2014 IEEE 8th International Symposium on Service Oriented System Engineering

DLSM: Decoupled Live Storage Migration with Distributed Device Mapper Storage

Zhaoning Zhang*, Ziyang Li*, Kui Wu†, Huiba Li*, Yuxing Peng*, Xicheng Lu*

* *National Key Laboratory for Parallel and Distributed Processing*
National University of Defense Technology, China
† *Computer Science Department*
University of Victoria, B.C., V8W 3P6, Canada

***Abstract*—As a key technique in cloud computing, live virtual machine (VM) migration makes cloud computing elastic and renders more efficient resource scheduling. In many cases, live storage migration is also desirable. Compared to live VM migration, however, live storage migration is relatively slow. The speed gap between live VM migration and live storage migration may lead to poor Quality of Service (QoS) for cloud users. In this paper, we present a new resource migration scheme, called Decoupled Live Storage Migration (DLSM), which decouples the live storage migration from the live VM migration. DLSM can migrate the VM immediately, while data blocks actually required by the VM are moved on demand at later times. It achieves the on-demand data migration at the storage level without the need of modifying the hypervisor. The key contribution of DLSM is that it reduces the network traffic and speeds up the live storage migration, by completely eliminating iterative copy of dirty data over the network. This good property is achieved without relying on any tracking mechanism for dirty data blocks. Experimental evaluation demonstrates the efficiency and effectiveness of DLSM.**

***Keywords*-Cloud Computing, IaaS, Live Storage Migration (LSM), Block Device**

I. Introduction

Resource virtualization is the foundation for cloud computing. Virtualization provides isolated, transparent, encapsulated, and manageable environment for both cloud service providers and end users. By supporting an elastic resource pool, virtualization allows cloud service providers and users to utilize the computing/storage resources more efficiently [7], such as load balancing, energy saving, host failure handling, and users' resource reassignment. In these basic functional modules, we often need to live migrate a virtual machine (VM) from one host to another without interrupting the current running applications in the VM [1].

In a nutshell, live VM migration needs to copy the memory and device state of a VM from one host to another host, with a negligible halt to handover the status information between hosts. The current live VM migration achieves negligible VM downtime, by allowing current running applications continues to run during the VM migration process. In particular, the live VM migration starts with copying the VM state from the source host to the destination host. The VM continues to run on the source, and the changes that it makes need to reflect to the destination. At a time near the end of the copy process, called the *halting time*, the source VM is suspended to allow the remaining differences to be copied to the destination, and after that the source VM is killed and the replica on the destination is made alive. The halting time is set to the time when only a small portion of data is left to move.

In the above live migration process, since the VM on the source host continues to run, new data may be created and such changes need to reflect on the destination host. We follow the tradition and call the new data "dirty data" [8], [5], even if the term may sound misleading because the "dirty data" are useful and must be updated to the destination host for consistency.

In early times, virtual disks are not migrated. Instead they reside on a shared volume, such as shared storage or distributed file system, accessible by both the source and destination hosts. These solutions, however, leave the problem to another layer of abstraction and cause semantic gap between the computing and storage resources, resulting in performance degradation and high management overhead. Therefore, live storage migration is often desirable due to a variety of reasons, such as better VM mobility and manageability, automatic storage load balancing, disaster recovery, storage maintenance, and storage upgrade [8, 12].

Live storage migration and live VM migration share some common technical challenges, in particular, the dirty data problem. Existing solutions focus on iterative data copying and use particular mechanisms such as as snapshot, dirty block tracking, and I/O mirroring [8]. Some work tries to optimize I/O intensive workload [10]. All of them iteratively copy live dirty blocks between the original host and the destination host. Iterative copying, intuitively, wastes a lot of network bandwidth and slows down the converging process, because the VM on the source host may make changes many times on the same address and each change needs to be updated on the destination. When the workload is data and I/O intensive, iterative copying poses a large penalty on guest QoS. In addition, iterative copying is not flexible for it needs to modify the hypervisor.

We observe that iterative data copying is actually unnecessary, because at the end only the last write on the same address needs to reflect to the destination and the intermediate changes during the live migration process can be ignored by the destination. In this paper, we propose a live storage migration method, called decoupled live storage migration (DLSM), without using any iterative data copying or dirty block tracking. DLSM strategically utilizes snapshot. It "migrates" the VM storage immediately without depending on the traditional shared storage, by building a snapshot in the destination host that acts as a shadow disk of the source. Actual data moving is deferred after the halting time of the live migration. Since iterative copying is eliminated, DLSM greatly reduces the amount of data transfer involved in the live migration. It decouples the VM migration and storage migration, and it is quite flexible since the storage migration can be deferred according to the demand of the user.

Compared to the state-of-the-art live storage migration method of VMWare ESX [8], DLSM brings the following benefits: (1) DLSM does not require dirty block tracking or I/O mirroring. VMWare ESX uses these mechanisms to improve the storage migration performance, but they incur high complexity in implementation and may introduce other problems such as a long convergence time. (2) DLSM involves less amount of data transfer and thus requires less network bandwidth. VMWare ESX also uses the snapshot technique but in a different way. It iteratively snapshots the image and consolidates the snapshots at the end of live migration. Iterative snapshotting, in principle, is similar to iterative copying and thus creates heavy network traffic. (3) DLSM does not need to modify the hypervisor.

II. Detailed Design of DLSM

The main goal of DLSM is to decouple the storage migration from the VM migration and to eliminate the iterative copying of dirty data over the network. In this section, we introduce DLSM in detail. Section II-A shows the image storage; Section II-B depicts the main components of DLSM, including snapshot and shadow, deferred transfer, I/O path switching, and the merge of local mapped devices. Furthermore, we analyze the amount of data transfer with DLSM and introduce an on-demand data transfer method.

A. Distributed Device Mapper Storage

Since live storage migration is closely related to live VM migration, we use the standard live VM migration method of the prevalent hypervisor, like KVM, to migrate the VM. Live storage migration is implemented at the storage layer. The storage layer solution makes storage migration transparent to the hypervisor and flexible enough for various cloud platforms.

To implement live storage migration, we choose the block level Device Mapper (DM) mechanism, which is supported in the Linux kernel. DM is a Linux kernel framework for mapping block devices onto higher-level virtual block devices. At the same time, to make the host-to-host storage migration work, we use the iSCSI (internet Small Computer System Interface) remote storage networking standard to map local devices to other host over the network. We call this mechanism Distributed Device Mapper Block Storage (DDMS), and it is much more light-weight than the distributed file system. DDMS works at the virtual block device level and is transparent to the virtual machine. We will show the implementation details of the DDMS in Section III.

B. Live Storage Migration with DDMS

Figure 1 shows the overall process of live storage migration with DDMS. Each subfigure has a source host and a destination host, both having DM block devices like snapshot, local cache, iSCSI virtual block device. The remote block devices are linked by iSCSI channel.

We describe the details as follows.

1) Snapshot and Shadow: Figure 1(a) shows the scenario before the migration request is sent. The VM is running on a block device like a logical volume based on LVM in the original host. Our goal is to migrate both the memory and the disk from the original host to the destination host.

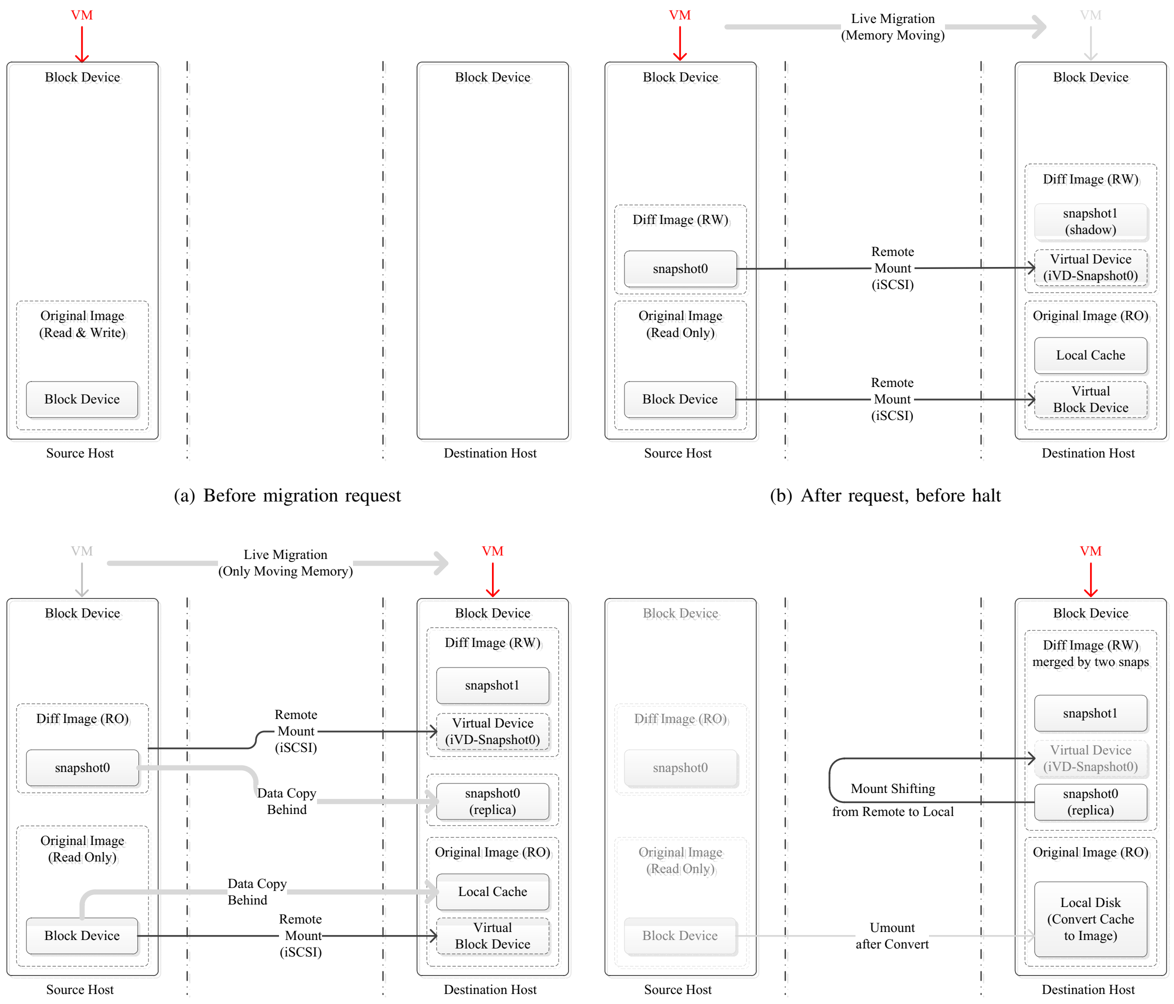
After the migration request is sent (the time point is labeled as t_0), and before the VM is halted for fully migration [1] (i.e., the halting time labeled as t_1), three main steps are carried out, as shown in Figure 1(b).

Step 1: A *diff* image based on redirected on write (RoW) snapshot is taken, which is labeled as $snapshot0$. This makes the original image a read-only, consolidated disk. All the write operations later on will be sent to the snapshot.

Before introducing the second step, we need to briefly introduce two types of snapshot: **Copy on Write** (CoW) snapshot and **Redirect on Write** (RoW) snapshot [11]. The two types of snapshot are similar but have different write directions. For example, when device B is taken as a snapshot of device A. CoW snapshot sets B logically invariant. It writes new data to A, and copy the old data to B for reservation. In contrast, RoW snapshot leaves A as read only, and writes all new data to B. In both methods, the snapshot shares the unwritten data logically on the original data, so we call the access to unwritten data as read through the clean data. Regularly, in the RoW mode, the original disk should be read only and never be written again, but in the CoW mode, we keep the snapshot disk untouched. The misuse of the two type of snapshot may cause chaos.

Step 2: Remotely build a "snapshot" of the $snapshot0$ via iSCSI channel (labeled as $snapshot1$). $snapshot1$ is a RoW snapshot based on virtual device of $snapshot0$ ($iVD - snapshot0$), and $iVD - snapshot0$ is a remote iSCSI initiator based on $snapshot0$. $iVD - snapshot0$ is only a logically mapped device of $snapshot0$ on the destination host, and it has no corresponding physical device. $snapshot1$ has its own physical disk, but has no data after the establishment. Importantly, $snapshot1$ will not be touched until time t_1, because the VM has not been migrated yet.

It is worthy noting that unlike regular snapshot, $snapshot1$ acts as neither CoW or RoW before t_1. During



(a) Before migration request

(b) After request, before halt

(c) After halt

(d) All done

Figure 1. Live storage migration with DDMS.

the period between t_0 to t_1, $snapshot1$ acts only as a **shadow** disk. Since the VM is running and may write new data to its disk (now still $snapshot0$), $snapshot0$ will be modified, but we do not send the dirty data to $snapshot1$. Logically, the data of $snapshot1$ should change when $snapshot0$ changes, like a shadow moving together with the object. Nevertheless, we make $snapshot1$ stay unchanged with no data inside. This would not cause any problem, because the VM in the destination host has not started running yet. After t_1, when the VM has been migrated to the destination host, this shadow disk is converted to a regular RoW snapshot. Because all the data written by the runtime VM between t_0 and t_1 is on $snapshot0$ (label $dataset0$) and is mapped logically to $snapshot1$, new data will be written on $snapshot1$ after t_1. Any read requests of $dataset0$ will be sent to the original host via network. In addition, $snapshot0$ is set to read-only after t_1.

Step 2 is the key step in our solution. It eliminates the consistency problem and there is no dirty data between t_0 and t_1 to be sent over the network. Furthermore, our solution does not need the I/O tracking or I/O duplicating operations to log each access explicitly.

Step 3: Another remotely mounted virtual block device will be established, and at the same time a local replica cache is set for this virtual block device. This cache device is another important device for the flexibility and better performance of live migration. When the cache is established, it works as a mirror device of the original virtual block device. When a read request comes from the upper layers, the cache will keep a replica for same request in the future. Furthermore, the cache can keep the minimum data on demand of the upper layers or keep the maximum data as the whole image.

Note that before the I/O path is switched (introduced in Section II-B3), the cache device is not actually used by

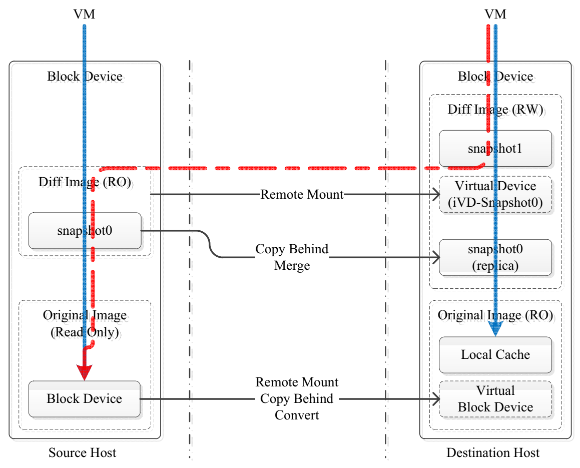


Figure 2. I/O path switch

the VM.

2) Deferred Data transfer: In Figure 1(c), after the VM is migrated to the destination host, the VM will be running on the virtual block device as same as before, since the underlying changes of virtual block device are transparent to the VM. To release the resource of the original host, the data and block device should be migrated to the destination host. However, such a move will not impact the running VM at all, because the data transfer is deferred after the halting time (i.e., the VM in the source host has been halted).

There are two main parts of data transfer: one is the read-only original image before t_0 and the other is $snapshot_0$, on which new data between t_0 and t_1 has been written. We name the new device to hold the $snapshot0$ data as $snapshot0(replica)$. Since $snapshot0$ is set read-only after t_1, copying data from $snapshot0$ to $snapshot0(replica)$ is easy. During the period of transferring the data, the VM will write new data in $snapshot1$, and read data from remote $snapshot0$ or local cache of the original image. $snapshot0(replica)$ is temporarily out of the I/O path and will not be accessed until the migration is finished.

3) I/O path switching: Figure 1(d) shows the scenario when all the data has been transferred (labeled as time t_2). Then the I/O path switches, and the old block device and data could be discarded on the original host. The live storage migration is done.

After t_2, $snapshot1$ shifts the mount point from $snapshot0$ to $snapshot0(replica)$, and $snapshot0$ will not be used any longer. Since all the image data has been transferred, the cache holds all the data from the original image. Because the $snapshot0$ and the original image are both read only, the new disks and old ones are exactly the same, i.e., there is no consistency problem.

Figure 2 illustrates the I/O path switching in more detail. The blue lines indicate the I/O path from the runtime VM through layers, and the red dashed line indicates the temporary I/O path during the time between t_1 and t_2.

4) Merge of Local mapped devices: The migration is finished after t_2. Yet, there are still some "house-cleaning" tasks. In our implementation, the snapshot and the cache device are based on the device mapper mechanism in Linux kernel. Device mapper maps logical devices to physical devices through a translation layer, and this translation layer may cause performance degradation. For better performance after live migration, we need to carry out two more tasks: (1) convert the cache to an original image disk, and (2) merge two snapshots into one snapshot.

First, caching every block of data will eventually makes the cache the same as the original image, and as such the only thing to do is to unmount the remote disk after the data transferring. Since we only need to transfer the actually used data on the disk, the transferring time of the original image is only relevant to the disk usage instead of the capacity of disk. To achieve this, an on-demand data transfer mechanism is used at the destination host.

Second, merging two snapshots can be easily done with the live snapshot merge technique such as Qemu. Live snapshot merge may have the data consistency problem and need to iteratively copy data. Nevertheless, the snapshot merge is performed on a local host, and can be done quickly. This is significantly different from iteratively copying data across the network.

In other words, we have shifted the data consistency problem from the network to the local host. We do not use any iterative data copying over the network and solve the dirty data problem with the merge of two local snapshots. This method significantly reduces the network traffic.

C. Analysis of Data Transfer

In this section, we analyze the data amount transferred during live migration.

1) One-time data transfer: We only transfer each data block once. The key idea is that the dirty data is written to the original host as if there is no migration, and these data blocks are transferred after the VM is migrated and are merged at the destination host.

In contrast, existing storage migration methods use iterative copying of dirty data in phases. In each phase, newly generated data needs to be copied to the remote host. The earlier versions of the data on these blocks are updated. This wastes time and network bandwidth – the earlier versions of the data have not been used, since the VM in the destination host has not started running. In addition, the iterative copying of dirty data over the network may introduce new problems for the write intensive applications. The migration may take a long time to converge, when the newly-generated data continuously overwrites the migrating data blocks. With dirty block tracking [8], if the dirty blocks are generated at a rate larger than the rate of data transfer over network, the storage migration cannot converge. To solve this problem, VMWare introduces a I/O mirroring method, which writes data to a region that is mirrored to the source and the destination. This solution, however, is much more complex than our snapshot based solution.

2) Less dirty data: The main reason of using iterative copying of dirty data across the network is that the storage migration starts ahead of VM migration, and the runtime VM on the source host may be still using the storage being migrated. In contrast, we postpone the starting time of storage migration behind the VM migration. By doing this, the time when the dirty data is created only depends on the time required for moving memory data.

The storage migration has a large impact on the VM migration. A coupled migration strategy may end up with a vicious circle of creating more and more dirty data, resulting in a long time for migration and a large amount of network traffic. All of these will bring significant performance penalty to guest applications. We call this phenomenon the *Storage Resistance* problem, i.e., the storage migration slows down the VM migration. This problem can be alleviated in DLSM, which decouples the storage migration and the VM migration.

Furthermore, when the rate of creating dirty data is higher than the network bandwidth, the migration process cannot converge. We call this situation the *migration convergence problem*. There are two types of convergence problem. One is related to the dirty data in memory, and the other is related to the dirty data written on disk. We call the former the *memory convergence problem*, and the latter the *storage convergence problem*. To alleviate the impact of the migration convergence problem, DLSM reduces the rate of creating dirty data, which will be demonstrated in our later experimental evaluation.

To summarize, we only transfer each block of data once over the network. More importantly, our method produces much less dirty data because the time for creating dirty data is reduced.

D. Deferred On-Demand Data Transfer

In some cases, the user or the service provider may not release the source host immediately. If this is the case, we can defer the transfer of untouched data on the original image of source host, and schedule their transfer on-demand, i.e., data transfer is triggered by I/O requests from the VM in the destination host.

Our method can easily support deferred on-demand data transfer. Because the VM live migration and the storage live migration are decoupled, the storage can be remotely and temporarily accessed after the VM has just been migrated to the destination host. In contrast, deferred on-demand data transfer is not possible in other methods, because storage data has to be copied before VM migration or when the VM is running.

III. Implementation

A. DLSM configuration

The live storage migration is based on distributed device mapper block storage, and thus the VMs are running on the block devices such as LVM and DM. There are block device modules in the current Linux kernel, but they are designed for local block devices. To turn the local block devices into distributed ones, we use the iSCSI protocol to provide the remote interface and the data transfer channel. Each iSCSI server(target)/client(initiator) pair converts a remote block device into a logically local one, called virtual block device. As the same as local block devices, distributed devices can also be stacked. Thus local block device can be established on the virtual devices created by the iSCSI initiator.

We established the cache device for the image on the destination host with a modified version of Flashcache [4], and the size of the cache device is set to equal the size of the original image. The original version of Flashcache cannot unmount the source image at any time. But in DLSM, after the VM is migrated and the whole data sets of the image are fetched from the source image on the source host into the cache image on the destination host, the cache image should be set to independent of the source image, and the link between both images is disconnected.

We reuse the snapshot module in the Linux kernel, but the shadowing mechanism is used differently from the regular usage of the RoW snapshot. In regular usage, if the device is the original device of the RoW snapshot, it cannot be written any more, and the new data is sent to the snapshot device. This is to avoid the consistency problem if we write on both the snapshot device and the original device. Our shadowing method is a bit different: we write to the original device after the RoW snapshot is created, and the snapshot is left unused afterward. Since the dirty data may make changes on the original device, the snapshot device should *logically* make changes following the original device. This creates no problem, because all the data is mapped to the snapshot device and there is no actual data on it, as described in Section II-B1 step 2. Actually, the RoW snapshot is based on the virtual device through the iSCSI channel on the destination host.

On the destination host, a simple on-demand image data fetcher is implemented to pull the data from the source host, and the local snapshot merger is reused as the merging mechanism of the DM snapshot.

Regarding the VM live migration in our test, we just use the KVM migration method.

B. Experimental Environment

We evaluate DLSM with two regular physical hosts, each having Intel Xeon quad core CPU, 16 GB memory, gigabyte ethernet (both card and switch), 2 TB SATA disk. CentOS 6.x OS are installed with KVM and libvirt as the VM hypervisor and virtualization API. The guest OS is Ubuntu server 12.04.3 (with 1 CPU core, 2GB RAM, 100GB virtual disk space and 1415 MB disk usage after installation).

C. Comparison Benchmark

The benchmark used for comparison is the libvirt/virsh method *Copy Storage Incremental*(CSI for short [6]). There are two types of LSM supported by KVM and libvirt, *–copy-storage-all* (CSA for short) and CSI. CSA migrates non-shared storage with full disk copy, while CSI migrates non-shared storage with incremental copy.

The incremental copy is supported with the specified image format, $qcow2$ [9]. The $incremental$ means that the image size can grow as data is added, and the copied data is only the data used (disk usage). This supports smaller image file sizes other than using the raw images always. CSA and CSI are the naïve methods realized by the hypervisor to copy the image data. Both copy dirty data iteratively from the source host to the destination. We use CSI for comparison because it tends to move smaller size of data and thus is generally faster.

IV. Evaluation

For fine-grained analysis, we recorded the DLSM migration time with two parts, the time for VM migration and the time for image migration. Due to the deferred on-demand data transfer, the image can be migrated any time after the VM migration. In our test, however, the on-demand data transfer starts right after the VM migration, and the image migration time is recorded as the time for transferring the total used data of the image. The migration time for CSI is set to be the time when the request is send to the time when the migration process is finished. Besides the migration time, we also test the guest performance penalty incurred by DLSM and CSI.

A. Comparison among Specified Workloads

We tested the performance with three different workloads: 1) CPU intensive workload, 2) memory intensive workload, and 3) I/O intensive workload.

1) CPU Intensive Workload: Table I presents the migration times of idle VMs. The guest operating systems both take about 103 MB memory. The results show that CSI migrates idle VM much slower than the DLSM. The traditional LSM like CSI considers VM migration and storage migration together and takes much longer time than DLSM, which decouples the two types of live migration. This is because the storage migration in CSI prolongs the time for dirty data creation and produces more data to move.

Table I
Migration time of idle VM

	CSI	DLSM		
		VM	Image	Total
Time (s)	38.36	5.34	16.37	21.71

Figure 3 shows the migration time under different CPU usages. We run the program of $quicksort$, which was controlled by $cgroup$ with a fixed memory usage (with OS running total 666 MB). To fix the memory usage, the computing scale of $quicksort$ is set to be a constant. In this test, both CSI and DLSM encountered the problem of migration convergence. The migration could not converge with CSI when the load is larger than 50%, and could not converge with DLSM when the load is larger than 90%. The DLSM convergence problem is caused by the VM migration method of KVM. There exists a critical point in all LSM, after which the rate of creating dirty data

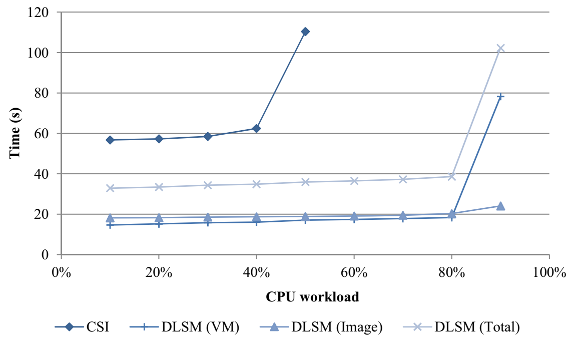


Figure 3. Comparison among varied CPU load

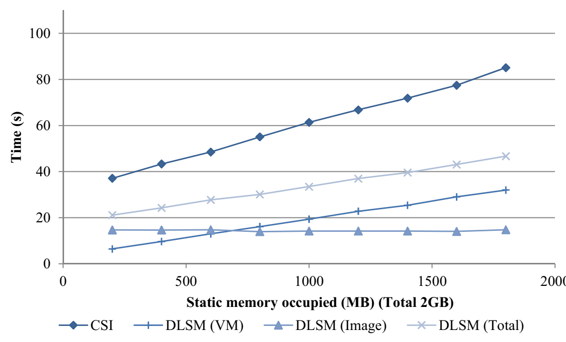
becomes larger than the network bandwidth. The above test result shows that the critical point of DLSM is much higher than that of CSI, indicating that DLSM can tolerate higher CPU workload without causing the convergence problem.

2) Memory Intensive Workload: The memory occupied by the guest OS has a large impact on the performance of LSM. Figures 4(a) and 4(b) show the migration time of CSI and DLSM under different memory loads.

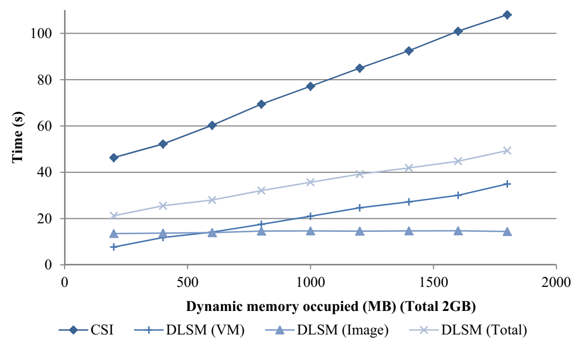
Figure 4(a) shows the results under static memory load. The test was performed by a custom benchmark using $malloc$ in C with the memory space filled with random data. Since the data does not change, very little dirty data was produced in the migration period. In this case, the migration time mainly depends on the size of occupied memory space. As shown in Figure 4(a), the image migration time with DLSM remains stable, even if the memory load varies. From the figure, we can see that the total migration time with CSI is higher than the total migration time with DLSM.

Figure 4(b) shows the results under dynamic memory load. The test was performed by running $quicksort$ controlled by $cgroup$. The CPU usage was fixed to about 30%. Under the same CPU usage, we changed the computing scale of the $quicksort$ to vary the memory usage, and test the migration time under each situation. Unlike the static memory case, this test changed the data in memory constantly. In this case, more dirty data was produced. Comparing Figures 4(a) and 4(b), we can see that CSI incurred a much higher increase in migration time under dynamic memory load. In contrast, the performance of DLSM only changed slightly under dynamic memory load. Table II further shows the numerical results of VM migration time of DLSM under the static and dynamic memory workloads.

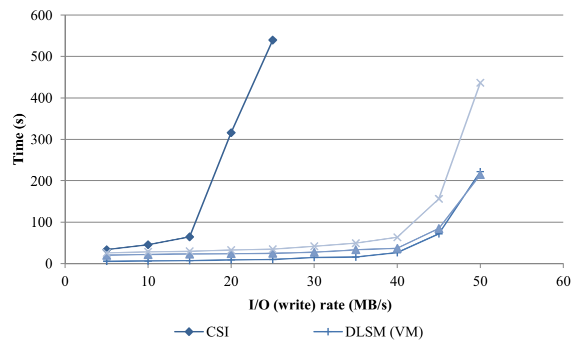
3) I/O Intensive Workload: Figures 4(c) and 4(d) show the migration time of CSI and DLSM under different I/O workloads (only with $write$). The test was performed by running FIO [3] benchmark. We synchronized the written data to the disk after each FIO test, so that the data was written to disk and not remained in memory. Figure 4(c) shows the evaluation results under different I/O rates, while Figure 4(d) shows the situation under different I/O



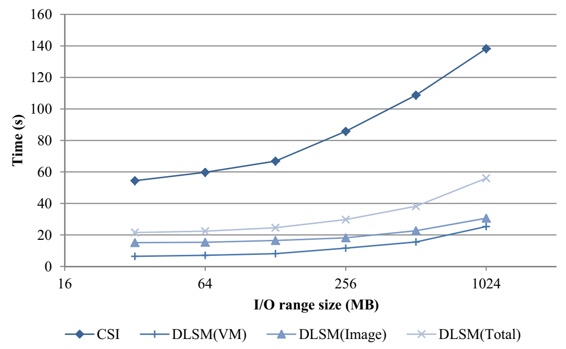
(a) Comparison among varied static memory load



(b) Comparison among varied dynamic memory load



(c) Different I/O rate (sequentially)



(d) Different I/O range (10MB/s)

Figure 4. Comparison among varied memory load and I/O (write) load

Table II
DLSM MIGRATION TIME BETWEEN STATIC AND DYNAMIC MEMORY WORK LOAD

	Static Time (s)	Dynamic Time (s)
200 MB	6.46	7.71
400 MB	9.62	11.82
600 MB	12.99	15.9
800 MB	16.13	17.51
1000 MB	19.32	21.01
1200 MB	22.79	24.66
1400 MB	25.37	27.22
1600 MB	29.04	30.05
1800 MB	31.97	34.94

locations (i.e., in one test, we randomly selected a write location and repeatedly wrote random data in the same location until the migration is finished).

From the figures, we can see that the migration time for CSI increases dramatically when the write rate is above 20 MB/s. CSI in this test encountered the problem of disk convergence. As shown in Figure 4(c), when the writing rate is higher than 20 MB/s, the disk convergence problem poses a large penalty to CSI, increasing its migration time quickly. In contrast, DLSM is resilient to I/O workload changes under 50 MB/s. Its performance mainly depends on CPU usage, which remained relatively stable in this test. However, it does not perform well if the writing rate is too high, due to the limitation of the VM live migration with KVM.

Table III
PARAMETERS FOR FILEBENCH WORKLOADS

Service	#File	#Thread	File size (KB)	I/O size (MB)	Dominate Oper
File Svr	10k	50	128	1	R|W
Web Svr	1k	100	16	1	R
DB Svr	10	0.2M	10	0.002	R|W|S
Mail Svr	1k	16	16	1	R|A|S

Figure 4(d) shows the test results of different writing locations with the constant write rate (10 MB/s). This result clearly shows that DLSM performs much better than CSI.

B. Macro-benchmarks

We used $Filebench$ [2] to test several popular macro-benchmarks, such as file server, web server, data base server, and mail server, to show how the LSM influences the guest performance for different real-world applications. All the benchmarks are set to run about 120 second. Table III shows the parameters in test (R=read, W=write, A=append, S=synchronize). We set a reference case that no migration was requested in the same evaluation environment. Table IV shows the test results.

The results show that with CSI, file server, mail server and DB server finish migration after the benchmark test is over (the red numbers in the table). This means that CSI encountered the migration convergence problem, which seriously hurts the performance of the cloud service. In contrast, DLSM migrates both the VM and the storage within the runtime of the benchmark, incurring negligible performance penalty. Web server with CSI did not have the migration convergence problem, because the dominate operations are read and thus no dirty data (or very little) was created.

Table III shows that DB server and Mail server seemed to perform better with CSI than with DLSM. Note that, the VM migration with CSI is postponed after the end of the benchmark test, so the impact of VM migration on the benchmark performance has not been counted here. The results do not indicate that the CSI is better, because the migration has not been actually fulfilled at the end of benchmark test.

Table IV
FILEBENCH RESULT (RUNTIME = 120S)

		MIG Time(s)	iops	Throughput (MB/s)	Latency (ms)
File Svr	no MIG		3294.938	78.6	34.4
	CSI	213.65	1709.83	40.6	97.1
	DLSM	90.55[a]	2541.53	60.3	32
Web Svr	no MIG		21470.093	107.6	<0.1
	CSI	78.49	16532.04	82.9	0.5
	DLSM	44.49[b]	18611.21	93.3	0.2
DB Svr	no MIG		940.252	1.8	0.5
	CSI	151.15	856.496	1.7	0.5
	DLSM	44.67[c]	807.59	1.6	0.8
Mail Svr	no MIG		520.789	1.9	99.4
	CSI	139.19	487.309	1.7	106.1
	DLSM	36.72[d]	323.5	1.2	154

a: 24.35+66.20; b: 5.62+38.87; c: 7.22+37.45; d: 5.71+31.01.

V. CONCLUSION

We designed, implemented, and tested a live storage migration (LSM) method, called DLSM. DLSM provides a new perspective on the interplay between LSM and live VM migration, as well as a new solution to alleviate the performance degradation caused by LSM. It has the following benefits: First, it is easy to implement and has a shorter convergence time, because it does not requires dirty block tracking or I/O mirroring. Second, it requires less data transfer across the network and thus significantly saves network bandwidth. Third, it does not need to modify the hypervisor. Experimental results show that DLSM outperforms other traditional LSM methods, such as the Copy Storage Incremental (CSI) method supported by KVM and libvirt.

ACKNOWLEDGMENT

This work was supported by the National Basic Research Program of China under Grant No.2011CB302601 and National Natural Science Foundation of China under Grant No.61272483.

REFERENCES

[1] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. NSDI, 2005.
[2] Filebench. http://sourceforge.net/projects/filebench/.
[3] FIO. http://freecode.com/projects/fio.
[4] Flashcache. https://github.com/facebook/flashcache/.
[5] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. ACM International Conference on Virtual Execution Environments, 2009.
[6] Libvirt.org. Nbd storage migration, http://wiki.libvirt.org/page/nbd_storage_migration.
[7] Xicheng Lu, Huaimin Wang, Ji Wang, Jie Xu, and Dongsheng Li. Internet-based virtual computing environment: Beyond the data center as a computer. *Future Generation Computer Systems*, 2011.
[8] Ali Mashtizadeh, Emr E. Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in vmware esx. USENIX ATC, 2011.
[9] M. McLoughlin. The qcow2 image format, http://people.gnome.org/∼markmc/qcow-image-format.html.
[10] Bogdan Nicolae and Franck Cappello. A hybrid local storage transfer scheme for live migration of i/o intensive workloads. HPDC. ACM, 2012.
[11] W. Xiao, Y. Liu, Q. Yang, J. Ren, and C. Xie. Implementation and performance evaluation of two snapshot methods on iscsi target storages. Mass Storage Systems and Technologies, 2006.
[12] Ruijin Zhou, Fang Liu, Chao Li, and Tao Li. Optimizing virtual machine live storage migration in heterogeneous storage environment. ACM International Conference on Virtual Execution Environments, 2013.