ACStor: Optimizing Access Performance of Virtual Disk Images in Clouds

Song Wu, *Member, IEEE*, Yihong Wang, Wei Luo, Sheng Di, *Member, IEEE*, Haibao Chen, Xiaolin Xu, Ran Zheng, *Member, IEEE*, and Hai Jin, *Senior Member, IEEE*

Abstract—In virtualized data centers, *virtual disk images* (VDIs) serve as the containers in virtual environment, so their access performance is critical for the overall system performance. Some distributed VDI chunk storage systems have been proposed in order to alleviate the I/O bottleneck for VM management. As the system scales up to a large number of running VMs, however, the overall network traffic would become unbalanced with hot spots on some VMs inevitably, leading to I/O performance degradation when accessing the VMs. In this paper, we propose an adaptive and collaborative VDI storage system (ACStor) to resolve the above performance issue. In comparison with the existing research, our solution is able to dynamically balance the traffic workloads in accessing VDI chunks, based on the run-time network state. Specifically, compute nodes with lightly loaded traffic will be adaptively assigned more chunk access requests from remote VMs and vice versa, which can effectively eliminate the above problem and thus improves the I/O performance of VMs. We implement a prototype based on our ACStor design, and evaluate it by various benchmarks on a real cluster with 32 nodes and a simulated platform with 256 nodes. Experiments show that under different network traffic patterns of data centers, our solution achieves up to $2-8\times$ performance gain on VM booting time and VM's I/O throughput, in comparison with the other state-of-the-art approaches.

Index Terms—Virtual disk image (VDI), collaborative, uneven traffic, adaptive

1 Introduction

Infrastructure-as-a-Service (IaaS) allows users to deploy customized *virtual machines* (VMs) in a data center based on their specific resource demands. *Virtual disk images* (VDIs) (or the containers of VMs) play a critical role in the management of VMs, especially in the deployment phase and the VM running period. In general, a cloud data center adopts a centralized storage system to store and manage VDIs for simplicity [1], [2]. The I/O bottleneck of such a storage system, however, may easily result in the phenomenon of booting storm in the deployment phase of VMs [3]. That is, the overall performance will become very low when a large number of VMs are booted up concurrently or their VDIs are frequently read/written simultaneously when initializing their operating systems or hosted services. In addition, the VM running period after the system boot-up phase may also suffer from the frequent VDI accesses because of the data-intensive applications deployed in VMs [4].

In order to improve the access performance of VDIs, some storage systems (such as [5], [6] and [7]) are designed with collaborative cache, being able to distribute VDI chunks across the local disks of multiple compute nodes. Such systems, on the one hand, significantly reduce the dependency of the centralized storages, as well as the I/O bottleneck. On the other hand, they create a big opportunity for the VMs to easily query and access the required chunks stored on compute nodes, due to the high similarity across the VDIs [5], [8], [9].

Such existing storage systems are suitable for the small-scale or medium-size cloud environment, but they are not suitable for large-scale environment because of inevitably unbalanced network traffic state. In particular, the network traffic volume is often distributed unevenly among compute nodes in a large-scale data center [10], [11]. What is even worse, the network traffic introduced by the remote chunk access in the VDI storage system will likely aggravate such uneven network status, especially when a large number of VMs are running for a long time. Since the existing collaborative storage methods are unaware of the dynamic traffic state, the compute nodes with heavy traffic will become the potential hot-spots, leading to serious I/O performance degradation of VMs. We give an example in Section 2.3 to illustrate such a problem.

In order to mitigate the negative effect of uneven traffic distribution, it is necessary to make the chunk access adapt to the dynamic traffic patterns.[1] Few existing solutions, however, can be adapted to dynamic traffic due to the following challenges. (1) The requested chunks of VMs are not always stored in the lightly traffic-loaded nodes because of the diversity of cloud service requests and user demands.

- *S. Wu, Y. Wang, W. Luo, X. Xu, R. Zheng, and H. Jin are with the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {wusong, yihwang, weiluo, xxl, zhraner, hjin}@hust.edu.cn.*
- *S. Di is with Argonne National Laboratory, Lemont, IL 60439. E-mail: sdi1@anl.gov.*
- *H. Chen is with Chuzhou University, Chuzhou 239000, China. E-mail: chb@chzu.edu.cn.*

Manuscript received 21 Feb. 2016; revised 16 Jan. 2017; accepted 30 Jan. 2017. Date of publication 2 Mar. 2017; date of current version 9 Aug. 2017. Recommended for acceptance by X. Gu. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2017.2675988

1. Traffic pattern refers to the incoming and outgoing traffic of VMs. It is discussed in Section 2.2 in more detail.

(2) The addressing mode[2] is generally static in existing collaborative VDI storage systems, so the chunk access patterns cannot be dynamically changed to avoid hot-spots. (3) The traditional traffic-aware scheduling like [10] and [12] is designed only for the data centers with centralized storage, which cannot fit the new collaborative environment.

Accordingly, we propose ACStor to solve this problem. In comparison to the previous work, our solution is able to dynamically handle the chunk access operations in the VDI storage system, based on the run-time network traffic load on compute nodes. Specifically, there are three significant contributions in this work.

- We characterize the existing VDI storage systems, and validate the uneven traffic distribution issue. Such a problem is due to the fact that the existing solutions consider only the chunk access in an ideal environment, neglecting the unbalanced traffic state among compute nodes. Such an unbalanced traffic is very common in data centers, and easily leads to serious I/O performance degradation of VMs.
- We design a set of optimization strategies for improving the VDI chunk access performance. (1) We devise a two-level cache, including a private cache and a public cache, which can handle the access of VDI chunks more efficiently on compute nodes than traditional approaches, because of the separate writes and reads. (2) Under our solution, the chunk access is enabled to adapt to the dynamic traffic patterns: the requests can be dynamically assigned to the nodes with light traffic load via an improved consistent hashing algorithm. (3) We propose a deduplication mechanism divided into distributed and local, to mitigate the storage consumption. The local deduplication based on chunk contents furthermore reduce cache updates.
- We rigorously implement a prototype system and carefully evaluate our solution. Experiments show that it can achieve $2-8\times$ performance improvement compared to other existing VDI storage approaches with collaborative cache under different network traffic patterns in data centers.

The rest of the paper is organized as follows. In Section 2, we present the background and research motivation. In Section 3, we describe the key design of ACStor. In Section 4, we elaborate the implementation of ACStor. We present evaluation results in Section 5, and discuss some key issues and the future work in Section 6. In Section 7, we discuss the related work. Finally, we provide concluding remarks in Section 8.

2 Background and Motivations

In this section, we first discuss the pros and cons of the existing VDI storage strategies. Then, we discuss the traffic patterns in the cloud environment with collaborative storage, followed by our motivations.

2.1 Existing VDI Storage Models

In order to store and operate VDIs more efficiently for cloud environment, it is necessary to customize a particular storage system based on the features of VDI data such as the similarity of data chunks, instead of directly using general-purpose storage systems for simplicity [13].

There are many existing solutions for storing VDIs in clouds. We classify them into three categories from the perspective of cache usage.

Non-Cache Based VDI Storage. This is the simplest model, because it stores all VDIs in a dedicated back-end storage device with a large capacity for simplicity. The dedicated storage device could be a *network file system* (NFS) server or a *parallel file system* (PFS), which can be shared by all VMs. Such a centralized management, however, suffers from the I/O performance bottleneck when the system scale is large, in that each VDI access has to go through a few *network interface cards* (NIC) of the storage device(s).

Local-Cache Based VDI Storage. The second strategy distributes VDIs across local disks of compute nodes before booting up VMs, and also makes use of local cache to reduce the chance of remote access to back-end storage, avoiding unnecessary data transmission. By this means, VMs can automatically fetch data on demand from the backing VDIs to local caches. A critical problem is that the local caches cannot store all of VDIs, as there are more and more discrepancies among the cached VDIs across nodes over time, which will definitely degrade the access performance.

Collaborative-Cache Based VDI Storage. In order to access VDIs more efficiently, collaborative storage model is proposed. This model splits a VDI into a series of fixed-size chunks and distributes them across compute nodes. Due to high similarity across VDIs, there will be many common chunks across the caches, such that these chunks can be shared across compute nodes in a *peer-to-peer* (P2P) manner, such as BitTorrent [14] or a zone-shared way [15]. The biggest advantage of this model is that it has a high cache-hit ratio.

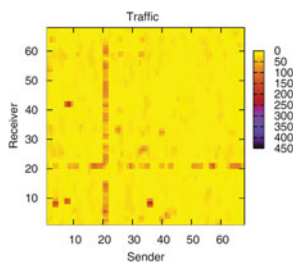
2.2 Traffic Patterns in Data Centers

Although a VDI storage system with collaborative cache outperforms other traditional storage systems, its I/O access performance is sensitive to network state in data center, in that quite a few chunk accesses have to go through the network to fetch the required data on remote nodes. The remote data access performance will be degraded significantly on the nodes with hot-spot traffic.

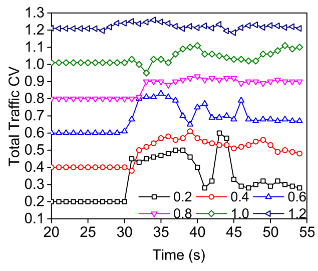
As more and more communication intensive applications (MapReduce style) are deployed in cloud data center, the bandwidth usage between *virtual machines* is rapidly growing. Therefore, the principle bottleneck often happens on the inter-node (or inter-VM) communication bandwidth, especially when a significant number of hot-spots exist in the network [16], [17], [18].

The uneven traffic distribution among compute nodes is very typical and common in clouds. The traffic here refers to the number of incoming and outgoing network packets received by VMs. Meng et al. [10] revealed two features of the traffic pattern in a typical cloud. The first one is that the VMs' traffic volumes are unevenly distributed in the system, as exemplified by Fig. 1a, which uses different levels of gray-scale to reflect traffic rates. We can clearly see that different pairs of VMs deliver various inter-VM traffic rates. The second one is that the per-VM traffic is usually stable at large time-scale. Shrivastava et al. [12] showed that the VMs

2. Addressing mode is the method that VMs can query and locate the required data in the VDI storage system.

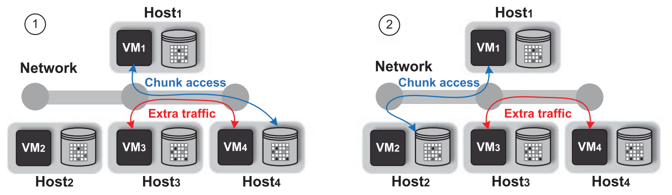


(a) traffic matrix from [10]



(b) total traffic CV

Fig. 1. Unbalance traffic state in clouds.



(a) before re-mapping (b) after re-mapping

Fig. 2. The basic idea of ACStor. As VM_3 starts to communicate with VM_4, VM_1's communication node will be redirected from $Host_4$ to $Host_2$, in order to avoid the traffic conflict.

comprising a multi-tier application may also introduce complex traffic interactions among the physical nodes.

For a cloud system facilitated with collaborative VDI cache, the traffic pattern (without considering the back-end storage) is rather more complicated. The traffic in a collaborative VDI storage system can be classified into two categories, the traffic generated by the remote chunk access and the traffic independent of chunk access. Note that the chunk access is not only triggered by local I/O operations of VMs, but also triggered by a remote node, leading to an uneven network traffic state in the system. Such a unbalanced traffic state could be even aggravated over time, due to the fact that the existing VDI collaborative storage systems are unaware of the traffic state. For instance, hosts with frequent inter-node (or inter-VM) communication have to deal with chunk access from other hosts. Fig. 1b shows the traffic variance (evaluated by *coefficient of variance*[3] *(CV)*) with different initial traffic states. We can clearly observe that the traffic distribution is fairly unstable and much higher in the VM booting/running phase (starting from second 31) than the initial period (from second 20 to 30), because of the simultaneous accesses to VDIs.

2.3 A Motivating Example

We use a motivating example to further illustrate the problem, as shown in Fig. 2a. Suppose VM_1 needs to access the chunk cached in $Host_4$. At the same time, there is a traffic generated by the inter-VM communication between VM_3 and VM_4. Then, the $Host_4$'s traffic may become overloaded when handling the two traffic links simultaneously, such that the original chunk access from VM_1 will be significantly affected.

We evaluate the I/O performance of collaborative storage under various traffic patterns on a 32-node testbed. The experimental results show that the degree of I/O performance degradation increases with the volume and the uneven degree of traffic distribution (detailed in Section 5). Such a phenomenon motivates us to design a collaborative VDI storage which can adaptively handle the chunk access operations based on the dynamic network traffic state.

3 Adaptive and Collaborative Storage System (ACStor)

In this section, we first introduce the basic idea, and then we summarize three critical issues to revolve in our design. Finally, we describe our solution in detail.

3. Traffic CV stands for the uneven degree of the traffic among compute nodes. The bigger value of traffic CV is, the more unevenly the traffic distribution exhibits. It is discussed in Section 5.3 in more detail.

3.1 Basic Idea

As mentioned above, the existing collaborative VDI storage systems cannot work efficiently in the situation of uneven traffic distribution. The key reason is that the remote chunk access cannot adapt to the varied traffic patterns at runtime. Thereby, the basic idea of our solution is to assign more chunk accesses from remote VMs to the compute nodes with relatively light traffic and vice versa. Fig. 2 illustrates the workflow of this approach. As shown in Fig. 2b, as the traffic load of $Host_4$ increases, VM_1's chunk access request will be redirected to $Host_2$, which has less traffic load compared with others. Through this way, the global traffic load can be balanced, so as to improve the network utilization as well as the I/O performance of VMs in turn.

Such a design, however, may introduce three new challenging issues to resolve.

How to Store and Organize the VDIs in Data Center? The traditional collaborative storage systems divide whole VDIs into a series of fixed-size chunks, and distribute them from back-end centralized storage to compute nodes, such that all of VMs can share them across the nodes. Since the nodes with the required chunks may not always have light traffic load, we need to redirect some of the chunk accesses from the heavily traffic-loaded hosts to the lightly loaded ones dynamically. However, after changing the target hosts for some requests, the corresponding accesses may be failed unexpectedly if the chunk requested cannot be found in the new destination host. For example, as shown in Fig. 2, after a chunk access of VM_1 is redirected from $Host_4$ to $Host_2$, it would be failed if the required chunk was not cached on $Host_2$. Hence, the first issue is how to store and organize the VDIs, such that each chunk can always be efficiently queried on any host.

How to Redirect a Chunk Access to the Host we Want? In general, the addressing mode of existing collaborative storage systems cannot dynamically change target hosts upon the change of chunk access requests, even though the target hosts have the chunks requested. Accordingly, it is necessary to devise a dynamic addressing mode to access the chunk stored in any host on demand (e.g., with less real-time traffic), for the purpose of traffic load balance. For example, suppose the requested chunk exists in both $Host_2$ and $Host_4$, and one wants to access the chunk stored in $Host_4$ at the beginning yet redirects the access to the $Host_2$ later on. The existing addressing modes (e.g., using the *Chunk-At-Host-Table* (CAHT) in [5] or using a Bloom Filter in [6]), however, will actually still always stick to $Host_2$ as long as $Host_2$ has the available requested chunk.

How to Balance the Global Traffic by Manipulating the Chunk Accesses? It is necessary to design a traffic load balancing policy according to the dynamic traffic state. In other words,

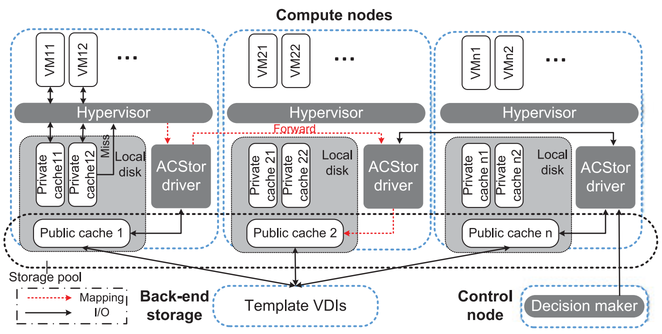


Fig. 3. The architecture of ACStor with two levels of caches (i.e., public cache and private cache).

the system is supposed to automatically determine when and where to forward an I/O request. Besides, we should also take into account the network topology in data center. As illustrated by Fig. 2, suppose both $Host_2$ and $Host_3$ have the chunk requested by VM_1. Note that the access distance between VM_1 and $Host_2$ is farther than that between VM_1 and $Host_3$, hence $Host_3$ is a better choice from the perspective of access distance. In this situation, suppose $Host_2$ has the lower traffic load than $Host_3$, then there will be a tradeoff between the traffic load and the access distance, when determining which host to access for VM_1.

3.2 Design Overview

In this section, we present the design overview of our ACStor, as shown in Fig. 3. We propose three key principles in order to deal with the three issues addressed in Section 3.1, respectively.

For the first one, we use a two-level cache architecture to store the VDIs in ACStor. The two-level cache consists of a private cache and a public cache in the local disk of a compute node, in order to handle the writes and reads, respectively.

For the second issue, we design an adaptive addressing mode based on the two-level cache architecture. It can change the target host of I/O requests of VDIs according to the traffic load of each compute node and the network topology of data center. This addressing mode takes full advantage of consistent hash [19] and content-based addressing [20] to query and identify the VDIs accurately, improving the hit ratio of public cache at the same time.

For the third issue, in order to make the remote VDI access adapt to the extra traffic patterns, we provide a mapping strategy based on both the dynamic traffic of compute nodes and the topology of data center. This work is performed by the decision maker in a control node, which notifies the ACStor drivers the dynamic policies according to the run-time network status.

3.3 Two-Level Cache

The two-level cache is used to cache the VDIs read from the back-end storage into the local compute nodes. In what follows, we describe private cache and public cache, respectively.

3.3.1 Private Cache

The first-level cache is a private cache which is deployed in the local disk of compute node. Each private cache employs *copy-on-write* (CoW) to serve a VM, i.e., all writes are conducted in this cache after copying the context from the back-end storage. That is, the write operations will not be passed to the public caches or the template VDI in back-end storage.

This design has two advantages: first, it can accelerate the writes and a part of reads in the local nodes; second, VMs can read the data from the public caches without concerning dirty data, because private cache handles all the write operations of a VM. Generally, we do not need to use cache replacement strategies in a private cache. When data size exceeds disk quota, we will dump part of data to the back-end storage.

3.3.2 Public Cache

Public cache (a.k.a., collaborative cache) will work when read missing occurs in the private cache. The public caches serve the reads sent from any VMs. A public cache deployed in a compute node occupies an independent partition of the local disk, and all of the public caches constitute a distributed storage pool. For sharing across compute nodes conveniently, VDIs are cached in public caches in the form of data chunks. As shown in Fig. 3, we illustrate the procedure of finding a chunk in public cache for VMs as follows. When a request misses the private cache, the hypervisor will pass it to the local host's ACStor driver. The driver forwards this request to a remote ACStor driver (or query the local public cache). Upon receiving the request, the remote ACStor driver will read the chunk in its local public cache. Such a procedure is called *mapping*. By modifying hypervisor's mapping strategy, we implement a dynamic request redirecting for traffic load balancing. If a public cache miss occurs, the ACStor driver in the target host will fetch the chunk from back-end storage.

In order to save disk space and ensure that the chunks are available in target hosts meanwhile, we do not distribute these chunks or their replicas in the storage pool in advance. Instead, as ACStor service starts, an empty public cache is created in each compute node. Every missing request from private caches needs to access the template VDIs, fetch the corresponding data blocks, and put them in the public caches on demand (i.e., copy-on-read). Users can decide the size of public cache according to the disk utilization. Various cache replacement algorithms can be applied to the public cache, such as *first in first out* (FIFO), *least recently used* (LRU), and *least frequently used* (LFU). The requests that cannot hit the public cache will be forwarded to the back-end storage device for searching the template VDIs.

3.4 Adaptive Addressing Mode

In the above section, we provide an overview of the I/O flow in ACStor. In what follows, we describe the addressing mode and discuss why it can adapt to the dynamic traffic in data centers.

3.4.1 Mapping Procedure

First of all, we elaborate the mapping procedure, which is the key to locate the target public cache.

As shown in Fig. 3, when a VM sends an I/O request, it will search the private cache first. Since the private cache is a file named the ID of VDI used by a VM, we can use the ID of VDI and file sector offset information to seek the requested data. If private cache miss occurs, the hypervisor will pass this request to the ACStor driver that will determine which

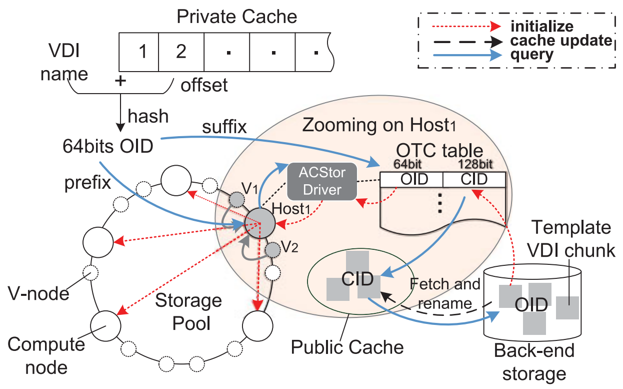


Fig. 4. The I/O flow of the adaptive addressing mode in ACStor.

host (i.e., public cache) to access. To locate the target host, we use consistent hashing algorithm [19] here, which maps each request to a host on a hash circle (similar to Chord [21]).

As shown in Fig. 4, all compute nodes are distributed on a hash circle. When a request with VDI ID and offset information comes, we convert this information to a 64 bit hash value called the *object ID* (OID). An OID consists of two parts: a prefix to identify the target public cache and a suffix to identify a chunk in the corresponding public cache. So an OID can uniquely identify a VDI chunk in ACStor. Actually, in order to implement the re-mapping to be introduced later, we use only the prefix of OID to locate the target public cache, instead of addressing the specific chunk. For example, the request in Fig. 4 locates the compute node $Host_1$ by using the prefix.

3.4.2 Re-Mapping Procedure

Before explaining how to address a chunk in public cache, we need to show the re-mapping procedure, which is used to redirect requests from a host to another.

The consistent hashing and OID are used to determine the target host for a chunk access. In this way, however, the I/O requests with the same OID will be always forwarded to the same host. In order to access different hosts with the same chunks dynamically, we improve the consistent hashing algorithm. Specially, the consistent hashing maps each host on the circle to multiple pseudo-randomly distributed points (we call them virtual node or v-node) on the same circle. If a request is mapped to a v-node, it will be essentially handled by the host this v-node belongs to.

The biggest advantage of using v-node is that we can control the distribution of requests directed to different hosts, by dynamically changing the number of v-nodes for each host at runtime. This is due to the fact that the hosts with more v-nodes will be selected to handle requests more frequently. Such a procedure is called re-mapping in this paper. Some existing storage systems such as Amazon's Dynamo [22] also adopt virtual node for consistent hashing. However, they are only used when a node leave or join the system, or used to estimate the storage load based on the capacities of disks. Note that the re-mapping will definitely disorganize the distribution of requests, so it is necessary to dynamically tune the number of v-nodes at runtime, while it is unchangeable in traditional systems.

There are two points worth mentioning. (1) The re-mapping operation only focuses on the distribution of requests handled by the public caches, instead of the specific request type. In fact, such a design is already able to adjust the global traffic load very well, and it can be compatible with different mapping policies. (Detailed in Section 3.6). (2) As the mapping strategy is changed, the requests previously forwarded to a host may be redirected to another host, leading to potential public cache misses. That is, the I/O performance would be degraded, even though the overall system availability is unaffected. In what follows, we will discuss how to deal with this problem, by designing an adaptive chunk addressing method in ACStor.

3.4.3 Chunk Addressing

If we use the OID to uniquely address/identify a chunk in ACStor, there must be a cache miss when the target host of the request is changed in the re-mapping procedure. Some studies like [8] indicate that there are a lot of common chunks with the same content after dividing VDIs into small chunks. That is, even though each chunk has an unique OID, many of them have the same contents actually, which inspires us to use the content-based addressing in our adaptive addressing model. Hence, we can avoid most of cache misses in the re-mapping procedure by using an index named as *object-to-content* (OTC) table. In the following, we will illustrate the process of chunk addressing in detail.

Initialization. As a new VDI is registered to ACStor, it will be preprocessed to initialize an OTC table. Specifically, the VDI will be divided into multiple fixed-size chunks, each of which is identified by its OID on the back-end storage and also identified by a hash code generated by the MD5 hashing algorithm. Accordingly, each chunk in ACStor has two identifiers, a 64 bit OID and a 128 bit hash code. The 128 bit hash code represents the chunk content, and we call it *content ID* (CID). As shown in Fig. 4, an entry of OTC table stands for a chunk which consists of an OID and a CID. The OTC table on every compute node is synchronized with those of other nodes through ACStor drivers, such that every host can query any chunk information around the whole system. Such a synchronization is performed only when a new VDI is registered to the system, so there will be no synchronization cost during the VM boot-up phase or VM running period. But when a new VDI is registered, the synchronization will happen again to make sure that each host has the same OTC table contents.

Query. A complete chunk addressing process is shown in Fig. 4. First, our solution locates the target host according to the prefix of OID. Second, the OTC table in this host will be searched in order to find the corresponding entry based on the suffix of OID, such that we can get the CID of the requested chunk. Last, we access the chunk based on the CID in public cache. If there is no requested chunk (i.e., cache miss), the cache update process will be conducted, that is, we will fetch the chunk from the back-end storage, put it in the public cache and index it by its CID for the future query.

Note that, different from previous Chord-like DHTs, such as Dynamo [22] and Cassandra [23] with static addressing mode, our method can dynamically adjust the mapping node (for example, by changing the v-node quantity of nodes) at runtime. And, It is optimized for virtualized environment especially with heavy inter-VM traffic. Moreover, our method is a chunk addressing mode based on VDI content and can deal with the potential public cache miss.

3.5 Public Cache Deduplication

Our proposed deduplication mechanism consists of distributed deduplication and local deduplication. The former deletes duplicated chunks in nodes according to OID, while the latter works in local host based on CID.

Re-mapping procedure probably produces chunks duplication, which means that different hosts may have the same chunks with the same OID. Distributed deduplication component calculates the amount of chunks with the same OID. If this amount is greater than a threshold value (set by users), chunks with that OID need to be de-duplicated. We adopt LRU algorithm to delete duplicated chunks.

As chunks with different OIDs may have the same contents, we propose local deduplication, namely content-based addressing in Section 3.4.3. It de-duplicates chunks based on CID in each node. By taking advantage of content-based addressing, the VDI chunks can be addressed based on their contents instead of only using the offset information in VDIs. It can avoid some cache update operations, also reducing the cache misses in re-mapping procedure.

3.6 Mapping Strategy

In order to balance the global traffic, we need to implement an efficient mapping and re-mapping strategy. In ACStor, this issue can be converted to the problem of associating v-nodes to hosts.

We propose a formula (Equation (1)) to determine the number of v-nodes for host i, where n is the total number of compute nodes, and λ is a factor (or coefficient) to avoid too large or too small numbers of v-nodes. We set it to 1.67 in our experiments because it can lead to a satisfying result. In fact, the system booting performance is not sensitive to λ as long as its value is around 1.67, to be shown in Section 5.4.1. $Traffic_i$ here refers to the host i's traffic load, which is not generated by remote chunk accesses. Intuitively, the hosts with less traffic are supposed to be assigned more chunk accesses, and vice versa. Thus, we make the number of v-nodes associated to each host inversely proportional to its traffic, as shown in the equation

$$vnode\_count(host_i) = \lfloor \frac{\lambda \times \sum_{j=1}^{j=n} traffic_j}{traffic_i} \rfloor. \tag{1}$$

After recalculating v-nodes associated to each host, we have a further v-node mapping adjustment in compute nodes, which is to reduce potential cache miss caused by re-mapping. We record the changes of v-node count associated to each host after recalculation. Adjustment algorithm does not change the v-node count in each host calculated by Equation (1), but only changes some v-node mapping. It depends on the similarity between compute nodes, that is, host which needs to reduce v-nodes will transfer its v-nodes to the host that needs to increase v-nodes and has the highest similarity with the former. Here, host similarity refers to the proportion of the same chunk between two hosts.

Besides, we take into account the topological information of data center, which is also important for the performance improvement. All these factors our strategy takes into are for balancing the workload and improving the performance. The pseudo-code is presented in Algorithm 1. In the beginning, it calculates the v-node quantity for each host using Equation (1) (lines 1-5 of Algorithm 1) and than make a v-node mapping adjustment. Then, it identifies the target host based on the requested chunk by using the consistent hashing. If the distance between the target host and the source host of the request is smaller than the *threshold* we set, we forward this request to the target host. Otherwise, we recalculate the target host without taking this host into account just during this request. (lines 6-13 of Algorithm 1).

Algorithm 1. A Mapping Strategy to Locate the Target Host

```
Input: 1) the list of compute nodes: $nodeList$; 2) the OID of the
       request: $oid$; 3) the source node of the request: $src$; 4) a
       threshold: $threshold$
Output: the destination of the request: $dest$
1:  $vnodeList$ = [] /* the list of the v-node count for each node */
2:  for each $node$ in $nodeList$ do
3:      calculate the v-node count by Equation (1)
4:      add the v-node count into $vnodeList$
5:  end for
6:  v-node mapping adjustment
7:  begin:
8:  calculate the $dest$ of request according to the $oid$ by using
    consistent hashing
9:  if the distance between $src$ and $dest$ $<$ $threshold$ then
10:     return $dest$
11: else
12:     remove the $dest$ from $nodeList$ /* remove the host from
    hash circle only during this request */
13:     goto begin
14: end if
```

The *threshold* in Algorithm 1 can be set according to the following strategy. If half of network switches at some level exhibit fairly heavy traffic usage (e.g., the traffic rate is over 80 percent of the bandwidth), then the *threshold* is set to the minimum hop counts required by the requests when they go through the switches in this level. The pseudo-code is presented in Algorithm 2.

Algorithm 2. Calculating the Threshold

```
Input: the 2-dimensional array of switches: $switchList[][]$,
       where $switchList[0]$ is the list of edge switches,
       $switchList[1]$ is the list of aggregate switches, and
       $switchList[2]$ is the list of root switches
Output: $threshold$
1:  $threshold = 7$
2:  for each $childList$ in $switchList$ do
3:      $count = 0$ /* the number of the overloaded switches */
4:      for each $switch$ in $childList$ do
5:          if $switch$.traffic $>$ $switch$.bandwidth*0.8 then
6:              $count$ ++
7:          end if
8:      end for
9:      if $count$ $>$ $childList$.length $\times$ 0.5 then
10:         $threshold$ = $childList[0]$.level $\times$ 2 + 1 /* the level of
    edge, aggregate, and root switch is 0, 1, 2, respectively */
11:         return $threshold$
12:     end if
13: end for
```

Note that the way to determine the threshold could be different in other topologies such as BCube. Specifically, in a three-level tree topology data center, if the root switch has a heavy traffic rate (line 5 of Algorithm 2), we can set it to 5 according to our strategy (line 10 of Algorithm 2). If more than half of aggregate switches have heavy traffic rates, the threshold is set to 3. In this way, we not only let chunk access adapt to the dynamic traffic of hosts, but also reduce the overhead of remote access.

Since the traffic in data center changes over time and transient workload spikes occurs from time to time, we should conduct the re-mapping operations at the right time. In order to avoid the system thrashing (i.e., the frequent cache replacement caused by re-mapping), we conduct the re-mapping operations only when a part of hosts are overloaded in two successive collection. There are two reasons why we adopt this strategy. First, if none of the hosts is overload, VMs' I/O performance will not be affected at all even though the traffic load is uneven. Second, if ACStor triggers remapping immediately in the face of transient workload spikes, it may suffer from potential cache misses caused by unnecessary re-mapping. Therefore, we conduct the re-mapping operations only when there is an overloaded host, which can reduce the system thrashing generated by short-term traffic burst.

3.7 Other Issues

As a VDI storage system, there are two more important issues (VDI update and fault tolerance issue in ACstor) which need to be considered in addition to the I/O performance, in comparison to the existing systems.

3.7.1 VDI Update

To avoid dirty data in public caches, the template VDIs in back-end storage are not allowed to be updated directly. All of the users' private data and modifications are stored in private cache temporarily. When VMs are shutdown, cloud middleware in the control node will move the corresponding private caches and their snapshots (detailed in Section 3.7.2) to the back-end storage and change their backing files (i.e., base VDI) to the template VDIs they used. These private caches in back-end storage can be considered as new customized template VDIs, though they cannot be booted up without the original template VDIs. That is to say, the template VDIs are not updated actually, but we append the modifications as incremental VDIs to the template VDIs and consider them as a whole. In order to avoid the performance degradation caused by massive incremental VDIs, we merge the child and grandchild of a template VDI when the depth of a VDI tree exceeds 3.

Note that there is a trade off whether the new template VDIs need to be registered to ACStor as presented in Section 3.4.3. For example, a new template VDI containing tens of gigabytes of someone's private data can hardly be shared by others. Thus, caching it to public caches would occupy a lot of space and reduce the number of common chunks in system, so as to reduce the cache hit ratio after re-mapping. However, in our prototype system, we still cache each template VDI to public cache for simplicity.

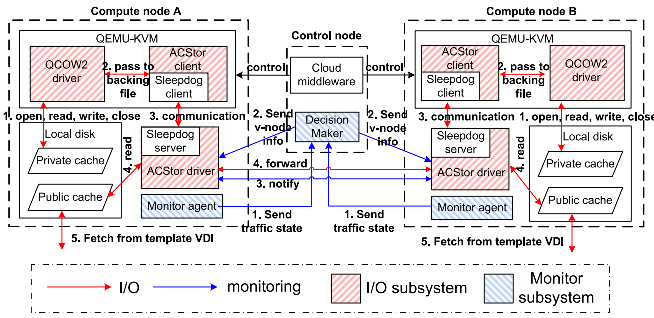


Fig. 5. The system module chart.

3.7.2 Fault Tolerance

In our implementation, ACStor is able to protect the execution of VMs and users' data against unexpected faults/failures such as node crashes, by adopting both checkpoint/restart model and replica mechanism. The fault tolerance is completely transparent to users, because it is automatically conducted by the system.

ACStor protects the private cache in case of failures, by periodically setting the checkpoint files for each running VM to the back-end storage. The checkpoint files (a.k.a., snapshots) are very easy to be generated in ACStor, because of our private cache designed to handle users' modifications to VMs. Specifically, when a checkpoint file needs to be created for a VM, what ACStor needs to do is just to save the current corresponding private cache into the back-end storage. In order to reduce the VM-recovery overhead upon failures, the checkpoint files generated will also be kept in the local disk for a certain period until the next checkpoint setting.

The back-end storage in ACStor is protected by the replica mechanism in case of storage failures, and each VDI file has three replicas in the back-end storage. As for the public cache, it can be constructed based on existing distributed file systems with inherent fault-tolerance mechanism.

Although we have fault tolerance considerations above, our system has a single-point failure on control node. Fortunately, if the decision maker on control node crashes during system running, ACStor still can work, which is because chunk access does not rely on decision maker. But, ACStor can not adapt to network loads anymore, due to re-mapping operation depending on the v-node information received from decision maker. In the follow-up study, we will solve this problem by adding a backup node.

4 Implementation

In this section, we elaborate the implementation of ACStor in detail (the code of ACStor is publicly available online at *https://github.com/CGCL-codes/ACStor*). The system module chart is shown in Fig. 5. ACStor includes two parts: I/O subsystem and monitor subsystem. We will present the work flows step by step.

Note that the core ACStor driver component is based on Sheepdog [24]. Sheepdog is an open-source distributed storage system, providing highly available block level storage volumes that can be attached to KVM virtual machines. It is chosen for sake of the following three reasons. First, the KVM's userspace tools (called qemu-kvm) support sheepdog block driver. Therefore, it is convenient for us to implement the data transmission from local VMs to remote VDI storage servers. Second, Sheepdog has no centralized master node, which alleviates the bottleneck and accelerate the I/O performance for public cache accesses. Last, Sheepdog has a built-in fault-tolerance mechanism, which can be used on the public cache layer.

4.1 I/O Flow

I/O subsystem is responsible for dealing with the I/O requests. The I/O flow is shown in Fig. 5. *Step 1:* Each request (e.g., open, read, write, close) from VMs accesses private cache first which is handled by the QCOW2 driver in QEMU-KVM. Private cache is an incremental image in QCOW2 format, whose backing file contains the chunks in public cache. *Step 2:* When a private cache miss happens, QCOW2 driver passes the request to its backing file driver (i.e., ACStor client). Since the size of private cache is restricted, when the size of a private cache exceeds the quota, we move it to the back-end storage and create a new private cache file whose base VDI is the older private cache. This process is controlled by the cloud middleware in control node. *Step 3:* ACStor client will directly send requests to ACStor driver (We use the Sheepdog's communication module). *Step 4:* After an ACStor driver receives a request, it parses this request into OID and forwards it to the target host by consistent hashing. Then, when an ACStor driver receives a request from itself (the target of this request is local host) or others, it gets the corresponding CID from OTC table and reads the data from the chunk. *Step 5:* If the chunk is not available, it generates a public cache missing signal, and then accesses the data from back-end storage. At the same time, a new thread is raised in charge of fetching the missing chunks from back-end storage to public cache. This asynchronous and non-block cache fetching process accelerates the chunk access.

Here we give a scenario example to describe the flow in detail. Suppose Node A has private cache misses, and the request is sent to Node B. If Node B has the public cache hit, B will transfer the chunk to A. The transferred chunk will not be saved to Node A. If Node B has the public cache miss, B will access data from back-end storage and the data is read to B's public cache. And Node B needs to transfer the data back to A.

4.2 Monitoring Flow

Monitor subsystem lets ACStor perceive the network state in data center, and helps I/O subsystem to forward requests adaptively. The Monitoring flow is shown in Fig. 5. *Step 1:* First, monitor agents collect traffic state in local hosts and send it to decision maker periodically. Each agent can distinguish the traffic generated by chunk accesses by monitoring the incoming and outgoing traffic in ACStor driver daemon. So, the traffic volume used in Equation (1) is set to the difference of the total traffic and the traffic generated by chunk accesses. *Step 2:* Decision maker calculates the suitable v-nodes information according to the network state by Equation (1). Then it sends this information to a ACStor driver, which will synchronize the information with others later (i.e., *Step 3*).

5 Performance Evaluation

In this section, we first measure the I/O performance of VMs by using different VDI storage approaches under various traffic patterns in a physical platform with 32 nodes. We then evaluate the scalability of our approach in a simulated platform with 256 nodes.

5.1 Experimental Setting

Platform. We perform the experiments on two platforms: a 32-node experimental testbed and a 256-node simulated platform. The simulated platform aims at simulating a large-scale data center, and it is only used for evaluating the scalability (detailed in Section 5.8). In the 32-node real cluster testbed, each blade node has two x86_64 CPUs (2.6 GHz) with hardware virtualization support, 64 GB memory and 300 GB local disk. The back-end centralized storage is served with one storage server, which has x86_64 CPUs (1.8 GHz), 16 GB of memory and 18 TB of storage, and it is mounted to compute nodes through NFS. These nodes are interconnected with Gigabit Ethernet. A blade center consists of 12 blade servers and the backplane bandwidth of the switch which connects the blade centers is 640 Gbps. The hypervisors running on all compute nodes are KVM 1.2.0, while the operating system is Red Hat Enterprise Linux Server release 6.2.

VDIs. We use a series of VDIs each with size of 4 GB. In order to simulate the real environment, we deploy some common softwares such as Apache, MySQL on each of them. The operating systems of the VDIs include 64-bit Centos 5.5, 64-bit Centos 6.6, 64-bit Ubuntu 12.04, 32-bit Ubuntu 12.04 and 64-bit Windows Server 2008. We upload these VDIs to back-end storage in advance, and each of them is divided into many 64 KB chunks.

Traffic. As mentioned in Section 2.2, the total traffic consists of two parts: the traffic generated by chunk accesses and the traffic independent of chunk accesses. We call the latter *extra traffic* which will be used as an independent variable in the following experiments to simulate different traffic patterns. We make use of two factors to generate various extra traffic patterns, *mean traffic load* and *traffic coefficient of variation*. The larger the size of mean traffic load is, the heavier traffic in data center is. In order to generate different traffic patterns, we let each VM send TCP or UDP messages with different sizes of packets to VMs in other compute node with a fixed rate.

5.2 Approaches to Evaluate

We compare ACStor to three well-known VDI storage approaches: non-cache based storage, local-cache based storage and collaborative-cache based storage.

Non-Cache Based Storage. Is the baseline in our evaluation. For fair comparison, we also deploy a QCOW2 format VDI in each node whose backing file is stored in NFS server (a centralized storage mode). So, when the VMs are booting or running, they always access the data from the centralized storage on demand.

Local-Cache Based Storage. Is the improvement of the previous one. We append local caches in the disks of compute nodes between the QCOW2 incremental VDIs and their backing files like [3]. These local caches, whose size is half of a VDI, are also chunk-based and adopt FIFO cache replacement strategy.

Collaborative-Cache Based Storage. Is pretty close to our proposal. However, there are two differences between this approach and ACStor. It has no re-mapping function, therefore I/O requests cannot be forwarded dynamically. And, its

TABLE 1
Booting Time Changing with λ in ACStor

λ value	1	1.40	1.67	1.80	2.20	2.60	3
Booting time	53.1 s	55.1 s	54.7 s	52.1 s	55.3 s	54.9 s	53.1 s

Mean traffic load is 20 MB/s, traffic CV is 0.8.

mapping policy only contains the topological information for simplicity. In this way, all I/O requests are forwarded to the compute nodes with relatively close topological range without considering the dynamic traffic like [5]. For fair comparison, we set the total size of private cache and public cache in each host to that of local cache in the former approach.

5.3 Evaluation Metrics

In this section, we present the indicators which are used to evaluate different approaches.

Average Booting Time for Each VM. In VMs' booting process, there are a lot of concurrent I/O requests which are dominated by reads. This metric reflects the performance of VM deployment phase. It is computed by measuring the time of each VM from the moment of issuing boot instruction to the completion of all start-up services.

I/O Throughput. This metric measures the sum of I/O throughput for all VMs when we conduct a series of reads or writes. Different from the booting time, it stands for the I/O performance during the running phase of VMs.

Traffic Coefficient of Variation. This metric stands for the uneven degree of the traffic among compute nodes in the data center. A traffic *coefficient of variation* can be calculated by

$$CV = \frac{traffic\ standard\ deviation}{mean\ traffic\ load}. \quad (2)$$

The CV value can be larger than 1. The bigger value of the traffic CV is, the more uneven the traffic distribution is. The minimum value of the traffic CV is 0, which means all compute nodes have the same traffic volumes.

Note that the global traffic CV is not only used as a metric to measure the uneven degree of overall traffic, but is also an independent variable to generate different extra traffic patterns (detailed in Section 5.4).

5.4 Evaluation of Booting Time

In this section, We evaluate booting time changing with different values of vnode-count factor λ (defined in Formula (1)), traffic states and amount of VMs. Traffic states include the size of mean traffic load and traffic CVs in data center.

5.4.1 Sensitivity of Vnode-Count Factor

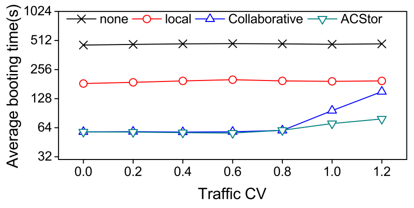
The vnode-count factor λ is designed for avoiding too large or too small number of v-nodes on a host. Its value has no clear influence on the system performance because it does not change the proportion of v-node between compute nodes. Table 1 shows the mean booting time changing with λ in Equation (1). We can see that it has little influence on the average booting times. In the following experiments, we set it to 1.67 in our experiment, which already is able to get a satisfying result in our evaluation.

5.4.2 Traffic States

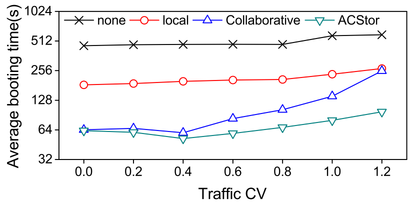
In this experiment, five VMs are co-running on each physical node. The template VDIs of these VMs are randomly selected from VDIs mentioned in Section 5.1. The number of VMs cannot be booted in four approaches is about 1-5, and it grows as traffic CV or mean traffic load increases. We discard the unbooted VMs when calculating the average booting time. Fig. 6 shows the mean booting times when the mean traffic loads are 40, 70, and 100 MB/s, respectively. In each experiment, the traffic of hosts follows a distribution of traffic CVs, ranging from 0 to 1.2 (some of VMs cannot be booted normally when the traffic CV is larger than 1.2).

There are two observations. First, the non-cache storage system performs stably but it suffers relatively low performance with various traffic CV values, while the local-cache based storage system obviously outperforms it. On average, booting time of local-cache storage system is 40-50 percent of the non-cache storage system. Since these two VDI storage systems are not involved in the cross-host accesses, the extra traffic among compute nodes can hardly effect their performance. When traffic CV increases up to a certain level (such as 0.8 with the mean traffic load as 70 MB/s), the performance of the non-cache storage system and local-cache storage system is still lower than that of the collaborative-cache based storage system. This is because the incoming traffic load of some compute nodes are even heavier than that of the outgoing traffic of the back-end storage device, leading to serious hot spots in the network.

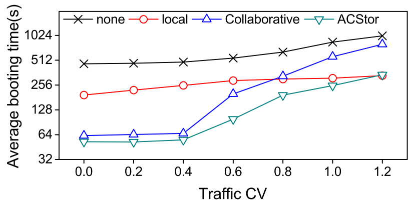
Second, the collaborative-cache based storage system suffers the interference from the extra traffic easily. The performance is further degraded with the increasing traffic CV. For example, when the traffic CV is over 0.8 with 100 MB/s mean traffic load, it performs worse than the local-cache based storage system. This is because when some compute nodes become overloaded, the new bottleneck is generated in these nodes, which leads the performance of chunk accesses from them to degrade. By comparison, our approach outperforms the traditional collaborative storage method under various traffic CVs. The booting time of ACStor is 38-95 percent of the traditional collaborative storage. The advantage of ACStor is rather clearer with higher traffic CV. For example, when the traffic CV is 1.2 and mean traffic is



(a) mean traffic load is 40MB/s



(b) mean traffic load is 70MB/s



(c) mean traffic load is 100MB/s

Fig. 6. Booting time changing with the traffic CV under different sizes of mean traffic load (160 VMs).

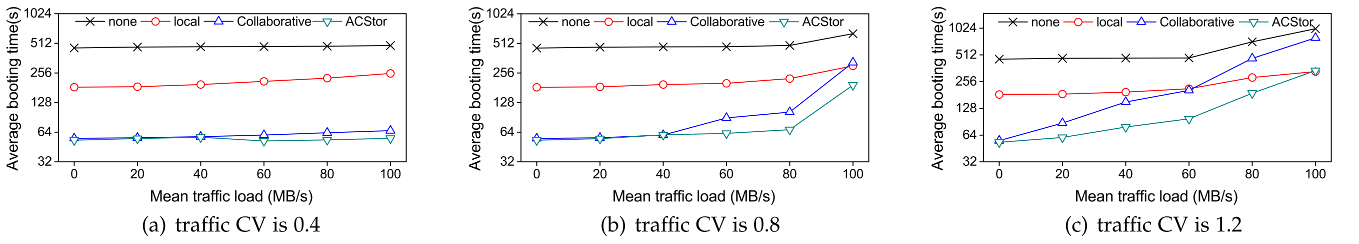


Fig. 7. Booting time changing with the size of mean traffic load under different traffic CVs (160 VMs).

70 MB/s, ACStor can get 2.6× performance gain compared to the traditional collaborative storage.

Fig. 7 shows the mean booting time that changes with the size of mean traffic load, when the traffic CV is 0.4, 0.8 and 1.2, respectively. In each experiment, the size of mean traffic load on compute nodes ranges from 0 to 100 MB/s (Some of VMs cannot be booted normally when the size of mean traffic load is larger than 100 MB/s). The changing trend of the performance under different sizes of mean traffic load is similar to that under different traffic CVs. The non-cache based storage and the local-cache based storage are hardly affected under different extra traffic loads. The performance of collaborative-cache based storage apparently degrades with increasing traffic loads. ACStor also has a performance degradation but less than the traditional collaborative storage, since the adaptive chunk access avoids the traffic hot spots on the potential overloaded nodes.

5.4.3 *Data Amount*

We further compared ACStor with the collaborative-cache based storage in this experiment. Fig. 8 shows the mean booting time changing with the amount of VMs, when the traffic CV is 0.4, 0.8 and 1.2, respectively. In each experiment, the amount of VMs ranges from 32 to 192. According to the increasing trend of booting time, We can find that ACStor has a lower slope than the collaborative-cache based storage system, and the scope gap is getting lager when traffic CV grows. Once again, It proves that ACStor has a lower performance degradation than the traditional collaborative storage.

In summary, we can draw two conclusions. First, the higher the traffic load or CV is, the more performance gains ACStor will get, due to the more benefits of traffic load balance. Second, the collaborative-cache based storage outperforms the local-cache storage in most cases. However, when the traffic load or CV across compute nodes is too high, the local-cache storage may be a better choice.

5.5 Evaluation of I/O Throughput

A good VDI storage system not only performs well in VM booting phase, but it is also supposed to have a good runtime performance. In this experiment, we choose three types of workloads with different characteristics of disk I/O (i.e., *web server*, *web proxy*, and *video server*) from Filebench [25] and measure the corresponding I/O throughput under different traffic patterns.

The results are shown in Fig. 9. The *x*-axis stands for 'mean traffic load-traffic CV'. For example, '40-0.4' stands for the scenario with the mean traffic load size of 40 MB/s and the traffic CV of 0.4. Since the *web server* in Fig. 9 a is dominated by read operations, most of the reads access the public cache in ACStor. We observe that during the running period, they always keep accessing the remote chunks with a stable rate in a large time-scale. So, ACStor does not need to conduct the re-mapping operation frequently. Thus, our approach can usually outperform the traditional collaborative storage, especially in the situation with uneven traffic.

Compared to the *web server*, the performance improvement of ACStor in *web proxy* (Fig. 9 b) is less, though the web proxy is also dominated by read operations, because the chunk access rate in web proxy is unstable and the size of requests are usually small, unlike the web server. This means the mapping policy is not always optimal. However, when the traffic is uneven, ACStor also has a big promotion.

The *video server* in Fig. 9 c consists of many writes as well as reads. So, all the storages have a performance improvement due to the Copy-on-Write (all the writes will only access local disk). Besides, as VMs read the newly updated data, they also access only the local disk. So, we find the different traffic patterns almost have no effect on I/O performance.

5.6 Evaluation of Traffic CV

In this experiment, we mainly observe the change of traffic CV, which is a critical metric indicating the traffic balancing state around the whole system. Fig. 10 illustrates the change of total traffic CV (regarding the traffic generated by chunk accesses and extra traffic) in the VM booting phase under different extra traffic CVs (0.4, 0.8 and 1.2). Since the non-cache storage and the local-cache storage have little effect on the traffic CV, we here mainly compare the collaborative-cache storage and our designed ACStor. It should be noted that we start the monitor process before we boot the

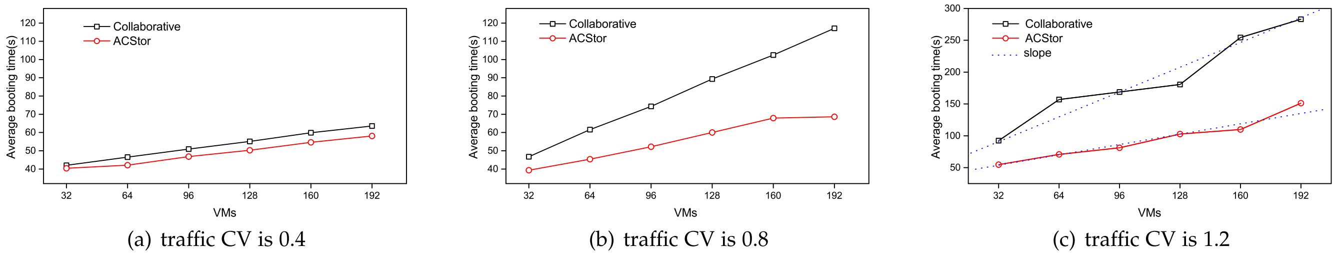


Fig. 8. Booting time changing with the amount of VMs under different traffic CVs (mean taffic load is 70 MB/s).

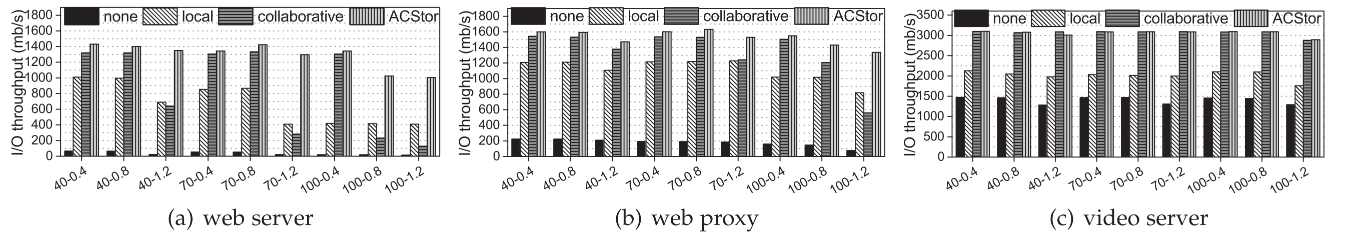


Fig. 9. I/O throughput for different workloads under various traffic CVs.

VMs, which leads to the stable CV during the first period of time (from second 0 to 30).

It is observed that the traffic CV exhibits the same for both storage systems at the beginning. It fluctuates through the booting phase with the collaborative-cache based storage. The total traffic CVs are maintained at a relatively higher level in this period than the beginning. The reason is that the traffic generated by chunk accesses aggravates the uneven degree of global traffic. However, if the original extra traffic CV is uneven enough (e.g., 1.2), the total traffic CV may not be affected a lot.

As expected, the total traffic CVs are significantly reduced under ACStor. Whereas, there are still fluctuations during the time of VM booting and it looks more serious than those in the traditional collaborative storage. This is because ACStor lets chunk accesses adapt to the extra traffic pattern.

5.7 Evaluation of Extra Overhead

There are three kinds of extra overhead in our approach: OTC-table query, OTC-table update and cache-miss caused by re-mapping. We evaluate them from the perspective of the traffic generation, time consumption and space consumption, during the VM booting by using a 4 GB VDI with 64 KB chunk size.

The most frequent OTC-table query operation takes only 569.4 ms with no extra space consumption and traffic generation. Since we maintain this OTC table in local memory, the overhead of the OTC table accesses are far less than I/O operations which can be ignored.

The OTC-table update is another important operation. As a new VDI is registered in ACStor, we should divide the VDI file into chunks and broadcast their OID and CID messages to all nodes. In the OTC table, each entry consists of 192 bits, so we also need to transfer more than 24×31 Bytes (excluding the size of TCP/IP header) to other compute nodes in our

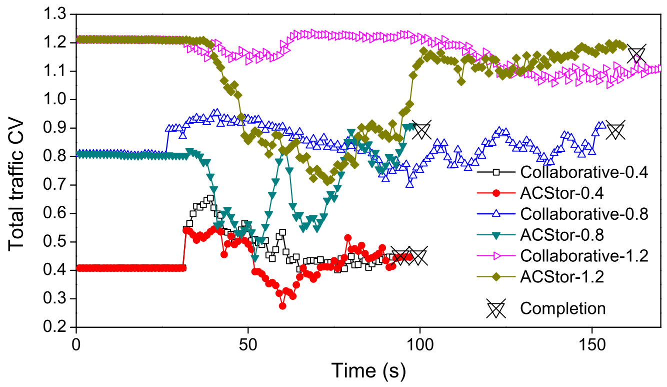


Fig. 10. The average traffic CV among compute nodes during monitoring of VMs booting.

32-node testbed. As for a 4 GB VDI which is divided into 64 KB chunks, we only need a 1.5 MB OTC table in each node. The overhead of synchronizing a OTC table with those on other nodes is about 46.5 MB. If we register two different 4 GB VDIs, the memory overhead of OTC table will be doubled. Since the update operations are not so frequent, the synchronization overhead is acceptable in most situations.

The last overhead is caused by the cache miss caused by re-mapping. Each cache miss generates more than 64 KB traffic, because it needs to fetch the chunk from back-end storage to public cache. We do not consider the normal cache miss, in that it will also appear in the traditional storages. The cache miss rate is related to the cross-similarity of VDIs [9]. Moreover, the larger the cross-similarity is, the more cache hits in the re-mapping procedure. Although the cache-miss ratio is higher than the traditional collaborative storage (since they do not have re-mapping, this rate is 0), ACStor still has the better I/O performance especially in the environments with high traffic CVs, because accessing the back-end storage will be more efficient than accessing the overloaded compute nodes sometimes.

5.8 Evaluation of Scalability

We evaluate the scalability of ACStor, by combining the real experiments and simulations.

In order to evaluate the performance in a more complex data center, we use CloudSim toolkit [26] to simulate a three-level tree topology data center with $4 \times 8 \times 8$ nodes. Specifically, we assign 1 Gbps bandwidth for edge and aggregate switches as well as 10 Gbps for root switch. The root switch connects four aggregate switches. Each of them connects eight racks, each consisting of eight nodes. For the back-end storage, we assume that the bandwidth from any host to it is 1 Gbps. We measure only the VM booting time in this set of experiments and the VM I/O patterns are set according to the trace generated in the real experiments. With scaling up of a data center, the inter-VM communications become more complex due to the topology. So, we use the following way to generate the extra traffic patterns. We let a VM in each node randomly choose 1 to 32 VMs to communicate with one another, and repeatedly generate many different traffic patterns (traffic CVs are 0.4, 0.8 and 1.2).

The evaluation results are presented in Fig. 11. The non-cache storage and local-cache storage exhibit stable with low performance under different traffic patterns. However, the performance of the collaborative-cache storage and our ACStor exhibit relatively high, though fluctuating significantly. This indicates the performance of collaborative storage system is not only related to the traffic CV among nodes, but also influenced by the traffic patterns in data

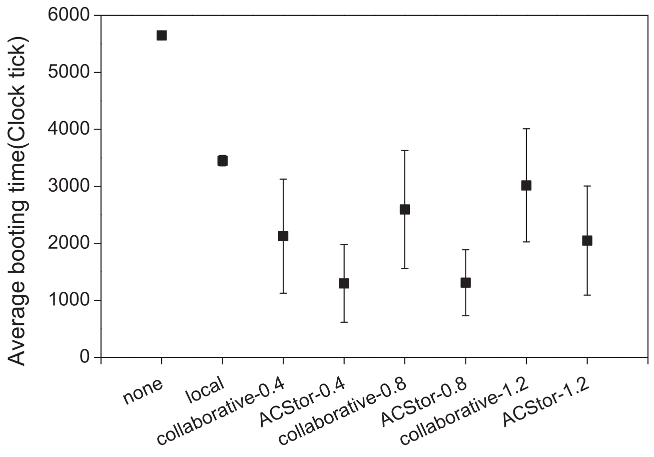


Fig. 11. Booting time in 128-nodes under various traffic patterns.

centers. For example, when a large number of VMs need to communicate with distant VMs (i.e., there are many hops between the source and the destination), the links associated with aggregate switches or root switch may be congested. Hence, the reads in ACStor will prefer to choose the neighboring hosts though they may be already overloaded. In the contrary, if most of inter-VM communications are short-distance traffic like [12], ACStor will have a big performance gain. As requests are forwarded to distant hosts, the links associated with aggregate switches or root switch will have no traffic bottleneck.

6 Discussion and Future Work

Alternative Caching Medium. ACStor uses local HDD in compute nodes to cache the VDIs in back-end storage, which is a cheap way to optimize the access performance of VDI. It is no doubt that using either *solid-state disk* (SSD) or in-memory caching of chunks will achieve better performance. However, the space of SSD and memory is far less than the normal HDD, so we need to improve the cache utilization further. For example, we plan to avoid the duplication between the private cache and the public cache in ACStor due to the usage of copy-on-write strategy. On the other hand, we also need to implement the data transmission in a high-speed network like Infiniband to match the access rate of high performance caches.

Cross-Similarity and Performance. In Section 5.7, we show the cross-similarity of VDIs in ACStor. More accurately, the I/O performance of VMs is related to the cross-similarity of chunks in different hosts rather than different VDIs. For example, in a single VDI environment, there are few same chunks within the same VDI (except the zero-filled chunks) [8]. So there must be lots of cache misses in the re-mapping procedure. In multi-VDI environment, it may also happen because of the low cross-similarity of VDIs or due to the fact that the chunks with same content are exactly hashed to the same host. In the future, we will focus on how to handle the I/O requests in this situation.

Mapping Policy Extensions. We show that the network traffic has effects on the remote chunk accesses and how ACStor deals with this negative effect by using mapping strategies. However, the strategies proposed may be not always efficient in all cases, especially when the targets and the sizes of I/O requests change frequently (as shown in Fig. 9b). In the future, we plan to add the I/O information to our mapping policy, in order to realize a fine-grained control.

Dedicated Storage Network. Some production environments may have dedicated storage and management networks. The uneven traffic pattern will become more complex in this situation. We will investigate how to optimize the I/O performance and improve the resource utilization further in this kind of heterogeneous network environment in our future work.

Transient Workload Spikes. We periodically collect network traffic. And we conduct the re-mapping operations only when there is an overloaded host in two successive collection. However, adopting workload prediction systems would be a better choice in the face of transient workload spikes. In the future, we will focus on a new re-mapping strategy which takes advantage of transient workload prediction.

Location Constrained Storage. Honestly, ACStor can not adapt to location constrained storage environments. The key reason is that one of the important features of our design is to dynamically adapt to diverse network load over time, while the environment with constrained storage locations has relatively stable network status because of the fixed data access amounts on each node. If we consider these requirements, ACStor will be rather complicated and our mapping strategy will be influenced greatly.

7 Related Work

Many studies aim to improve the I/O performance of VMs from the perspective of VDI storage system. A straightforward way is to cache a part of data from the VDIs in back-end storage to the local disks of the compute nodes. One representative work is *Fast Virtual Disk* (FVD) [27], which integrates the *Copy-on-Read* (CoR) mechanism and the data prefetching method to accelerate remote reads. Razavi et al. [3] also improve the traditional QCOW2 VDI format [28] by using on-demand CoR caches and validate the efficiency of a warm local cache. However, these approaches adopt only single-VDI caches which neglect the similarities across VDIs.

Considering the common data blocks across VDIs, an efficient way for VMs to access the local caches (content-based address) is provided in [20], by which the VMs booted from different VDIs are able to leverage the data blocks with the same content in local cache. The similar method is adopted by [9], which compresses the local caches by using the cross-similarity of VDIs. Through these methods, they can make full use of the small local caches. However, the hit ratio of local caches may decrease with increasing variety of VDIs. [29] propose a method called probabilistic deduplication to improve the resource utilization, though it is used in cluster-based storage systems.

Accordingly, a straightforward solution is to discard the centralized storage, such as [4], which aggregates the local disks of compute nodes to a distributed storage pool, substituting the original back-end storage device. Meyer et al. [30] and Hansen et al. [31] propose specific distributed file systems to store VDIs respectively. Their solutions split VDIs into small chunks and store them on distributed compute nodes. Such designs can effectively alleviate the I/O bottleneck of centralized storage. Another two solutions [6], [7] are proposed to make use of collaborative cache across compute nodes with centralized storage as back-end storage. Compared with the former methods, these two methods try to leverage the cross-similarity of VDIs in the distributed environment. Jayaram et al. [8] analyze the similarities across different VDIs by splitting whole VDI files into small chunks like [4]. They design a VDI storage system with collaborative cache in [5], which distributes VDI chunks to local disks of compute nodes in data centers and make chunks be shared in a P2P manner. They also optimize the chunk addressing by utilizing the topological information of a data center to reduce the overhead of remote chunk accesses. Unfortunately, it neglects the influence of dynamic traffic across compute nodes (e.g., generated by inter-VM communication), which may suffer from the performance degradation in a large-scale system. By contrast, our solution takes into account both the dynamic traffic load and the topology information, which can resolve the performance issue caused by the unbalanced network traffic very well.

Some previous researches are orthogonal to our work. For example, OpenFlow [32] provides bandwidth guarantee so as to moderate the I/O performance degradation issue caused by uneven network traffic, and [10] proposes a traffic-aware VM placement which also does good to traffic load balancing.

8 Conclusion

VDI Storage systems significantly affects overall performance of virtualized cloud platforms. In this paper, we propose Adaptive and Collaborative Storage System to optimize the I/O performance of virtualized cloud platforms. Our proposed solution employs a two-level cache to achieve the chunk sharing across different compute nodes. We devise an adaptive addressing mode to locate the requested chunk in compute nodes according to the traffic patterns by using mapping and re-mapping mechanisms. By taking into account both traffic load and the network topology, our mapping and remapping strategies are able to adaptively control the chunk accesses under the dynamic traffic in data centers.

We use a real cluster environment with 32 physical nodes and a large-scale simulated platform with up to 256 nodes to evaluate our solution, as well as three other state-of-the-art approaches. Based on our experiments, we have following key findings. First, VMs' I/O performance significantly depends on traffic status of data centers, especially when the traffic distribution exhibits unevenly across compute nodes. Second, ACStor significantly outperforms other state-of-the-art approaches (with performance gain up to $2-8\times$ than others), with respect to the VM booting time and VM's I/O throughput. Third, the total traffic CVs are significantly reduced under ACStor, which means ACStor has a better load balancing capability. Last, ACStor also exhibits an excellent scalability, with the increase on the number of VMs or nodes in data center.

Acknowledgments

This research was supported by 863 Hi-Tech Research and Development Program under grant No. 2015AA01A203, National Key Research and Development Program under grant 2016YFB1000501, and National Science Foundation of China under grants No. 61472151 and 61232008, and National Science Foundation of Anhui Province under grant No. 1608085QF147, and Key Project of Support Program for Excellent Youth Scholars in Colleges and Universities of Anhui Province under grant No. gxyqZD2016332. This research was also based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357. Sheng Di is the corresponding author.

References

[1] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The collective: A cache-based system management architecture," in *Proc. 2nd Conf. Symp. Netw. Syst. Des. Implementation*, May 2–4, 2005, pp. 259–272.

[2] VMware vSphere VMFS, 2016. [Online]. Available: http://www.vmware.com/files/pdf/vmfs-bestpractices-wp.pdf

[3] K. Razavi and T. Kielmann, "Scalable virtual machine deployment using VM image caches," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, Nov. 17–22, 2013, pp. 65:1–65:12.

[4] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu, "Going back and forth: Efficient multideployment and multisnapshotting on clouds," in *Proc. 20th Int. Symp. High Performance Distrib. Comput.*, Jun. 8–11, 2011, pp. 147–158.

[5] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *Proc. IEEE INFOCOM*, Mar. 25–30, 2012, pp. 181–189.

[6] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li, "Liquid: A scalable deduplication file system for virtual machine images," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1257–1266, May 2014.

[7] Z. Zhang, et al., "VMThunder: Fast provisioning of large-scale virtual machine clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3328–3338, Dec. 2014.

[8] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei, "An empirical analysis of similarity in virtual machine images," in *Proc. Middleware Industry Track Workshop*, Dec. 12–16, 2011, pp. 6:1–6:6.

[9] K. Razavi, A. Ion, and T. Kielmann, "Squirrel: Scatter hoarding VM image contents on IaaS compute nodes," in *Proc. 23rd Int. Symp. High Performance Distrib. Comput.*, Jun. 23–27, 2014, pp. 265–278.

[10] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, Mar. 14–19, 2010, pp. 1–9.

[11] H.-C. Hsiao, H.-Y. Chung, H. Shen, and Y.-C. Chao, "Load rebalancing for distributed file systems in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 5, pp. 951–962, May 2013.

[12] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee, "Application-aware virtual machine migration in data centers," in *Proc. IEEE INFOCOM*, Apr. 10–15, 2011, pp. 66–70.

[13] B. Pfaff, T. Garfinkel, and M. Rosenblum, "Virtualization aware file systems: Getting beyond the limitations of virtual disks," in *Proc. 3rd Conf. Symp. Netw. Syst. Des. Implementation*, May 8–10, 2006, pp. 353–366.

[14] J. Reich, et al., "VMTorrent: Scalable P2P virtual machine streaming," in *Proc. 8th Int. Conf. Emerging Netw. Experiments Technol.*, Dec. 10–13, 2012, pp. 289–300.

[15] X. Xu, H. Jin, S. Wu, and Y. Wang, "Rethink the storage of virtual machine images in clouds," *Future Generation Comput. Syst.*, vol. 50, pp. 75–86, 2015.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, Aug. 17–22, 2008, pp. 63–74.

[17] W. Hou and L. Guo, "Virtual network embedding for hybrid cloud rendering in optical and data center networks," *Sci. China Inf. Sci.*, vol. 59, no. 2, pp. 1–14, 2016.

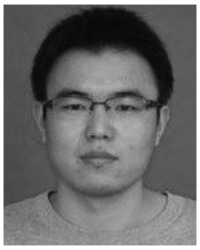
[18] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 1–3, 2010, pp. 267–280.

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide Web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, May 4–6, 1997, pp. 654–663.

[20] S. Tang, Y. Chen, and Z. Zhang, "Machine Bank: Own your virtual personal computer," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 26–30, 2007, pp. 1–10.
[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, Oct. 2001.
[22] G. DeCandia, et al., "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, Oct. 14–17, 2007, pp. 205–220.
[23] Apache cassandra, 2017. [Online]. Available: http://cassandra.apache.org/
[24] Sheepdog, 2017. [Online]. Available: http://sheepdog.github.io/sheepdog/
[25] Filebench, 2017. [Online]. Available: http://sourceforge.net/projects/filebench/
[26] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw.: Practice Experience*, vol. 41, no. 1, pp. 23–50, 2011.
[27] C. Tang, "FVD: A high-performance virtual machine image format for cloud," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, Jun. 15–17, 2011, pp. 18–18.
[28] M. McLoughlin, "The QCOW2 image format," 2016. [Online]. Available: https://people.gnome.org/ markmc/qcow-image-format.html
[29] D. Frey, A.-M. Kermarrec, and K. Kloudas, "Probabilistic deduplication for cluster-based storage systems," in *Proc. 3rd ACM Symp. Cloud Comput.*, Oct. 14–17, 2012, Art. no. 17.
[30] D. T. Meyer, et al., "Parallax: Virtual disks for virtual machines," in *Proc. 3rd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 31st Mar.–4th Apr., 2008, pp. 41–54.
[31] J. G. Hansen and E. Jul, "Lithium: Virtual machine storage for the cloud," in *Proc. 1st ACM Symp. Cloud Comput.*, Jun. 10–11, 2010, pp. 15–26.
[32] N. McKeown, et al., "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

Song Wu received the PhD degree from Huazhong University of Science and Technology (HUST), in 2003. He is a professor of computer science with Huazhong University of Science and Technology, China. He currently serves as the director of Parallel and Distributed Computing Institute, and the vice head of Service Computing Technology and System Lab (SCTS), HUST. He has published more than 100 papers and received more than 30 patents in the area of parallel and distributed computing. His current research interests include cloud resource scheduling, system virtualization, and large-scale datacenter management. He is a member of the IEEE.

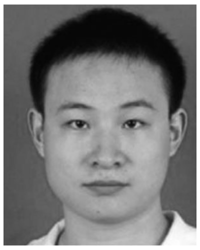
Yihong Wang received the MS degree from Huazhong University of Science and Technology (HUST), China, in 2015. His research interests include the area of cloud computing and virtualization, focusing on resource management in the cloud.



Wei Luo is working toward the master's degree in the Service Computing Technology and System Lab (SCTS), Cluster and Grid Computing Lab (CGCL), Huazhong University of Science and Technology, China. His research interests include virtualization and distributed storage.

Sheng Di received the master's degree from Huazhong University of Science and Technology, in 2007 and the PhD degree from the University of Hong Kong, in 2011. He is currently working with Argonne National Laboratory. His research interests involves resilience on high-performance computing (such as silent data corruption, optimization of checkpoint model, characterization and analysis of supercomputing log, and in-situ data compression) and broad research topics on cloud computing (including optimization of resource allocation, cloud network topology, and prediction of cloud workload/hostload). He is a member of the IEEE.

Haibao Chen received the PhD degree in computer architecture from Huazhong University of Science and Technology, China. He is now serving as the director of Department of Network and Communication Engineering, Chuzhou University, China. His research interests include system virtualization, cloud computing, and wireless sensor network.



Xiaolin Xu received the MS and PhD degrees from Huazhong University of Science and Technology (HUST), China, in 2009 and 2015, respectively. His research interests include the area of cloud computing and virtualization, focusing on resource management in the cloud.

Ran Zheng received the MS and PhD degrees from Huazhong University of Science and Technology, China, in 2002 and 2006, respectively. She is currently an associate professor of computer science and engineering with Huazhong University of Science and Technology (HUST), China. Her research interests include distributed computing, cloud computing, high-performance computing, and applications. She is a member of the IEEE.

Hai Jin received the PhD degree in computer engineering from HUST, in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with Huazhong University of Science and Technology (HUST), China. He is currently the dean of the School of Computer Science and Technology, HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He has coauthored 15 books and published more than 500 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a senior member of the IEEE and a member of the ACM.