Live and Incremental Whole-System Migration of Virtual Machines Using Block-Bitmap

Yingwei Luo [#1], Binbin Zhang [#], Xiaolin Wang[#], Zhenlin Wang [*2], Yifeng Sun[#], Haogang Chen[#]

[#] *Department of Computer Science and Technology, Peking University, P.R.China, 100871*

[1] lyw@pku.edu.cn

[*] *Dept. of Computer Science, Michigan Technological University, Houghton, MI 49931, USA*

[2] zlwang@mtu.edu

***Abstract*—In this paper, we describe a whole-system live migration scheme, which transfers the whole system run-time state, including CPU state, memory data, and local disk storage, of the virtual machine (VM). To minimize the downtime caused by migrating large disk storage data and keep data integrity and consistency, we propose a three-phase migration (TPM) algorithm. To facilitate the migration back to initial source machine, we use an incremental migration (IM) algorithm to reduce the amount of the data to be migrated. Block-bitmap is used to track all the write accesses to the local disk storage during the migration. Synchronization of the local disk storage in the migration is performed according to the block-bitmap. Experiments show that our algorithms work well even when I/O-intensive workloads are running in the migrated VM. The downtime of the migration is around 100 milliseconds, close to shared-storage migration. Total migration time is greatly reduced using IM. The block-bitmap based synchronization mechanism is simple and effective. Performance overhead of recording all the writes on migrated VM is very low.**

I. INTRODUCTION

VM migration refers to transferring run-time data of a VM from one machine (the source) to another machine (the destination). After migration, VM continues to run on the destination machine. Live migration is a migration during which the VM seems to be responsive all the time from clients' perspective. Most research focuses on migrating only memory and CPU state assuming that the source and destination machines use shared disk storage. But in some scenarios, the source and destination machines cannot share the disk storage. So the local disk storage should also be migrated. This paper describes a whole-system live migration, which moves all the VM state to the destination, including memory data, CPU state, and local disk storage. During the migration, the VM keeps running with a negligible downtime.

We propose a **Three-Phase Migration (TPM)** scheme to minimize the downtime while maintaining disk storage data integrity and consistency. The three phases are pre-copy, freeze-and-copy, and post-copy. The original VM is only suspended during the freeze-and-copy phase and then resumes on the destination machine. In the pre-copy phase, before the local memory is pre-copied, local disk storage data are iteratively transferred to the destination while using a block-bitmap to track all the write accesses. In the freeze-and-copy phase, the block-bitmap, which contains enough information for later synchronization, is sent to the destination. In the post-copy phase, we take an approach that combines *pull* and *push*. According to the block-bitmap, the destination *pulls* a dirty block if it is accessed by a read request, while the source *pushes* the dirty blocks continuously to ensure that the synchronization can be completed in a finite time. A write request in the destination to a dirty block will overwrite the whole block and thus does not require pulling the block from the source VM.

We developed an **Incremental Migration (IM)** algorithm to greatly reduce the migration time. The block-bitmap continues to track all the write accesses to the disk storage in the destination after the primal migration and only the new dirty blocks need to be synchronized if the VM needs to migrate back to the source machine later on. IM will be very useful when the migration is used for host machine maintenance and the migration back and forth between two places to support telecommuting, for instance.

In our design and implementation, we intend to minimize downtime and disruption time such that the clients can barely notice the service interruption and degradation. We further control total migration time and amount of data transferred. These metrics will be explained in detail in section III.

The rest of the paper is structured as follows. In section II we discuss related work. In section III we analyze the problem requirements and describe the metrics to evaluate the VM migration performance. In section IV and section V we describe TPM and IM in detail, including their design and some implementation issues. In section VI we describe our evaluation methodology and present the experimental results. Finally we conclude and outline our future work in section VII.

II. RELATED WORK

In this section, we discuss the existing research on VM migration, including live migration with shared disk storage and whole-system migration with local disk storage.

A. Live Migration with Shared Disk Storage

Two representative live migration systems, Xen live migration [1, 11] and VMware VMotion, share similar implementation strategies. Both of them assume shared disk storage. Take Xen live migration as an example. It uses a pre-copy mechanism that iteratively copies memory to the destination, while recording dirty memory pages. Then at a right time, it suspends the VM, and copies the remaining dirty memory pages and CPU state to the destination. It resumes the

VM at the destination after all the memory has been synchronized. Because only a few pages may be transferred during VM pausing, the downtime is usually too short for a client to notice. Both Xen live migration and VMotion only focus on the memory state and run-time CPU state; So VM can be migrated only between two physical machines using shared storage.

B. Whole-System Migration with Local Disk Storage

Whole-system migration will migrate the whole-system state of a VM, including its CPU state, memory data, and local disk storage data, from the source to the destination machine.

A simple way to migrate a VM with its local storage is freeze-and-copy, which first freezes the VM to copy its whole-system state to the destination, and then restarts the VM at the destination. Internet Suspend/Resume [3, 5] is a mature project using freeze-and-copy to capture and transfer a whole VM system. A copy and only the copy of all the VM run-time state are transferred without any additional redundancy. It results a severe downtime due to the large size of the storage data. The Collective [4, 10] project also uses the freeze-and-copy method. It introduces a set of enhancements to decrease the size of transmitted data. All the updates are captured in a Copy-on-Write disk. So only the differences of the disk storage need to be migrated. However, even transferring disk updates could causes significant downtimes.

Another method is on-demand fetching [5], which first migrates memory and CPU state only with delayed storage migration. The VM immediately resumes on the destination after the memory and CPU state migration. It then fetches storage data on-demand over network. The downtime is the same to the shared-storage migration downtime. But it will incur residual dependence on source machine, even an irremovable dependence. So on-demand fetching can't be utilized for source machine maintenance, load-balance migration, or other federated disconnected platforms such as Grids and PlanetLab. Furthermore, it actually decreases system availability, for its dependency on two machines. Let p (p<1) stand for a machine's availability, then the migrated VM system's availability is p^2, which is less than p. Considering the network connection failure, the actual availability must be less than p^2.

Bradford *et al*. propose to pre-copy local storage state to the destination while VM still running on the source [6]. During the migration all the write accesses to the local storage are recorded and forwarded to the destination, to ensure consistency. They use a *delta*, a unit consisting of the written data, the location of the write, and the size of the written data, to record and forward the write access for synchronization. After the VM resumes on the destination, all the write accesses must be blocked before all forwarded deltas are applied. It shows the same downtime to the shared-storage migration. But it may cause a long I/O block time for the synchronization. Furthermore there may be some redundancy in the *delta* queue, which can frequently happen because of locality of storage accesses.

In conclusion, there is still much to do to find out how to migrate large-size local storage in an endurable migration time while remaining a short downtime, how to synchronize storage state using as less redundant information as possible, and how to keep a finite dependency on the source machine. This paper addresses these questions.

III. Problem Analysis and Definition

The goal of our system is to migrate the whole-system state of a VM from the source to the destination machine, including its CPU state, memory data, and local disk storage data. During the migration time the VM keeps running. This section describes the key metrics and requirements for a whole-system live migration.

A. Definition of the Metrics

The following metrics are usually used to measure the effectiveness of a live migration scheme:

- **Downtime** is the time interval during which services are entirely unavailable [1]. It is the time from when VM pauses on the source machine to when it resumes on the destination. Synchronization is usually performed in downtime. So the synchronization mechanism impacts on downtime.
- **Disruption time** is the time interval during which clients connecting to the services running in the migrated VM observe degradation of service responsiveness—requests by the client take longer response time [6]. It is the time during which the services on the VM show lower performance due to the migration from a client's perspective. The transfer rates and methods for synchronization have influence on disruption time.
- **Total migration time** is the duration from when the migration starts to when the states on both machines are fully synchronized [1]. Decrease the size of transferred data, e.g. to compress the transferred data before sending it, will show a reduction in total migration time.
- **Amount of migrated data** is the amount of data transmitted during the whole migration time. The minimal amount is the size of the run-time states, including the memory size, storage size, CPU state size, etc.. Usually it will be larger than the actual run-time state size, except for the freeze-and-copy method, because there must be some redundancy for synchronization and protocols.
- **Performance overhead** is the decrement of the service performance caused by migration. It is evaluated by the comparison of the service throughput during the migration and without migration.

A high-bandwidth network connection between the source and the destination will decrease downtime, disruption time, and migration time to a certain extent.

B. Requirements for a Whole-System Live Migration

Based on the metrics discussed in section III-A, an ideal VM migration is a whole-system migration with short downtime, minimized disruption time, endurable migration time, and negligible performance overhead. And it only transfers the run-time states without any redundancy. But this

ideal whole-system live migration is hard to implement. Transferring large-volume local storage incurs a long migration time. It is difficult to maintain the consistency of the storage between the source and destination during such a long migration time while retaining a short downtime. The design of our system focuses on the following requirements:

- **Live migration**: VM keeps running during most time of the migration process. In other words, clients can't notice that the services on the VM are interrupted during the migration.
- **Minimal downtime**: An ingenious synchronization method is required to minimize the size of the data transmitted in the downtime.
- **Consistency**: The VM's file system is consistent and identical during migration except downtime.
- **Minimizing performance overhead**: A non-redundant synchronization method and a set of simple protocols must be designed. And the bandwidth used by the migration process should be limited to ensure the performance of the services on the migrated VM.
- **Finite dependency on the source machine**: The source machine can be shutdown after migration. That means synchronization must be completed in a finite period of time.
- **Transparency**: Applications running on the migrated VM don't need to be reconfigured.
- **Minimizing migration time**: This can be achieved if a part of the state data need not be transmitted.

Our TPM and IM algorithms are designed to satisfy these requirements. The following two sections will describe TPM and IM in detail.

IV. Three-Phase Migration

The TPM algorithm aims at whole-system live migration. This section describes its design and implementation.

A. Design

Migration is a process to synchronize VM state between the source and the destination machine. Live migration requires the synchronization complete with a short downtime, while whole-system migration requires a large amount of state data be synchronized. TPM is designed to migrate the whole system state of VM while keeping a short downtime.

1) Three Phases of TPM: The three phases of TPM are pre-copy, freeze-and-copy, and post-copy. Most of the run-time data are transferred in pre-copy phase. The VM service is not available only in freeze-and-copy phase. And local disk storage data needs to be synchronized in post-copy phase. The process of TPM is illustrated in Figure 1.

In the **pre-copy** phase, the storage data are pre-copied iteratively. During the first iteration, all the storage data should be copied to the destination. For the later iterations only the latest dirtied data during last iteration need to be sent. We limit the maximum number of iterations to avoid endless migration. In addition, if the dirty rate is higher than the transfer rate, the storage pre-copy must be stopped proactively.

In the **freeze-and-copy** phase, the migrated VM is suspended on the source machine. Dirty memory pages and CPU states are transferred to the destination. All inconsistent blocks that have been modified during the last iteration of storage pre-copy are marked in the bitmap. So only the bitmap needs to be transferred.

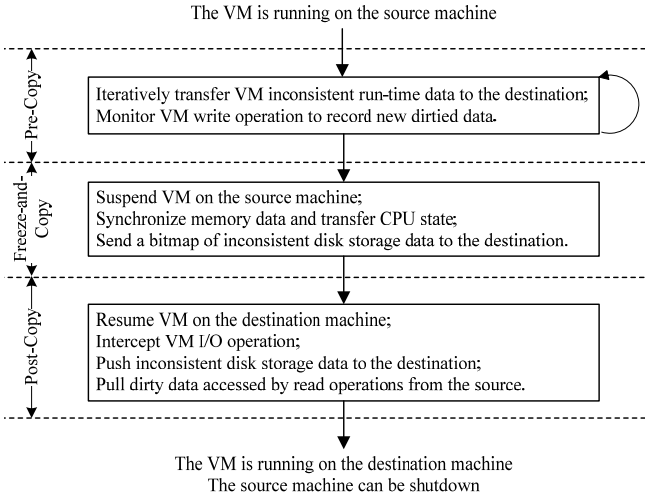

Fig. 1. Three-Phase whole-system live migration

In the **post-copy** phase, the migrated VM is resumed on the destination machine. The source begins to push dirty blocks to the destination according to the bitmap, while the destination uses the same block-bitmap to pull the dirty blocks requested by the migrated VM. The pulling occurs and only occurs when the VM submits a read access to a dirty block. So the destination must intercept all I/O requests from VM and check if a block must be pulled.

2) Block-bitmap: A bitmap is used to record the location of dirty disk storage data during migration. A bit in the bitmap corresponds to a unit in disk storage. 0 denotes that the unit is clean and 1 means it is dirty.

Bit Granularity. Bit granularity means the size of a unit in disk storage described by a bit. Though 512B sector is the basic unit on which physical disk performs reading and writing, modern OS often reads from or writes to disk by a group of sectors as a block, usually a 4KB block. So we prefer to choose the bit granularity at block level rather than at sector level, that is, to map a bit to a block rather than to a sector. For a 32GB disk, a 4KB-block bitmap costs only 1MB memory, but a 512B-sector bitmap will use up to 8MB. When disk size is not too large, a 4KB-block bitmap works very well.

Layered-Bitmap. For each iteration in the pre-copy phase, the bitmap must be scanned through to find out all the dirty blocks. If the bitmap is large, the overhead is severe. I/O operation often show high locality, so bit 1's are often clustered together, and the overall bitmap remains sparse. A layered bitmap can be used to decrease the overhead. That is, a bitmap is divided into several parts and organized as two layers. The upper layer records whether these parts are dirty. If the bitmap must be checked through, the top layer is

checked first, and then only the parts marked dirty need to be checked further. When using layered-bitmap, the lower parts are allocated only when there is a write access to this part, which can reduce bitmap size and save memory space.

Bradford *et al*. [6] use a forward and replay method to synchronize disk storage data. During pre-copy phase, all the write operations are intercepted and forwarded to the destination. On the destination all these writes are queued and will apply to the migrated disk after disk storage pre-copy is completed. Write throttling must be used to ensure that the network bandwidth can catch up with the disk I/O throughput in some disk I/O intensive workloads. And after migrated VM is resumed on the destination, its disk I/O must be blocked until all the records in the queue have been replayed. Furthermore, there will be some redundant records which write to a same block. It will increase the amount of migrated data so as to enlarge the total migration time and I/O blocked time. We have checked the storage write locality using some benchmarks. When we make a Linux kernel, about 11% of the write operations rewrite those blocks written before. The percentage is 25.2% in SPECweb Banking Server, and 35.6% while Bonnie++ is running.

In our solution all the inconsistent blocks are marked in the block-bitmap, and can be lazily synchronized until VM resumed on the destination. It works well in I/O intensive workloads, avoiding I/O block time on the destination and essentially solving the redundancy problem in recording and replaying all the write operations. Our solution may increase the downtime slightly due to transferring the block-bitmap. But in most scenarios, the block-bitmap is small (1MB-bitmap per 32GB-disk, and smaller if layered-bitmap is used) and the overhead is negligible.

3) Local Disk Storage Synchronization: We use a block-bitmap based method to synchronize local disk storage. In the pre-copy phase, a block-bitmap is used to track write operations during each iteration. At the beginning of each iteration, the block-bitmap is reset to record all the writes in the new iteration, during which all the data marked dirty in the previous iteration must be transferred.

In the freeze-and-copy phase, the source sends a copy of the block-bitmap, which marks all the inconsistent blocks, to the destination. So at the beginning of the post-copy phase, the source and the destination both have a block-bitmap with the same content. The post-copy synchronizes all the inconsistent blocks according to these two block-bitmaps. At the same time, a new block-bitmap is created to record the disk storage updates on the destination, which will be used in IM described in section V. The source pushes the marked blocks continuously and sends the pulled block preferentially if a pull request has been received, while the destination performs as follows:

DEFINE:

- An I/O request *R*<O, N, VM>, where O is the operation, WRITE or READ, N is the operated block number, and VM is the ID of the domain which submits the request.
- Transferred_block_bitmap: A block-bitmap marks all the blocks inconsistent with the source at the beginning of the post-copy.
- New_block_bitmap: A block-bitmap marks the new dirtied blocks on the destination.

```
1.  An I/O request R<O, N, VM> is intercepted;
2.  Queue R in the pending list P;
3.  IF R.VM != migrated VM
4.    THEN goto 14;
5.  IF R.O == WRITE // no pulling needed
6.    THEN{
7.          new_block-bitmap[N] = 1;
8.          transferred_block_bitmap[N] = 0;
9.          goto 14;
10.       }
11. IF transferred_block-bitmap[N] == 0 //clean block
12.   THEN goto 14;
13. Send a pulling request to the source machine for
    block N, goto 16;
14. Remove R from P;
15. Submit R to the physical driver;
16. End;
```

The destination intercepts each I/O request. If the request is from other domain than the migrated VM (line 3), submit it directly. Otherwise, if the request is a write (lines 5-10), we use a new block bitmap to track this update (line 7) and reset the corresponding state in the bitmap for synchronization (line 8). If the request is a read (lines 11-13), a pulling request is sent to the source machine only when the accessed block is dirty (line 13).

Finally the destination must check each received block to determine if it is a pushed block or a pulled one:

```
1.  A block M is received;
2.  IF transferred_block-bitmap[M] == 0
3.    THEN goto 12;
4.  Update block M in the local disk;
5.  transferred_block-bitmap[M]=0;
6.  For each request R_i in P
7.    IF R_i.N == M
8.      THEN{
9.            Remove R_i from P;
10.           Submit R_i;
11.         }
12. End;
```

The pushed block is dropped if there was a write in the destination that reset the bitmap (lines 2-3). If it is a pulled block, the pulling request is removed from the pending request queue (lines 6-11) and local disk will be updated accordingly (line 10).

4) Effectiveness Analysis on TPM: TPM is a whole-system live migration, which satisfies the requirements listed in section III.

Live migration and minimal downtime: In the freeze-and-copy phase, only dirty memory pages and the block-bitmap need to be transferred. So the downtime depends on the block-bitmap transfer time and memory synchronization time. In most scenarios, the dirty bitmap is small. The size can be even reduced greatly if we use the layered block-bitmap as analyzed in section IV-A-2. And memory synchronization time is very short as indicated in the Xen live migration research [1].

To keep consistency: In the post-copy phase, all the I/O requests from the migrated VM are intercepted and synchronization is necessary only if it is a read to dirty data.

To minimize performance overhead: The performance overhead can be limited if we limit the bandwidth used by migration, which will increase total migration time correspondingly (see section VI-C-3). Another approach is to use a secondary NIC (Network Interface Card) for the migration, which can help limit the overhead on network I/O performance, but it has no effect on releasing the stress on disk during migration.

To make a finite dependency on the source machine: We use push-and-pull to make the post migration convergent, avoiding a long residual dependency on the source by the pure on-demand fetching approach.

To be transparent: Storage migration occurs at the block level. The file system cannot observe the migration.

B. Implementation

We expand Xen live migration to implement a prototype of TPM. To make our description easy to follow, we first introduce some notations in Xen. A running VM is named **Domain**. There are two kinds of domains. One is privileged and can handle the physical devices, referred to as **Domain0**. The other is unprivileged and referred to as **DomainU**. Split drivers are used for DomainU disk I/O. A **frontend driver** in DomainU acts as a proxy to a **backend driver**, which works in Domain0 and can intercept all the I/O requests from DomainU. **VBD** is the abbreviation of Virtual Block Device acting as a physical block device of a Domain.

The process of our implementation of TPM is illustrated in Figure 2. The white boxes show Xen live migration process, and the grey boxes shows our extension.

Disk storage data are pre-copied before memory copying because memory dirty rate is much higher than disk storage and the disk storage pre-copy lasts very long. A large amount of dirty memory can be produced during the disk storage pre-copy. Simultaneous or premature memory pre-copy is useless.

We design a user process named *blkd* to do most work of storage migration. Xen's original functions *xc_linux_save* and *xc_linux_restore* are modified to direct *blkd* what to do at certain time. We modify the block backend driver, *blkback*, to intercept all the write accesses in the migrated VM and record the location of dirtied blocks into the block-bitmap. All the modifications are described as follows.

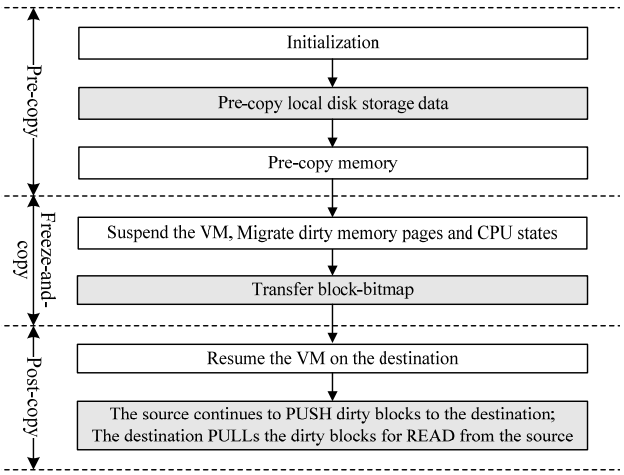

Fig. 2. Process of TPM implemented based on Xen Live Migration

- Modify initialization of migration to ask the destination to prepare a VBD for the migrated VM.
- Modify *xc_linux_save*. Before the memory pre-copy starts, it will signal *blkback* to start monitoring write accesses, and then signal *blkd* to start pre-copying local disk storage and block itself until the disk storage pre-copy completes. After the pre-copy phase, it will signal *blkd* to send the block-bitmap and enter the post-copy phase.
- Modify *xc_linux_restore*. Before receiving pre-copied memory pages, it will signal *blkd* to handle local disk storage pre-copy, and block itself until disk storage pre-copy completes. After the migrated Domain is suspended, it will signal the *blkd* to receive the block-bitmap and enter the post-copy phase before resuming the migrated Domain.
- Modify *blkback* to register a Proc file and implement its read and write functions to export control interface to *blkd* for communication. Then *blkd* can write the Proc file to configure *blkback* and read the file for the block-bitmap. *Blkback* maintains a block-bitmap and intercepts and records all the writes from the migrated domain. The block-bitmap is initialized when the migration starts. At the beginning of each iteration of pre-copy, after the block-bitmap is copied to *blkd*, it is reset for recording dirty blocks in the next iteration. If the *blkback* intercepts a write request, it will split the requested area into 4K blocks and set corresponding bits in the block-bitmap.

The user process *blkd* acts according to the signals from *xc_linux_save* and *xc_linux_restore*. When it receives a local disk storage pre-copy signal, it starts iterative pre-copy. During each iteration, it first reads the block-bitmap from the backend driver, *blkback*. Then it sends the blocks which are marked dirty in the block-bitmap.

In the freeze-and-copy phase, *xc_linux_save* signals *blk* to send the block-bitmap to the destination.

In the post-copy phase, as illustrated by Figure 3, the *blkd* on the source machine pushes (action 1) the dirty blocks to the destination according to block-bitmap BM_1, while it listens to the pull requirements (action 3) and sends the pulled block preferentially. On the destination, the *blkback* intercepts the requests from the migrated VM and forward them to *blkd* (action 2). *Blkd* checks if the blocks accessed by a request must be pulled according to the block-bitmap BM_2 and the rules described in section IV-A-3. It will send the source a request if the block must be pulled (action 3). And *blkd* will tell *blkback* (action 4) which requests can be submitted to the physical disk driver after a pulled block has been received and write into the local disk (action 5). All the writes in DomU are intercepted in *blkback* and marked in block-bitmap BM_3, which will be used in IM described in section V.

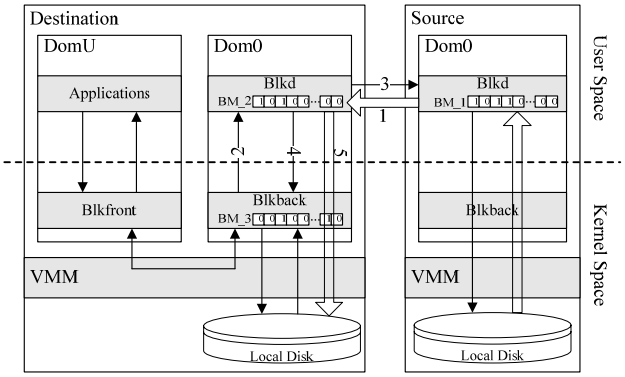

Fig. 3. The Implementation of Post-copy

V. Incremental Migration

Our experiments show that the TPM can also result a long migration time, due to the large size of the local storage data. Fortunately, in many scenarios, migration is used to maintain the source machine, or to relocate the working environment from office to home, for instance. A VM migrated to another machine may be migrated back again later, e.g., after the maintenance is done on the source machine, or the user need to move the environment back to his/her office. In these scenarios, if the difference between the source and the destination is maintained, only the difference needs to be migrated. Even in those I/O intensive scenarios, the storage data to be transferred can be decreased significantly using this **Incremental Migration (IM)** scheme. Figure 4 illustrates the process of IM.

The grey box shows that in the pre-copy phase, the block-bitmap should be checked to find out all the dirty blocks after last migration. Only those dirty blocks need to be transferred back in the first iteration. So after the VM is resumed on the destination all the newly dirtied blocks of the migrated VM must be marked in a block-bitmap as mentioned in section IV-A. So in the post-copy phase of TPM, two block-bitmaps are used. One is transferred from the source and records all the unsynchronized blocks; the other is initialized when the migrated VM is resumed on the destination, and is used for recording the newly dirtied blocks on the destination. When the migrated VM needs to be migrated back to the source, only the blocks marked in the new block-bitmap need to be transferred.

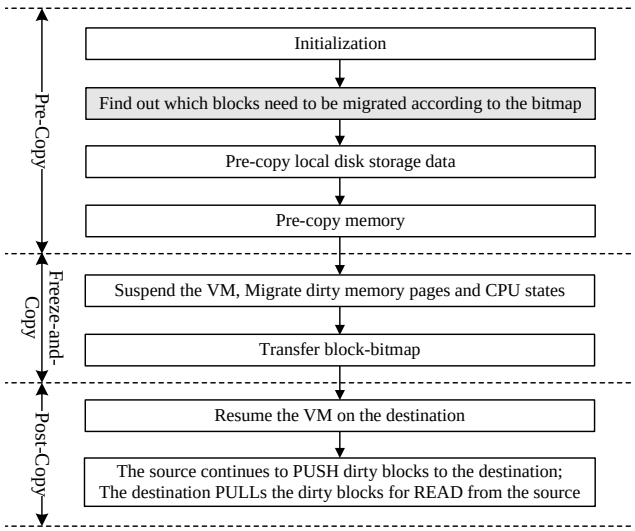

Fig. 4. Process of IM

The implementation is a minor modification to the TPM. We check if the bitmap exists before the first iteration. If it does, only the blocks marked dirty in the block-bitmap need to be migrated. Otherwise an all-set block-bitmap is generated, suggesting that all the blocks need to be transmitted.

VI. Evaluation

In this section we evaluate our TPM and IM implementation using various workloads. We first describe the experimental environment and list the workloads. We then present the experimental results including downtime, disruption time, total migration time, amount of migrated data, and performance overhead.

A. Experimental Environment

We use three machines for the experiments. Two of them share the same hardware configuration, which is Core 2 Duo 6320 CPU, 2GB memory, SATA2 disk. The software configuration is also the same: Xen-3.0.3 with XenoLinux-2.6.16.29 running on the VM. Two Domains run concurrently on each physical machine. One is an unprivileged VM configured with 512MB of memory and 40GB VBD. The other is Domain0, which consumes all the remaining memory. To reduce the context switches between VMs, the two VMs are pinned to different CPU cores. The unprivileged VM is migrated from one machine to the other to evaluate TPM and migrated back to evaluate IM. The third machine emulates the clients to access the services on the migrated VM. They are connected by a Gigabit LAN.

B. Workloads for Migration Evaluation

Our system focuses on local storage migration, so we choose some typical workloads with different I/O loads. They are a web server serving dynamic web application, which

generates a lot of writes in bursts, a video stream server performing continuous reads and only a few writes for logs to represent latency-sensitive streaming applications, and a diabolical server which is I/O-intensive, producing a large number of reads and writes all the time. These workloads are typical for evaluating the VM migration performance in the past research.

C. Experimental Results

In all the experiments, services on the migrated VM seem to keep running during the whole migration time from clients' perspective. Table I shows experimental results of our prototype of TPM. From the results, we can see that it achieves the goal of live migration with very short downtime. The migration can be completed in a limited period of time. The amount of migrated data is just a little larger than the size of the VBD (39070MB), which means that the block-bitmap based synchronization mechanism is efficient.

TABLE I
RESULTS FOR DIFFERENT WORKLOADS

	Dynamic web server	Low latency server	Diabolical server
Total migration time (s)	796	798	957
Downtime (ms)	60	62	110
Amount of migrated data (MB)	39097	39072	40934

1) Dynamic web server: We configure the VM as a SPECweb2005 [12] server that serves as a banking server. 100 connections are configured to produce workloads for the server. Figure 5 illustrates the throughput during the migration. We can see that during the migration time using our TPM, no noticeable drop can be observed in terms of throughput.

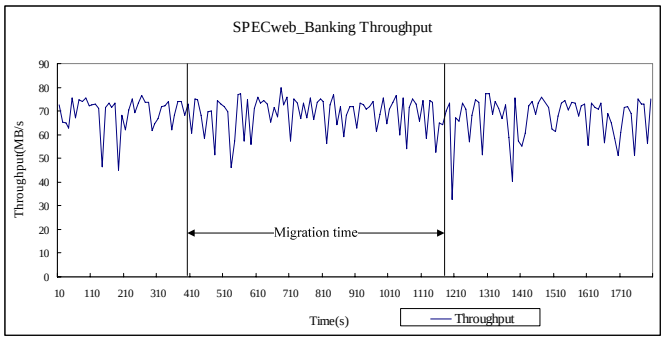

Fig. 5. Throughput of the SPECweb_Banking server while migration

In this experiment, three iterations are performed in the pre-copy phase. 6680 blocks have been retransferred. And 62 blocks are left dirty to be synchronized in the post-copy phase which lasts only 349 milliseconds. Only one block is pulled, the others are pushed by the source. The downtime is only 60ms.

2) Low latency server: We configure the VM as a Samba [13] server. It shares a 210MB video file (.rmvb) with a Windows client. The VM is migrated from the source to the destination, while the shared video is played on the client with a standard video player. During the whole migration time, the video is played fluently, without any observable intermission by the viewer. The write rate is very low in video server, so only two iterations are performed and only 610 blocks have been retransferred in the second iteration of the pre-copy phase which lasted for about 796 seconds. Five blocks are left unsynchronized which are pushed to the destination in the post-copy phase in 380 milliseconds. The downtime is only 62 milliseconds. The video stream is transferred at a rate less than 500kbps. The server works well even when the bandwidth used by the migration process is not limited at all.

3) Diabolical server: We migrate the VM while Bonnie++ [14] is running on it. Bonnie++ is a benchmark suite that performs a number of simple tests for hard disk drive and file system performance, including sequential output, sequential input, random seeks, sequential create, and random create [14].

Bonnie++ writes the disk at a very fast rate. Many blocks have been dirtied and must be resent during migration. During the pre-copy phase which lasts for 947 seconds, 4 iterations are performed and about 1464 MB dirtied blocks are retransferred. So the total migration time seems a little longer. But the block-bitmap is small. The downtime is still kept very short. The migration process reads the disk at a high rate. The Bonnie++ shows a low performance in terms of throughput during migration as illustrated by Figure 6.

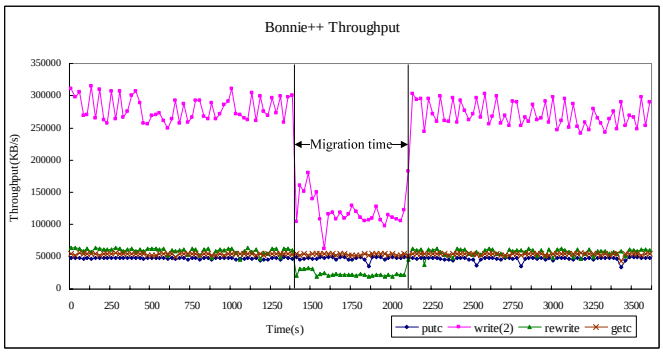

Fig. 6. Impact on Bonnie++ throughput

If we limit the migration transfer rate, the impact can be reduced about 50%. We just simply limit the network bandwidth used by the migration process in the pre-copy phase. Correspondingly, the disk bandwidth used by the migration will be decreased. The results show that the Bonnie++ works much better. But the migration time rose significantly. The pre-copy phase is about 37% longer than the unlimited one. It suggests that the disk I/O throughput is the bottleneck of the whole system performance.

4) Incremental migration: We perform migration from the destination back to the source after the primary migration using our IM algorithm. Table II show the results.

TABLE II
IM RESULTS COMPARED WITH TPM

	Dynamic web server		Low-latency server		Diabolical server	
	Migration time (s)	Amount of migrated data (MB)	Migration time (s)	Amount of migrated data (MB)	Migration time (s)	Amount of migrated data (MB)
Primary TPM	796.1	39097	798.0	39072	957	40934
IM	1.0	52.5	0.6	5.5	17	911.4

The amount of data that must be migrated using IM is much smaller than the primary TPM migration. So the total migration time is decreased substantially.

5) I/O performance overhead of synchronization mechanism based on block-bitmap: We configure Bonnie++ to run in the VM where all the writes are intercepted and marked in the block-bitmap. Table III shows the results compared with Bonnie++ running in the same VM without writes tracked.

TABLE III
I/O PERFORMANCE COMPARISON (KB/S)

	putc	write(2)	rewrite
Normal	47740	96122	26125
With writes tracked	47604	95569	25887

The results show that the performance overhead is less than 1 percent. So performance won't drop notably when all the writes are tracked and recorded in the block-bitmap preparing for IM after the VM has been migrated to the destination.

VII. CONCLUSION AND FUTURE WORK

This paper describes a Three-Phase Migration algorithm, which can migrate the whole-system state of a VM while achieving a negligible downtime and finite dependency on the source machine. It uses a block-bitmap based approach to synchronize the local disk storage data between the source and the destination. We also propose an Incremental Migration algorithm, which is able to migrate the migrated VM back to the source machine in a very short total migration time. The experiments show that both algorithms are efficient to satisfy those requirements described in section III for an effective live migration.

These two algorithms take the migrated VM as a black-box, all the data in VBD must be transmitted including unused blocks. If the Guest OS running on the migrated VM can take part in and tell the migration process which part is not used, the amount of migrated data can be reduced further. Another approach is to track all the writes since the Guest OS installation. Then all the dirty blocks are marked in the block-bitmap. Only these dirty blocks need to be transferred to a VM using the same OS image.

Our implementation of IM can only act between the primary destination and the source machine. The future work will focus on local disk storage version maintenance to facilitate IM to decrease the total migration time of a VM migrated among any recently used physical machines.

ACKNOWLEDGMENT

This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2007CB310900, National Science Foundation of China under Grant No. 90718028, MOE-Intel Information Technology Foundation under Grant No. MOE-INTEL-08-09, and HUAWEI Science and Technology Foundation under Grant No.YJCB2007002SS. Zhenlin Wang is also supported by NSF Career CCF0643664.

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] M. Nelson, B. Lim, and G. Hutchins, *Fast Transparent Migration for Virtual Machines*. 2005 USENIX Annual Technical Conference, 2005.
[3] Kozuch, M., and Satyanarayanan, M. *Internet Suspend/Resume*. Fourth IEEE Workshop on Mobile Computing Systems and Applications, 2002.
[4] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. *Optimizing the Migration of Virtual Computers*. OSDI, 2002.
[5] M. Kozuch, M. Satyanarayanan, T. Bressoud, C. Helfrich, S. Sinnamohideen. *Seamless Mobile Computing on Fixed Infrastructure*. Computer, July 2004.
[6] R. Bradford, E. Kotsovinos, A. Feldmann, H. Schioberg, *Live Wide-Area Migration of Virtual Machines with local persistent state*. VEE'07, June 2007.
[7] 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*. SOSP, 2003.
[8] J.G. Hansen and E. Jul, *Self-migration of Operating Systems*. Proc. Of the 11th ACM European SIGOPS Workshop, September 2004.
[9] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. D. Laat, J. Mambretti, I. Monga, B. Oudenaarde, S. Raghunath, Phil Y. Wang. *Seamless live migration of virtual machines over the MAN/WAN*. Future Generation Computer Systems, 2006.
[10] R Chandra, N Zeldovich, C Sapuntzakis, MS Lam. *The Collective: A Cache-Based System Management Architecture*. NSDI '05: 2nd Symposium on Networked Systems Design & Implementation, 2005.
[11] Xen, http://www.cl.cam.ac.uk/research/srg/netos/xen/.
[12] SPECweb2005, http://www.spec.org/web2005/.
[13] Samba, http://us1.samba.org/samba/.
[14] Bonnie++, http://www.coker.com.au/bonnie++/.