Multi-layered I/O Virtualization Cache on KVM/QEMU

Jaechun No
College of electronics and information engineering
Sejong University
98 Gwanjin-gu, Gunja-dong, Seoul, Korea

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

***Abstract*—** While virtualization has become an important component in VDI, I/O is still an open challenge in the virtualization. Although several researches attempted to mitigate virtualized I/O overheads, many of them concentrated on either VM-level optimization or hypervisor-level one, not both. In this paper, we propose a multi-layered cache implementation, called MultiCache, which combines the guest-level optimization with the hypervisor-level I/O optimization. We executed the performance measurement of MultiCache to verify its effectiveness.

Keywords—multi-layered; VDI; guest-level; hypervisor-level; QEMU

I. INTRODUCTION

In recent years virtualization has become an essential component in VDI(Virtual Desktop Infrastructure) due to its flexibility of decoupling virtual machine (VMs) from physical resources while allowing various ways of utilizing host resources to the maximum extent. This is possible because VDI multiplexes hardware resources of the host among VMs, which can improve server resource utilization and density.

While VDI offers several benefits, there exist problems that can deteriorate the system performance, including I/O virtualization overhead [1-3]. 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 [4,5].

Due to the thick software stack of VDI, implementing the virtualization cache method needs to take into account several layers, with each I/O request passing through [6,7]. For example, considering only the guest VM for the cache may not be enough to achieve the desirable I/O performance, because the latency occurring in the hypervisor, such as the context switching between the non-root mode and the root mode, can substantially deteriorate application executions [8,9]. Also, the OS dependency makes it difficult to port the guest-level cache method across VMs, especially in the case that VMs execute different guest operation systems.

In this paper, we propose the virtualization cache mechanism on top of KVM, called MultiCache (Multilevel virtualization Cache implementation), which combines VM's guest-level component with QEMU's hypervisor-level component. The main goal of the guest-level component of MultiCache is to alleviate the I/O overhead occurring in the file transmission between the backend, shared storage and the guest [10]. Also, caching on the guest-level can give the better chance to retain the application-specific data. The hypervisor-level component of MultiCache attempts to reduce I/O latency by supplying the desired data in QEMU as many as possible, instead of accessing the physical device of the host. The other contribution of the hypervisor-level component is to provide fast responsiveness by reducing the application process block time before I/O completion.

This paper is organized as follows: In section 2, we describe the overall structure of MultiCache. In section 3, we present the performance measurement and, in section 4, we conclude with a summary.

II. IMPLEMENTATION DETAILS

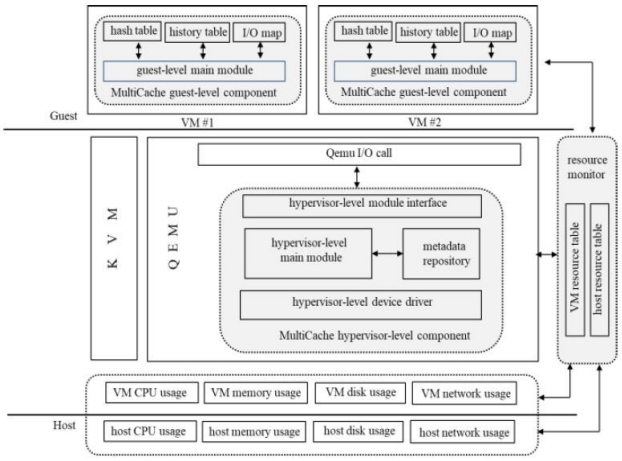

Fig. 1. MultiCache Structure

Fig. 1 represents an overall structure of MultiCache. As can be seen in the figure, MultiCache is divided into three components: guest-level component, hypervisor-level component, and resource monitoring component. The main goal of the guest-level component is to mitigate the I/O latency between the shared storage and the guest, by utilizing the history information of application I/O executions. Furthermore, by retaining the application-specific data in the guest, it can reduce I/O accesses to the physical device attached to the host. Finally,

it tries to determine the effective cache size while taking into consideration VM and host resource usages in real-time.

The guest-level component works at VM and is consisted of three tables, including hash table, history table and I/O map, to detect application's I/O activities and to retain the associated metadata representing the execution history logs. Those logs are used to predict the next I/O behavior to preload the preferential files from the shared storage and also used to maintain recently referenced files in VM.

The hypervisor-level component was implemented in QEMU. The primary objective of this component is to minimize the I/O latency incurred in the virtual to hypervisor transition, by using the I/O access frequency measured in QEMU. Also, by intercepting I/O requests before they go to the host kernel, the component tries to reduce I/O contention among VMs.

The first attribute of the component is the module interface interacting with QEMU I/O call while exchanging the associated I/O metadata with it, such as sector numbers requested. The main module of the component receives the I/O metadata from the interface and determines the hit or miss, while communicating with the metadata repository that contains the history logs of hypervisor's I/O execution, such as I/O access frequency. The device driver of the component is responsible for managing the hypervisor cache memory.

A. MultiCache Guest-Level Component

The guest-level component of MultiCache was implemented to optimize network and I/O overheads incurring in file transmissions between the shared storage and the guest VM. Furthermore, by monitoring and accumulating I/O history information, MultiCache enables to provide better I/O responsiveness and data reliability.

To maintain the history information, MultiCache uses two kinds of tables: hash table and history table. The hash table is constructed with hash keys and is used to locate the associated history table containing the corresponding file metadata. There are λ history tables organized to solve the hash collision. One of the important file metadata in the history table is the usage count. Every time files are accessed for read and write operations, their associated usage count is increased by one to indicate the file access frequency. Also, MultiCache uses two I/O maps to determine the number of files to prefetch it from and to replace it to the shared storage.

Fig. 2 shows the structure of MultiCache guest-level component. First, with the file inode, the hash key to access the hash table is calculated. The associated hash table entry contains the current history table address and its entry number where the desired file metadata can be retrieved. If the new file is used for I/O, then the next empty place in the current history table is provided to store its metadata.

In order to maintain the appropriate cache memory size in the guest, only the files with each having the usage count no less than *USAGE_THRESHOLD* are stored in the cache and their file metadata is inserted into the read or write map, based on file read or write operations.

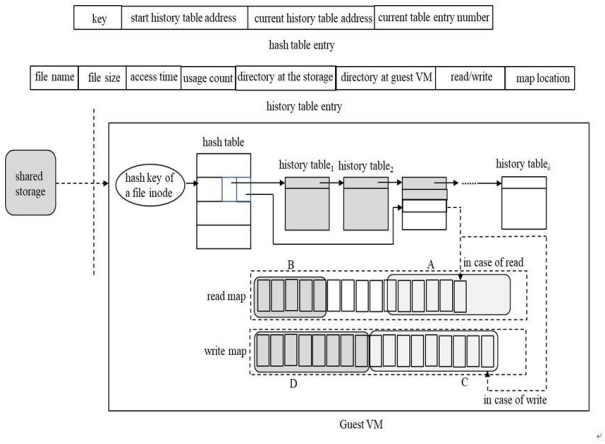

Fig. 2. MultiCache guest-level component

In Fig. 2, sections *A* (cache window size) and *C* in the read and write maps, respectively, illustrate the files that should be maintained in the cache memory; sections *B* and *D* are the candidates to be replaced under the cache memory pressure.

MultiCache can enhance the read responsiveness by caching more files whose most recent I/O accesses are read operations. Such a process involves replacing less files mapped in the section *B*. Similarly, MultiCache can replace more dirty files mapped to the section *D* for data reliability and availability.

Let M_g be the guest-level cache memory size and *MEM_THRESHOLD* be the memory usage limitation over which files designated at sections *B* and *D* must be flushed out to maintain the appropriate cache memory capacity. Finally, let f_a, f_b, g_c, and g_d be files whose metadata are mapped to sections *A*, *B*, *C*, and *D*, respectively.

At each time epoch, MultiCache checks M_g by communicating with the resource monitor, to see if the following condition is satisfied:

$$\left\{\sum_{f_a \in A} s(f_a) + \sum_{f_b \in B} s(f_b) + \sum_{g_c \in C} s(g_c) + \sum_{g_d \in D} s(g_d)\right\} / M_g \leq MEM\_THRESHOLD$$

B. MultiCache Hypervisor-Level Component

The hypervisor-level component was implemented to minimize the I/O overhead caused by the software stack between the guest VM and the host. Before completing I/O requests, there are several mode transitions taking place between non-root mode and root mode, which incurs the application execution being blocked.

Furthermore, because those requests require to access the data from the physical device attached to the host, the optimization at the hypervisor needs a way of reducing I/O contention on the device during the service time.

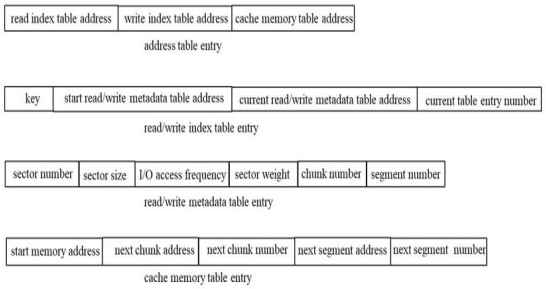

Fig. 3. Metadata repository of MultiCache hypervisor-level component

MultiCache hypervisor-level component uses several tables, called the metadata repository, to maintain I/O-related metadata at the hypervisor. Fig. 3 shows the tables in the metadata repository.

The address table stores the addresses of the read and write index tables containing the hash key and the start and current addresses of the read and write metadata tables. Similar to the history tables of the guest-level component, λ read metadata tables and λ write metadata tables are organized to target the collision problem.

The read and write metadata tables contain the access information about the sectors transferred from QEMU I/O calls; the cache memory table maintains the next chunk and segment addresses of the hypervisor cache memory.

The hypervisor-level component uses I/O access frequencies of sectors to determine if those sectors should be retained in the cache memory. The I/O access frequency indicates how many times the associated sectors were used in I/O requests.

There are two reasons for utilizing I/O access frequency. First, because the cache memory maintained in MultiCache is of a restricted size, a criteria is needed to filter sectors before storing them in the cache memory. In MultiCache, only those sectors who have been accessed no less than a threshold (*FREQ_THRESHOLD*) are stored in the cache memory.

Second, besides optimizing the mode transition and I/O contention aforementioned, MultiCache gives an opportunity to prioritize I/O requests, according to the VM's different importance. In other words, I/O requests issued in the high-priority guest VM can be executed first, despite their access frequency. In MultiCache, the priority of VM is determined by the number of CPUs and the memory capacity with which the VM was configured: the more number of CPUs and the larger memory size it is assigned, the higher priority the guest is given.

Let S be a set of sectors consisting of I/O requests in a guest. Consider a host where N number of VMs are currently executing. Also, each VM(i) is configured with u_i number of CPUs and v_i memory capacity.

Definition 1) A sector $sc \in S$ issued from VM(i) is defined by four components: p_{sc}, w_{sc}, δ_{sc}, and m_{sc} :

1. p_{sc} is the I/O access frequency of sc
2. w_{sc} is the weight of sc satisfying

$$w_{sc} = p_{sc} \times \frac{u_i}{\sum_{k=1}^{N} u_k} \times \frac{v_i}{\sum_{k=1}^{N} v_k}$$

and the weight of VM(i) is

$$\frac{u_i}{\sum_{k=1}^{N} u_k} \times \frac{v_i}{\sum_{k=1}^{N} v_k}$$

3. δ_{sc} is the mapping function, indicating either cache hit (δ_{sc}=1) or miss (δ_{sc}=0)
4. m_{sc} is the position of the cache memory where sc is stored if $w_{sc} \geq FREQ\_THRESHOLD$

III. Performance Evaluation

We executed all experiments on a host server equipped with an AMD FX 8350 eight core processors, 24GB of memory, and 1TB of Seagate Barracuda ST1000DM003 disk. Also, the other server having the same hardware specification as the host server is configured as the shared storage node. Two servers are connected with 1Gbit of network. The operating system was Ubuntu release 14.04 with 3.13.0-24 generic kernel. We installed the virtual machine on top of the host server by using KVM hypervisor.

We first evaluated the guest-level component of MultiCache. In order to analyze the accurate I/O performance pattern of the guest-level component, we used the original KVM/QEMU version that is not integrated with the MultiCache hypervisor-level component. Also, we modified postmark to connect between the host server and the shared storage node. As a result, when files are generated from postmark, the files already brought into the guest from the storage node are read from MultiCache (*cached*) and the other files not residing in MultiCache are read from the storage through NFS (*not cached*).

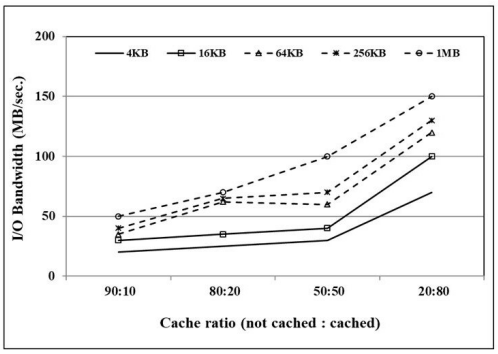

Fig. 4. I/O bandwidth of MultiCache

Fig. 4 shows I/O bandwidth while varying file sizes from 4KB to 1MB. The *x*-axis represents the ratio of *not cached* to *cached*. For example, (90:10) implies that 90% of files to be needed during transactions are exchanged with the shared

storage node. The number of transactions is 20000 and the ratio of read to write is 50:50.

In the figure, as the percentage of files being accessed from MultiCache becomes high, better I/O bandwidth is achieved. Moreover, the effect of MultiCache is more apparent with large files. For example, with (20:80) where 80% of files are accessed from MultiCache, about 53% of I/O bandwidth improvement is observed with 1MB of files as compared to that of 4KB of files. The reason is that as more number of large files is accessed from MultiCache the network overhead to transfer data to VM becomes small, resulting in the bandwidth speedup.

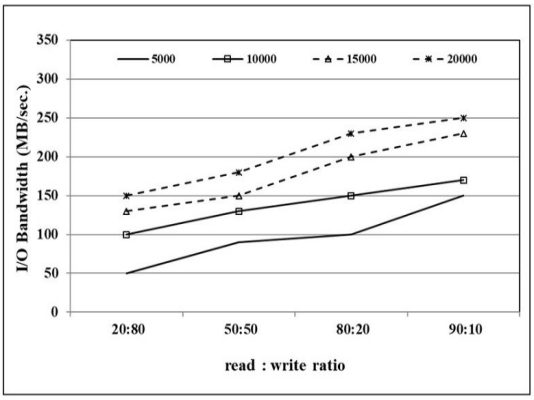

Fig. 5. Performance evaluation based on I/O accesses

In the evaluations, we observed that the effect of MultiCache is especially obvious with read operations, as shown in Fig. 5. In order to see the impact of MultiCache in the mixed I/O operations, we varied the read and write percentages while increasing the number of transactions.

In Fig. 5, (80:20) means that 80% of transactions are read operations and 20% of transactions are writes. Also, we used 1MB of file size. Fig. 5 exhibits that better I/O throughput is generated with the large number of transactions and especially with the larger percentage of read operations. This is because write operations inevitably incur network and I/O overheads to store data to the shared storage and such burdens may lower the throughput.

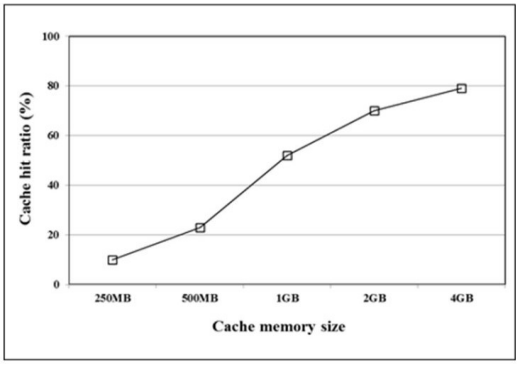

Fig. 6. The effect of the cache memory

We measured the I/O performance of the hypervisor-level component. In this experiment, there is no file transmission between the shared storage and the guest. In other words, all files for I/O were generated from the postmark benchmark running on the guest. The file sizes vary between 4KB and 1MB.

First of all, we observed the effect of the hypervisor cache memory in Fig. 6, while changing the cache memory size from 250MB to 4GB. To warm the cache, we executed the modified postmark for 5 seconds and took the average value of each test case. Fig. 6 shows the cache hit ratio obtained while changing the cache memory size. The figure shows that as the cache memory size becomes large, so does the hit ratio. For example, increasing the memory size from 250MB to 4GB shows the hit ratio improvement by up to 6.9x.

However, there is a subtle difference worthwhile to observe in the figure. While the hit ratio improves 126% from 500MB to 1GB, the hit ratio from 1GB to 2GB increases 34%. Extending the cache memory from 2GB to 4GB produces even the smaller percentage of hit ratio improvement. We guess that this is because the locality is shifted as time goes on. Also, the metadata stored in the metadata repository are replaced to the new ones due to the space restriction.

IV. Conclusion

We proposed a multi-layered cache mechanism, called MultiCache, to optimize I/O virtualization overhead. The first layer of MultiCache is the guest-level component whose main goal is to optimize the I/O overhead between the backend, shared storage and the guest. Also, caching the application-specific data in the guest can contribute to accelerate the performance speedup.

To achieve this goal, the guest-level component uses the history logs of file usage metadata to preload preferential files from the shared storage and to maintain recently referenced files in the guest. The second layer of MultiCache is to minimize the I/O latency between the guest and the host, by utilizing the I/O access frequency in QEMU. Also, by intercepting I/O requests in QEMU before they are transferred to the host kernel, the component can mitigate I/O contention on the physical device attached to the host.

The performance measurement with the postmark demonstrates that our approach is beneficial in achieving high I/O performance in the virtualization environment. As a future work, we will evaluate MultiCache with more real applications to prove its effectiveness in improving I/O performance.

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 of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT)(No.2019-0-00150, Decentralized cloud technologies for edge/IoT integration in support of AI applications).

REFERENCES

[1] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter(and Safer) Than Your Container," Proc. Of the 26th Symposium on Operaing Systems Principles(SOSP), 2017.

[2] J. Santos, Y. Turner, G. Janakiraman, I. Pratt, "Bridging the Gap between Software and Hardware Techniques for I/O Virtualization," Proc. 2012 USENIX Annual Technical Conference, Boston, MA, 2008.

[3] W. Jia, C. Wang, X. Chen, J. Shan, X. Shang, H. Cui, X. Ding, L. Cheng, F. Lau, Yuexuan Wang, Yuangang Wang, "Effectively Mitigating I/O Inactivity in vCPU Scheduling," Proc. Of the 2018 USENIX Annual Technical Conference, Boston, MA, USA, July 2018.

[4] 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.

[5] 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.

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

[7] 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.

[8] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, A. Traeger A, "Towards Exitless and Efficient Paravirtual I/O," Proc. 5th Annual International Systems and Storage Conference, Haifa, Israel, June 2012.

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

[10] V. Tarasov, D. Hidebrand, G. Kuenning, E. Zadok, "Virtual Machine Workloads: The Case for New Benchmarks for NAS," Proc. 11th USENIX Conference on File and Storage Technologies (FAST'13), Santa Clara, CA, pp. 307-320, 2013.