UKAI: Centrally Controllable Distributed Local Storage for Virtual Machine Disk Images

Keiichi Shima
Research Laboratory, IIJ Innovation Institute Inc.
Chiyoda-ku Tokyo, Japan 101-0051
Email: keiichi@iijlab.net

***Abstract*—In this paper, we introduce a new distributed storage system for virtual machine disk images. A key concept of this storage system is that an operator has full control over which distributed storage nodes are used to keep a disk image. Unlike many other existing distributed file/storage systems, the proposed system does not automatically decide the location of distributed objects. This is important, especially when virtual machine infrastructure is deployed over many different geographical locations. Virtual machines are sometimes moved across hypervisors due to resource constraints or maintenance operations. When moving virtual machines, how efficiently the virtual machines communicate with their storage images is a serious issue. By acquiring full control of disk image placement, an operator can move related disk images to a nearer location. We present the design of the proposed system and provide performance measurements for our prototype implementation. The preliminary results show that the prototype code performs as well or better than existing virtual disk image serving methods under realistic operating conditions.**

I. Introduction

Virtual machine operation has become one of the core tasks of successful service providers. One problem still to be solved in this area is that of resource migration. For instance, it is sometimes necessary to perform actions such as move a virtual machine from one hypervisor to another in order to decrease the load of the origin hypervisor, upgrade a hypervisor operating system, move virtual machines to a newly built datacenter, or discontinue an existing datacenter. A virtual machine basically consists of three parts: i) CPU and memory resources, ii) a network resource, and iii) a storage resource. To keep a virtual machine running after migration, an operator must transparently supply the above three resources to the virtual machine. For i), the recent major virtualization technologies [1]–[5] already have the ability to migrate CPU and memory resources. For ii), recent progress in software defined networking technology means that is now a candidate solution for network resource migration [6]. For iii), there are a few technologies to support storage migration [7]–[9].

This paper focuses on the storage resource migration issue. We believe the following three requirements are important when operating a virtual machine storage backend.

Req.1 *Controllablity*: to provide the ability to store a disk image to a specific location for each virtual machine

Req.2 *Availability*: to provide a flexible level of redundancy to avoid data loss

Req.3 *Locality*: to place a disk image as near its owner virtual machine as possible to increase performance and robustness against network failures

Based on the above requirements, we have designed a new storage system that serves virtual machine disk images to hypervisors. The proposed system offers full control over the storage location of disk image data for each virtual machine. An operator can assign multiple locations for the same disk image to increase redundancy and can even specify location information on a portion of a disk image to protect important parts.

The location specification can be added or removed at any time. When a virtual machine moves from one location to another, an operator can add local locations, migrate the existing data to local locations without disrupting machine operations, and remove remote locations after migration. Unlike autonomously distributed file/storage systems, an operator can specify storage location. This is important because network services usually consist of multiple virtual machine instances. How storage access is optimized for one machine may affect the collaborative operational performance of the set of virtual machines.

As might be anticipated, providing full location control could lead to a lot of configuration tasks when the number of virtual disks is large. In our proposal, we do not consider the automated support of location control; however, it is possible to design a super-layer for storage management policy. This higher-layer module may provide automatic location determination functions. To do that, the lower-layer module must have detailed location management functions. This proposal aims to provide the basis for more intelligent management systems in future.

II. Related Work

The most basic technique of storage migration is incremental block migration. In this mechanism, the entire storage image is moved from the source to the destination location. A simple method was proposed in [7] that moves disk image blocks from head to tail after a virtual machine has been moved to a destination location. If the virtual machine tries

978-1-4799-2851-4/13/$31.00 ©2013IEEE

to access a not-yet moved block, then an on-demand copy is initiated. These mechanisms satisfy requirements 1 and 3. However, since the disk image is stored at single location, requirement 2 is not achieved.

To achieve redundancy, a distributed filesystem is sometimes used. Distributed Replicated Block Device (DRBD) [10] provides a block device, replicated over a network. Since DRBD is a type of mirrored disk, the operations for disk management are similar to those of local disk mirroring operations. In this sense, requirement 1 is satisfied; however, it is not possible to use this system for each virtual machine. DRBD also partially satisfies requirement 2 as it can have up to two replicated disks in a basic configuration. When more than two replicas are needed, a cascading configuration is required. For requirement 3, DRBD allows a dual primary operation mode that enables concurrent access to both mirrored volumes. In this sense, data locality can be achieved. However, it is difficult to change the mirroring volume from a remote site to a local site. Even though a user can access the local mirrored volume, the remote mirrored volume is still located remotely. The reconfiguration operations in DRBD are less flexible.

There are other distributed filesystems such as Ceph [11] or GlusterFS [12]. There is also a distributed block storage mechanism called Sheepdog [13] that is a special block device designed for virtual machine disk images. These distributed systems use a consistent hash mechanism to determine the locations for data distribution. An operator cannot control which part of a disk image is stored on which storage node. This is problematic when an operator wants to move a virtual machine to a distant location. If a storage cluster is configured locally near the origin hypervisor, the migrated virtual machine will suffer from poor disk performance because of the long network delay. If the storage cluster is deployed over a wide area network, the daily disk operation performance will worsen. Hence, these systems only satisfy requirement 2.

III. The UKAI System

In this section, we discuss the design of the *UKAI* system[1], a centrally controllable distributed local storage system.

A. Constraints and Advantages

When designing a new storage system that satisfies the requirements defined in Section I, we must first clarify the constraints of a virtual machine operation mechanism. We believe the operational environment of a virtual machine is unique, as a storage interface is not always required to provide a fully functional distributed filesystem. The constraints are given below.

- Con.1 No concurrency: there is only one entity for accessing a specific disk image at one time.
- Con.2 Limited metadata elements: the metadata information of a disk image is limited.

Since we are designing a storage system for virtual machine disk images, we can assume the disk images will only be used by hypervisors and virtual machines. Considering that a disk image is associated with a specific virtual machine, it is not necessary to allow concurrent access to a specific disk image from multiple entities. This means that it is not necessary to design a distributed resource locking mechanism.

A disk image is seen as a block device from a virtual machine's point of view. Creation, access, or modification times are not meaningful in such an environment. The size of a disk image is also meaningless, as it does not usually change. Based on these constraints, a UKAI storage can be implemented with only simple data I/O interfaces.

B. Disk Image Design

In the UKAI system, a disk image is divided into small blocks. Each block has its own location record. If redundancy is required, a block may have multiple location records. A block is the unit of a synchronization operation, and its location record contains a flag that indicates if the block at that location is in-sync or out-of-sync. If the disk image does not have redundancy, then all the locations must be in-sync. Having an out-of-sync location means that the disk image is broken, if a block has only one location record. If there are multiple locations, the operational requirement is to have at least one in-sync location. When reading data from a block, one of the in-sync locations is chosen for data retrieval. When writing, data is transferred to all locations and written to a local storage device at that location. If a location is in the out-of-sync state, data from one of the in-sync blocks is transferred to the out-of-sync location. Once the data transfer completes, the state is changed to in-sync. Fig. 1 shows the concept of the UKAI disk image structure.

In the figure, a disk image is divided into four blocks. Each block has two location records, some of which are flagged as out-of-sync. Data is read from the node indicated by the white location information box. When writing, for example, to block 3, the data stored in node B is copied to node A before any data is written. Then the location of node A is changed to an in-sync state and the actual data is written to both locations.

C. Metadata Design

Metadata information for a virtual disk is stored in the hypervisor on which the virtual machine using that disk is run. It consists of the size and name of the disk image, the size of the block, and location information of each block. The size and name are used as a filename when the disk image is exposed as a file through a filesystem or as a block device name of a hypervisor, depending on how the system is implemented. The block size and location records are used in the UKAI system internally.

D. Error Handling

When operating a distributed system, an error is not an option. For example, when reading data from one of the nodes defined in the location record, it may not be available because of node failure, network failure, or some other reason. In that case, the UKAI system records the node address in the

[1]UKAI is named after a traditional Japanese fishing method that uses cormorants (http://en.wikipedia.org/wiki/Cormorant_fishing).

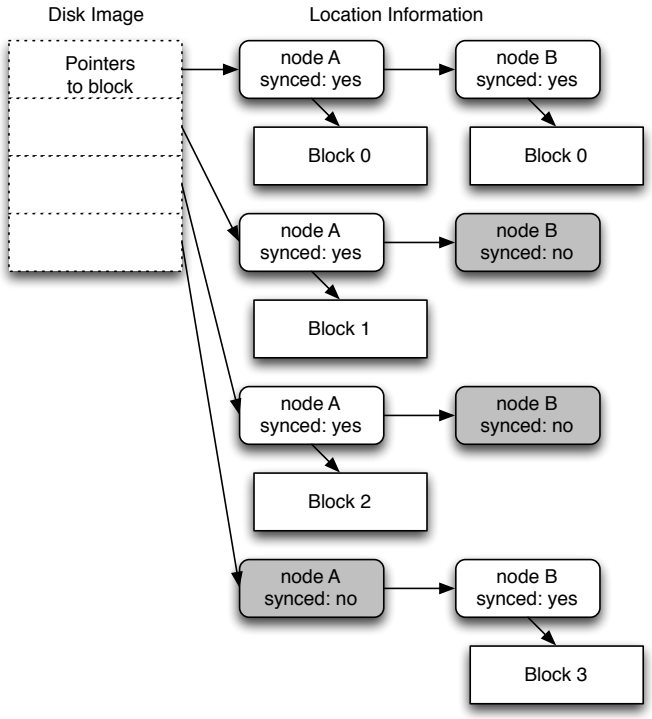

Fig. 1. The concept of the UKAI disk image structure consisting of four blocks and two location records

locally managed failure node list. When accessing a block, the UKAI system first checks the failure node list. If the node is listed in when the UKAI system is writing data, the location information is marked as out-of-sync. The data is written to all the other nodes listed in the location record list. When reading, the UKAI system looks for another candidate from the list of locations of the block being accessed. If there is no in-sync candidate, the situation is considered a fatal error.

The failure node list has a time limit for each entry. When this limit expires, access to the node may be resumed. If the node recovers before the time limit expires, then the data synchronizing process will be initiated when a write operation occurs on blocks marked as out-of-sync.

E. Control

The following control operations are defined for the minimum operation of the UKAI system.

Add image: adds a new virtual disk image to the UKAI system. An image must be added before being used as a disk image by a virtual machine.
Remove image: removes an existing virtual disk image from the UKAI system. Before removing a disk image, the virtual machine that uses the target disk image must be powered off.
Add location: adds a location specification to a range of blocks.
Remove location: removes a location specification from a range of blocks. When removing a location, the system must ensure that at least one in-sync location is left for each block. Otherwise, the data of the disk will be lost and a fatal error will occur.
Get metadata: returns the metadata information of a specific disk image containing the name of the image, the size of the image, the block size of the image, and the list of locations for all the blocks.
Synchronize: synchronizes a range of blocks between nodes defined in the location records of the specified blocks.

Other operations may be defined in future UKAI systems.

```
{
 "name": "disk01",
 "size": 200000000,
 "block_size": 50000000,
 "blocks": [{
   "192.0.2.100": {"synced": true},
   "192.0.2.101": {"synced": true}
  },{
   "192.0.2.100": {"synced": true},
   "192.0.2.101": {"synced": false}
  },{
   "192.0.2.100": {"synced": true},
   "192.0.2.101": {"synced": false}
  },{
   "192.0.2.100": {"synced": false},
   "192.0.2.101": {"synced": true}
  }]
}
```

Fig. 2. The metadata structure that represents the disk image shown in Fig. 1, where the size of the image is set to 200 MB, the size of each block is set to 50 MB, and the addresses of nodes A and B are set to 192.0.2.100 and 192.0.2.101, respectively

IV. Implementation

We have implemented the concepts of the UKAI system in a prototype using FUSE [14] and Python. Each disk image is exposed through the FUSE mechanism as a file. We used QEMU as a hypervisor since it has the ability to use a file as a virtual disk image. Any other hypervisors can also be used if they can use a file as a virtual disk image. Note that there is no limitation on how to implement the UKAI system. An implementation as a filesystem interface is just one example. It could also be implemented in a special block driver form for QEMU or other virtualization systems.

The prototype code is available at the GitHub repository[2].

A. Metadata Handling

Fig. 2 shows the metadata structure for the example disk image shown in Fig. 1. The data is structured using a JSON [15] format. The name of the disk image is `disk01` and this name is used as a file name under the FUSE mount point of the UKAI system. The size of this disk image is defined to be 200 MB. Since the block size is 50 MB, the total number of blocks will be four blocks.

The metadata file for a virtual disk is stored in the hypervisor that runs its virtual machine. When a virtual machine

[2]https://github.com/keiichishima/ukai/

is migrated to another hypervisor, the related metadata must be accessible from the destination hypervisor. This can be achieved in several ways. One method is to copy the entire metadata file at the last moment when the virtual machine state is migrated to the destination. Since the size of a metadata file is small compared to memory or storage data, copying a metadata file does not affect migration performance. The other method is to use some kind of distributed filesystem, such as GlusterFS, to share metadata files. In the latter case, it is necessary to operate a wide-area distributed filesystem; however, the update of metadata information occurs only when the synchronization status changes, so its influence is small. In the test operations discussed in Section V, we simply share metadata files using NFS. For performance reasons, we plan to integrate the former metadata transition method in the final form of the implementation.

B. FUSE Interface

To implement a FUSE interface in Python, we utilized fusepy[3], a FUSE-Python binding library. As we have discussed in Section III-A, not all the filesystem interfaces must be implemented. In the prototype implementation, only the interfaces shown in TABLE I are implemented.

Other filesystem interfaces are either not implemented (such that the call falls back to the default behavior of fusepy), or do nothing.

C. Remote Read/Write Operations

In some cases, read or write operations may require access to a remote UKAI node. This is implemented using the XML-RPC mechanism. As we discussed in Section III-A, we do not need to take care of conflicting accesses to a disk image since only one virtual machine accesses a specific disk image at one time. When receiving read/write requests from a remote node, the UKAI storage node just performs read or write operations on the local disk without any exclusive control because it is certain that there is no other entities accessing the disk.

D. Control Operation

Control operations described in Section III-E are also implemented using the XML-RPC mechanism. The RPC interface is open to a local node. An operator can issue management commands such as adding an image or location through this interface.

V. Performance Measurement

We measured the I/O bandwidth with our prototype implementation. We prepared three 8 GB UKAI disk images with different block size values (5 MB, 10 MB, and 20 MB) and three different locations as shown in TABLE II.

For comparison, a disk image was created on local storage as a single file, and a disk image served by NFS was also prepared.

Two physical machines connected with a 1 Gbps Ethernet switch were prepared and configured as hypervisors and UKAI nodes. These nodes were located in the same network segment. One of the nodes was also configured as a NFS server for the NFS disk image mentioned above. TABLE III shows the specifications of the equipments.

The virtual machines used for the measurement ran Ubuntu 12.04 LTS with 512 MB memory.

Fig. 3 shows the system diagram of the measurement system. There were five different virtual machine configurations, each using a different disk image configuration. The three that used UKAI disk images also had three different block size configurations as described earlier. TABLE IV shows all the combination cases of the disks and virtual machines used in the measurement.

We used the fio[4] measurement tool. The configuration parameters for fio are shown in TABLE V. Measurement operations were done one-by-one for each virtual machine. While one virtual machine was running, the other four machines

TABLE II
Configuration of location information of three UKAI disk images

Location type	Configuration detail
Local	A virtual machine and its disk image are located on the same node.
Remote	A virtual machine and its disk image are located on different nodes.
Mirror	A disk image has two locations: one is on the same node as the virtual machine and the other is on a different node.

TABLE III
Specifications of the measurement equipments

Nodes	
Product ID	EPSON Endeavor AT971
CPU	Intel® Core™2 Duo E8400 3.00 GHz
Memory	4 GB
HDD	250 GB SATA
NIC	Intel® PRO/1000 Gigabit Network Adapter
Switch	
Product ID	Corega CG-SW05GTLXW

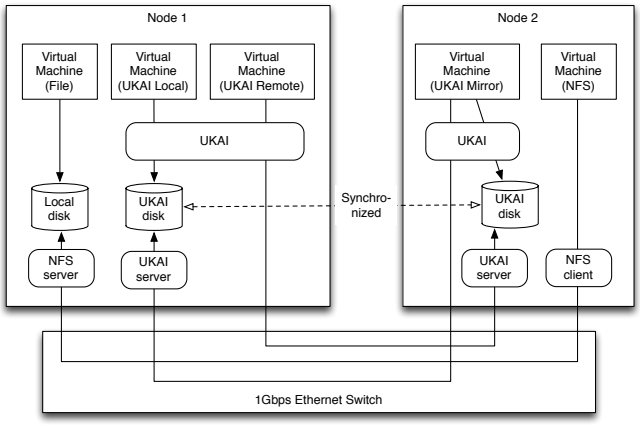

Fig. 3. System diagram of the performance measurement system

[3]https://github.com/terencehonles/fusepy/

[4]http://freecode.com/projects/fio

TABLE I
FUSE INTERFACES MANDATORY FOR THE UKAI SYSTEM

Interface	Action
`init()`	Initializes the UKAI system. The function launches two threads, one for receiving read/write operation requests from remote UKAI nodes as described in Section IV-C and the other for receiving control commands as described in Section IV-D.
`getattr()`	Returns a `stat` structure of a disk image file. The total image size is the only meaningful information.
`open()`	Returns a file descriptor of the specified disk image.
`readdir()`	Returns a list of disk image file names. To conform to the normal filesystem `readdir()` operation, the current and parent directories of the UKAI mount point are also returned.
`read()`	Reads data from a virtual disk and returns it to the caller. One read operation may contact multiple blocks depending on the specified read size and offset value.
`write()`	Writes data to a virtual disk. Just as for the read operation, multiple blocks may be accessed depending on the specified write size and offset. The write operation may initiate a synchronization operation if the blocks to be accessed have location information with an out-of-sync status.

TABLE IV
MAPPING TABLE OF VIRTUAL DISK LOCATION TYPES, VIRTUAL DISK CONFIGURATION TYPES, AND THEIR OWNER VIRTUAL MACHINES

Location type	Virtual disk type	Owner virtual machine
Local	Local file	Local
	UKAI 5 MB BS	UKAI Local
	UKAI 10 MB BS	
	UKAI 20 MB BS	
Remote	NFS file	NFS
	UKAI 5 MB BS	UKAI Remote
	UKAI 10 MB BS	
	UKAI 20 MB BS	
Mirror	UKAI 5 MB BS	UKAI Mirror
	UKAI 10 MB BS	
	UKAI 20 MB BS	

TABLE V
CONFIGURATION PARAMETERS OF THE FIO COMMAND

I/O pattern	Random read and random write
I/O file size	128 MB, 256 MB, 512 MB, and 1 GB
I/O block size	4 KB
I/O API	Use standard `read()` and `write()` system calls and `fseek()` library call

were shut down. As shown in TABLE V, four different file sizes, 128 MB, 256 MB, 512 MB, and 1 GB were used in the experiment to compare the effect of files size on performance variation.

A. Local Storage

Fig. 4 shows the results for local storage cases. The graph shows four different types of local disk: the first three are UKAI images with different block sizes and the fourth is a local file disk image.

In the random read case, we see unstable behavior for UKAI disks of 5 MB and 10 MB block sizes in the 1 GB file test case. However, the overall performance is not worse than that of the local file storage method. For the random write case, the UKAI disks often achieved better bandwidth than the local file storage method.

B. Remote Storage

Fig. 5 shows the results for remote location cases. In this case, we used a NFS mounted disk image for comparison instead of a local file disk image.

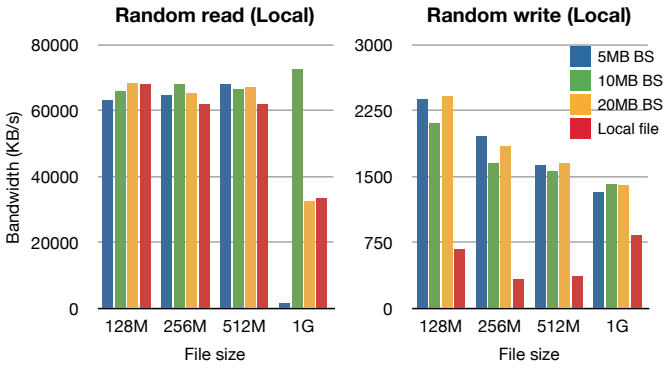

Fig. 4. Comparison of I/O bandwidth of local disk images

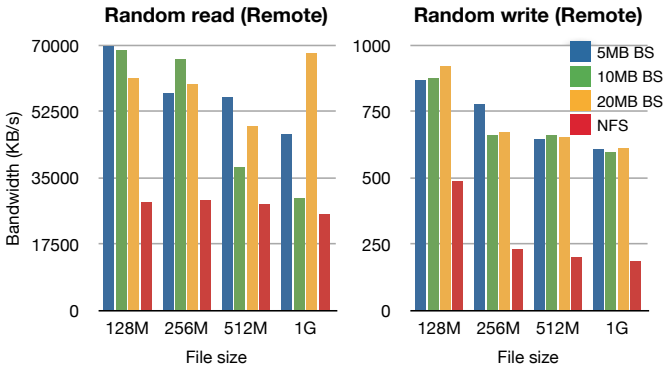

Fig. 5. Comparison of I/O bandwidth of remote disk images

In the remote location cases, the UKAI disks achieved better performance in both the random read and random write cases. However, we observed unstable behavior when reading, just as for the local case.

C. Mirrored Storage

Fig. 6 shows the results for three mirrored (one on local and the other on remote) UKAI disk images. For comparison, a local file disk image result is shown for the random read case and a NFS disk image result is shown for the random write case. Since the prototype UKAI implementation prefers reading from a local node whenever available, it is natural to compare it to a local file when reading. For writing, the UKAI filesystem has to write to both locations. Since a network write

Fig. 6. Comparison of I/O bandwidth of mirrored UKAI disk images and other images

operation always happens in this case, a NFS disk is used for comparison.

For reading, the UKAI performance should be similar to that of a local disk because data is read from the local side of a disk in the local-remote mirror case. However, for the 512 MB and 1 GB file size cases, the actual performance was worse than expected. For write operations, even though UKAI writes to two locations, the performance was better than that of NFS.

D. Measurement Summary

The results show that performance degrades when the size of a test file is increased. There were some cases where the performance of the UKAI disk was much worse than expected, for example, for random reads in the local storage and mirrored cases. On the contrary, for all the random write cases, its performance was better than local file and NFS storage.

The block size configuration of UKAI storage does not have a serious impact on the overall read/write performance; however, we do not recommend using a large block size because it will impact synchronization performance. If a block size is large, the possibility of accessing a block that is being synchronized increases. Such access results in device-level disk I/O blocking and will cause serious performance degradation at the virtual machine operation level.

VI. Discussion

We have not identified the reason for the variation in the UKAI storage system performance that we observed in the previous section. Our current hypothesis is that the load of other user-space programs might affect its performance because the UKAI system is implemented in user space. Another hypothesis is that the variation is caused by the nature of a layered filesystem. A virtual machine has its own filesystem and buffering mechanism. Its disk device is in reality provided by a hypervisor and is built on top of FUSE and hypervisor files, both of which also have buffering mechanisms. Because of these layered mechanisms, it is difficult to gain the control needed to optimize the I/O operations of a virtual machine. Where or how to buffer I/O data is currently a vital topic in virtual storage research [16]. We need to investigate the true source of this behavior to achieve a more stable and predictable performance.

We initially thought that using a smaller block size would increase access overhead, especially when operating with large size files. However, it seems that block size did not significantly influence read/write operations. The analysis of its impact on synchronization operations and the determination of the best block size during synchronization are future issues we plan to address.

It was surprising that the random write performance was better than for the local file and NFS storage cases, considering that the current UKAI is implemented in user space in Python and FUSE. We think this is because the filesystem buffering mechanism works efficiently for random write operations on the UKAI disk image blocks that have a much smaller file size compared to the disk image file used by local file and NFS storage. We have not yet measured the amount of resources consumed for UKAI I/O operations when handling a large number of block files, but this may have to be investigated in order to understand the overhead of the UKAI system.

One negative observation not mentioned in the previous section is that we found particularly poor sequential write performance in every case (Fig. 7). This is probably because the UKAI system uses small files to build a disk image. Buffering may not work efficiently in a sequential write operation that spreads over many small files.

VII. Conclusion

Flexible virtual machine location and/or relocation is a key function for efficient virtual machine resource management. Research on storage management mechanisms for virtual machine image storage is vitally needed to enable relocatable virtual machines. We defined three requirements necessary for a distributed virtual machine image storage system: controllability, redundancy, and locality and proposed the UKAI system. We implemented the concepts of the UKAI system in a prototype and achieved as good or better throughput compared to existing virtual disk mechanisms in most of the random read/write cases common to real-life operations. However, we also found that in some cases, the performance was not stable. We also found that sequential write performance was particularly poor. We continue to investigate the reason for these behaviors and will improve the design and implementation of the UKAI system to provide a better virtual machine image storage mechanism.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, 2003, pp. 164–177.

[2] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC'05)*, April 2005, pp. 41–41.

[3] R. Harper, A. Aliguori, and M. Day, "KVM: The Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[4] VMware, Inc., http://www.vmware.com/.

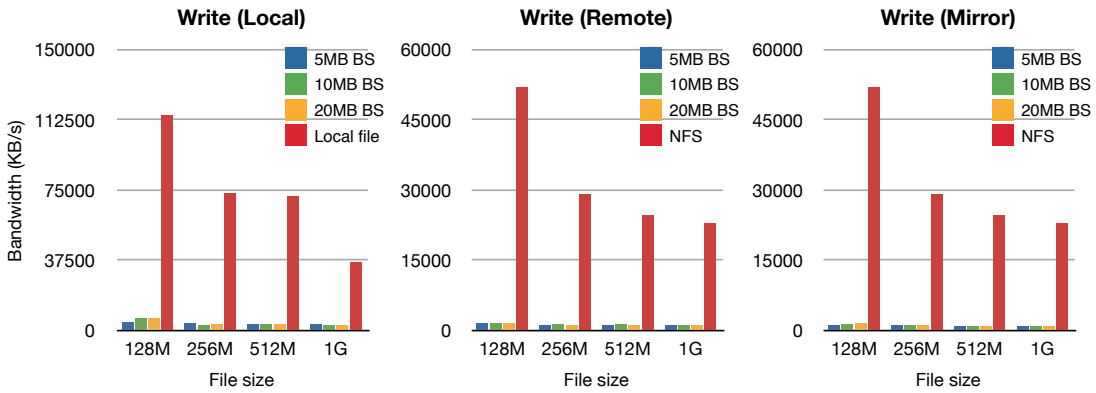

Fig. 7. Comparison of I/O bandwidth for sequential write operations over eleven different disk image configurations

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'05)*, vol. 2, 2005, pp. 273–286.

[6] J. Rexford, "Programming Languages for Programmable Network," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'12)*, 2012, pp. 215–216.

[7] T. Hirofuchi, H. Ogawa, H. Nakada, S. Itoh, and S. Sekiguchi, "A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services on Clouds," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09)*, 2009, pp. 460–465.

[8] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live Wide-Area Migration of Virtual Machines Including Local Persistent State," in *Proceedings of the 3rd internatiolan conference on Virtual execution environments*, 2007, pp. 169–179.

[9] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-Aware Live Storage Migration for Clouds," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'11)*, 2011, pp. 133–144.

[10] L. Ellenberg, "DRBD® 9 & Device-Mapper Linux® Block Level Storage Replication," in *Proceedings of Linux-Kongress 2008*, October 2008.

[11] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*. USENIX, 2006, pp. 307–320.

[12] Gluster Inc., "Gluster File System Architecture," Gluster Inc., Tech. Rep., 2010.

[13] K. Morita, "Sheepdog: Distributed Storage System for QEMU/KVM," Linux.conf.au 2010, January 2010.

[14] M. Szeredi, "FUSE: Filesystem in Userspace," http://fuse.sourceforge.net/.

[15] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, IETF, July 2006, rFC4627.

[16] V. Tarasov, D. Jain, D. Hildebrand, R. Tewari, G. Kuenning, and E. Zadok, "Improving I/O Performance Using Virtual Disk Introspection," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, June 2013.