hyperCache: A Hypervisor-Level Virtualized I/O Cache on KVM/QEMU

Taehoon Kim, Seungho Choi, Jaechun No*
College of electronics and information engineering
Sejong University
98 Gwangjin-gu, Gunja-dong, Seoul, Korea

Sung-soon Park
Dept. of Computer Engineering
Anyang University and Gluesys Co. LTD
Anyang 5-dong, Manan-gu,

***Abstract*—While VDI offers several benefits, such as the increased resource utilization and the private data protection, there exist problems that can deteriorate the system performance, including I/O virtualization overhead. Before I/O requests issued in VMs are completed in VDI, they should go through multiple software layers, such as the layer from the backend, shared storage to the host server and the layers between guest operating system, hypervisor, and eventually host operating system. In this paper, we present a hypervisor-level cache, called hyperCache, which is possible to provide a shortcut in KVM/QEMU by intercepting I/O requests in the hypervisor, while taking into account the I/O access frequency. Our experimental results demonstrate that our design improves I/O bandwidth over the existing QEMU.**

***Keywords*—access frequency; metadata repository; writeback, QEMU**

I. INTRODUCTION

In the cloud computing environment, VDI (Virtual Desktop Infrastructure) is widespread in use due to its advantages, such as improved resource utilization, protection, and consolidation [1-3]. VDI is a server-hosted solution leveraging the thin-client architecture and the centralized resource allocation. It virtually multiplexes hardware resources of the host across virtual machines (VMs), with the help of hypervisor [4-6].

Although many VDI-related researches have successfully been performed to generate the comparable system bandwidth to that of bare-metal machine, I/O virtualization overhead is still the main obstacle to boost the application performance. This is because I/O requests issued from the guest VM should go through multiple software layers until they eventually reach the physical device. The I/O overhead to be caused by such a thick I/O path can be solved either by applying the direct device assignment [7,8], or by using a VM-specific software in the paravirtual I/O [9,10]. The direct device assignment enables guest VMs to directly interact with physical devices, experiencing I/O performance close to that of bare-metal machine. However, such a performance advantage of the direct device assignment comes at the expense of complicated VM migration and less flexibility [11-13].

The second way of speeding up I/O bandwidth is to use VM-specific software such as caching in the paravirtualized I/O. The I/O cache for virtualization can be implemented either in VM layer, or in hypervisor. Although caching in VM layer is easy to implement and manage, its heavy guest OS dependency makes it difficult to port the cache component to other OSs (such as Linux to Window, or vice versa). In this paper, we present a hypervisor-level cache implemented on top of KVM/QEMU, called hyperCache, which enables to cut down the I/O virtualization path by intercepting I/O requests in QEMU and to efficiently manage the cache memory by taking into account the access frequency of the associated data in memory.

*Jaechun No is the corresponding author

II. PROBLEM DEFINITION

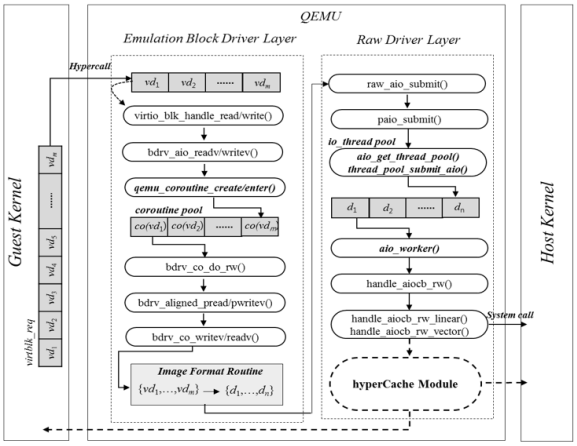

Fig. 1. QEMU I/O emulation path

QEMU [14] is an efficient binary translator for machine emulation. In the paravirtualized I/O using virtio[5], KVM hypervisor passes I/O requests issued on VMs to the virtual disk emulated by QEMU, using hyper calls. QEMU in turn translates those requests to a specific image format and stores in a regular file to be transmitted to the host kernel. Fig. 1

represents the I/O emulation path in KVM/QEMU in which threads QEMU iomode is used for the emulation. Let $\{vd_1, vd_2,\ldots,vd_m\}$ be the data blocks to be kicked off from the virtio ring buffer in the guest kernel. After performing several functions in QEMU emulation driver layer, those data blocks are converted to blocks in a specific image format, $\{d_1, d_2,\ldots,d_n\}$, by the associated coroutines. In QEMU raw driver layer, iothreads submit the converted data blocks to the host kernel by calling pread/pwrite system calls.

After thoroughly investigating the I/O emulation path depicted in Fig.1, we noticed that two performance overheads can be optimized to accelerate I/O bandwidth. The first overhead is one caused by the application waiting latency due to the thick I/O path between guest kernel, hypervisor including QEMU, and host kernel. This includes the switching overhead from the non-priviledged mode of VM to the priviledged mode of hypervisor, and vice versa. Although, several methods are proposed to mitigate the number of context switches between two modes [9], I/O-intensive applications still experience the performance degradation. Secondly, in the case of write operations, the data is frequently submitted to the host kernel in small pieces, aggravating the write performance. Such an overhead becomes apparent when applications on VMs execute heavy write operations. The primary objective of our proposed method attempts to minimize those overheads in the hypervisor to generate better I/O performance.

III. IMPLEMENTATION DETALS

A. HylerCache

We developed the hyperCache for two objectives. First, the hyperCache enhances I/O throughput by providing the shortcut of I/O execution path between guest VMs and host. To alleviate I/O latency, the hyperCache intercepts I/O requests in QEMU and checks the availability of the necessary data using the in-memory metadata tables. Second, the hyperCache utilizes I/O access frequency of data blocks in a way that it can guarantee the fast responsiveness for the important I/O requests. Also, by coalescing data blocks in QEMU before transferring them to the host kernel, the hyperCache can alleviate the disk contention on the physical I/O device attached to the host.

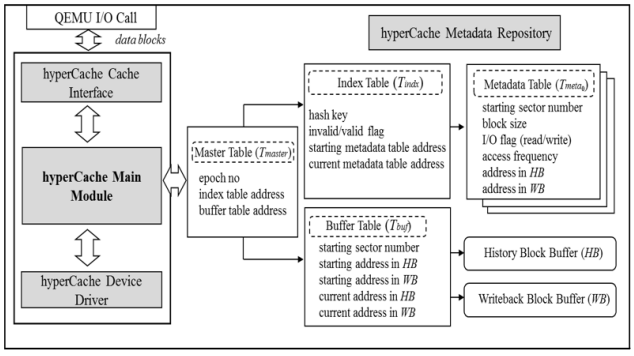

Fig.2 An overall structure of hyperCache

Fig. 2 shows an overall structure of the hyperCache. The hyperCache provides two I/O modes: *hC*-read and *hC*-writeback. The *hC*-read accumulates the memory-resident data blocks representing high I/O access frequency and, instead of forwarding I/O requests to the host, it provides the desired data from the accumulated blocks to applications on VMs. The *hC*-writeback merges the image formatted data blocks to hand over them to the host together, while synchronizing with *hC*-read for data consistency.

Fig. 2 represents the four components of the hyperCache. The I/O requests are captured by the hyperCache cache interface to determine the cache hit or miss. Based on the availability check, the main module processes the necessary steps to retain the data in the hyperCache memory buffer, using the metadata repository. Besides the four tables depicted in Fig. 2, the metadata repository maintains the two in-memory buffers, called writeback block buffer (WB) for *hC*-writeback mode and history block buffer (HB) for *hC*-read mode. Finally, the data in WB would be flushed by the hyperCache device driver, if necessary.

B. HyperCache Procedures

Definition 1: The structure of the hyperCache can be expressed by (E, D, ϕ, *MT, HB, WB*), where

- E is a set of epochs at which *WB* for *hC*-writeback is flushed out to the host and *HB* for *hC*-read is invalidated for the initialization.
- $D=\{d_1, d_2, \ldots, d_n\}$ is a number of data blocks to be transmitted to the hyperCache interface. For a block $d_i \in D$, $s(d_i)$ and $\Delta(d_i)$ are the starting sector number and the number of I/O access frequency of d_i, respectively. Also, $o(d_i)$ represents the current I/O activity for d_i. For instance, $o(d_i)$ = *read*/*write* indicates that d_i belongs to read/write requests.
- ϕ is a map function determining the hit or miss:
- $\phi : D \rightarrow \{0,1\}$ ($\phi(d_i) = 1$: hit, $\phi(d_i) = 0$: miss)
- $MT=\{Tmaster, Tindx, Tmeta, Tbuf\}$ is the metadata repository. Let $\{t_1, t_2,\ldots,t_n\}$ be the table entries of *Tindx*. Then, each $t_i \in Tindx$ is linked with δ number of *Tmeta* to resolve the hash collision problem.
- *HB* and *WB* are the hyperCache buffers for *hC*-read and *hC*-writeback.

Definition 2: Let $d_i \in D$ be a data block with which $o(d_i)$=*read* and *HIT* be the threshold of I/O access frequency. Then, d_i satisfies the followings:

case 1) $\Delta(d_i) > HIT \Rightarrow \phi(d_i) = 1$ and $d_i \in HB$

case 2) $\Delta(d_i) = HIT \Rightarrow \phi(d_i) = 0$ and $HB = HB \cup d_i$

case 3) $\Delta(d_i) < HIT \Rightarrow \phi(d_i) = 0$

Also, for any block $d_k \in D$ where $o(d_k)$=*write*, $WB = WB \cup d_k$.

The hyperCache calculates the I/O access frequency of blocks to be accessed. Since the blocks with the high access

frequency are likely to belong to the working-set due to their locality, the hyperCache stores them in HB to reuse for further I/O requests. Selectively retaining blocks by taking into account the access ratio makes it possible to control the memory capacity of HB, preventing from the memory shortage. On the other hand, the blocks involved in write requests are collected in WB, before issuing the system call to the host kernel. Both buffers are initialized for every epoch, while flushing out blocks in WB to the physical I/O device. Algorithm 1 shows the steps for the hyperCache initialization.

Algorithm 1: *hC*-initilization

Input: Null
Output: $MT = \{Tmaster, Tindx, Tmeta, Tbuf\}$, *HB*, and *WB*
1. Create the metadata repository and memory buffers
2. Set the new epoch number

Algorithm 2: *hC*-read and *hC*-writeback at an epoch E_v

Input: $d_i \in D$
Output: $\phi(d_i)$ indicating cache hit or miss, *HB*, and *WB*

```
1.  for (each epoch transition from E_u to E_v){
2.      flush out WB; invalidate MT and HB;
3.  }
4.  Calculate the hash key using s(d_i);
5.  if (s(d_i) is not available in Tindx ) {
6.      update Tindx and Tmeta with the metadata of d_i;
7.  }
8.  if (o(d_i) = read) { // hC-read
9.      Δd_i ++;
10.     if (Δd_i < HIT) {
11.         φ(d_i) =0;
12.         exit hC-read to go to the host kernel;
13.     }
14.     else if (d_i is not mapped to HB) {
15.         φ(d_i) =0; // let hp be the current address of HB
16.         HB = HB ∪ d_i ; update Tbuf with hp;
17.     }
18.     else {
19.         φ(d_i) =1; // cache hit
20.         access HB to retrieve d_i and return;
21.     }
22. }
23. else { // hC-writeback
24.     if ( read history of d_i is available ) {
25.         update Tmeta with the new write of d_i;
26.         HB = HB – d_i;
27.     }
28.     // let wp be the current address of WB
29.     WB = WB ∪ d_i ;  update Tbuf with wp;
30. }
```

Algorithm 2 illustrates the steps involved in *hC*-read and *hC*-writeback. The HB and WB are invalidated per epoch transition, and at the same time the data in WB is flushed out to the host kernel. In the case of *hC*-read, after locating and updating the associated metadata, the hyperCache accounts for the I/O access frequency, to check the data availability in HB. First, in the case that the value is less than the threshold (HIT), the I/O path exits the hyperCache to enter the host kernel. Second, if the value is equal to the threshold, then the data block is duplicated in HB, although such a case is still considered as a miss. Finally, the map function, ϕ(di), is defined as a hit and returned to VM. In the case of *hC*-writeback, based on the availability of the read history, the synchronization with HB should be taken for the consistency, while updating WB and Tbuf to reflect write operations in applications.

IV. PERFORMANCE EVALUATION

A. Experimental Platform

We evaluated the hyperCache performance on a desktop PC equipped with AMD FX-8350 8 core and with 32GB of memory. The Operating system was Ubuntu 14.04 and the kernel version was 3.13.0.24-generic. Also, the hard disk was 1TB of 7200RPM Seagate SATA-3. On top of the host, we created VMs using KVM hypervisor, and each of VMs was configured with 2 core CPU, 8GB of memory, and 20GB of hard disk. The operation system for VMs was CentOS 6.4 and the kernel version was 2.6.32-431. We modified the iozone benchmark, in order to continuously generate the necessary files for a specific period of time.

B. I/O Performance of hyperCache

Fig. 3 and 4 show the bandwidth of the *hC*-read. In the figures, *hC*-read(A) means that the HB capacity is configured as retaining 25% of data involved in I/O requests. Also, *hC*-read(B) and *hC*-read(C) indicate that 75% and 100% of the desired data, respectively, are designated in HB. We compared the read performance of three hyperCache types to that of the baseline configuration where the original module is used for handling I/O requests in QEMU. In Fig. 3, we varied file sizes from 64KB to 1MB. In the case of reading 1MB of data, *hC*-read(A) generates about 10 times higher read performance than the baseline configuration. Also, we notice that increasing HB capacity size can contribute to accelerate the read performance, because *hC*-read(C) shows 27% better bandwidth than *hC*-read(A).

However, in Fig. 4 where file sizes are varied between 4MB and 512MB, we can see that the overhead for accessing the metadata repository is not negligible in the read performance. In this figure, the bandwidth of *hC*-read(A) becomes almost the same to that of the baseline configuration because the benefit of serving data in the hyperCache is offset by the overhead incurred to search for the associated metadata in the metadata repository. As a result, in the case of reading large files, the overhead of looking for the desired table entries using the hash function has a critical impact in the read

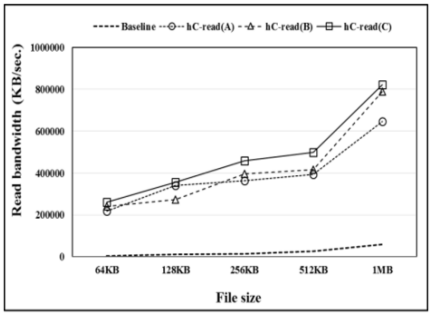

Fig. 3. *hC*-read bandwidth (case 1)

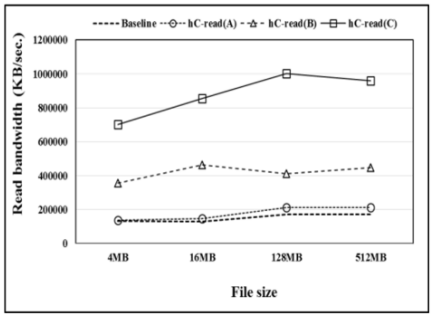

Fig. 4. *hC*-read bandwidth (case 2)

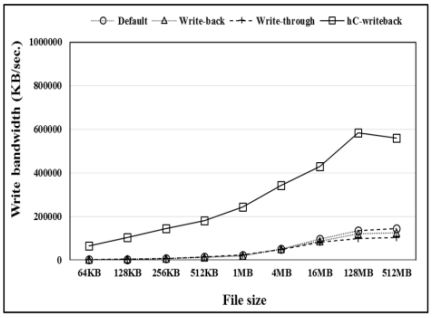

Fig. 5. *hC*-writeback bandwdith

performance, due to the large number of data blocks involved in the read requests. Nonetheless, with the appropriate HB capacity, the bandwidths of *hC*-read(B) and *hC*-read(C) increase by a factor of 1.6x and 4.7x than that of the baseline configuration with 4MB of files.

Fig. 5 shows the write bandwidth of *hC*-writeback, while comparing it to I/O modes of QEMU, including the default mode, write-back mode, and write-through mode. In the case of write operations, regardless of file sizes, the bandwidth of the hyperCache shows the consistent performance improvement. As soon as write requests take place, the data is retained in the WB of hyperCache. Either the WB is occupied with blocks or the epoch transition occurs, the system call to the host kernel is issued to flush out WB. Fig. 5 shows that such an I/O behavior mitigates the performance latency in accessing the associated table entries in the metdadata repository.

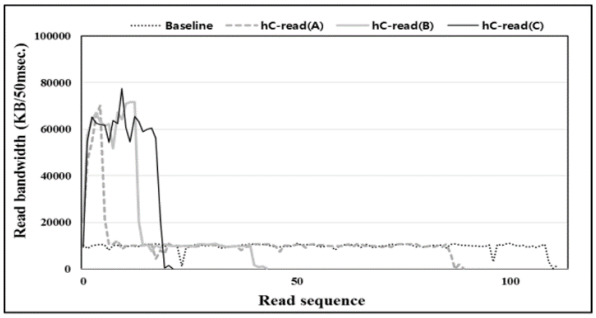

Fig. 6. *hC*-read bandwidth change per 50msec.

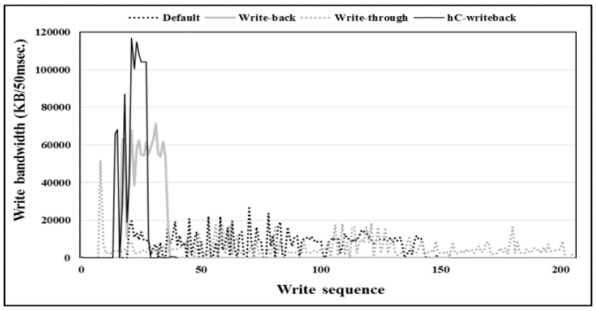

Fig. 7. *hC*-writeback bandwidth change per 50msec.

In order to verify the I/O bandwith of the hyperCache, in Fig. 6 and 7, we present how the read and write bandwidths are varied as times progress. Both bandwidths were obtained at every 50msec in reading and writing 1GB of file using 4KB of record size. In Fig. 6, *hC*-read(C) completes much faster than the others, generating the highest read performance, as shown in Fig. 4. Also, Fig, 7 depicts that the write sequence of the hyperCache is shorter than that of QEMU three modes, verifying the highest write bandwidth of *hC*-writeback.

V. CONCLUSION

The hyperCache utilizes the I/O access frequency of the data in QEMU to provide a shortcut in the virtualized I/O stack. It intercepts I/O requests based on the accumulated access history information, while merging data for write operations in a large chunk. The performance evaluation shows that such an optimization can improve the read bandwidth by up to 4.7x as compared that of the baseline configuration. Also, we can notice that coalescing data in the hyperCache works effectively for write operations. However, the overhead of metadata repository has a critical impact in the I/O bandwidth, which we should address in our future work.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2016R1D1A1B03930683). Also, this work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT)(R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services).

REFERENCES

[1] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment," Proc. USENIX Annual Technical Conference, pp. 133-144, June 2014.

[2] T. Ferreto, M. Netto, R. Calheiros, and C. DeRose, "Server consolidation with migration control for virtualized data centers" Journal of Future Generation Computer Systems, vol. 27, no. 8, pp.1027-1034, 2011.

[3] K. Razavi and T. Kielmann, “Scalable Virtual Machine Deployment Using VM Image Cache,”, Proc. SC’13 on High Performance Computing, Networking, Storage and Analysis, Denver, USA, November 2013.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “KVM: the Linux virtual machine monitor,” Proc. Ottawa Linux Symposium, pp.225-230, 2007.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 164-177, 2003.

[6] A. Menon, A. Cox, and W. Zwaenepoel, “Optimizing network virtualization in Xen,” Proc. 2006 USENIX Annual Technical Conference, Boston, MA, 2006.

[7] B.-A. Yassour, M. Yehuda, and O. Wasserman, “Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines,” Tech. Rep. H-0263, IBM Research Division, Haifa, Israel, 2008.

[8] S. Bhosale, A. Caldeira, B. Grabowski, C. Graham, A. Hames, V. Haug, M. Kahle, C. Maciel, M. Mangalur, and M. Sanchez, “IBM Power Systems SR-IOV,” IBM redpaper, 2014.

[9] N. Har’El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, “Efficient and Scalable Paravirtual I/O System,” Proc. USENIX Annual Technical Conference, June 2013.

[10] R. Russell, “Virtio: Towards a de-facto standard for virtual I/O devices,” ACM SIGOPS Operating Syst. Review, vol. 42, no. 5, pp.95-103, July 2008.

[11] P. Lu and K. Shen, “Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache,” Proc. 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 2007.

[12] M. Saxena, P. Zhou, and D.A. Pease, “vDrive: An Efficient and Consistent Virtual I/O System,” Proc. ACM SOSP, Oct. 2015.

[13] H. Kim, H. Jo, and J. Lee, “XHive: Efficient Cooperative Caching for Virtual Machines,” IEEE Transactions on computer, vol. 60, no, 1, pp. 106-119, Jan. 2011.

[14] F. Bellard, “QEMU, a fast and portable dynamic translator,” Proc. USENIX Annual Technical Conference, April 2005.