Hindawi Publishing Corporation
Scientific Programming
Volume 2016, Article ID 3780163, 13 pages
http://dx.doi.org/10.1155/2016/3780163

Research Article

MultiCache: Multilayered Cache Implementation for I/O Virtualization

Jaechun No[1] and Sung-soon Park[2]

[1] *College of Electronics and Information Engineering, Sejong University, 98 Gunja-dong, Gwangjin-gu, Seoul 143-747, Republic of Korea*
[2] *Department of Computer Engineering, Anyang University and Gluesys Co. LTD, Anyang 5-dong, Manan-gu 430-714, Republic of Korea*

Correspondence should be addressed to Jaechun No; jano@sejong.edu

Received 16 February 2016; Revised 18 June 2016; Accepted 3 July 2016

Academic Editor: Zhihui Du

As the virtual machine technology is becoming the essential component in the cloud environment, VDI is receiving explosive attentions from IT market due to its advantages of easier software management, greater data protection, and lower expenses. However, I/O overhead is the critical obstacle to achieve high system performance in VDI. Reducing I/O overhead in the virtualization environment is not an easy task, because it requires scrutinizing multiple software layers of guest-to-hypervisor and also hypervisor-to-host. In this paper, we propose multilayered cache implementation, called MultiCache, which combines the guest-level I/O optimization with the hypervisor-level I/O optimization. The main objective of the guest-level optimization is to mitigate the I/O latency between the back end, shared storage, and the guest VM by utilizing history logs of I/O activities in VM. On the other hand, the hypervisor-level I/O optimization was implemented to minimize the latency caused by the "passing I/O path to the host" and the "contenting physical I/O device among VMs" on the same host server. We executed the performance measurement of MultiCache using the postmark benchmark to verify its effectiveness.

1. Introduction

Recently, VDI (Virtual Desktop Infrastructure) is becoming an essential aspect of the cloud-based computing environment due to its advantages such as user customization, easy-to-maintain software, and location-transparent accesses [1–3]. VDI multiplexes hardware resources of the host among VMs, which can improve server resource utilization and density. Also, VDI is capable of isolating VMs on the same host platform, which can offer the performance isolation and the secure application execution in the guest. This is performed by using the hypervisor that is responsible for coordinating VM operations and for managing physical resources of the host server.

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 [4, 5]. Before I/O requests issued in VMs are completed in VDI, they should go through multiple software layers, such as the layer from the back end, shared storage to the host server [6, 7], and the layers between guest operating system, hypervisor, and eventually host operating system.

Figure 1 shows the I/O virtualization path using KVM hypervisor and QEMU emulator. The application I/O requests are first handled by the guest kernel before being passed to the virtual, emulated device executing in the user space. After executing several modules including ones for the image format, those requests are entered to the host kernel by calling posix file system interface. The virtual disk is typically a regular file from the perspective view of the host file system. The files necessary for I/O requests can be stored either in the local disk attached to the host or in the shared storage connected by network.

As depicted in Figure 1, because the I/O virtualization path is organized with multiple software layers, optimizing I/O cost is very challenging, which requires scrutinizing various virtualization aspects. In this paper, we are interested in mitigating such an overhead by implementing the

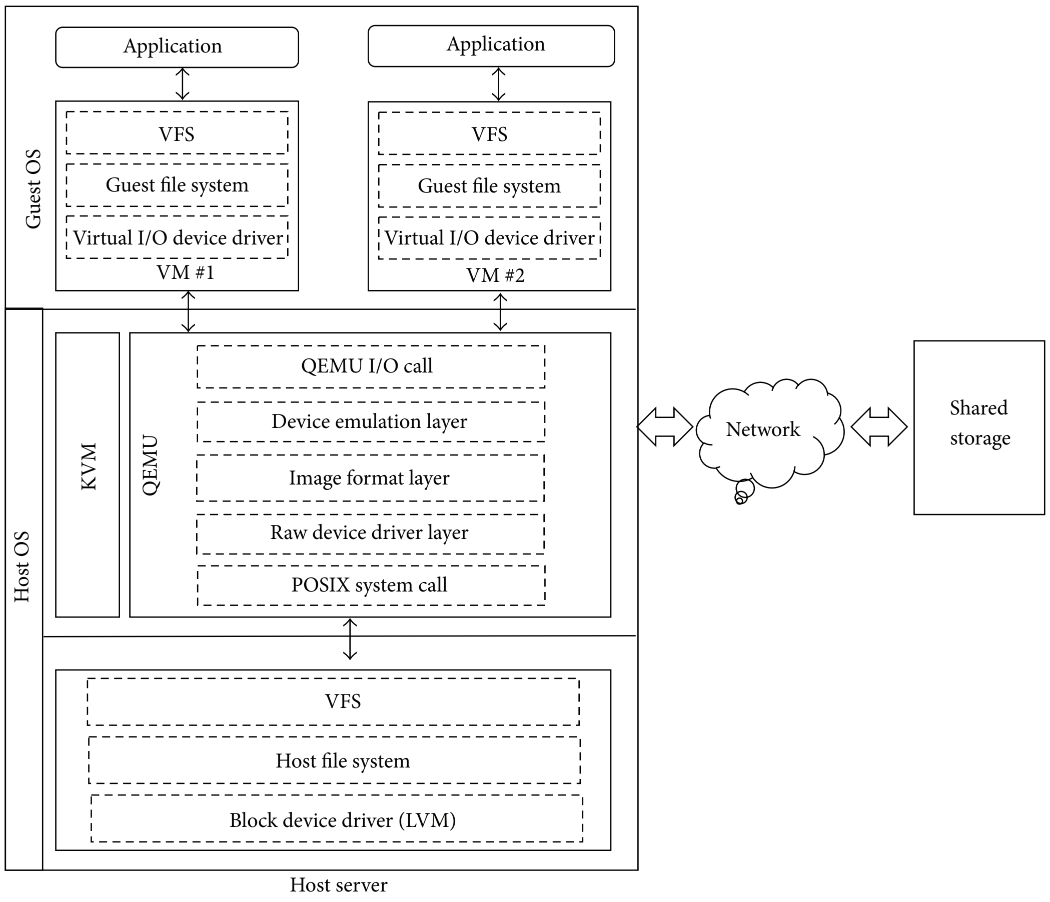

Figure 1: I/O virtualization path in KVM/QEMU.

appropriate cache mechanism in the guest and KVM using QEMU emulator [8–10].

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. 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 nonroot mode and the root mode, can substantially deteriorate application executions. Also, the OS dependency makes it difficult to port the guest-level cache method across VMs, especially in the case where 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 back end, shared storage, and the guest. Also, caching on the guest level can give the better chance to retain the application-specific data. This is because while the guest needs to consider only the applications running on top of it, the hypervisor should control all the data necessary for VMs on the same host, which can cause the cache miss for the desired data due to the limited cache size or the swapping activity. Finally, by tightly coupling with the light-weight resource monitoring module, the component can manage the effective cache size in the guest.

The hypervisor-level component of MultiCache attempts to reduce I/O latency by supplying the desired data in QEMU 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. The hypervisor mainly uses the hypercall to transit process control from the guest operation system to the hypervisor itself. Because such a transition requires the mode switching between nonroot mode and root mode, the application process on the guest should remain blocked, lagging the I/O performance behind. The hypervisor-level component tries to optimize such an overhead by providing the necessary data in QEMU.

This paper is organized as follows. In Section 2, we discuss the related studies and, in Section 3, we describe the overall structure of MultiCache. In Section 4, we present the performance measurement and, in Section 5, we conclude with a summary.

2. Related Studies

Reducing I/O virtualization cost is the critical issue to accelerate I/O bandwidth of virtual machines. There have been several researches targeting I/O virtualization overhead. First of all, most VDI schemes use the back end and shared storage as a persistent data reservoir, such as DAS (host's direct attached storage), NAS (network-attached storage), or SAN (storage area network) [6, 11]. This storage is used to store read-only image templates or shared libraries and files for VMs. As the virtual machine has gained a widespread use in the cloud computing, managing the optimal cost for transferring the image contents and files between the storage and the host is becoming the essential research aspect. For example, Tang [6] proposed FVD (Fast Virtual Disk) consisting of VM image formats and the block device driver for QEMU. FVD enables supporting the instant VM creation and migration on the host by using copy-on-write, copy-on-read, and adaptive fetching.

As the technology of SSD (Solid State Disk) is rapidly growing, there have been several attempts to boost I/O bandwidth by adopting SSD in the virtualized environment [12–15]. For example, in vCacheShare [12], instead of proportionally allocating the flash cache space on the shared storage, vCacheShare uses the information about I/O accesses from VMs and trace processing data to extract reuse patterns in order to calculate the appropriate flash cache size. Mercury [13] is the client-server, write-through based flash cache method in the hypervisor. Byan et al. [13] argued that placing the flash cache either in the networked storage server or in VM may not be beneficial for speeding up I/O performance due to the network latency or VM migration [16, 17]. Also, utilizing flash cache with the write-back policy might not satisfy high I/O demand, because every write should still be written to the shared storage via a network hop for data consistency and availability.

S-CAVE [14] is a hypervisor-level flash cache to allocate the cache space among VMs. Similar to vCacheShare, S-CAVE monitors I/O activities of VMs at runtime and uses them to determine their cache space demand. Arteaga et al. [15] proposed a flash cache on the client-side storage system (VM host). They used dm-cache [18] block-level cache interface in their method and also argued that write-back policy is beneficial in the cloud environment. Razavi and Kielmann [19] tried to reduce the network overhead to be occurring during VM startup time, by placing VM cache either on the compute node or on the storage memory. They found that when the cloud environment supports a master image to be shared among multiple VMs, caching VM images on the compute node would efficiently reduce network traffics. Also, with the cloud environment, where many compute nodes simultaneously use multiple VMs, placing VM cache image to the storage memory can help reduce the disk queuing delay.

Besides between the shared storage and the host, the I/O latency taking place in the hypervisor should also be addressed to achieve the desirable system bandwidth in the virtualization environment. One of such overheads is VM exit. The I/O requests issued in VMs are asynchronously handled by the host while passing through the hypervisor and the emulator such as QEMU. Since VMs run on the nonroot mode and the hypervisor runs on the root mode, servicing I/O requests causes exiting VM first to go to the hypervisor, which incurs the context switching overhead. Also, the replies from the hypervisor to VMs adversely affect I/O performance. Since the application which issued those I/O requests remains blocked on VM, such a switching overhead can eventually slow down the application execution.

There are several researches on this issue. For example, SR-IOV [20, 21] was implemented to obtain the benefits of direct I/O on physical devices, by defining extensions to the PCIe specification. In SR-IOV, VDD running in the guest either is connected to VF executing on the shared sources for direct data movement or forwards the request to dom 0 where PF driver manages and coordinates the direct accesses to the shared resources for VFs. Yassour et al. [22] proposed a device assignment, where VM can access physical I/O resources directly, without passing through the host emulation software.

However, the direct device assignment cannot work for virtual resources such as virtual disk, losing the strength of virtualization flexibility. To overcome such a drawback, Har'El et al. [23] proposed a new form of paravirtual I/O, which tried to overcome the weakness of the existing paravirtual I/O scheme [4, 24, 25]. Their I/O scheme attempts to alleviate I/O overhead by providing the dedicated I/O core controlled by a single I/O thread. Instead of mixing I/O and guest workloads in the same core, using a dedicated I/O not only can assign more cycles to guests but also can improve overall system efficiency by reducing the context switching cost.

The other issue of the I/O virtualization overhead is that I/O requests should go through a thick I/O stack to complete. In the case of KVM using QEMU, the typical way of writing data in the guest is that, after passing the file system and device driver layer of the guest kernel, the data necessary for the write should be transferred to the emulated device driver in the hypervisor. Also, the data enters the host kernel that has the similar software structure to the guest kernel (assuming the guest and host run the same OS) and reaches the physical I/O device attached to the host. Appropriately placing cache is a way of reducing such traffics in the virtualization environment [26, 27].

Capo [27] uses local disks as a persistent cache. Shamma et al. insisted that the majority of requests on VMs are redundant and can be served by local disk. In order to justify their argument, they first traced a production VDI workload and found that caching below the individual VMs is effective to improve I/O performance. Capo was integrated with XenServer [28], by putting it into domain 0. Also. Gupta et al. [29] studied the page sharing and memory compression to save the memory consumption of VMs. Their difference engine method searches for the identical pages by using the hashing function. If pages have the same value, then it reclaims the pages and updates the virtual memory to point out the shared copy. Detecting the page sharing in their method goes further by eliminating the subpage sharing using page patching and by adapting in-core memory compression.

Ongaro et al. [30] studied the impact of Xen scheduling policy on I/O bandwidth with several applications showing the different performance characteristics. They found that

Xen's credit scheduler does not lower the response latency in the situation where several domains are concurrently performing I/O, even with BOOST state. One of the reasons is that the event channel driver always scans the pending vector from the beginning, instead of resuming from where it left. Also, they found the possibility of priority inversion of which delivering the highest-priority packet is postponed by preemption. Lu and Shen [31] traced the page miss ratio of VMs, by employing the hypervisor-level exclusive cache. They captured the pages evicted from VM memory into the hypervisor exclusive cache, while avoiding containing the same data in VM and exclusive cache. Jones et al. [32] also proposed a way of inferring promoting and evicting pages of buffer cache in the virtual memory. In order to correctly infer page cache activities, they observed some sensitive events causing control to be transferred to VMM, such as page faults, page table updates, and disk I/Os.

However, optimizing either in the guest or in the hypervisor might not be enough to produce the desirable performance because I/O path in the virtualization involves several software layers including the shared storage to guest and the guest to host. In this paper we attempted to target both layers by implementing the guest-level component and the hypervisor-level component.

3. MultiCache

3.1. System Structure. MultiCache was implemented to exploit I/O optimizations targeting multiple layers of I/O virtualization stack. Figure 2 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 consists 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 behaviour 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.

The third component of MultiCache is the real-time resource monitoring component. The monitoring module works at the hypervisor independently of guest operating systems, collecting the resource usage information from all VMs and the host server. The monitoring information is used by both components of MultiCache to effectively perform I/O optimization schemes. There are two tables associated with the monitoring component: VM resource table for storing VM resource usages and host resource table for host resource usages.

3.2. Differences between Two Components of MultiCache. There are four differences between the guest-level component and the hypervisor-level component of MultiCache. First, the main goal of the guest-level component is to mitigate I/O overhead between the shared storage and the guest VM, by prefetching and retaining files that will likely be used in the near future. On the other hand, the hypervisor-level component is to minimize I/O overhead between the guest VM and the host, by cutting down I/O software stack inside QEMU.

Second, two components of MultiCache use the different I/O unit: files in the guest-level component and sectors in the hypervisor-level component. While the guest-level component uses files for I/O optimization, the hypervisor-level component uses sectors that have been divided from files in the guest kernel before arriving at QEMU I/O call.

Third, to mitigate I/O overhead, the guest-level component utilizes the usage count that indicates how many times files have been referenced after they were brought into the guest. By caching the files that have high usage counts, the component attempts to reduce the network and I/O overheads between the shared storage and the guest VM. Also, this information is used to reduce I/O accesses from the host. The hypervisor-level component utilizes the I/O access frequency that implies how often sectors have been accessed from the host. Instead of forwarding sectors having frequently been used to the host, the hypervisor-level component caches those sectors in memory to reduce application process block time and I/O contention on the host.

Finally, while the guest-level component reserves the cache memory in the guest VM, the hypervisor-level component reserves the cache memory in the hypervisor, which is managed independently of guest operation systems. Table 1 illustrates the brief description about the differences between two MultiCache components.

3.3. 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,

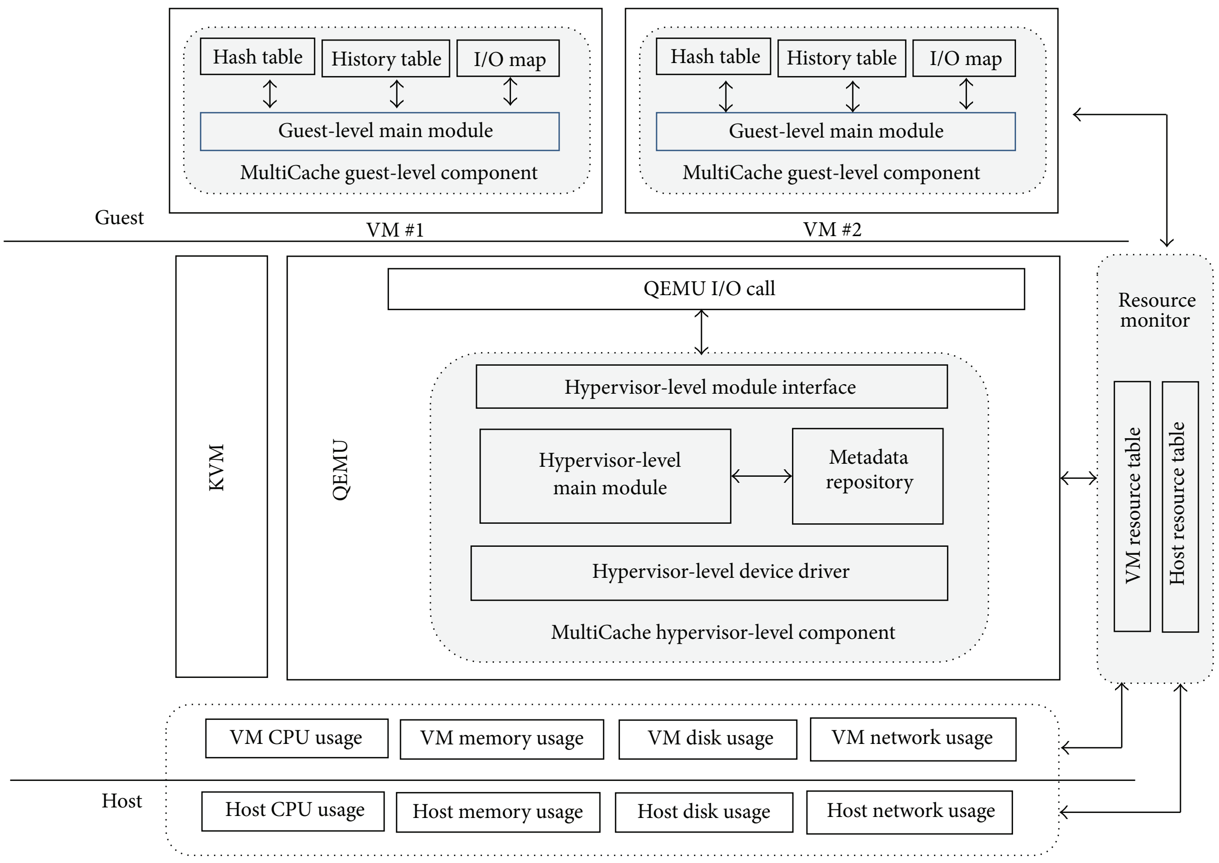

Figure 2: MultiCache structure.

Table 1: Difference between two MultiCache components.

	Guest-level component	Hypervisor-level component
Objective	To minimize the overhead between the shared storage and the guest VM	To minimize the overhead between the guest VM and the host
I/O unit	File	Sector
Optimization hint	File usage count	Sector I/O access frequency
Cache memory	Placed in the guest VM	Placed in the hypervisor
OS dependency	Yes	No

MultiCache enables providing 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.

Figure 3 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. Separately maintaining read and write maps offers two benefits. First, it enables cashing more files showing frequent read executions in order to support the better chance for the fast read responsiveness. Second, it can contribute to enhancing data reliability and availability by flushing out more dirty files at the replacement phase. Besides, the I/O map enables maintaining files in the guest according to their frequency and recentness to reduce I/O accesses to the host.

In Figure 3, 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

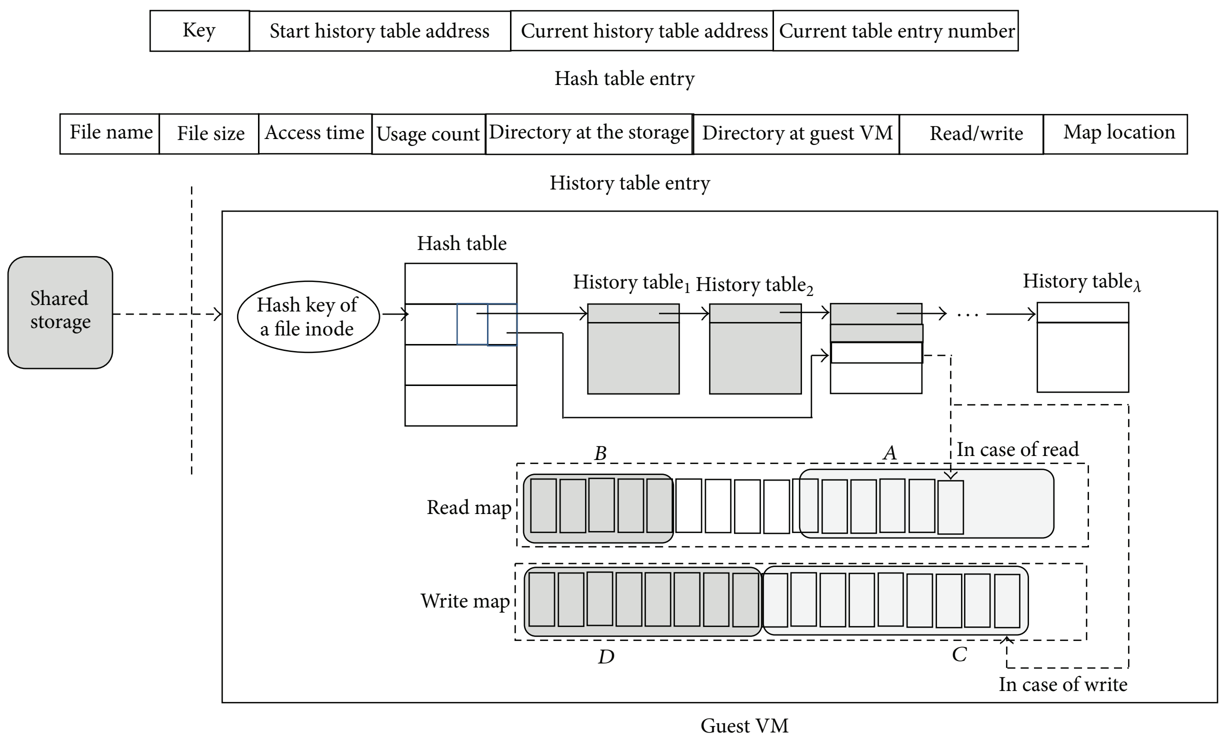

FIGURE 3: MultiCache guest-level component.

by caching more files whose most recent I/O accesses are read operations. Such a process involves replacing less files mapped in section B. Similarly, MultiCache can replace more dirty files mapped to section D for data reliability and availability. Let M_g be the guest-level cache memory size and let *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:

$$\frac{\left\{\sum_{f_a\in A}\text{size}\left(f_a\right)+\sum_{f_b\in B}\text{size}\left(f_b\right)+\sum_{g_c\in C}\text{size}\left(g_c\right)+\sum_{g_d\in D}\text{size}\left(g_d\right)\right\}}{M_g}\leq MEM\_THRESHOLD. \quad (1)$$

Algorithm 1 shows the steps involved in the guest-level component of MultiCache. Let i and k be the most recent positions of the read and write maps, respectively. Also, let f be the file for read and let g be the file for write.

In steps (1) and (2), MultiCache calculates the hash keys of f and g to access their file metadata from two tables. Also, the usage counts of two files are increased. In steps (4) to (17), if the usage count is larger than or equal to the threshold, then the metadata of f and g are inserted into the read and write maps, respectively, to store the associated data to MultiCache. In particular, if the last access of f was write, then the metadata is migrated from the write map to the read map, while erasing its history from the write map. The same procedure is applied for g to save its metadata to the write map. Steps (18) to (24) describe the procedure to maintain the appropriate cache size by taking into account condition (1). In the case that the condition is not satisfied, files mapped to sections B and D are flushed out to eliminate the memory pressure.

3.4. 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 nonroot mode and root mode, which incurs the application execution being blocked. Furthermore, because those requests require accessing 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.

MultiCache hypervisor-level component uses several tables, called the metadata repository, to maintain I/O-related metadata at the hypervisor. Figure 4 shows the tables

```
(1)  calculate the hash keys of $f$ and $g$ to retrieve their file metadata from the hash and history tables;
(2)  **if** (not found) **then** insert their metadata to the hash table and the history table **end if**
(3)  increase the usage counts of $f$ and $g$ by one;
(4)  **if** (the usage count of $f \geq$ USAGE_THRESHOLD) **then**
(5)      **if** ($f \in$ read map and its position in read map is $a$ where $a < i$) **then**
(6)          $i$++; move the metadata of $f$ from $a$th position to $i$th position of read map;
(7)      **else if** ($f \in$ write map) **then**
(8)          $i$++; move the metadata of $f$ to $i$th position of read map; delete it from write map;
(9)      **else** $i$++; insert the metadata of $f$ to $i$th position of read map **end if**
(10) **end if**
(11) **if** (the usage count of $g \geq$ USAGE_THRESHOLD) **then**
(12)     **if** ($g \in$ write map and its position in write map is $b$ where $b < k$) **then**
(13)         $k$++; move the metadata of $g$ from $b$th position to $k$th position of write map;
(14)     **else if** ($g \in$ read map) **then**
(15)         $k$++; move the metadata of $g$ to $k$th position of write map; delete it from read map;
(16)     **else** $k$++; insert the metadata of $g$ to $k$th position of write map **end if**
(17) **end if**
(18) **for each time epoch**
(19)     receive the current guest VM memory size from the resource monitor;
(20)     check the condition specified in (1);
(21)     **if** (the condition is not satisfied) **then**
(22)         flush out files depicted in the section $B$ or $D$ until memory pressure is eliminated;
(23)     **end if**
(24) **end for**
```

Algorithm 1: MultiCache guest-level component.

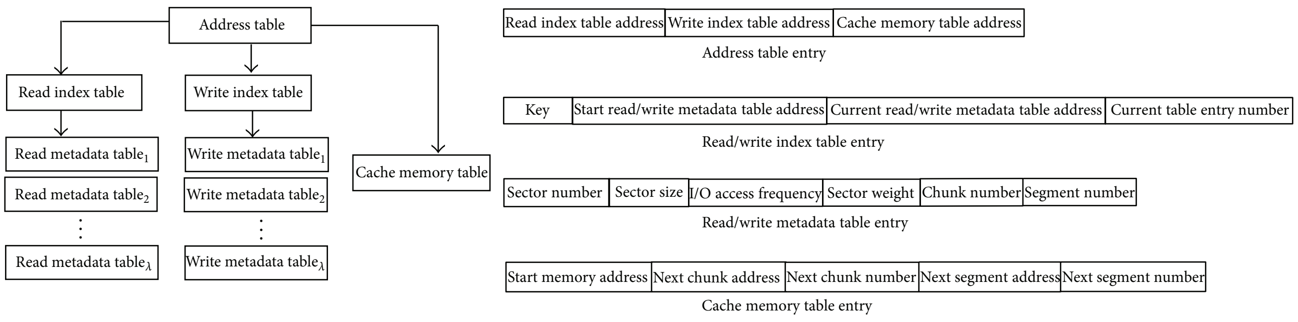
Address table
Read index table
Write index table
Read metadata table$_1$
Read metadata table$_2$
Read metadata table$_\lambda$
Write metadata table$_1$
Write metadata table$_2$
Write metadata table$_\lambda$
Cache memory table
Read index table address | Write index table address | Cache memory table address
Address table entry
Key | Start read/write metadata table address | Current read/write metadata table address | Current table entry number
Read/write index table entry
Sector number | Sector size | I/O access frequency | Sector weight | Chunk number | Segment number
Read/write metadata table entry
Start memory address | Next chunk address | Next chunk number | Next segment address | Next segment number
Cache memory table entry

Figure 4: Metadata repository of the hypervisor-level component.

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 criterion is needed to filter sectors before storing them in the cache memory. In MultiCache, only those sectors that 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)  apply the hash function to obtain a hash key using sc;
(2)  access the read index table with the hash key to retrieve the metadata of sc from the read metadata table;
(3)  if no metadata about sc is available in the read index table then
(4)     store it in the read index table and the current read metadata table;
(5)     update the read index table to point out the next entry of the current read metadata table;
(6)  end if
(7)  $p_{sc}$++; $w_{sc} = p_{sc} \times$ the weight of guest VM;
(8)  if ($w_{sc}$ < FREQ_THRESHOLD) then $\delta_{sc} = 0$; exit to access sc from the host end if
(9)  if sc has not been mapped to the cache memory then
(10)    $\delta_{sc} = 0$;
(11)    map sc to the cache, by retrieving the chunk and segment numbers from the cache memory table;
(12) else
(13)    $\delta_{sc} = 1$;
(14)    access the cache memory with the chunk and segment numbers of sc retrieved;
(15) end if
```

Algorithm 2: MultiCache hypervisor-level component.

(1) p_{sc} is the I/O access frequency of sc.

(2) w_{sc} is the weight of sc satisfying $w_{sc} = p_{sc} \times (u_i/\sum_{k=1}^{N} u_k) \times (v_i/\sum_{k=1}^{N} v_k)$, where $(u_i/\sum_{k=1}^{N} u_k) \times (v_i/\sum_{k=1}^{N} v_k)$ is the weight of VM(i).

(3) δ_{sc} is the mapping function, indicating either cache hit ($\delta_{sc} = 1$) or miss ($\delta_{sc} = 0$).

(4) m_{sc} is the position of the cache memory, where sc is stored if $w_{sc} \geq$ *FREQ_THRESHOLD*.

Algorithm 2 represents the steps for reading sc at the hypervisor-level component of MultiCache.

Suppose that sc is one of the sectors consisting of a read request in the guest. MultiCache calculates a hash key to access the read index table containing the corresponding read metadata table address. After retrieving the associated metadata from the table, the I/O access frequency is multiplied by the VM weight to obtain the weight of sc. In the case where the weight of sc is less than *FREQ_THRESHOLD*, MultiCache passes sc to the host kernel to access it from the physical I/O device. Otherwise, from step (9) to step (15), MultiCache checks to see if sc has been stored in the cache memory. If not, sc is stored in the memory by using the chunk and segment numbers retrieved from the cache memory table. In the case where sc is found in the cache memory, it returns to the guest without going down to the host kernel. In the write operation, after updating the associated metadata to the write index table and the write metadata table, the sector is mapped to the cache table. If the associated metadata is available in the table, then the sector having been mapped in the cache memory is overwritten to update.

The cache memory handled by the hypervisor-level component is partitioned into chunks that consisted of a number of pages. In case of the write cache memory, the sectors stored in the chunk are transmitted to the host kernel, either after the chunk is filled with valid sectors or when the current checkpoint (currently every 30 seconds) for the chunk comes. Let M_h be the size of the cache memory; let C_i be the ith chunk; and let $|C_i|$ be the size of C_i. Also, let seg_k be the kth segment of C_i whose size, $|\text{seg}_k|$, is the same as that of a sector. The chunk validity and segment validity are determined by the chunk map and the segment map, respectively.

Definition 2. The allocation status of C_i in the chunk map and the one of seg_k of C_i in the segment map are defined as follows:

For any chunk C_i, bit: $C[i] \rightarrow \{0, 1\}$, $1 \leq i \leq M_h/|C_i|$.

For any segment $\text{seg}_k \in C_i$, bit: $\text{seg}[i, k] \rightarrow \{0, 1\}$, $1 \leq k \leq |C_i|/|\text{seg}_k|$.

If $\text{bit}(\text{seg}[i, k]) = 1$, then the segment contains a valid sector that should be transferred to the host. Otherwise, $\text{bit}(\text{seg}[i, k]) = 0$. Also, $\text{bit}(C[i]) = 1$ implies that all the segments consisting of C_i contain the valid sectors. Algorithm 3 shows the steps involved in the write process for the cache memory.

3.5. MultiCache Resource Monitor.

The resource monitor calculates the resource statuses of guest VMs and host server at the hypervisor level because it should monitor the usage information independently of guest operating systems. Also, it is organized with the light-weight modules so that it rarely affects I/O bandwidth on VMs. During application executions, the monitor periodically notifies the resource usage information to the guest-level and hypervisor-level components to help them maintain the effective cache capacity for I/O improvement.

The resource monitor is composed of three modules: resource collection module, resource calculation module, and usage container. The resource collection module works on top of the server, while communicating with *proc* file system and *libvirt* to collect the resource status information such as CPU, memory, disk I/O, and network status. The resource calculation module calculates resource usages and, finally, the usage container stores the calculated information to offer it to both components of MultiCache.

Figure 5 represents the functions to be called in the resource monitor. To activate the monitor, the resource

```
(1)  for each chunk $C_i$
(2)     if (bit($C[i]$) = 1) then transfer $C_i$ to the host; bit($C[i]$) = 0 end if
(3)  end for
(4)  /* let sc be the current sector to be stored in the cache memory */
(5)  /* let $i$ and $k$ be the chunk and segment numbers, respectively, selected from the cache memory table */
(6)  store sc to $\mathrm{seg}_k$ of $C_i$; bit(seg[$i, k$]) = 1; update bit($C[i]$) while reflecting bit(seg[$i, k$]) value;
(7)  $k$++;
(8)  if (($k < |C_i|/|\mathrm{seg}_k|$) and bit(seg[$i, k$]) = 0) then
(9)     store $i$ and $k$ to the cache memory table as the next position in the cache memory; exit;
(10) end if
(11) /* find the next empty chunk in the cache memory */
(12) search for $C_i$ such that ($i = (i + 1)\%(M_h/|C_i|)$ and bit($C[i]$) = 0);
(13) store $i$ and 0 to the cache memory table as the next position in the cache memory;
```

Algorithm 3: Cache memory management.

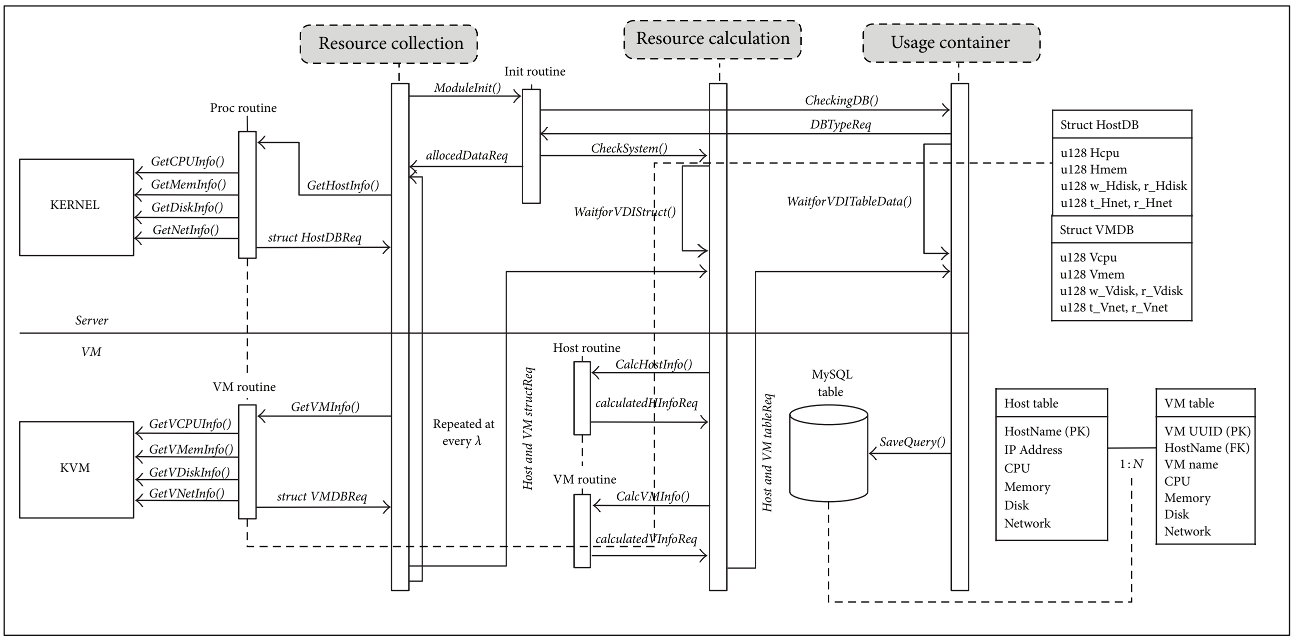

Figure 5: Resource monitor structure in MultiCache.

collection first initializes the functions to be called in the resource calculation and the usage container, by issuing *ModuleInit*( ). Also, it communicates with /*proc* and *libvirt* at every time period λ by calling *GetXXXInfo*( ) and accumulates the resource status information to store it to the usage container. The resource calculation module retrieves the status information, by calling *calcHostInfo* and *calcVMInfo*( ), and calculates the resource usages by applying the formulas described in Table 2. Finally, the results are stored in the usage container.

4. Performance Evaluation

4.1. Experimental Platform. We executed all experiments on a host server equipped with an AMD FX 8350 eight-core processors, 24 GB of memory, and 1 TB 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 1 Gbit 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. Each VM was configured with two-core processors, 8 GB of memory, and 50 GB of virtual disk using virtIO. The operating system of each VM was CentOS release 6.5 with 2.6.32-431 kernel. We used postmark benchmark for the evaluation.

4.2. MultiCache Guest-Level Component Evaluation. 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

Table 2: Formulas and data structures for calculating resource usages.

Resource usage formula	Resource monitor data structure

$\text{CPU\%} = 100 - \frac{100}{\text{total}} \times (\text{idle}_{\text{now}} - \text{idle}_t)$
(i) Total: total CPU usage
(ii) idle_{now}, idle_t: idle values at the moment and at t
	struct S_ProcCpuInfo {
	_u128 user, nice, system, idle, iowait;
	_u128 irq, softirq, steal, guest;
	}

$\text{Memory usage} = 100 \times \frac{\text{total} - \text{free}}{\text{total}}$
(i) Total, free: total and free memory sizes, respectively
	struct S_ProcMemInfo {
	_u128 total, free;
	_u128 buffers, cached;
	}

$\text{Disk usage} = (\text{sectr}_{\text{now}} - \text{sectr}_t) \times 512$
(i) $\text{sectr}_{\text{now}}$, sectr_t: sector usages up to now and at t, respectively
	struct S_ProcDiskInfo {
	char disk_name[20];
	_u128 r_compl, r_merge, r_sectr, r_milsc;
	_u128 w_compl, w_merge;
	_u128 w_sectr, w_milsc;
	_u128 io_c_prc, io_milsc, io_w_milsc;
	}

Network usage
$\text{PPS} = \text{packet}_{\text{now}} - \text{packet}_t$
$\text{PacketSize} = \frac{(\text{byte}_{\text{now}} - \text{byte}_t) \times 8}{\text{PPS}} + 12 + 7 + 1$
$\text{BPS} = \text{PPS} \times \text{PacketSize}$
(i) PPS: number of packets transmitted per second
(ii) BPS: network bandwidth in bit per second
	struct S_ProcNetInfo {
	char net_name[20];
	char net_hwaddr[20];
	_u128 r_byte, r_pack, r_err, r_drop;
	_u128 r_fifo, r_frm, r_cmp, r_mult;
	_u128 t_byte, t_pack, t_err, t_drop;
	_u128 t_fifo, t_col, t_cal, t_cmp;
	}

MultiCache (*cached*) and the other files not residing in MultiCache are read from the storage through NFS (*not cached*).

Figure 6 shows I/O bandwidth while varying file sizes from 4 KB to 1 MB. 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 1 MB of files as compared to that of 4 KB 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.

In the evaluations, we observed that the effect of MultiCache is especially obvious with read operations, as shown in Figure 7. 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 Figure 7, 80 : 20 means that 80% of transactions are read operations and 20% of transactions are writes. Also, we used 1 MB of file size. Figure 7 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.

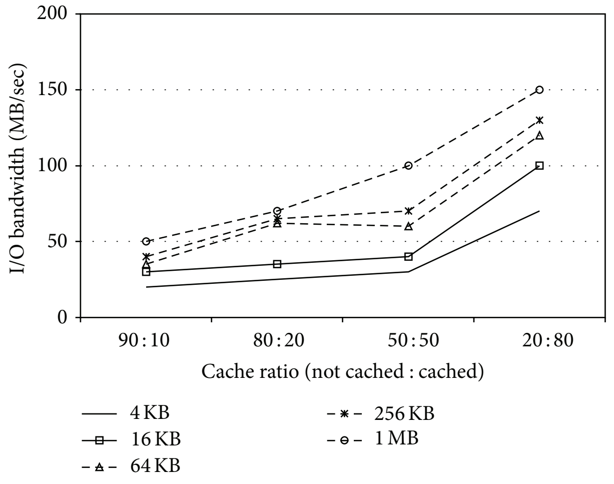

Figure 6: I/O bandwidth of virtCache.

However, I/O latency between the shared storage and the VM is not the only one that should be addressed to achieve the desirable performance. As mentioned, the I/O path from the guest to the host should also be scrutinized because there are multiple places causing the performance slowdown, such as I/O contention to physical devices and mode transition between the guest and the hypervisor. We will observe how the hypervisor-level component of MultiCache can achieve better bandwidth by overcoming such latencies.

4.3. I/O Bandwidth of Hypervisor-Level Component. We measured the I/O performance of the hypervisor-level component. In this experiment, there is no file transmission between

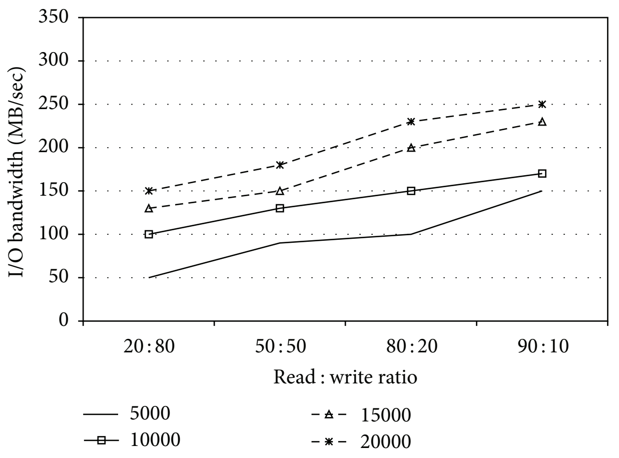

Figure 7: Performance evaluation based on I/O accesses.

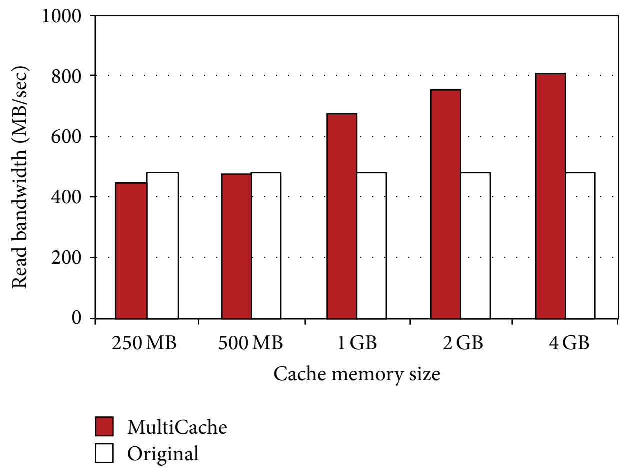

Figure 9: Read bandwidth based on the cache memory.

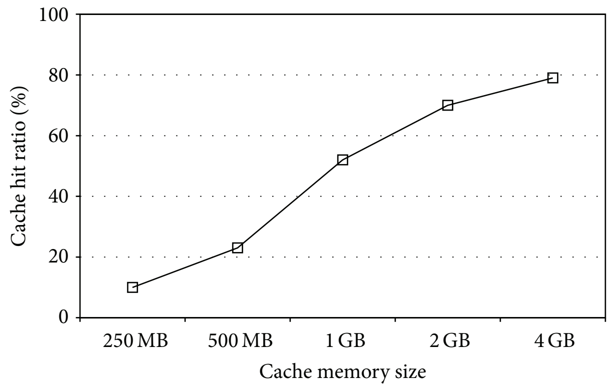

Figure 8: The effect of the cache memory.

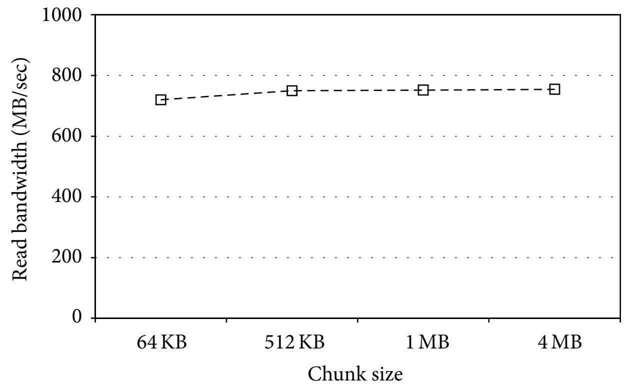

Figure 10: The effect of the chunk size on read.

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 4 KB and 1 MB.

First of all, we observed the effect of the hypervisor cache memory in Figure 8, while changing the cache memory size from 250 MB to 4 GB. To warm the cache, we executed the modified postmark for 5 seconds and took the average value of each test case. Figure 8 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 250 MB to 4 GB 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 500 MB to 1 GB, the hit ratio from 1 GB to 2 GB increases 34%. Extending the cache memory from 2 GB to 4 GB 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.

Figure 9 shows the I/O bandwidth obtained while varying the cache memory size from 250 MB to 4 GB. We can notice that Figure 9 depicts the similar performance pattern to that of Figure 8: the larger cache memory size is, the better I/O bandwidth is. Also, while increasing the cache memory size from 500 MB to 1 GB shows about 43% of bandwidth speedup, the cache memory extension from 1 GB to 2 GB produces only 12% of performance improvement.

In Figure 9, we compared the I/O performance of Multi-Cache to that of the original KVM/QEMU. With the small cache memory size such as 250 MB, the I/O bandwidth of MultiCache is less than that of the original version because of the cache miss incurred by space restriction. However, the performance difference becomes large as the memory size of MultiCache increases. With 2 GB of cache memory size, MultiCache produces about 37% of I/O bandwidth improvement compared to that of the original version. We currently use 2 GB of cache memory size. The RAM size of the host is 24 GB; therefore we use only about 8% of the total size as the cache memory.

Figure 10 shows the read results while changing the chunk size from 64 KB to 4 MB in the cache memory. As can be seen in the figure, changing the chunk size does little affect I/O bandwidth in the read operation because no write occurred to the host.

Figure 11 shows the write bandwidths of MultiCache while comparing to those of the original KVM/QEMU. The original version supports three write modes: default, write-through,

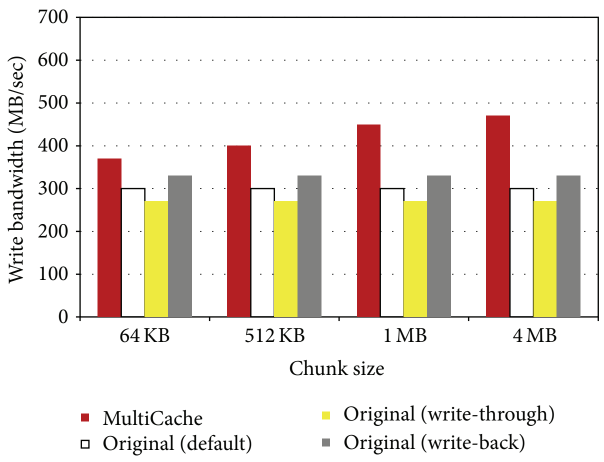

Figure 11: Write bandwidth based on the chunk size.

and write-back. In the case of using 1 MB of chunk size in MultiCache, it generates about 33% and 27% higher bandwidth than the default mode and the write-back mode of the original version, respectively. There are two reasons to explain such I/O bandwidth improvements. In the case of write-back mode of the original version, it buffers the data for I/O in the host kernel so that I/O requests issued in the guest should go through the guest kernel and QEMU before arriving at the host. Second, instead of flushing the data out to the physical device, the hypervisor-level component intercepts them in QEMU and collects in the cache memory in a big I/O unit. Such a method can contribute to accelerating I/O bandwidth, because, in Figure 11, we can notice that as the chunk size increases, the write performance also becomes large. However, based on the result with 4 MB of chunk size, increasing the size more than 1 MB might not produce the significant performance speedup due to the write latency in the host.

5. Conclusion

We proposed a multilayered 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 back end, shared storage, and the guest. Also, caching the application-specific data in the guest can contribute to accelerating the performance speedup. In order 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. In the component, we accumulated the I/O access information about application executions in the metadata repository and used it to retain data with high I/O access frequency in the cache memory. Both components of MultiCache were integrated with the real-time resource monitoring module collecting the resource usage information of VMs and host at the hypervisor. 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.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2014R1A2A2A01002614).

References

[1] R. Spruijt, “VDI Smackdown,” White Paper, v.1.4, 2012.

[2] J. Hwang and T. Wood, “Adaptive dynamic priority scheduling for virtual desktop infrastructures,” in *Proceedings of the IEEE 20th International Workshop on Quality of Service (IWQoS ’12)*, pp. 1–9, IEEE, Coimbra, Portugal, June 2012.

[3] D.-A. Dasilva, L. Liu, N. Bessis, and Y. Zhan, “Enabling green IT through building a virtual desktop infrastructure,” in *Proceedings of the 8th International Conference on Semantics, Knowledge and Grids (SKG ’12)*, pp. 32–38, Beijing, China, October 2012.

[4] J. Santos, Y. Turner, G. Janakiraman, and I. Pratt, “Bridging the Gap between Software and Hardware Techniques for I/O Virtualization,” in *Proceedings of the USENIX Annual Technical Conference*, Boston, Mass, USA, 2008.

[5] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, “Towards high-quality I/O virtualization,” in *Proceedings of the Israeli Experimental Systems Conference (SYSTOR ’09)*, article 12, May 2009.

[6] C. Tang, “FVD: a high-performance virtual machine image format for cloud,” in *Proceedings of the USENIX Annual Technical Conference*, Portland, Ore, USA, June 2011.

[7] D. Le, H. Huang, and H. Wang, “Understanding performance implications of nested file systems in a virtualized environment,” in *Proceedings of the 10th USENX Conference on File and Storage Technologies (FAST ’12)*, San Jose, Calif, USA, February 2012.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “KVM: the Linux virtual machine monitor,” in *Proceedings of the Ottawa Linux Symposium (OLS ’07)*, pp. 225–230, July 2007.

[9] R. Russell, “Virtio: towards a De-Facto standard for virtual I/O devices,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[10] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Anaheim, Calif, USA, April 2005.

[11] V. Tarasov, D. Hidebrand, G. Kuenning, and E. Zadok, “Virtual machine workloads: the case for new benchmarks for NAS,” in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST ’13)*, pp. 307–320, Santa Clara, Calif, USA, 2013.

[12] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vCacheShare: automated server flash cache space management in a virtualization environment," in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC '14)*, pp. 133–144, Philadelphia, Pa, USA, June 2014.

[13] S. Byan, J. Lentini, A. Madan et al., "Mercury: host-side flash caching for the data center," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*, pp. 1–12, IEEE, San Diego, Calif, USA, April 2012.

[14] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-CAVE: effective SSD caching to improve virtual machine storage performance," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*, pp. 103–112, Edinburgh, UK, September 2013.

[15] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *Proceedings of the 7th ACM International Systems and Storage Conference (SYSTOR '14)*, Haifa, Israel, June 2014.

[16] H. Jin, W. Gao, S. Wu, X. Shi, X. Wu, and F. Zhou, "Optimizing the live migration of virtual machine by CPU scheduling," *Journal of Network and Computer Applications*, vol. 34, no. 4, pp. 1088–1096, 2011.

[17] T. C. Ferreto, M. A. S. Netto, R. N. Calheiros, and C. A. F. De Rose, "Server consolidation with migration control for virtualized data centers," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1027–1034, 2011.

[18] dm-cache, http://visa.lab.asu.edu/dmcache.

[19] K. Razavi and T. Kielmann, "Scalable virtual machine deployment using VM image caches," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*, Denver, Colo, USA, November 2013.

[20] S. Bhosale, A. Caldeira, B. Grabowski et al., *IBM Power Systems SR-IOV*, IBM Redpaper, IBM, 2014.

[21] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[22] B. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," Tech. Rep. H-0263, IBM Research, 2008.

[23] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual I/O system," in *Proceedings of the USENIX Annual Technical Conference*, pp. 231–242, San Jose, Calif, USA, 2013.

[24] A. Menon, A. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *Proceedings of the USENIX Annual Technical Conference*, Boston, Mass, USA, 2006.

[25] P. Barham, B. Dragovic, K. Fraser et al., "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[26] H. Kim, H. Jo, and J. Lee, "XHive: efficient cooperative caching for virtual machines," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 106–119, 2011.

[27] M. Shamma, D. Meyer, J. Wires, M. Ivanova, N. Hutchinson, and A. Warfield, "Capo: recapitulating storage for virtual desktops," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, San Jose, Calif, USA, February 2011.

[28] C.-H. Hong, Y.-P. Kim, S. Yoo, C.-Y. Lee, and C. Yoo, "Cache-aware virtual machine scheduling on multi-core architecture," *IEICE Transactions on Information and Systems*, vol. E95-D, no. 10, pp. 2377–2392, 2012.

[29] D. Gupta, S. Lee, M. Vrable et al., "Difference engine: harnessing memory redundancy in virtual machines," *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[30] D. Ongaro, A. L. Cox, and S. Rixner, "Scheduling I/O in virtual machine monitors," in *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pp. 1–10, Seattle, Wash, USA, March 2008.

[31] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," in *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, Calif, USA, June 2007.

[32] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," *ACM SIGPLAN Notices*, vol. 40, no. 5, pp. 14–24, 2006.