VMAR: Optimizing I/O Performance and Resource Utilization in the Cloud

Zhiming Shen[1,⋆], Zhe Zhang[2], Andrzej Kochut[2], Alexei Karve[2], Han Chen[2], Minkyong Kim[2], Hui Lei[2], and Nicholas Fuller[2]

[1] Cornell University
zshen@cs.cornell.edu
[2] IBM T. J. Watson Research Center
{zhezhang,akochut,karve,chenhan,minkyong,hlei,nfuller}@us.ibm.com

Abstract. A key enabler for standardized cloud services is the encapsulation of software and data into VM images. With the rapid evolution of the cloud ecosystem, the number of VM images is growing at high speed. These images, each containing gigabytes or tens of gigabytes of data, create heavy disk and network I/O workloads in cloud data centers. Because these images contain identical or similar OS, middleware, and applications, there are plenty of data blocks with duplicate content among the VM images. However, current deduplication techniques cannot efficiently capitalize on this content similarity due to their warmup delay, resource overhead and algorithmic complexity.

We propose an instant, non-intrusive, and lightweight I/O optimization layer tailored for the cloud: *V*irtual *M*achine I/O *A*ccess *R*edirection (VMAR). VMAR generates a block translation map at VM image creation / capture time, and uses it to redirect accesses for identical blocks to the same filesystem address before they reach the OS. This greatly enhances the cache hit ratio of VM I/O requests and leads to up to 55% performance gains in instantiating VM operating systems (48% on average), and up to 45% gain in loading application stacks (38% on average). It also reduces the I/O resource consumption by as much as 70%.

1 Introduction

The economies of scale of cloud computing, which differentiates it from transitional IT services, comes from the capability to elastically multiplex different workloads on a shared pool of physical computing resources. This elasticity is driven by the standardization of workloads into moveable and shareable components. To date, virtual machine images are the *de facto* form of standard templates for cloud workloads. Typically, a cloud environment provides a set of "golden master" images containing the operating system and popular middleware and application software components. Cloud administrators and users start with these images and

⋆ This work was conducted when Zhiming Shen was an intern at IBM and a Ph.D. student at North Carolina State University.

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 183–203, 2013.

create their own images by installing additional components. Through this process, a hierarchy of deviations of VM images emerges. For example, in [24], Peng *et al.* have studied a library of 355 VM images and constructed a hierarchical structure of images based on OS and applications, where the majority of images contain Linux with variation only on minor versions (i.e., v5.X).

Today's production cloud environments are facing an explosion of VM images, each containing gigabytes or tens of gigabytes of data. As of August 2011 Amazon Elastic Compute Cloud (EC2) has 6521 *public* VM images [4] (data on *private* EC2 VM images is unavailable). Storing and transferring these images introduces heavy disk and network I/O workloads on storage and compute/hypervisor servers. On the other hand, the evolutionary nature of the VM "ecosystem" determines that different VM images are likely to contain identical chunks of data. It has been reported that a VM repository from a production cloud environment contains around 70% redundant data chunks [15]. This has indicated rich opportunities to deduplicate the storage and I/O of VM images.

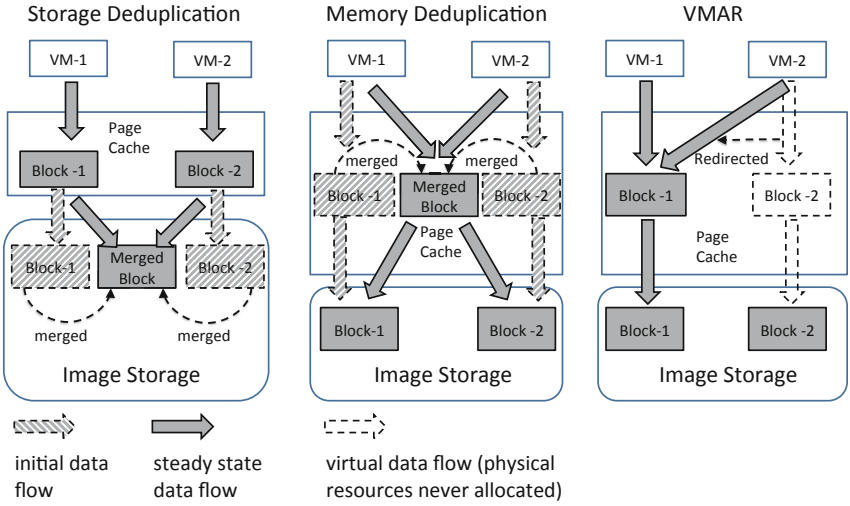


Fig. 1. Comparison of storage deduplication, memory deduplication, and `VMAR`.

To exploit this content redundancy, storage deduplication techniques have been actively studied and widely used [10,11,12,13,21,26,31]. As illustrated in Figure 1, storage deduplication mostly works on the block device layer and merges data blocks with identical content. The scope of storage deduplication is mainly to save storage capacity rather than to optimize the performance and resource consumption of I/O operations. As a matter of fact, most of them cause various degrees of overhead to both write and read operations.

On the other hand, memory deduplication techniques [5,7,16,18,28,29] save memory space by scanning the memory space and compressing identical pages. They also reduce the I/O bandwidth consumption by improving cache hit ratio. However, existing memory deduplication methods suffer from 2 fundamental

drawbacks when applied to VM I/O optimization. First, savings can only be achieved after a "warm-up" period where similar data chunks are brought in the memory and become eligible for merging. Second, the merging process, including content identification, page table modification, as well as the copy-on-write logic (triggered when a shared page is updated), requires complex programs and competes with primary applications for computing resources.

As an alternative, this paper proposes VMAR, an *instant*, *non-intrusive*, and *lightweight* I/O optimization method tailored for cloud environments. VMAR is based on the idea of *V*irtual *M*achine I/O *A*ccess *R*edirection. It is a lightweight extension to the virtualization layer that can be easily deployed into the cloud incrementally, and does not need any modification to the guest OS or application stack. Compared to existing deduplication and I/O optimization methods, VMAR has two key distinctions.

1. *Ahead* of I/O requests: VMAR detects identical data blocks *when VM images are captured* and generates a block translation map. This way, even before a VM starts running, VMAR has rich knowledge on its future I/O accesses and is capable of linking them to other VMs' data blocks. The data hashing and comparison can be done lazily because a VM image is typically captured when the VM using it has just been terminated. By batching these operations at image capture time, VMAR also avoids keeping large amount of hash values as deduplication metadata at compute nodes.
2. *Upstream* in the I/O architecture: Using the block translation map, VMAR redirects VM read accesses for identical blocks to the same filesystem address *above the hypervisor Virtual Filesystem (VFS) layer*, which is the entry point for all file I/O requests into the OS. Since I/O operations are merged from the upstream instead of on the storage layer, each VM has a much higher chance to hit the file system page cache, which is already "warmed up" by its peers. The reduction of warmup phase is critical to cloud user experience, especially in development and test environments where VMs are short-lived.

We have implemented VMAR as a QEMU image driver. Our evaluation shows that in I/O-intensive settings VMAR reduces VM boot time by $39 \sim 55\%$ (48% on average) and application loading time by $24 \sim 45\%$ (38% on average). It also saves up to 70% of I/O traffic and memory cache usage.

The reminder of this article is organized as follows. Section 2 provides a background of VM image I/O. Section 3 details the design and implementation of VMAR. Section 4 presents the evaluation results. Section 5 surveys related work in storage and memory deduplication. Finally, section 6 concludes the paper.

2 Background

Most virtualization technologies present to VMs a *virtual disk* interface to emulate real hard disks (also known as *VM image*). Virtual disks typically appear as regular files on the hypervisor host (i.e., image files). I/O requests received at virtual disks are translated by the virtualization driver to regular file I/O requests to the image files.

Due to the large amount and size of VM images, it is impossible to store all image files on every hypervisor host. A typical cloud environment has a shared image storage system, which has a unified name space and is accessible by each hypervisor host. One commonly used architecture is to set up the shared storage system on a separate cluster from the hypervisor hosts, and connect the storage and hypervisor clusters via a storage area network. Another emerging scheme is to form a distributed storage system by aggregating the locally attached disks of hypervisor hosts [14]. In either scenario, when a VM is to be started on a hypervisor host, the majority of its image data is likely to be located remotely.

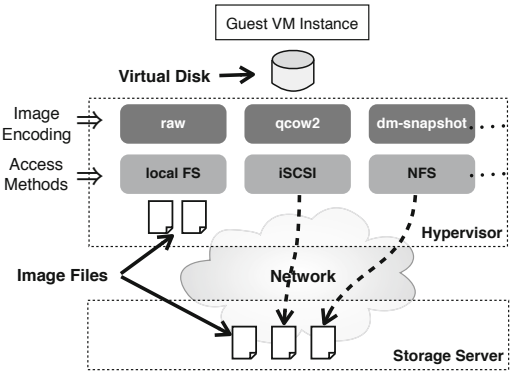


Fig. 2. Different configurations of virtual disks

Figure 2 illustrates different combinations of virtual disk configurations. First, VM images can be stored in different formats. The most straightforward option is the *raw* format, where I/O requests to the virtual disk are served via a simple block-to-block address mapping. In order to support multiple VMs running on the same base image, copy-on-write techniques have been widely used, where a local snapshot is created for each VM to store all modified data blocks. The underlying image files remain unchanged until new images are captured. As shown in Figure 2, there are different copy-on-write schemes, including Qcow2 [1], dm-snapshot, FVD [27], VirtualBox VDI [3], VMware VMDK [2], and so forth.

The second dimension of virtual disk configuration is how VM images are accessed. One way is to *pre-copy* the entire image from the image storage to the local file system of the target hypervisor before starting up a VM instance. Since a typical VM image file contains multiple gigabytes, or even tens of gigabytes of data, it may take a long time to start up a VM instance under this scheme. To overcome this problem, an alternative method is to fetch parts of a VM image from the storage system *on-demand*. Under the *on-demand* configuration, image data may need to be fetched from the remote storage during runtime, causing extra delay. However, as shown in [8], the runtime performance degradation is very limited. Therefore, in the rest of the paper, we have focused on applying `VMAR` on top of the *on-demand* configuration.

3 Design and Implementation

Figure 3 illustrates how VMAR interacts with a VM during its lifetime. First, when a new VM image is inserted into the image repository, either copied from external sources or captured from the disk of a VM, VMAR compares it against the existing images in the repository. It then generates the meta-data of the new image, including a block map that identifies common blocks between this image and other images in the repository. Section 3.1 discusses details of the block map generation process. When a new VM is created from the base image, the meta-data is forwarded to the compute node, and an image in VMAR format is created. With the VMAR images serving as the backing files for the Qcow2 images, I/O accesses to VM images are redirected and consolidated. Section 3.2 describes the access redirection mechanism. Finally, Section 3.3 presents techniques to optimize block map size and lookup performance.

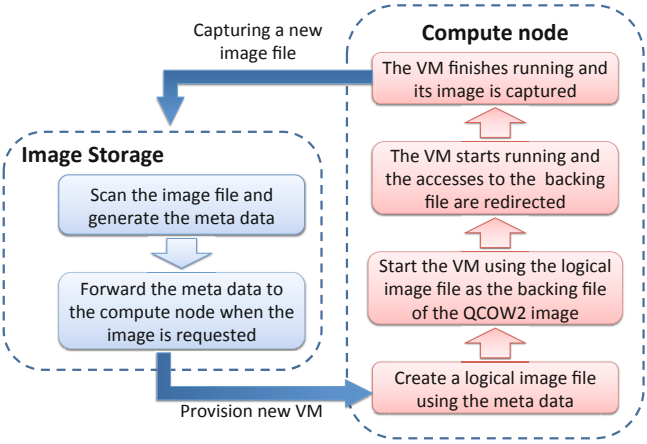


Fig. 3. Flow of VMAR

3.1 Hash-Based Block Map Generation

The block map generator of VMAR uses 4 KB blocks as the base unit. Each data block is identified by its hash value as the fingerprint. In capturing the content similarities among VM images, we leverage the concept of metadata *clusters* proposed in [17]. Each cluster represents the set of blocks that are common across a subset of images. The main benefit of using clusters in VMAR is that they greatly facilitate the search of all VM images having content overlaps with a given image. Therefore, when an image is modified or deleted from the image repository, it is easy to identify entries in the block map that should be updated.

For completeness we first briefly describe the concept of metadata clusters. Consider a simple example of three images: Image-0, Image-1 and Image-2 as shown in Fig. 4. In this illustration, CL-001, CL-010, CL-100 are singleton clusters, containing the blocks only from Image-0, 1 and 2, respectively. For example,

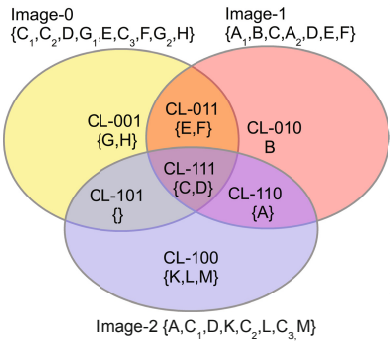


Fig. 4. Illustration of clusters for three example images

block with hash G is unique to Image-0. CL-011 is the cluster with blocks from Image-0 and 1, which have hash values E and F. We use subscripts to denote identical blocks within an image. For example, hash value C appears in Image-0 3 times, as C_1, C_2 and C_3.

When a new image is added to the library, the system computes the SHA1 hash for each block and compare it against existing clusters. Then each cluster is divided into two new clusters: one that contains the new block and another one that doesn't. The hash values in the new images that do not belong to the any current clusters are put into a new singleton cluster. A certain hash value can appear in multiple images. The block mapping protocol should be consistent and and ensure all requests for identical blocks are redirected to the same address. For this purpose we always use the image with the smallest sequence ID as the mapping destination. Alternative consistent mapping protocols can be considered as future work – for instance, the least fragmented image [20] or the most used image can be used as the target. These optimizations can potentially improve I/O sequentiality.

Hash	Contained images	Block list
C	Image-0	0,1,5
	Image-1	2
	Image-2	1,4,6
D	Image-0	2
	Image-1	4
	Image-2	2

Fig. 5. Meta-data of cluster CL-111

Block number	Hash	Cluster	Target image	Target block number
0	A	CL-110	Image-1	0
1	B	CL-010	Image-1	1
2	C	CL-111	Image-0	0
3	A	CL-110	Image-1	0
4	D	CL-111	Image-0	2
5	E	CL-011	Image-0	4
6	F	CL-011	Image-0	6

Fig. 6. Block map for Image-1

Fig. 5 illustrates the meta-data of cluster CL-111. Cluster CL-111 contains two hash values that are shared by all three images, so the accesses to any block belonging to CL-111 should be redirected to Image-0, which has the smallest ID. It is possible that there are multiple blocks having the same hash value in

```
function update_block(s, block)
  hash_prev: previous hash value of the block;
  hash_new: new hash value of the block;
  add s to update_list;
  let c = find_cluster_from_hash(hash_prev);
  remove block in the block list of hash_prev for image s in c;
  if the updated block list becomes empty:
    move the entry of hash_prev in c to the corresponding cluster;
  if the minimal image ID containing hash_prev is changed:
    for each image t that contains hash_prev do:
      add t to update_list;
    end for;
  let c' = find_cluster_from_hash(hash_new);
  if c' = None:
    add block into singleton of s;
  else:
    add block to c';
    if s does not contain c':
      move the entry of hash_new in c' to the corresponding cluster;
    if the minimal image ID containing hash_new is changed:
      for each image t that contains hash_new do:
        add t to update_list;
      end for;
  for each image t in update_list do:
    re-construct the block map for image t;
  end for;
end function;
```

Fig. 7. Pseudo-code of updating an image

Image-0, such as the blocks with hash value C. In this case, we always map them to the block with the smallest block number. For example, in the illustrated case, any block with hash value C will be mapped to block 0 in Image-0. Given the hash value of a block, we can quickly identify the target image and block we should map by looking up the hash table in each cluster. Fig. 6 shows the map for Image-1 and the cluster meta-data we use to construct the map.

The method update_map in Fig. 7 is executed when a VM image is updated. It search for all other images having blocks pointing to this image with the cluster data structure, and consequently update the map entries. A hash value can also be moved to another cluster if the ownership is changed due to the update.

Finally, to illustrate the offline computational overhead for creation of clusters and map, that is a one time cost to prepare the image library for redirection, we have run an experiment on a VM with 2.2 GHz cpu and 16 GB memory. We have used an image library with 84 images with total size of 1.5 TB. The images were a mix of Windows and Linux images of varying sizes ranging between 4 GB and 100 GB (used in a production Cloud). This image library resulted in creation of 453 clusters. The total time to create the clusters and mappings for all images was 15 minutes.

3.2 I/O Deduplication through Access Redirection

Figure 8 illustrates the overall architecture of VMAR's access redirection mechanism. The VMAR image serves as the backing file of the Qcow2 image. When a read request R is received by the QEMU virtual I/O driver, the copy-on-write

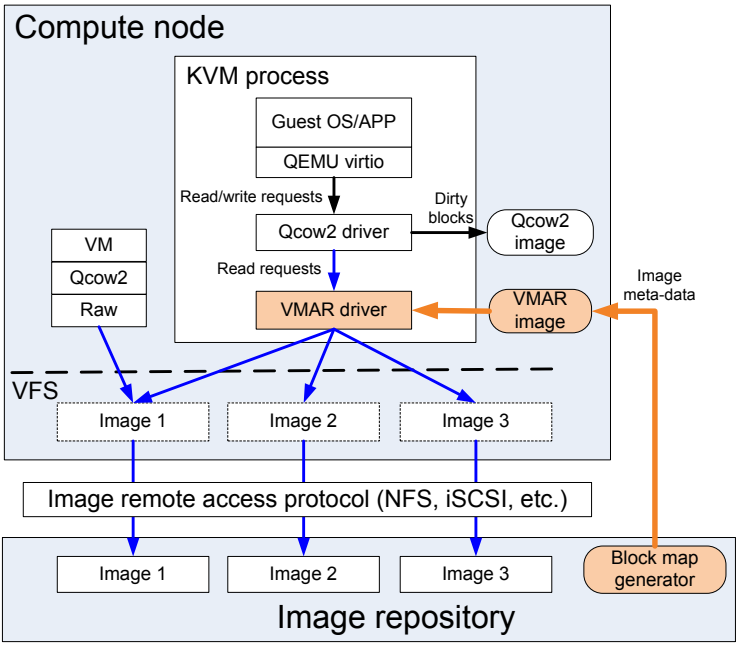


Fig. 8. Architecture of `VMAR`

logic in Qcow2 first checks whether it is for base image data or VM private/dirty data. If R is for VM private/dirty data, Qcow2 forwards the request to a local copy-on-write file. If R is for base image data, the Qcow2 driver forwards the request to the backing image. In both cases, R is translated as a regular file request which is handled by the VFS layer of the host OS. Unless the file is opened in direct I/O mode, R will be checked against the host page cache before being sent to the host hard disk drive.

The `VMAR` image driver implements address translation and access redirection. When a read request R is received, `VMAR` looks up the block map introduced in Section 3.1 to find the destination addresses of the requested blocks. If the requested blocks belong to different base images, or are noncontinuous in the same base image, then R is broken down into multiple smaller "descendant" requests. The descendant requests are sent to the corresponding base images. Upon the completion of all descendant requests, the `VMAR` driver returns the whole buffer back to the Qcow2 driver.

The descendant requests are issued concurrently to maximize throughput. We leverage the asynchronous I/O threadpool in the KVM hypervisor to issue concurrent requests. To serve a request R, the application's buffer is divided into multiple regions and a set of I/O vectors are created. Each I/O vector represents a region of the buffer and fills the region with the fetched data. A counter for the application buffer keeps track of the number of issued and completed descendant requests. The last callback of the descendant request will return the buffer back to the application.

VMAR updates the *inode* numbers of the descendant requests of R to the destination / redirected base image files before sending them to the host OS VFS layer and checked against the page cache. If the corresponding blocks in the destination files have been read into the page cache by other VMs, the new requests will hit the cache as "free riders". As discussed in Section 3.1, if a block appears in multiple images, the block map entry always points the image with the smallest ID. Therefore, all requests for the same content are always redirected to the same destination address, which increases the chance of "free riding".

VMAR redirects accesses to VM images, but not to private/dirty data. The reason is twofold. First, the data generated during runtime has a much smaller chance to be shared than that of the data in the base images, which contain operating systems, libraries and application binaries. Second, deduplication of private/dirty data incurs significant overhead because the content of each newly generated block has to be hashed and compared to existing blocks during runtime.

3.3 Block Map Optimizations

Block Map Size Reduction. A straightforward method to support redirection lookup is to create a block-to-block map. Based on the offset of the requested block in the source image, we can calculate the position of its entry in the block map directly. Each map entry has two attributes: {ID_{target}, Block_{target}}. The lookup of block-to-block map is fast. However, the map size will grow linearly with the image size. For example, Figure 9 shows that the map size for a 32 GB image can grow up to 64 MB before optimization.

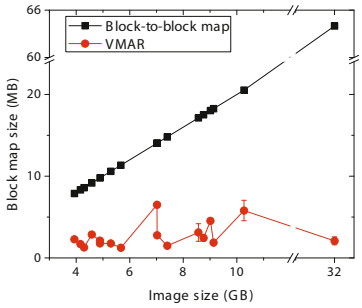


Fig. 9. Block map size optimization.

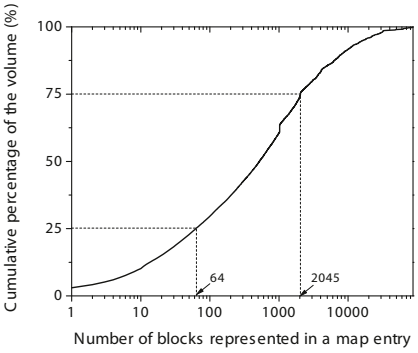


Fig. 10. CDF of the volume in the clusters with different sizes.

To reduce the map size and increase the scalability, we merge the map entries for the blocks that are continuous in the source image, and are also mapped continuously into the same target image. Since they are mapped continuously, we can use a single entry with four attributes to represent all of them: {offset_{source}, length, ID_{target}, offset_{target}}. Note that the length that each entry represents may be different. Thus, the lookup of the map requires checking whether a given block number falls into the range represented by an entry.

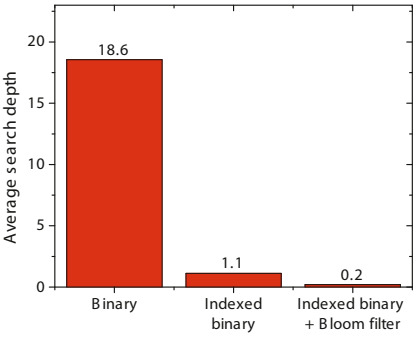


Fig. 11. Average binary search depth for each search scheme

To further reduce the map size, we also eliminate map entries for zero blocks. If a block cannot find a corresponding entry in the map, it is a zero block. In this case, the `VMAR` driver simply uses *memset* to create a zero-filled memory buffer. This saves the time and bandwidth overheads of a full memory copy.

Figure 9 shows that after optimization, the map size for `VMAR` is reduced significantly (mostly under 5 MB). In the VM images we have worked on, many continuous clusters have been detected. This is because the common sharing granularity between pairs of VM images is the files stored on their virtual disks. For example, the ram-disk file of the kernel, application binaries and libraries. Figure 10 presents the cumulative percentage of the the number of blocks represented in a single map entry. Map entries containing more than 64 blocks cover around 75% of the blocks. Some "big" map entry covers a significant portion of blocks. For example, map entries with a size more than 2,045 blocks covers around 25% of blocks.

Block Map Lookup Optimization. After the above optimization for the map size, each map entry represents different lengths. Thus, we cannot perform a simple calculation to get the position of the desired map entry. A linear search is inefficient. Note that the block map is sorted according to the source block offset. So we adopt *binary search* as the basic lookup strategy.

Since we still have many entries in the map, the depth of the binary search is typically high. So we have applied two mechanisms to further reduce the lookup time. First, we create an index to divide a large map into equal-sized sections. Each index entry has two pointers pointing the first and the last entry in the map that covers the corresponding section. Since the sections are equal-sized, given a block offset we can directly calculate the corresponding index entry. From the index entry, we can get the range within which we should perform binary search. This mechanism reduces search depth significantly. Second, to avoid searching to the maximum depth for zero blocks, we use a bloom filter to quickly identify them. Figure 11 shows the average search depth during the VM instantiation and application loading stage. We can see that our optimization mechanisms reduce the average search depth from 18.6 to 0.2.

4 Evaluation

4.1 Experiment Setup

We have implemented VMAR based on QEMU-KVM 0.14.0, and conducted the experiments using two physical hosts. Each host has two Intel Xeon E5649 processors (12 MB L3 Cache, 2.53 GHz) with 12 hyper-threading physical cores (24 logical cores in total), 64 GB memory, and gigabit network connection. The hosts run Red Hat Enterprise Linux Server (RHEL) release 6.1 with kernel 2.6.32 and libvirt 0.8.7. One host serves as the image repository and the other one is the compute node on which the VMs will be created. The compute node accesses the images repository using the iSCSI protocol.

To drive the experiments, we have obtained a random subset of 40 images from a production enterprise cloud. The size of the images ranges from 4 GB to over 100 GB. The VMs are instantiated using libvirt. Each VM is configured with two CPU cores, 2 GB memory, bridged network and disk access through virtio in the Qcow2 format. 23 of the images run RHEL 5.5, and 17 of them run SUSE Linux Enterprise Server 11.

The impact of VMAR on the VM instantiation performance is assessed by starting VMs from the images and measuring the time it takes before the VMs can be accessed from the network. This emulates the service response time that a customer perceives for provisioning new VMs in an Infrastructure as a Service (IaaS) cloud. In each image, we have added a simple script to send a special network packet right after the network is initialized. Most time is spent on booting up the OS and startup services. A daemon on the compute node waits for the packet sent by our script and records the timing.

After VM instantiation, another time-consuming step in cloud workload deployment is to load the application software stack into the VM memory space. This can take even longer in complex enterprise workloads, where a software installation (e.g., database management system) contains hundreds of megabytes or gigabytes of data. Due to the lack of semantic information on the production images, we added four additional images into the repository. On each image, we installed IBM DB2 database software version X and WebSphere Application Server (WAS) version Y, where $X \in \{9.0, 9.1\}$ and $Y \in \{7.0.0.17, 7.0.0.19\}$ [1]. These images run RHEL 6.0 and use the same VM configuration as other images. We have measured the application software loading time in the four images, while instantiating other images as a background workload.

As discussed in section 2, our evaluation uses the *on-demand* policy as the *baseline* configuration, where VM images stay in the storage server and the compute node obtains required blocks through the iSCSI protocol. Besides VMAR we have also included *lessfs* [19] and *KSM* [5] in the evaluation, which are widely used storage and memory deduplication mechanisms for Linux. Therefore, the rest of this section compares 4 configurations to the *baseline*: 1) VMAR used to start VMs on compute node; 2) *lessfs* used on storage server to store VM images; 3) *KSM* used on compute node to merge memory pages (*KSM* is triggered

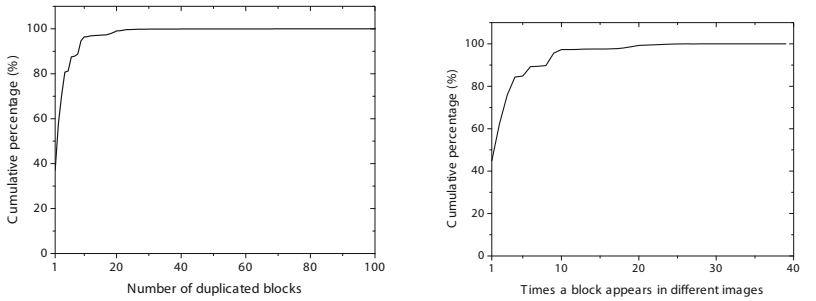
[1] DB2+WAS is commonly used in online transaction processing (OLTP) workloads.

only when the system is under memory pressure, therefore only evaluated in such settings); 4) *lessfs* (on storage server) +VMAR (on compute node). The first 3 configurations represent the typical usage of the individual optimization techniques. The fourth configuration explores using VMAR on top of storage deduplication to save both storage and I/O resources.

In our experiments, the arrival of VM instantiation commands follows a Poisson distribution. Different Poisson arrival rates have been used to emulate various levels of I/O workload. Each experiment is repeated three times and average values are reported with the standard deviation as error bars.

4.2 Experiment Results

This section shows the experiment results, including an analysis of content similarity in the VM image repository we use, the results for VM instantiation and application loading, and the overhead of VMAR.



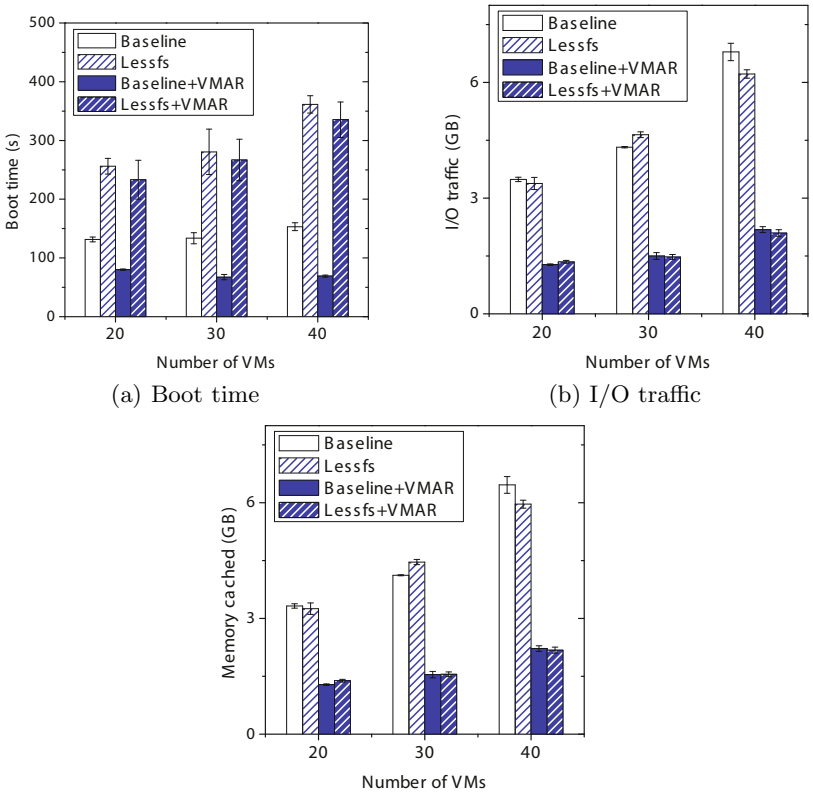
(a) Block duplication in the whole repository

(b) Inter-image duplication

Fig. 12. Image blocks similarity statistics

Similarity in the Image Repository. We first analyze the content similarity among our 40 images. In this analysis, we only consider non-zero data blocks. Figure 12(a) shows the CDF of the number of duplicated blocks in the entire repository of 40 images. More than 60% of the blocks are duplicated at least twice, and 10% of the blocks are duplicated more than eight times. This verifies the intuition that duplicated blocks are common in the VM image repositories of production clouds. A block can be duplicated within the same image, or across different images. Figure 12(b) shows the CDF of of the number of times that a block appears in different images. More than 50% of the blocks are shared by at least two images. Around 25% of the blocks are shared by more than three images. Therefore, opportunities are rich for VMAR to deduplicate accesses to identical blocks.

VM Instantiation Figure 13 shows the performance and resource consumption of VM instantiation when different numbers of VMs are booted. In this experiment, a new VM is provisioned every five seconds on average. During the



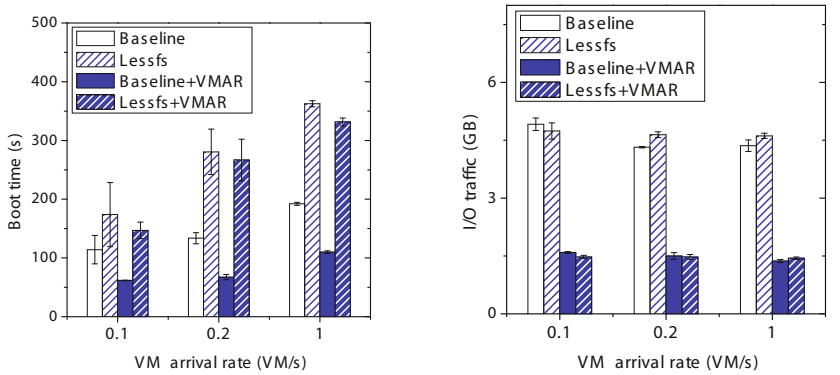
(a) Boot time (b) I/O traffic

(c) Memory cached

Fig. 13. Comparison of performance and resource utilization in VM instantiation, with different number of VMs

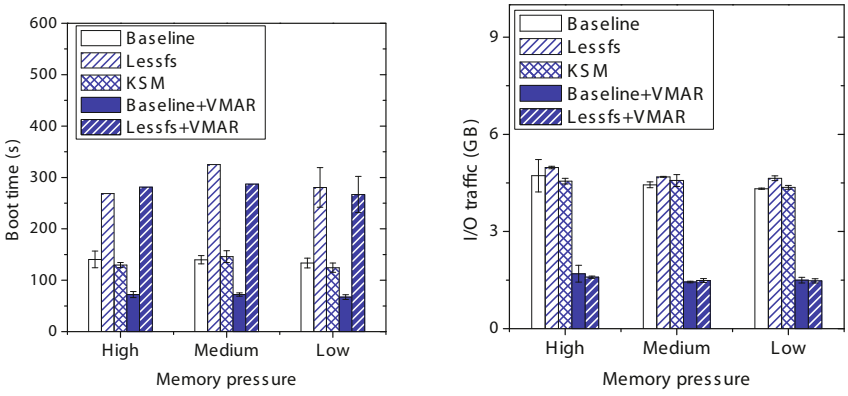
VM instantiation phase, the majority of the I/O workload is to load the OS into the VM's memory, causing few data re-accesses *within a single VM*. Therefore, under the *baseline* configuration, almost every read request goes through the network and the disk, and the data block eventually enters the memory cache of the compute node. As shown in Figures 13(b) and 13(c), the amount of I/O traffic and memory cache space usage are roughly the same, both increasing almost linearly with the number of VMs. Consequently, as shown in Figure 13(a), the average time it takes for a VM to boot up is over 100 seconds. The boot time increases when more VMs are booted, causing the disk and the network to be more congested.

With VMAR, each VM benefits from the data blocks brought into the hypervisor's memory page cache by other VMs that are booted earlier. Therefore, the average boot time is significantly reduced (by $39 \sim 55\%$). Moreover, the average boot time with VMAR decreases when more VMs are booted and the cache is "warmer". VMAR also reduces I/O traffic and memory consumption by $63 \sim 68\%$, by trimming unnecessary disk and network accesses up in the memory cache.



(a) Boot time (b) I/O traffic

Fig. 14. Comparison of performance and I/O traffic in VM instantiation, with different VM arrival rates

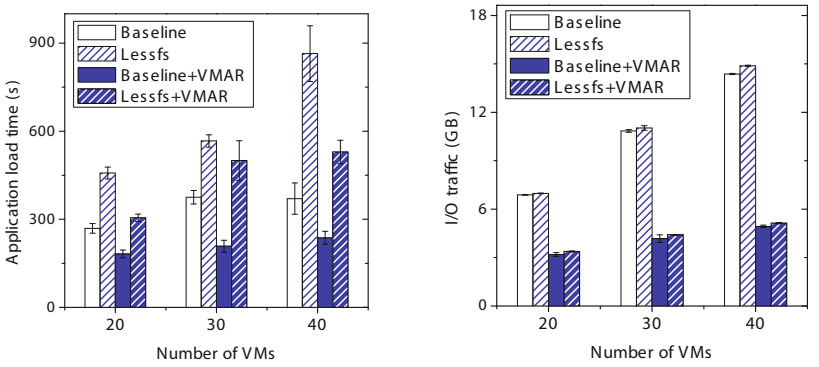


(a) Boot time (b) I/O traffic

Fig. 15. Comparison of VM boot time and I/O traffic in VM instantiation, with different available memory sizes

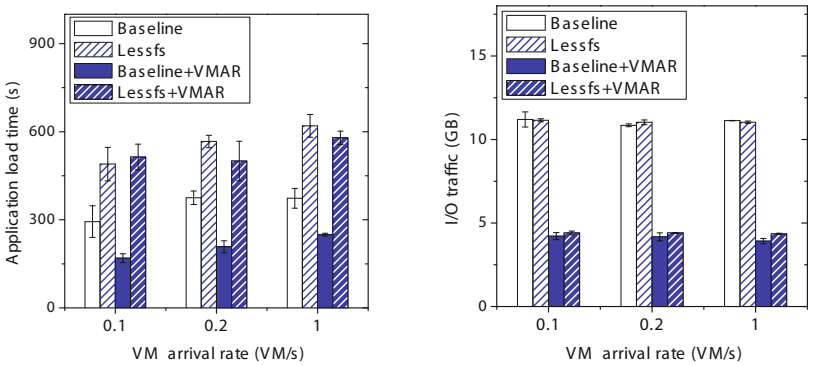
More importantly, the I/O traffic grows at a much slower rate than the *baseline* because the amount of "unique" content in every incoming VM image drops quickly as the hypervisor hosts more images. This is a critical benefit in resource overcommitted cloud environments.

With *lessfs*, the I/O traffic and memory cache usage are about the same as the *baseline*. This is because *lessfs* compresses data on the block storage layer, which is below VFS and thus doesn't change cache hit/miss events or the number of disk I/O requests. The VM boot time is worse than the *baseline*, mainly because it runs in the user space (based on *FUSE*), and incurs high context switch overhead. Deduplication techniques implemented in the kernel could have smaller overhead, but similar to *lessfs*, they will not improve filesystem cache performance and utilization. When *lessfs*+`VMAR` is used, the majority of I/O



(a) Application load time (b) I/O traffic

Fig. 16. Comparison of performance and I/O traffic in application loading, with different number of VMs

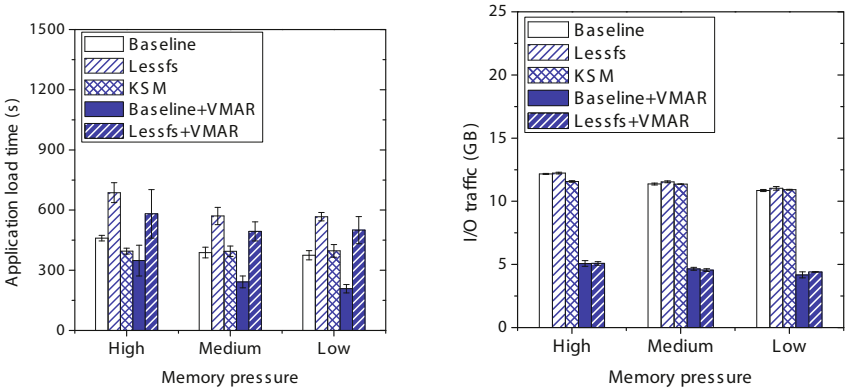


(a) Application load time (b) I/O traffic

Fig. 17. Comparison of performance and I/O traffic in application loading, with different VM arrival rates

requests hit the page cache of the compute node without reaching to *lessfs* at all. This improves the boot time results. However, the degree of performance improvement (7% on average) is much lower than the saving in I/O traffic (66.6% on average). This is because both VMAR and *lessfs* break sequential I/O patterns, thereby exaggerating the context switch and disk seek overhead. To mitigate this issue, replica selection optimizations similar to [18] can be investigated as interesting future work.

Figure 14 presents the performance and resource consumption of VM instantiation under different VM arrival rates, while the total number of instantiated VMs is fixed at 30. Figure 14(a) shows the average boot time when a new VM is provisioned every $\{10 - 5 - 1\}$ seconds on average. Since higher VM arrival rates lead to more severe I/O contentions, the average boot time with the *baseline* scheme increases quickly. In contrast, with the help of VMAR, a lot of disk



(a) Application load time (b) I/O traffic

Fig. 18. Comparison of performance and I/O traffic in application loading, with different available memory sizes

accesses from the VMs hit the memory cache and return directly without triggering any real device access. Therefore, in comparison to the *baseline*, the average boot time with `VMAR` is much lower, and increases slowly with the arrival rate. Figure 14(b) shows that the VM arrival rate does not significantly affect the total amount of I/O traffic [2]. This confirms that the increase in boot time under *baseline* is due to the increased I/O contention, which is mitigated by `VMAR`. Finally, it can be observed that the overhead of *lessfs* grows fast with the level of I/O contention.

Figure 15 presents the performance and I/O traffic of VM instantiation with different available memory sizes on the host. In this experiment, the number of VMs is set to 30 and the arrival rate is set to 0.2. From previous experiments, which uses all 64 GB memory, we observe that the memory usage of the host during runtime is around 11 GB, 4 GB of which is for caching. Thus, we test the scenarios where the available memory size is 9 GB and 11 GB respectively. Under all configurations, the instantiation time is insensitive to memory pressure, and the reason is twofold. First, `VMAR` has consolidated the I/O traffic and only requires very small amount of memory ($\sim$1.5 GB) to cache all I/O requests, which can be satisfied even under high memory pressure. Second, without `VMAR`, data re-access rate is very low, which diminishes the benefit of abundant memory. The *page_sharing* counter of *KSM* indicates that it saves $\sim$ 3.5 GB of memory by compressing similar pages. However, because the saving is achieved after the data blocks are loaded into memory, it incurs almost the same amount of I/O traffic as *baseline*, and therefore does not lead to notable performance improvement.

Application Loading. Figures 16, 17, and 18 show the results of application loading performance and I/O traffic. Again, in Figure 16, $\{20 - 30 - 40\}$ VMs are booted with a fixed arrival rate of 0.2; in Figure 17, the number of VMs is set

[2] In the rest of this section, memory usage results will be omitted because they are similar to the amount of I/O traffic.

to 30, and $\{10-5-1\}$ VMs are booted every second; in Figure 18, 30 VMs are booted at a rate of 0.2, under different memory pressures. As discussed above, we replace 4 of the production images with 4 new images installed with different versions of IBM DB2 and WAS, and only measure the application loading time of the 4 images. Other VMs serve as the background workload.

Loading enterprise applications is an I/O intensive workload, where a large number of application binaries and libraries are read into the memory. The 4 images we measure contain different versions of the same application stack, and thus share a lot of data blocks. Therefore, the results demonstrate a similar trend as that of the VM instantiation experiments. With the help of VMAR, the average load time and I/O traffic are much lower, and increase at a much slower pace with resource contention than the *baseline*. The *lessfs* scheme still causes significant overhead to I/O performance. When the system is under memory pressure, *KSM* is not able to reduce I/O traffic or improve performance.

Compared to *lessfs*, the *lessfs*+VMAR configuration improves application loading time by 15% on average, which is more than twice the improvement in VM instantiation time (7%). This is because in the application loading workload, data sharing among images is in large sequential chunks, which enables VMAR to redirect large I/O requests without creating too many descendants.

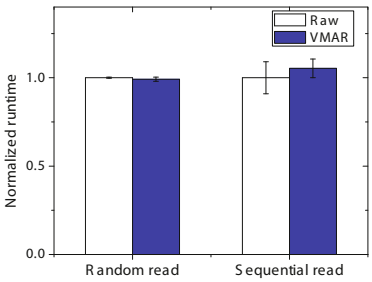


Fig. 19. Comparison of runtime for running random/sequential reading benchmark

Runtime Overhead. VMAR intercepts each read request to the VM image and incurs additional processing (address translation and redirection). We test this overhead with both random and sequential I/O by issuing dd commands within a VM, with 1 MB block size and direct I/O mode. For random I/O, 3,000 non-zero blocks are read on random locations of the virtual disk. For sequential I/O, a 350 MB non-zero file is read. To eliminate the impact of other factors, the benchmarks run twice when the VM is idle. After the first run, all the data has been brought into the host page cache. We measure the runtime of the second run, which only copies the data from the host memory. Figure 19 shows the runtime normalized to using a raw image. The result shows that the overhead of VMAR is within 5% for sequential I/O and negligible for random I/O.

5 Related Work

This section surveys existing efforts on I/O resource optimization by leveraging data content similarities in various workload scenarios.

5.1 Deduplicated Storage and File Systems

Deduplication for Backup Data. Due to the explosive generation of digital data, deduplication techniques have been widely used to reduce the storage capacity in backup and archival systems. In general, storage deduplication techniques break each dataset (file or object) into smaller chunks, compare the content of each chunk, and merge chunks with the same content. Much research effort has been made to enhance the effectiveness and efficiency of these operations [10,11,13,21,31]. For instance, Zhu et al. [31] have proposed three techniques to improve the deduplication throughput, which improve the content identification performance, deduplicated storage layout, and metadata cache management respectively. Meyer et al. [21] have provided the insight that deduplication on the whole-file level can achieve about $\frac{3}{4}$ of the space savings of block-level deduplication, while significantly reducing disk fragmentations.

Deduplication for Primary Data. Many recent papers have focused on the deduplication of primary data, namely datasets supporting runtime I/O requests [12,18,19,23,26]. They tackle the problem of I/O latency caused by deduplication from different angles. In [12], a study has been presented to analyze the file-level and chunk-level deduplication approaches using the dataset of primary data collected from Windows servers. Based on the findings, a deduplication system has been developed, where data scanning and compression are performed offline without interfering with file write operations. Ng et. al have proposed optimized metadata management schemes for inline deduplication of VM images [23]. iDedup [26] has used a minimum sequence threshold to determine whether to deduplicate a group of blocks, and thereby preserving the spatial locality in the disk layout. DEDE [9] focuses on distributing the workload of duplicate detection to the cluster of compute notes. It also demonstrates that the VM instantiation time can be significantly improved by improving the storage array cache hit rate.

5.2 Memory Deduplication

Many techniques have been proposed to leverage the similarities among processes or VMs running on a physical server and reduce their memory usage. Back in 1997, Disco [7] has introduced page sharing in NUMA multiprocessors. More recently, VMware ESX Server [28] has proposed *content-based page sharing*, in which pages with identical content can be shared by modifying the page table supporting the VMs. When a shared page is modified, the copy-on-write logic is triggered and a private copy of the page is created. Many optimizations have

been proposed to reduce the memory scanning overhead and increase sharing opportunities [16,18,22,25,29].

Among the above techniques, Satori [22] and I/O Deduplication [18] are the most relevant to VMAR. The sharing-aware block device in Satori and the content-based cache in I/O Deduplication both capture short-lived sharing opportunities by detecting similar pages at *page loading time*. However, Satori consolidates pages belonging to different VMs by modifying the guest OS, while VMAR works entirely on the host level and stays transparent to VM guests. I/O Deduplication introduces a secondary content-based cache under the VFS page cache, making it difficult to avoid duplicates across the two caching levels. As a matter of fact, VMAR may complement both by providing the block maps as hints for identical pages, which they need at page loading time.

6 Conclusion

In this paper we propose VMAR, which is a thin I/O optimization layer that improves VM instantiation and runtime performance by redirecting data accesses between pairs of VM images. By creating a content-based block map during image capture time and always directing accesses of identical blocks to the same destination address, VMAR enables VMs to give each other "free rides" when bringing their image data to the memory page cache. Compared to existing memory and I/O deduplication techniques, VMAR operates *ahead* of VM I/O requests and *upstream* in the I/O architecture. As a result, VMAR incurs small overhead and optimizes the entire I/O stack. Moreover, implemented as a new image format, VMAR is a configurable option for each VM. This enables cloud administrators to "test drive" it before complete deployment.

On top of the main access redirection mechanism, VMAR also includes two optimizations of the block map. The first one is to reduce block map size by merging contiguous map entries. The second one is to reduce the number of block map lookup operations by using an index to quickly guide a request into the correct region of the map. Experiments have demonstrated that in I/O-intensive settings VMAR reduces VM boot time by as much as 55% and reduces application loading time up to 45%.

VMAR is a disk image driver and does not rely on any specific CPU/memory virtualization technology. Thus, it is straightforward to make it work with other virtualization platforms such as Xen [6]. Currently VMAR works entirely on the host level. As future work, we plan to integrate VMAR with our previous work on VM exclusive caching [30] to achieve further savings on the VM level. We also plan to evaluate VMAR in an image pool with a larger scale and more types of operating systems, and explore adding a second level of redirection on the block storage layer to enhance sequential I/O pattern.

References

1. The QCOW2 Image Format, http://www.linux-kvm.org/page/Qcow2
2. Virtual machine disk format (VMDK), http://www.vmware.com/technical-resources/interfaces/vmdk.html

3. Virtualbox vdi image storage, http://www.virtualbox.org/manual/ch05.html
4. Amazon Web Services (AWS). Elastic Compute Cloud (EC2), http://aws.amazon.com (VM image data retrieved from AWS console on July 08, 2011)
5. Arcangeli, A., Eidus, I., Wright, C.: Increasing memory density by using KSM. In: Linux Symposium 2009 (2009)
6. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003 (2003)
7. Bugnion, E., Devine, S., Govil, K., Rosenblum, M.: Disco: running commodity operating systems on scalable multiprocessors. ACM Trans. Comput. Syst. 15(4) (November 1997)
8. Chen, H., Kim, M., Zhang, Z., Lei, H.: Empirical study of application runtime performance using on-demand streaming virtual disks in the cloud. In: Middleware 2012 (2012)
9. Clements, A.T., Ahmad, I., Vilayannur, M., Li, J.: Decentralized deduplication in SAN cluster file systems. In: USENIX ATC 2009 (2009)
10. Dong, W., Douglis, F., Li, K., Patterson, H., Reddy, S., Shilane, P.: Tradeoffs in scalable data routing for deduplication clusters. In: FAST 2011 (2011)
11. Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M.: HYDRAstor: a Scalable Secondary Storage. In: FAST 2009 (2009)
12. El-Shimi, A., Kalach, R., Kumar, A., Oltean, A., Li, J., Sengupta, S.: Primary Data Deduplication Large Scale Study and System Design. In: USENIX ATC 2012 (2012)
13. Guo, F., Efstathopoulos, P.: Building a high-performance deduplication system. In: USENIX ATC 2011 (2011)
14. Gupta, K., Jain, R., Koltsidas, I., Pucha, H., Sarkar, P., Seaman, M., Subhraveti, D.: GPFS-SNC: An enterprise storage framework for virtual-machine clouds. IBM Journal of Research and Development 55(6) (November-December 2011)
15. Jayaram, K. R., Peng, C., Zhang, Z., Kim, M., Chen, H., Lei, H.: An empirical analysis of similarity in virtual machine images. In: Middleware 2011 (2011)
16. Kim, H., Jo, H., Lee, J.: XHive: Efficient Cooperative Caching for Virtual Machines. IEEE Trans. Comput. 60 (January 2011)
17. Kochut, A., Karve, A.: Leveraging Local Image Redundancy for Efficient Virtual Machine Provisioning. In: NOMS 2012 (2012)
18. Koller, R., Rangaswami, R.: I/O Deduplication: Utilizing content similarity to improve I/O performance. Trans. Storage 6(3) (September 2010)
19. Koutoupis, P.: Data deduplication with Linux. Linux Journal 207 (2011)
20. Liang, S., Jiang, S., Zhang, X.: STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. In: ICDCS 2007 (2007)
21. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. In: FAST 2011 (2011)
22. Miłós, G., Murray, D.G., Hand, S., Fetterman, M.A.: Satori: enlightened page sharing. In: USENIX ATC 2009 (2009)
23. Ng, C.-H., Ma, M., Wong, T.-Y., Lee, P.P.C., Lui, J.C.S.: Live deduplication storage of virtual machine images in an open-source cloud. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 81–100. Springer, Heidelberg (2011)
24. Peng, C., Kim, M., Zhang, Z., Lei, H.: VDN: Virtual machine image distribution network for cloud data centers. In: INFOCOM 2012 (2012)

25. Sharma, P., Kulkarni, P.: Singleton: system-wide page deduplication in virtual environments. In: HPDC 2012 (2012)
26. Srinivasan, K., Bisson, T., Goodson, G., Voruganti, K.: iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In: FAST 2012 (2012)
27. Tang, C.: FVD: a high-performance virtual machine image format for cloud. In: USENIXATC 2011 (2011)
28. Waldspurger, C.A.: Memory Resource Management in VMware ESX Server. In: OSDI 2002 (2002)
29. Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., Corner, M.D.: Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In: VEE 2009 (2009)
30. Zhang, Z., Chen, H., Lei, H.: Small is big: functionally partitioned file caching in virtualized environments. In: HotCloud 2012 (2012)
31. Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: FAST 2008 (2008)