.1.2 Impact on Reads

As stated in Section 3, BASS may change the behavior of the reads that access the pages with "holes", as a result of the non-blocking approach. To measure the impact of BASS on such reads, we performed a *worst-case* scenario evaluation: We used one thread Fio to first write a non-cached file (2 GB) with various I/O sizes, leaving holes in the partially accessed pages. Afterwards, each of these same pages is read. Note that, from the I/O traces (in Section 2.4), an average of 15% of the partially dirtied pages will be accessed by the following reads, meaning most page fetches can be avoided by BASS.

Figure 12 shows the completion time for writes and reads, separately. As expected, under Legacy, writes require much longer time than reads because of RWM, while reads complete quickly because all data are cached after writes. In contrast, with BASS, writes complete fast due to the non-blocking approach, while reads take more time to finish because of fetching the *missing part* of partially dirtied pages (as presented in Section 3.3). However, as BASS only fetches the missing part of a page from the remote storage, the total completion time (i.e., the whole *read-after-write* process) with BASS is faster than Legacy – 30% shorter time is observed in the 2 KB case. Thus, from the perspective of applications, the overall I/O performance improves with BASS. Besides, BASS does not incur any cost to the well-aligned reads and writes as shown in the 4 KB case – both BASS and Legacy lead to the same completion time.

This experiment shows that, even in the worst case, BASS can improve the overall I/O performance by only fetching the necessary read data with byte-addressability.

5.1.3 Benefits of Fast Storage Devices

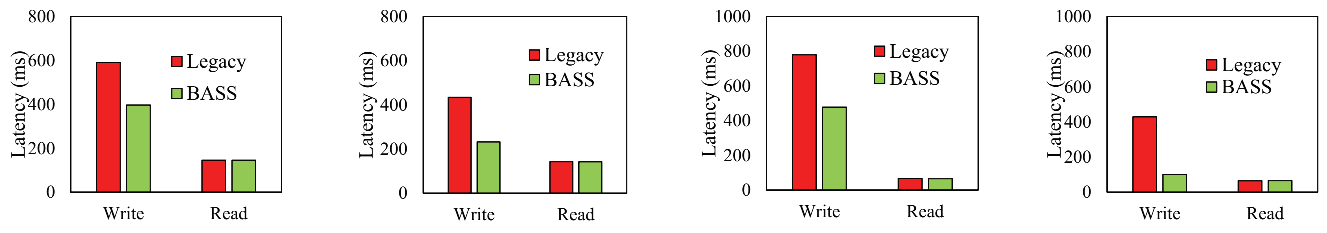
As shown in Section 5.1.1, the maximal performance of BASS is limited by the capacity of the storage device. To simulate a fast-device scenario, we loaded accessed data in the server-side cache in advance. Note that, this configuration removes the loads of fetching data from the storage device, but dirty pages are still periodically flushed to the storage device. We kept other experimental configurations the same as that in Section 5.1.1.

Under such circumstance, further throughput improvement for both random-write (Figure 13(a)) and sequential-write (Figure 13(b)) are observed with BASS. Specifically, with BASS, the throughput of both partial random-write and sequential-write jump from ∼27,000 IOPS to more than 60,000 IOPS. In contrast, with Legacy, the throughput of random-write increases from 3,000 to 5,500 IOPS, while the throughput of sequential-write slightly increases from 5,000 to 6,000 IOPS. We also notice that, the throughput of partial writes is equal to or even much higher than that of well-aligned writes. The reason is that, for partial writes, less data is transferred because of BASS's byte-addressability.

These results clearly demonstrate that the blocking issue of Legacy cannot be mitigated by fast devices, as most I/O time is spent on block storage's software layers and data transmission. On the contrary, BASS can benefit from the fast devices with both the non-blocking approach and byte-addressability.

5.2 I/O Trace Replay

We tested BASS with MobiBench's two traces, the Facebook and Twitter I/O traces, obtained from an Android device



(a) Facebook-sync. (b) Facebook-nonsync. (c) Twitter-sync. (d) Twitter-nonsync.

Figure 14. I/O latency breakdown for write and read when replaying the Facebook and Twitter I/O traces.

when using the Facebook and Twitter applications [3]. We enabled MobiBench's replay tool to work under Linux. We replayed each trace in a container. Before replaying the I/O traces, all files to be accessed were generated in advance to make sure all I/O operations can be correctly performed during replaying. Before each run, we cleared the page cache of the container and the storage server. The I/O operation latency was reported as the performance metric.

In comparison with Legacy, BASS leads to a reduction in overall latency of writes by 33% for the Facebook trace (Figure 14(a)) and 40% for the Twitter trace (Figure 14(c)). Both BASS and Legacy lead to the same overall latency of reads, indicating BASS brings no overheads to the read operations. In addition, both traces involved a significant number of "sync" operations, which required data synchronization between the container and storage sides and thus incurred relatively long I/O latency. To demonstrate the maximal benefits of BASS, we further removed all "sync" operations from the traces (we simulate the scenario of using persistent, durable memories) and replayed the traces again. Under this condition, BASS reduces the overall latency of writes by 47% for the Facebook trace (Figure 14(b)) and 77% for the Twitter trace (Figure 14(d)).

5.3 Application Workloads

We evaluated BASS with realistic application workloads, including Sysbench [6], YCSB [16], and PostMark [4].

For each workload, we evaluated the write-dominant operations in a KVM-based VM, with 4 vCPUs and 4 GB memory. One remote virtual volume was attached to the VM directly (i.e., we bypassed the hypervisor's storage I/O stack). BASS_Cache and BASS_Block were installed in the VM's kernel. To emulate a consolidation cloud environment, we ran four additional VMs with each running one *iperf* thread connecting to the storage server. The four traffic-generator VMs and the workload VM equally shared the 10 Gigabit storage network. Before each run, we cleared the page cache of the workload VM and the storage server.

Sysbench (MySQL) Sysbench is an OLTP application benchmark running on top of a transactional database. We chose MySQL (version 5.5.47) with its default storage engine, MyISAM. This storage engine uses a self-managed key buffer to cache *index* data, while leveraging the operating system's page cache to cache *row* data. We ran 20 Sysbench threads remotely in a client, which generated "INSERT" requests to MySQL (200,000 requests per run).

As Figure 15(a) shows, the "INSERT" throughput of MySQL increases by 80% with BASS compared with Legacy. The reason for this improvement is that, MyISAM manages its data with a fixed block size (e.g., 1 KB by default) that is smaller than the operating system's page size (i.e., 4 KB), thus resulting in the *partial write blocking problem* under Legacy. The blocking issue becomes severe for *write-intensive* workloads such as "INSERT". In contrast, equipped with the non-blocking approach, BASS avoids such blocking and increases the throughput significantly. With byte-addressability, BASS further reduces the network bandwidth by 22% as shown in Table 2 Row 1.

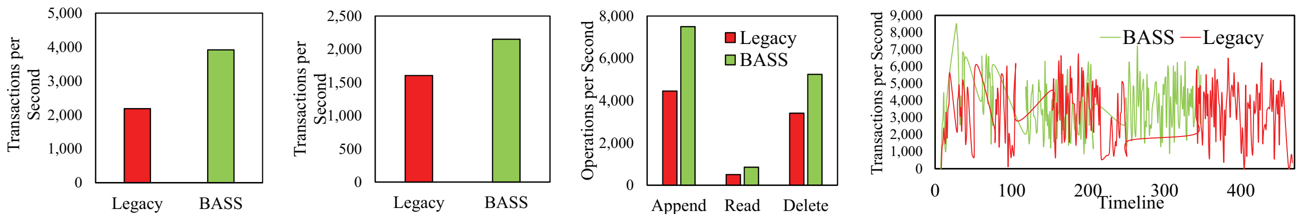
Workloads	Partial Write(%)	Bandwidth Saving(%)
Sysbench	38	21
YCSB	59	59
PostMark	46	48

Table 2. Partial write percentage and bandwidth saving.

YCSB (MongoDB) Yahoo Cloud Serving Benchmark (YCSB) is an industry-standard performance benchmark for NoSQL databases. We ran YCSB against a popular document-oriented NoSQL – MongoDB (version 3.0.10). We simply deployed a single shared MongoDB with its default storage engine, MMAPV1. We selected "INSERT" of YCSB as the core workload. For each run, the workload inserted 750,000 1-KB records.

Figure 15(b) shows that BASS improves the "INSERT" throughput of MongoDB by 34%. Different from Sysbench, the performance improvement is mainly due to the reduced network bandwidth. We observe that, during each run, ~60% of the total pages have a small number of bytes that are modified (less than 10 bytes). This is probably due to the metadata update in the page. Under Legacy, this small modification incurs a whole page flush, while BASS only flushes the dirty portion, resulting in reduced bandwidth (by 59% in Table 2 Row 2).

This is further confirmed in Figure 15(d), which displays the throughput over time. Note that, MongoDB flushes in-memory data to the backing store periodically (e.g., by default every 60 seconds). We observe that the flushing du-



(a) Sysbench. (b) YCSB. (c) PostMark. (d) YCSB's throughput over time.

Figure 15. Performance of (a) Sysbench, (b) YCSB and (c) PostMark, and (d) YCSB's throughput over time.

ration under Legacy can be 100 seconds (e.g., the interval between 250 and 350 depicted in Figure 15(d)), while the flushing duration for BASS is only 50 seconds (e.g., the interval between 200 and 250 in the same figure). During data flushing, the performance of MongoDB can be greatly impacted, meaning the shorter flushing period allowed by BASS benefits performance.

PostMark PostMark emulates the real-world usage of a file system using small file operations similar to busy mail servers and news servers. We generated 100,000 small files with sizes ranging from 1KB to 5KB. To evaluate the write-intensive scenario, we set the operation ratio of "read" to "append" to 1:9. The total transaction number of each run was 400,000. As Figure 15(c) shows BASS improves the performance (i.e., operations per second) of "append" operations by 68%. As the workload follows a strict operation ratio, the performance of "read" and "delete" operations also improves with BASS. In Table 2 Row 3, we observe a large fraction of partial writes for Postmark (i.e., 63%). Meanwhile, BASS reduces the network bandwidth usage of Postmark by 48%. Again, this experiment shows that BASS can boost the performance of small writes and reduce the storage network usage.

6. Related Work

For decades, file systems have been built on the assumption that the lower layer is the slow, block-based persistent devices, and provided the block-based interfaces. In response to these slow devices, extensive caching and prefetching techniques were studied [17, 21, 22, 26, 27]. This trend continued, even with the advent of non-volatile memory (NVM) – the same block-based interfaces were exported via the mapping layer, FTL [8, 11, 24]. Isotope [31] proposed an abstraction of a transactional block store that provides isolation in addition to atomicity and durability. To realize the benefits of emerging byte-addressable storage technologies, the authors of [15] proposed a byte-addressable file system for phase change memory (PCM). Following the similar direction, BASS bridges the addressability gap between the storage and network stacks in cloud, and boosts I/O performance for block storage.

Recently, researchers have identified challenges in storage area network (SAN) environments caused by complex hardware and software layers, which contributed the major request time and obscured the performance of fast non-volatile memory technologies [10, 14]. Further, value-added services (e.g., security and reliability) running on top of the block storage platforms compound such challenges [29]. By merging layers and offloading software functions to hardware, the optimized SAN architecture [14] greatly reduced the software overhead. BASS, instead, demonstrates the possibility of improving I/O performance by removing addressability discrepancy between software layers.

I/O latency caused by virtualization has been well-studied, and many optimization approaches have been proposed [19, 23, 28, 32]. These approaches mainly focused on the I/O stacks running in the local storage systems. vTurbo [32] accelerated I/O processing for VMs by offloading I/O processing to a designated core. Vanguard [30] tried to eliminate I/O performance interference by provisioning VMs with dedicated I/O resources. In contrast, BASS optimizes I/O performance in a cloud storage environment, and addresses the I/O performance problems by re-designing the storage stack to be compatible with the new storage system architecture.

For local block-based file systems, the partial write blocking issue can be mitigated by either using fast storage devices, or the non-blocking approach [12]. However, the hardware solutions do not work in cloud-based block storage, as the main I/O bottleneck comes from the software layers [14]. The non-blocking approach [12] eliminated the write blocking by buffering multiple data updates. However, this approach did not eliminate page fetching, and required considerable memory space. With a different scope, BASS resolves the blocking issue by taking advantage of the byte-addressable network layer. BASS not only avoids the complexity of keeping multiple updates for one page, but also eliminates unnecessary page fetching.

7. Conclusion

We have presented BASS, a byte-addressable storage stack that avoids unnecessary data transfer between VMs/containers and storage servers using variable-length I/O requests and improves performance of partial writes with a new non-blocking approach. Our evaluation with micro-benchmarks, I/O traces and realistic applications indicates the effectiveness and general applicability (to both VMs and containers) of BASS.

References

[1] Fio - flexible I/O tester synthetic benchmark. `http://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark`.

[2] Linux scsi target framework. `http://stgt.sourceforge.net/`.

[3] Mobibench traces. `https://github.com/ESOSLab/Mobibench/tree/master/MobiGen`.

[4] Postmark. `http://www.dartmouth.edu/~davidg/postmark_instructions.html`.

[5] Production file system syscall traces. `http://sylab-srv.cs.fiu.edu/dokuwiki/doku.php?id=projects:nbw:traces:start`.

[6] Sysbench OLTP benchmark. `http://www.storagereview.com/sysbench_oltp_benchmark`.

[7] Umass trace repository. `http://traces.cs.umass.edu/index.php/Storage/Storage/`.

[8] Understanding the flash translation layer (ftl) specification. `http://developer.intel.com/`.

[9] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 2013.

[10] J. Arredondo. Performance benchmark for cloud block storage, 2013. `http://c1776742.cdn.cloudfiles.rackspacecloud.com/downloads/pdfs/CloudBlockStorage_Benchmark.pdf`.

[11] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *ACM SIGOPS Operating Systems Review*, 2007.

[12] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche. Non-blocking writes to files. In *13th USENIX Conference on File and Storage Technologies*, 2015.

[13] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[14] A. M. Caulfield and S. Swanson. Quicksan: a storage area network for fast, distributed, solid state disks. In *ACM SIGARCH Computer Architecture News*, 2013.

[15] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Symposium on Operating Systems Principles*, 2009.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[17] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference*, 2007.

[18] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive nfs tracing of email and research workloads. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2003.

[19] S. Gamage, C. Xu, R. R. Kompella, and D. Xu. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[20] G. Gibson and G. Ganger. Principles of operation for shingled disk devices. *USENIX HotStorage*, 2011.

[21] B. S. Gill and L. A. D. Bathen. Optimal multistream sequential prefetching in a shared cache. *Trans. Storage*, 2007.

[22] B. S. Gill and D. S. Modha. Sarc: sequential prefetching in adaptive replacement cache. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[23] A. Kangarlou, S. Gamage, R. R. Kompella, and D. Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[24] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 2007.

[25] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX Annual Technical Conference*, 2008.

[26] M. Li, E. Varki, S. Bhatia, and A. Merchant. Tap: Table-based prefetching for storage caches. In *File and storage technologies*, 2008.

[27] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: mining block correlations in storage systems. In *Proceedings of USENIX conference on File and storage technologies*, 2004.

[28] H. Lu, B. Saltaformaggio, R. Kompella, and D. Xu. vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[29] H. Lu, A. Srivastava, B. Saltaformaggio, and D. Xu. Storm: Enabling tenant-defined cloud storage middle-box services. In *Dependable Systems and Networks (DSN), 46th Annual IEEE/IFIP International Conference on*, 2016.

[30] Y. Sfakianakis, S. Mavridis, A. Papagiannis, S. Papageorgiou, M. Fountoulakis, M. Marazakis, and A. Bilas. Vanguard: Increasing server efficiency via workload isolation in the storage i/o path. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[31] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Isotope: Transactional isolation for block storage. In *14th USENIX Conference on File and Storage Technologies*, 2016.

[32] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. USENIX Association, 2013.

[33] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.