RC25349 (WAT1209-080) September 27, 2012
Computer Science

IBM Research Report

VMAR: Virtual Machine I/O Access Redirection to Optimize Instantiation Performance and Resource Utilization

Zhiming Shen, Zhe Zhang, Alexei Karve, Andrzej Kochut, Han Chen, Minkyong Kim, Hui Lei, Nicholas Fuller
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

VMAR: Virtual Machine I/O Access Redirection to Optimize Instantiation Performance and Resource Utilization

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 high overhead and complexity.

We propose a new simple and non-destructive deduplication layer tailored for the cloud: *V*irtual *M*achine I/O *A*ccess *R*edirection (*VMAR*). *VMAR* generates a block translation map when images are captured, and uses it to redirect accesses for identical blocks to the same filesystem address. This greatly enhances the cache hit ratio of VM I/O requests and leads to more than 50% performance gains in instantiating VM operating systems, and over 70% gain in loading application stacks. It also reduces the I/O resource consumption significantly. Another strength of *VMAR* is that it does not change the storage layout of VM images, and thus each VM can make an independent decision on whether to use *VMAR*. This allows cloud administrators to adopt *VMAR* in an incremental way.

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 create their own images by installing additional components. Through this process, a hierarchy of deviations of VM images can be formed. For example, in [20] 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).

A typical cloud data center hosts hundreds or thousands of VM images, each containing gigabytes or tens of gigabytes of data. Transferring these images from storage servers to hypervisor servers introduces heavy disk and network I/O workloads. 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 [12]. This has indicated rich opportunities to deduplicate the storage and I/O of VM images.

Multiple approaches have been developed to exploit high degree of content redundancy. However, they have significant shortcomings when applied to runtime virtual machine optimization. Traditional storage deduplication techniques [7, 8, 10, 17, 25] break down each file into small chunks of fixed or variable sizes and merge chunks with the same content. Identical chunks can be discovered through bitwise comparison or content hashing, whereas the latter is more popular due to its speed advantage. These methods have been widely used in archival storage systems to compress the storage space. However, as discussed in [21], they cannot be easily used to support latency-sensitive primary data workloads due to several drawbacks, including increased write delay, metadata management overhead, and broken storage layout.

Recently, several efforts have been been proposed to support primary data deduplication [9, 15, 21] through techniques like selective deduplication and background data merging. However, these optimizations are still based on the idea of *storage compression*, and only mitigate instead of eliminating the aforementioned drawbacks. Moreover, both traditional and primary data deduplication techniques mainly reduce the storage space without saving other more expensive I/O resources, including memory cache space and I/O bandwidth.

As an alternative, this paper proposes *VMAR*, a *simple* and *non-destructive* deduplication method designed based on the characteristics of cloud environments. *VMAR* is based on the idea of *V*irtual *M*achine I/O *A*ccess *R*edirection. It detects identical data blocks among VM images when they are captured and generates a block translation map. The data hashing and comparison operations are offline and incur no runtime overhead to cloud customers. Using the map, *VMAR* redirects VM read accesses to identical blocks to the same filesystem address before they are received at the hypervisor Virtual Filesystem (VFS) layer. In non-virtualized I/O workloads, VFS is the entry point for all data requests, and no I/O deduplication can be performed before this layer. 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 filesystem page cache, which is already "warmed up" by its peers. The reduction of warmup phase is critical to cloud user experience because the lifetime of VM instances is typically only in the order of minutes or hours. Moreover, *VMAR* keeps the storage layer intact instead of breaking each image into chunks and rearranging the storage layout. Each individual VM can be configured to go through or bypass the *VMAR* deduplication layer. From a system administrator's standpoint, this allows *VMAR* to be *incrementally deployed* to a cloud system, greatly reducing the risk of adopting a new storage technology.

We have implemented *VMAR* based on the QEMU Qcow2 driver. Our evaluation shows that in I/O-intensive settings *VMAR* reduces VM boot time by over 50% and reduces application loading time by over 70%. The performance gains is even larger when the system becomes more memory stressed.

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

2 Background

For the past decades hard disk drives have been the dominant storage device for general purpose computers. As a consequence, most virtualization technologies present to virtual machines a *virtual disk* interface to emulate real hard disks (also known as *disk image* or *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.

Today's production cloud environments are facing an explosion of VM images. Amazon Elastic Compute Cloud (EC2) has 6521 *public* VM images [5] (data on *private* EC2 VM images is unavailable). Therefore, it is impossible to store all image files on every hypervisor host. Typically, each cloud environment has a shared storage system for image files, 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 [11]. 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.

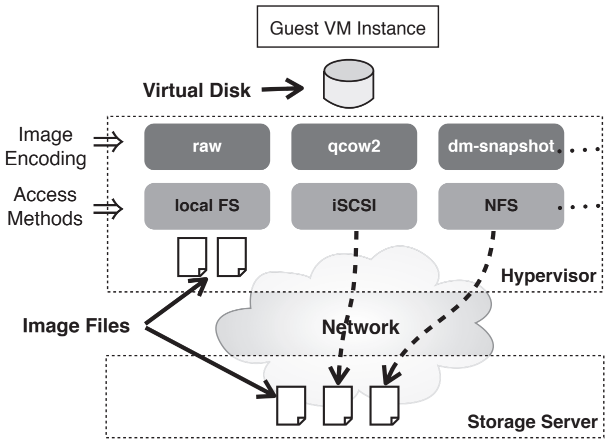


Figure 1: Different configurations of virtual disks

As there are many VM images that are in the order of gigabytes or even bigger, it is not efficient to store and transfer them in the uncompressed *raw* format. Popular meth of encoding VM images include qcow2 [2] and Device Mapper snapshot (dm-snapshot) [1]. We will discuss VM image formats in Section 2.1. Another 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. Another way is to fetch parts of a VM image from the storage system *on-demand*. We will discuss different approaches for accessing VM images in Section 2.2. Figure 1 illustrates different combinations of virtual disk configurations.

2.1 VM Image Formats

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 1, there are different copy-on-write schemes, including qcow2 , dm-snapshot, FVD [22], VirtualBox VDI [4], VMware VMDK [3], and so forth. In some schemes, such as Qcow2, a separate file is created to store all data blocks that have been modified by the provisioned VM. Some other schemes, such as dm-snapshot, work on the device level, without going through the operating system's virtual file system (VFS) layer. *VMAR* aims to utilize the VFS page cache to reduce I/O accesses to blocks with identical content. Therefore, its implementation is based on the Qcow2 format.

2.2 Accessing VM Images

In some cloud environments, the entire virtual disk is copied to the target hypervisor before an instance is started. If an instance uses an image that the target hypervisor does not have, it may take a long time to start up that instance. A typical VM image file contains multiple gigabytes, or even tens of gigabytes of data. Therefore, this delay could be severe in a heavily loaded cloud environment. Subsequent instances that use the same image on that host can start up faster as the image is locally available. If the total number of images is small, each image can be pre-copied to all compute nodes (target hypervisors). However, as the number of images increases this *pre-copy* approach would not be a feasible solution.

To overcome this problem, other cloud operating environments do not require to have the entire image at the target hypervisor. Instead, the parts of an image are copied as needed from the shared storage system to hypervisor hosts. This *on-demand* streaming approach avoids expensive network transfers of entire image files and accelerates VM start up time. However, VM runtime performance can be degraded, as blocks of data may need to be fetched from the remote storage during runtime. In contrast, all requests to the virtual disk are served locally with the *pre-copy* approach.

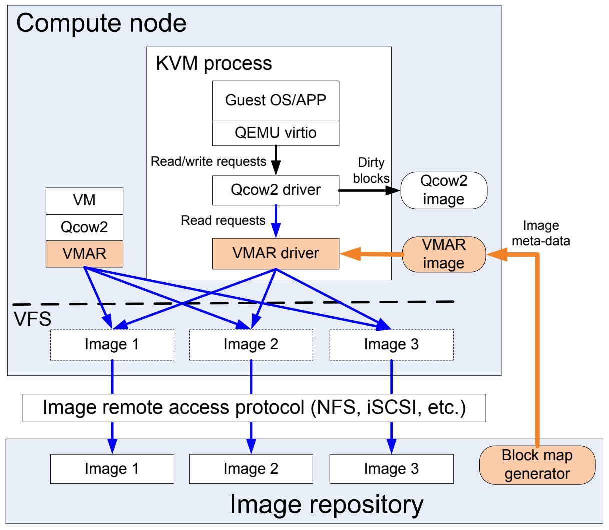


Figure 2: Architecture of VMAR

The benefit of *VMAR* is more pronounced when used with the *on-demand* policy, because a large portion of the on-demand network I/O can be reduced through the its access deduplication. On the other hand, the overhead of pre-copying images cannot be easily mitigated by *VMAR* because the I/O operations are not from VMs. Therefore, in the following discussion of the *VMAR* design and evaluation, we have assumed that it is applied on top of the *on-demand* policy.

3 Design and Implementation

Figure 2 illustrates the overall architecture of *VMAR*. *VMAR* consists of two main components: block map generator creates a map during image capture, and VMAR driver redirects accesses. To support sharing, we first identify common blocks. When the virtual disk of a VM is captured as a new base image, we compare it against the existing images in the repository. We generate the mete-data of the new image, including a block map that identifies common blocks between this image and the rest. After the block map is generated, when a new VM based on the image is requested on a compute node, the meta-data is forwarded to the compute node, and a VMAR image is created. The VMAR image format is a new format that we have added into the KVM hypervisor. It serves as a backing file for the Qcow2 image. The VMAR image does not contain any real data; it only stores the metadata for the VMAR driver, and the driver will redirect the image accesses to different base images. We further optimize *VMAR* to reduce block map size and look up time.

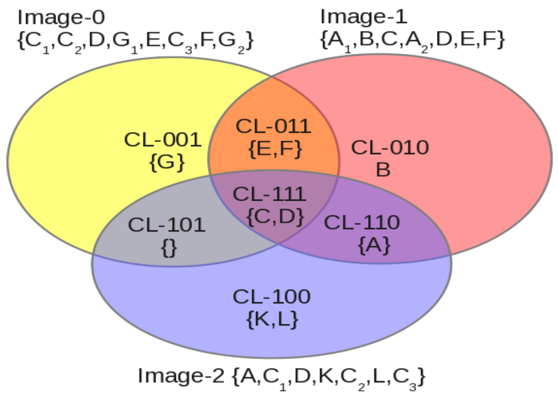


Figure 3: Illustration of clusters for three example images.

The rest of this section details each component of *VMAR*. Section 3.1 discusses the details of the block map generation. Section 3.2 describes the access redirection mechanism and Section 3.3 presents optimization techniques.

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 fingerprint computed from a collision resistant hash of the content of the data block. In capturing the content similarities among VM images, we leverage the concept of metadata *clusters* proposed in [14]. Each cluster represents the sets 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 briefly describe the mechanism of generating metadata clusters. Followed by that we discuss how to generate the block map from the clusters.

Consider a simple example of clusters for three images: Image-0, Image-1 and Image-2 in an image library as shown in Figure 3. In this illustration, CL-001, CL-010, CL-100 are singleton clusters, each containing the blocks only from Image-0, 1 and 2, respectively. The figure shows the actual cluster data blocks shared by the images. For example, block with hash G is unique to Image-0, while blocks with hashes C and D are shared by all three images. In practice, we only store the cluster metadata for the unique blocks shared by the images. CL-011 is the cluster with blocks from Image-0 and 1. CL-101 is the cluster with blocks from Image-0 and Image-2; in our example, it is empty. CL-110 is the cluster with blocks from Image-1 and Image-2. CL-111 is the cluster with blocks from Image-1, Image-2 and Image-3. For internal redundancy within an image, we use subscripts to denote same blocks. For example, Image-0 contains C_1, C_2 and C_3; this indicates that hash C is repeated three times in the image.

When a new image is added to the library, the system computes the SHA1 hash for each block. Each hash from the new image is compared against the existing clusters. We split each cluster into two: one containing blocks that are present in the new VM image and the other containing the blocks not appearing in the new image. It is possible that a cluster is not splited. This happens when either all hashes in a cluster are present in the image or none of the hashes in a cluster are present in the image. If a cluster has overlapping content with the new image, it is updated with the block positions in the new image. The hashes in the new images that do not belong to the any current clusters, are put into a new singleton cluster for the new image.

A certain hash value can appear in multiple images. We need a consistent mapping protocol to ensure that all requests for blocks with same hash are redirected to the same address. For that purpose we use the unique image IDs that are assigned to each image when it is added to the library, the value of which is incremented sequentially. If a block appears in multiple different images, our block map always points it to the image with the smallest ID. Instead of the image with the smallest ID, alternative consistent mapping protocols can be considered as future work. For instance, the least fragmented image [16] or the most used image can be used as the target. These optimizations will maximize I/O sequentiality and take advantage of prefetching.

The method compute_map in Figure 4 shows an algorithm for computation of mapping for a given image. We look through all clusters containing the image number s. For each hash in selected cluster, we select the smallest block number in the image$_{min}$ in the cluster. For every source block number in image s, we return the target image number t and the target block offset.

Figure 5 and Figure 6 illustrate redirecting the read request for block t to lowest numbered block within the lowest numbered image t for example images Image-1 and Image-2, respectively. The superscript in the figures for the hash indicates the Image number and the subscript indicates the block number if it is repeated. For Image-0, all blocks are retrieved from Image-0 itself because it is the least numbered image. From the complete list of 7 clusters, only 4 clusters contain blocks from Image-1. The CL-111 that shared blocks with all three images gives blocks C and D that will be redirected to least numbered Image-0. CL-011 that shares blocks with Image-

```
let $s$ be identifier of the source image

function compute_map($s$)
  let count=0;
  for each cluster $c$ containing s do
    let t=lowest_numbered_image(c);
    for each hash $h \in$ c do
      // h[s] contain block numbers in requested image s with hash h
      for each block $b_s \in$ h[s] do
        // h[t] contain block numbers in target image t with hash h
        let $b_t$ = min(h[t]);
        let map[count] = { $b_s$, t, $b_t$ };
        let count = count + 1;
      end for;
    end for;
  end for;
  return map;
end function;
```

Figure 4: Pseudo-code of mapping computation.

h[t]	s	h[s]
A_1^1	Image-1	A_1^1
B^1	Image-1	B^1
C^1	Image-0	C_1^0
A_2^1	Image-1	A_1^1
D^1	Image-0	D^0
E^1	Image-0	E^0
F^1	Image-0	F^0

Figure 5: Image-1 map

h[t]	s	h[s]
A^2	Image-1	A_1^1
C_1^2	Image-0	C_1^0
D^2	Image-0	D^0
K^2	Image-2	K^2
C_2^2	Image-0	C_1^0
L^2	Image-2	L^2
C_3^2	Image-0	C_1^0

Figure 6: Image-2 map

0 and 1 gives blocks E and F that will be also be redirected to least numbered Image-0. CL-110 has Image-1 as the least numbered image, so block A will be retrieved from Image-1. Finally CL-010 is the singleton cluster, so block B will also be retrieved from Image-1. Only 3 clusters contain blocks from Image-2. As in Image-1, blocks C and D are redirected to Image-0 for cluster CL-111. Block A is redirected to the least numbered Image-1 for CL-110. Blocks K and L belong to singleton CL-100 and are retrieved from Image-2.

The method update_map in Figure 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.

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 required to create the clusters was 3.3 hours. Creation of maps required time ranging from 13 s for the 4 GB image to 4.5 minutes for the 100 GB image. The total time to create the mappings for all images was 1.5 hours.

```
function update_block($s, block$)
  $hash\_prev$: previous hash value of block;
  $hash\_new$: new hash value of block;
  add $s$ to $update\_list$;
  let $c$ = find_cluster_from_hash($hash\_prev$);
  remove $block$ in $c$;
  if minimal image ID in $c$ that contains $block$ is changed:
    for each image $t$ that contains $c$ 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 minimal image ID in $c'$ that contains $block$ is changed:
      for each image $t$ that contains $c'$ do:
        add $t$ to $update\_list$;
      end for;
  for each image $t$ in $update\_list$ do:
    compute_map(t);
  end for;
end function;
```

Figure 7: Pseudo-code of updating an image.

3.2 I/O Deduplication through Access Redirection

The VMAR image serves as the backing file of the Qcow2 [2] image. When a read request R is received by the QEMU virtual I/O driver, the copy-on-write 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 with the *O_DIRECT* flag, 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 does a lookup in the block map introduced in Section 3.1 to find the designation 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. After all of the descendant requests are finished, the VMAR driver returns the whole buffer back to the Qcow2 driver.

The descendant requests are issued concurrently to maximize the 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

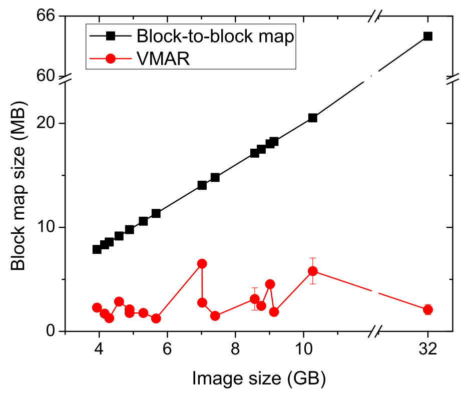


Figure 8: Block map size optimization.

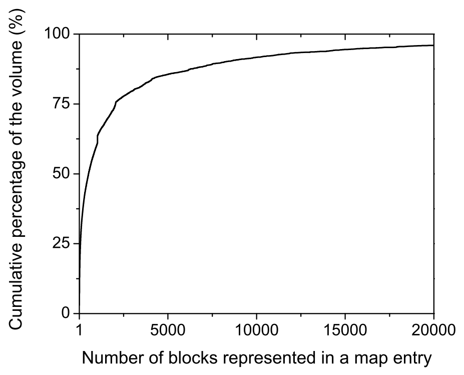


Figure 9: CDF of the volume in the clusters with different sizes.

vectors are created. Each I/O vector, representing each region of the buffer, fills the region with the fetched data. A counter for the application buffer keeps track of how may descendant requests are issued and how many of them are finished. The last callback of the descendant request will return the buffer back to the application.

When the descendant requests of R are sent to the host OS VFS layer and checked against the page cache, they will have *inode* numbers of the base image files. If the corresponding blocks in the base image 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

We further optimize *VMAR* to reduce the block map size and block-map lookup overhead.

Block map size reduction A straightforward method to support redirection lookup is to create a block-to-block map. Base 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: $\{\text{ID}_{target}, \text{Block}_{target}\}$. The lookup of block-to-block map is fast. However, the map size will grow linearly with the image size. For example, Figure 8 shows that the map size for a 32 GB image can grow up to 64 MB before optimization.

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: $\{\text{offset}_{source}, \text{length}, \text{ID}_{target}, \text{offset}_{target}\}$. 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.

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 8 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 9 presents the cumulative percentage of the the number of blocks represented in a single map entry. Map entries containing more than 64 blocks covers 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

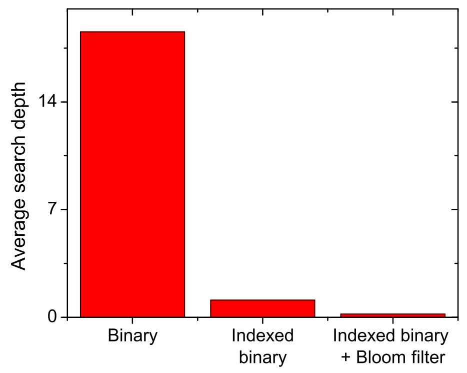


Figure 10: Average binary search depth for each search scheme.

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 10 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 18.5 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 hyperthreading physical cores (24 logical cores in total), 64 GB memory, and gigabit network connection. The hosts run Red Hat Enterprise Linux Server release 6.1 (Santiago) 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 in the repository using the iSCSI protocol.

To drive the experiments, we have obtained a random subset of 40 images from the image repository of a production enterprise cloud. The size of the images ranges from 4 GB to over 100 GB. The VMs are instantiated using ibvirt. Each VM is configured with two CPU cores, 2 GB memory, bridged network and disk access through virtio. 23 of the images run Red Hat Enterprise Linux Server release 5.5 (Tikanga), 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 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. Because it is hard to identify software already installed in the existing 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 Red Hat Enterprise Linux Server release 6.0 (Santiago) 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.

Most production clouds use either the *pre-copy* or the *on-demand* policy. We have evaluated *VMAR* on top of *on-demand* comparing to these baselines. *VMAR* on top of *pre-copy* is not discussed, because the benefit of precopying the images to the local disk is mostly diminished by the high cache hit ratio resulting from *VMAR*. In the experiments, we have evaluated the following three configurations:

- Pre-copy: The VM images are copied to the local disk of the compute node in advance.

- On-demand: The VM images are accessed through iSCSI protocol. The iSCSI devices serve as the backing device of the Qcow2 images directly, and blocks are fetched on-demand.

[1]DB2+WAS is a typical software stack used in online transaction processing (OLTP) workloads.

- VMAR: The Qcow2 image of each VM uses a *VMAR* image as the backing device. The *VMAR* image fetches blocks on-demand through iSCSI.

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.

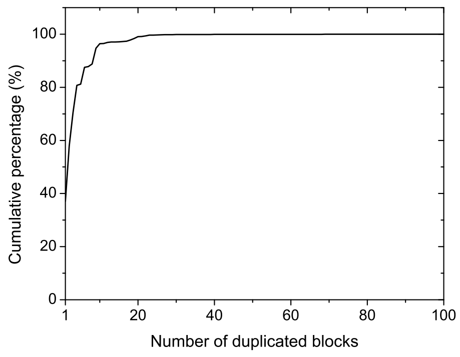
4.2 Experiment results

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*.

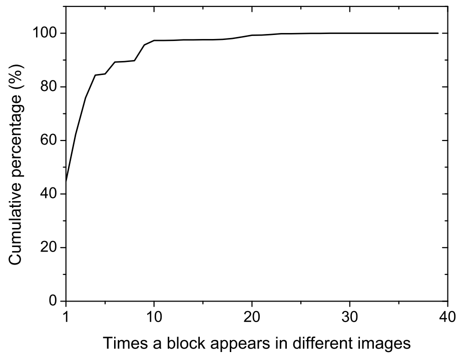
Similarity in the image repository We first analyze the content similarity among our 40 images. In this analysis, we only consider non-zero data blocks. Figure 11(a) shows the CDF of the number of duplicated blocks in the entire repository of 40 images. More than 60% of the blocks are duplicated at least 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 11(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 12 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. Figure 12(a) shows the average time it takes for a VM to boot up. With both the *pre-copy* and the *on-demand* schemes, a large number of data blocks needs to be copied either from the hard disk or over the network. In these two schemes, the boot time also increases quickly when more VMs are booted and the disk or network gets 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. Moreover, in contrast to *pre-copy* and *on-demand*, the average boot time with *VMAR* decreases when more VMs are booted. This is because the accesses of VMs that are started later are likely be served by the cache.

Figures 12(b) shows memory cache consumption and 12(c) shows I/O traffic during VM instantiation. In the *pre-copy* scheme, the I/O traffic is from the local disk



(a) Block duplication in the whole repository



(b) Inter-image duplication

Figure 11: Image blocks similarity statistics.

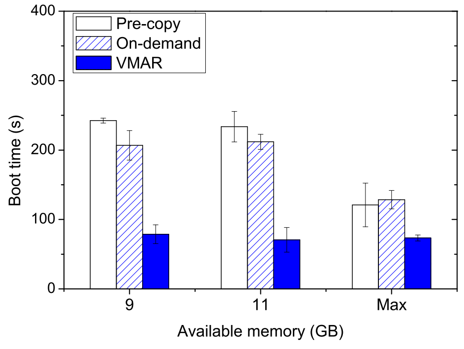
since the image is copied prior to our measurement. In case of the *on-demand* and *VMAR* schemes, the I/O traffic is from the network connecting to the image repository since the image is fetched on-demand. It can be seen that with the *pre-copy* and *on-demand* schemes, the amount of consumed memory is roughly the same as the incurred I/O traffic, both increasing almost linearly as the number of VMs increases. This is because the data accesses in the VM installation phase has poor temporal locality. Every read request of a data block is likely to go to the disk or network and the block is eventually cached in the memory. On the other hand, *VMAR* reduces the memory usage and I/O traffic for $37 \sim 61\%$ by trimming unnecessary disk and network accesses up in the memory cache. More importantly, the I/O resource consumption grows at a much slower rate than the two baseline schemes 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.

Figure 13 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 13(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 *pre-copy* and *on-demand* schemes increases quickly. In contrast, with the help of *VMAR*, a lot of disk accesses from the VMs hit the memory cache and return directly without triggering any real device access. Therefore, in comparison to the baselines, the average boot time with *VMAR* is much lower, and increases much slower as the arrival rate increases. Figures 13(b) and 13(c) show the memory usage and I/O traffic. As expected, the VM arrival rate does not affect the memory and I/O demands much, which confirms that the increase in boot time is due to the increased I/O contention, and *VMAR* reduces I/O contention to improve the VM instantiation performance.

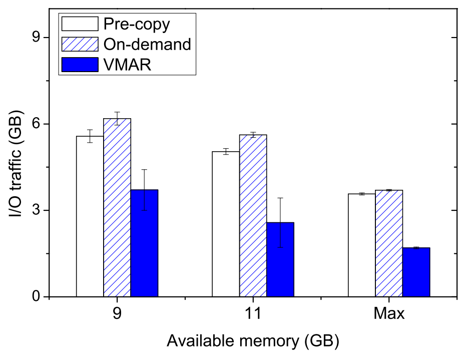
Figure 14 presents the performance and I/O traffic of VM instantiation with different available memory sizes. We insert a kernel module to the host kernel which uses *vmalloc* to occupy a set of non-swapable memory pages to control the available memory size. In this experiment, the number of VMs is set to 30 and the arrival rate is set to 0.2. From previous experiments, which uses all 64 GB memory, we figure out 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 in which the available memory size is 9 GB and 11 GB respectively. With *pre-copy* and *on-demand* schemes, the average boot time of the VMs increases significantly with a higher memory pressure because we have less memory for caching and more requests need to go to the physical devices. In contrast, the memory pressure has little impact on the VM boot time with *VMAR* scheme. The reason is that *VMAR* requires much less memory for effectively caching disk data for the applications.

Application loading Figures 15, 16, and 17 show the results about application loading performance. Again, in the experiment for Figure 15, the arrival rate of the VMs is set to 0.2, and different numbers of booted VMs are tested; in the experiment for Figure 16, the number of VMs is set to 30, and different arrival rates are tested; in the experiment for Figure 17, the VM number is set to 30 and the arrival rate is set to 0.2, and different available memory sizes are tested. As above mentioned, we replace four of the images with our own images which contain different versions of IBM DB2 and WAS, and only measure the application loading time of the four images. Other images will load different applications and serve as a background workload.

Loading an application is an I/O intensive workload. The VMs have to read a lot of application binaries and libraries into the memory. Although the four images we adopted contain different versions of the application, a lot of data can be common. Moreover, the applications loaded by the rest of images also have a lot of common



(a) Boot time

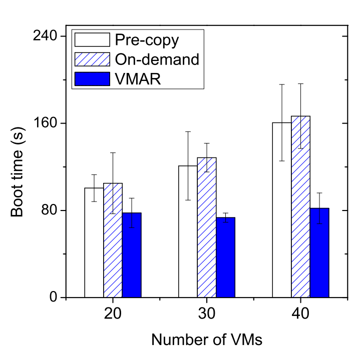


(b) I/O traffic

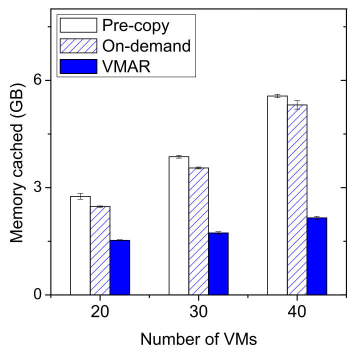
Figure 14: Comparison of VM boot time and I/O traffic with different available memory sizes.

data. The disk accesses to the common data provide a good opportunity for VMAR to share the memory cache and reduce the I/O contention. Thus, the results demonstrate a similar trend as that of the VM instantiation experiments. With the help of VMAR, the average load time, memory usage, and I/O traffic are much lower, and increase much slower than the *pre-copy* and *on-demand* schemes.

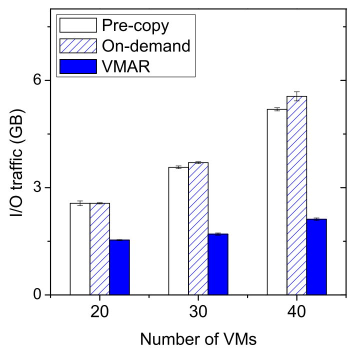
Overhead of VMAR The major overhead of VMAR resides in the address translation and breaking a big request into a number of descendant requests. We also conduct experiments to measure the overhead of VMAR. Both random read and sequential read are tested. For random read, we create a simple benchmark program that runs inside a VM and issues `dd` commands to randomly read 3,000 chunks of non-zero blocks (the size of each chunk is 1 MB, i.e., 256 blocks). For sequential read, we use a single `dd` command to read a 350 MB non-zero chunk directly from the disk. The block size for the *dd* command is set to 1 MB, and the *idirect* flag is used. 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 18 shows the runtime



(a) Boot time

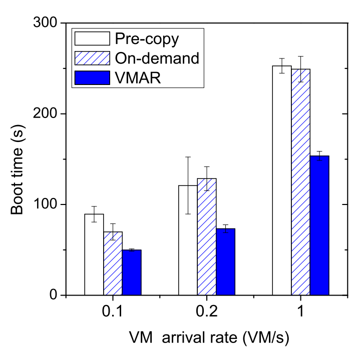


(b) Memory cached

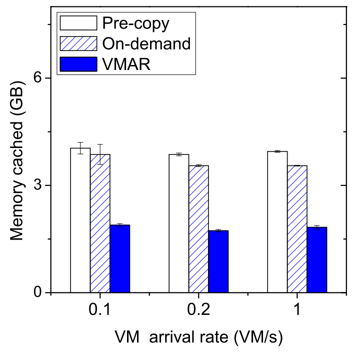


(c) I/O traffic

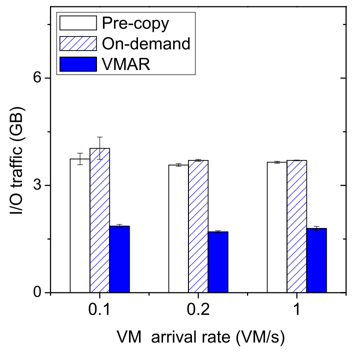
Figure 12: Comparison of VM boot time, memory cached and I/O traffic with different number of VMs.



(a) Boot time



(b) Memory cached



(c) I/O traffic

Figure 13: Comparison of VM boot time, memory cached and I/O traffic with different VM arrival rates.

normalized to using a raw image. The result shows that the overhead of *VMAR* is less than 5%. The runtime for random read is even smaller than raw image because we issue multiple descendant requests concurrently, which increases the throughput by leveraging multi-threading.

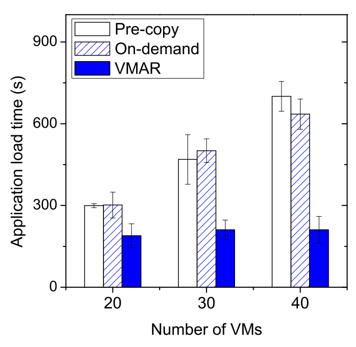
5 Related Work

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

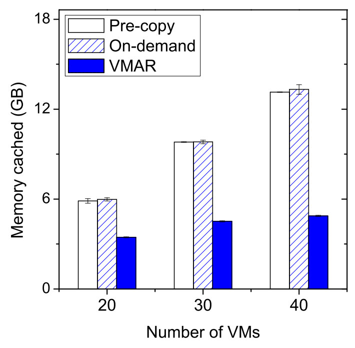
Deduplicated Storage Systems Due to the explosive generation of digital data, deduplication techniques have been widely used in to reduce the storage capacity in backup and archival systems. In general, storage deduplication techniques break each dataset (file or object) into smaller chunks, compare 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 [7, 8, 10, 17, 25]. For instance, Zhu et al. [25] 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. [17] 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.

The storage deduplication techniques discussed above mainly focus on optimizing the performance of data backup workloads which are sensitive to throughput and space. As summarized in [21], they cannot be easily used to support latency-sensitive runtime I/O, because of the bookkeeping overhead in the write path and increased disk seeks in the read path. Nevertheless, our work has leveraged the wisdom of many storage deduplication techniques. For example, Bloom filter is commonly used in examining the existence of a block in deduplicated storage repositories, which is also used by *VMAR* to speedup block map lookups. Our decision of using a fixed chunk size of 4 KB is also based on findings of the impact of chunking schemes on the deduplication ratio [12].

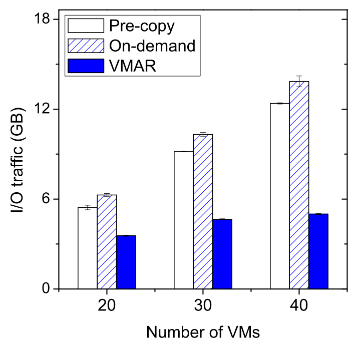
Deduplication for Primary Data Many recent papers have focused on the deduplication of primary data, namely datasets supporting runtime I/O requests [9, 15,



(a) Application load time

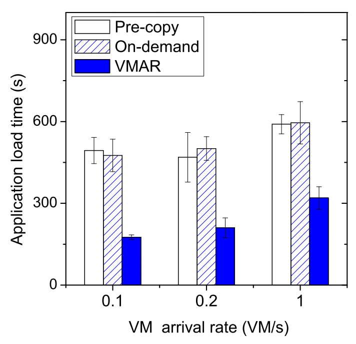


(b) Memory cached

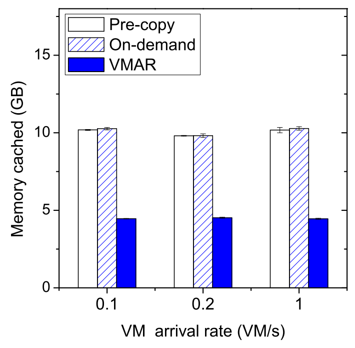


(c) I/O traffic

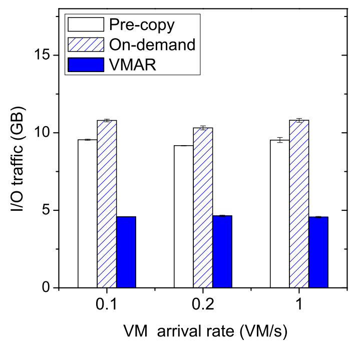
Figure 15: Comparison of application load time, memory cached and I/O traffic with different number of VMs.



(a) Application load time



(b) Memory cached



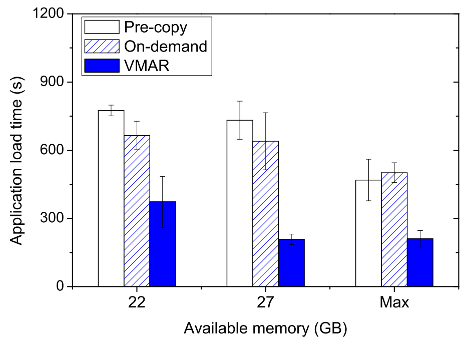
(c) I/O traffic

Figure 16: Comparison of application load time, memory cached and I/O traffic with different VM arrival rates.

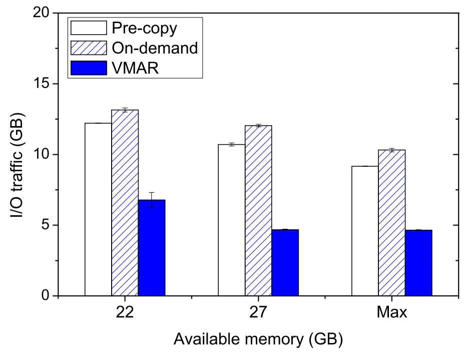
19, 21]. They tackle the problem of I/O latency caused by deduplication from different angles. In [9], a study has been presented to analyze the file-level and chunk-level deduplication approaches using the dataset of primary data collected from Windows servers. Based on the findings, a deduplication system has been developed, where data scanning and compression are performed offline without interfering with file write operations. The I/O deduplication technique proposed in [15] includes several optimizations, including an additional content-based buffer cache under the VFS page cache. Ng et. al have proposed optimized metadata management schemes for inline deduplication of VM images [19]. iDedup [21] 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.

However, most of the optimizations discussed above still deduplicate at the storage layer by merging identical data chunks. This saves the storage space without reducing the usage memory cache space and I/O bandwidth, which are of higher demand in cloud environments. In contrast, *VMAR* leverages the special characteristics of I/O virtualization and trims duplicate data accesses above the VFS layer, which leads to resource savings along the entire I/O path. This is not possible in non-virtualized I/O workloads, where VFS is the entry point of each request.

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. Disco [6] has introduced page sharing in NUMA multiprocessors, which requires modifications of the (guest) OS. Unlike traditional multiprocessor systems where there is only one OS (and thus no distinction between host and guest OS), in today's virtualized systems in the cloud, guest OS cannot be easily modified. To support page sharing without modifying guest OS, VMware ESX Server [23] uses *content-based page sharing*, in which pages can be shared as long as their content is same. To reduce the cost of identifying identical pages, the hash value of a page's content is used as the key to look up the pages with the identical hash value. Many optimizations have been proposed to reduce the memory scanning overhead and increase sharing opportunities [13, 18, 24]. For instance, Satori [18] captures short-lived sharing opportunities by detecting similar pages at page loading time.



(a) Application load time



(b) I/O traffic

Figure 17: Comparison of application load time and I/O traffic with different available memory sizes.

The page sharing techniques discussed above incurs CPU and memory bandwidth overhead in scanning memory pages. *VMAR* saves this runtime overhead by scanning and compressing VM images offline during image capture time. Moreover, in existing page sharing techniques, any requested data block needs to be copied from the hard disk drive to the memory cache at least once to compare with and potentially merge with other blocks. *VMAR* can greatly reduce these "cold misses" when a VM is instantiated, because many of the requested data blocks have already been brought into the cache by its peers.

6 Conclusion

In this paper we propose *VMAR*, which is a thin I/O optimization layer that optimizes 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 data deduplication and memory page sharing techniques, the proposed mechanism is tailored for the cloud with three key advantages. First, the content identification of data blocks is offline and does not incur perceivable overhead to foreground applications. Second, by redirecting accesses before the VFS layer,*VMAR* deduplicates the entire I/O path and eliminates "cold misses" in memory and disk caches. Finally, the simplicity and non-destructive nature *VMAR* make it an ideal candidate for progressive/gradual deployment in production systems.

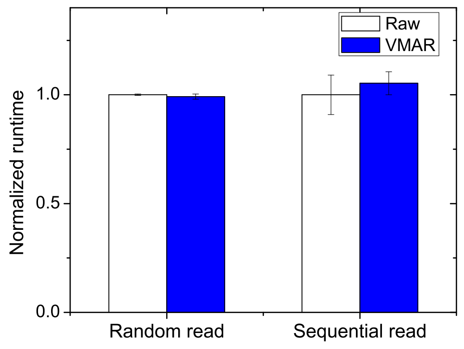


Figure 18: Comparison of runtime for running random/sequential reading benchmark.

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 over 50% and reduces application loading time by over 70%.

References

[1] Device-mapper snapshot support. See `http://www.kernel.org/doc/Documentation/device-mapper/snapshot.txt`.

[2] The QCOW2 Image Format. See `http://people.gnome.org/~markmc/qcow-image-format.html`.

[3] Virtual machine disk format (VMDK). See `http://www.vmware.com/technical-resources/interfaces/vmdk.html`.

[4] Virtualbox vdi image storage. See `http://www.virtualbox.org/manual/ch05.html`.

[5] AMAZON WEB SERVICES (AWS) INC. Elastic Compute Cloud (EC2). See `http://aws.amazon.com`. VM image data retrieved from an author's AWS console on Aug 7, 2011.

[6] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst. 15*, 4 (Nov. 1997), 412–447.

[7] DONG, W., DOUGLIS, F., LI, K., PATTERSON, H., REDDY, S., AND SHILANE, P. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and stroage technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 2–2.

[8] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. Hydrastor: a scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), FAST '09, USENIX Association, pp. 197–210.

[9] EL-SHIMI, A., KALACH, R., KUMAR, A., OLTEAN, A., LI, J., AND SENGUPTA, S. Primary data deduplication large scale study and system design. In *2012 USENIX Annual Technical Conference* (Boston, MA, USA, June 2012).

[10] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 25–25.

[11] GUPTA, K., JAIN, R., KOLTSIDAS, I., PUCHA, H., SARKAR, P., SEAMAN, M., AND SUBHRAVETI, D. Gpfs-snc: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development 55*, 6 (nov.-dec. 2011), 2:1 –2:10.

[12] JAYARAM, K. R., PENG, C., ZHANG, Z., KIM, M., CHEN, H., AND LEI, H. An empirical analysis of similarity in virtual machine images. In *Proceedings of the Middleware 2011 Industry Track Workshop* (New York, NY, USA, 2011), Middleware '11, ACM, pp. 6:1–6:6.

[13] KIM, H., JO, H., AND LEE, J. Xhive: Efficient cooperative caching for virtual machines. *IEEE Trans. Comput. 60* (January 2011), 106–119.

[14] KOCHUT, A., AND KARVE, A. Leveraging local image redundancy for efficient virtual machine provisioning. *IEEE Network Operations and Management Symposium* (2012).

[15] KOLLER, R., AND RANGASWAMI, R. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage 6*, 3 (Sept. 2010), 13:1–13:26.

[16] LIANG, S., JIANG, S., AND ZHANG, X. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proceedings of the 27th International Conference on Distributed Computing Systems* (2007), p. 64.

[17] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and stroage technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 1–1.

[18] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 1–1.

[19] NG, C.-H., MA, M., WONG, T.-Y., LEE, P. P. C., AND LUI, J. C. S. Live deduplication storage of virtual machine images in an open-source cloud. In *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware* (Berlin, Heidelberg, 2011), Middleware'11, Springer-Verlag, pp. 81–100.

[20] PENG, C., KIM, M., ZHANG, Z., AND LEI, H. Vdn: Virtual machine image distribution network for cloud data centers. In *INFOCOM* (2012), pp. 181–189.

[21] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association.

[22] TANG, C. Fvd: a high-performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 18–18.

[23] WALDSPURGER, C. A. Memory resource management in vmware esx server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (2002).

[24] WOOD, T., TARASUK-LEVIN, G., SHENOY, P., DESNOYERS, P., CECCHET, E., AND CORNER, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 31–40.

[25] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), FAST'08, USENIX Association, pp. 18:1–18:14.