Memory/Disk Operation Aware Lightweight VM Live Migration Across Data-centers with Low Performance Impact

Bin Shi
School of Computer Science and Engineering,
Beihang University
Beijing, 100191, China
shibin@act.buaa.edu.cn

Haiying Shen
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
hs6ms@virginia.edu

***Abstract*—Live virtual machine migration technique allows migrating an entire OS with running applications from one physical host to another, while keeping all services available without interruption. It provides a flexible and powerful way to balance system load, save power and tolerant faults in data centers. Meanwhile, with the stringent requirements of latency, scalability, and availability, an increasing number of applications are deployed across distributed cloud data-centers. However, existing live migration approaches still suffer from long downtime and serious performance degradation in cross data-center scenes due to the mass of dirty retransmission, which limits the ability of cross data-center scheduling. In this paper, we propose a system named Memory/disk operation aware Lightweight VM Live Migration across data-centers with low performance impact (MLLM). It significantly improves the cross data-center migration performance by reducing the amount of dirty data in the migration process. In MLLM, we predict disk read workingset (i.e., more frequently read contents) and memory write workingset (i.e., more frequently write contents) based on the access sequence trace. And then we adjust the migration models and data transfer sequence based on the workingset information. We also present two optimizing methods to filter unused blocks and to de-duplicate data content by a hot data cache, thereby greatly decreasing the amount of data to be transferred. We implement MLLM on the QEMU/KVM platform and conduct several real-world experiments. The experimental results show that our method averagely reduces 67.0% of total migration time and 41.6% service downtime over existing methods.**

I. Introduction

In recent years, cloud computing has experienced vigorous growth due to its capacity to utilize the IT infrastructure efficiently while providing the high quality of service (QoS) to users. Virtual machine (VM) live migration [1], which is a key technique of cloud computing, enables the administrator to move a running VM from one physical host to another without imposing appreciable service downtime. This provides a graceful and powerful means to 1) enable dynamic load balancing when a physical host is overloaded by migrating VMs out of this host; 2) implement hardware maintenance, e.g. upgrade the hypervisor after migrating all the running VMs out; 3) enable load-concentration by migrating other VMs in when a physical host becomes underutilized; and 4) continue the service when a physical host is at the end of its lifecycle by migrating its VMs to other running physical hosts [2]–[4].

Live migration needs to transfer CPU state, memory, networking and storage content of the running VM, guaranteeing that the final destination copies of VM states are identical to the source ones. At present, the most widely employed live migration method is pre-copy [1], in which the target VM starts running after the entire VM is copied to the target host. Since the source VM keeps running while it is migrating, some states could be updated (i.e., get dirty), rendering inconsistency between the source states and destination states. Therefore, all the dirty states should be recorded and retransferred iteratively until the remaining dirty data is small enough. Then, the VM stops running and the remaining dirty pages are copied to the destination (i.e., stop-and-copy phase). Pre-copy has less guest performance penalty because all the data is completely copied before the target VM starts. However, it is considered to be an unreliable operation, and only 87% of those migrations are successful [5], [6]. A common failure is that the migration process takes too much time transferring the frequently-dirty data repeatedly using limited bandwidth, and consequently the migration cannot be completed within the required time. To handle this problem, traditional methods simply cancel the migration when a failure occurs or enter the stop-and-copy phase ahead of schedule. Alternatively, other methods [1], [7] employ remedies that slow down the VM states dirty rate by using process-stunning and write-throttling. However, all the solutions above are considered as degraded live-migration since they sacrifice QoS [6].

Pre-copy also introduces much extra traffic due to the large amount of dirty retransmission. This problem becomes more significant in cross data-center migration scenario. Compared with migration within a single data-center, cross data-centers migration 1) **requires disk migration** since it is less likely to use a global network-attached storage in cross data-center environments, and 2) **has limited bandwidth** since the network bandwidth assigned for migration is limited considering that the outlet bandwidth which contains massive data is quite expensive in data-centers. Actually, an increasing number of VM applications are deployed at a very large scale across multiple cloud data-centers as the requirements of both data volume and calculation ability have grown rapidly. For this reason, it is urgent to solve the problem and improve the performance of migration.

Some efforts have been made to solve the problem of the large amount of dirty retransmission. Post-copy [8] starts the destination VM first and it fetches data from the source VM

when it needs the data. However, since the VM needs to wait until it receives the data, such on-demand data fetching generates high latency for the VM running. To mitigate this problem, hybrid-copy [9], [10] was proposed that combines pre-copy and post-copy. It copies the states from the source host to the destination host along with a bitmap indicating dirty data, and then the destination VM starts running. When it requires data that is dirty, it fetches the data from the source VM, which avoids considerable retransmission of the dirty data that is not needed. However, the on-demand data fetching still leads to certain latency for the VM running.

The works in [11]–[13] try to reduce the amount of dirty retransmission by transferring more frequently updated contents (write workingsets) later, so that the probability for them to get dirty will decrease. Those works treat memory and disk content separately ([11] only focuses on disk migration while [12], [13] only focus on memory migration). However, they need to be jointly considered since disk accesses are usually associated with memory accesses in modern operating systems. In addition, the works in [12], [13] introduce much overhead to guest VM by tracing memory access online without hardware assistance. [12] also faces the problem that it uses a fixed size for a workingset pre-defined by users. However, the workingset size is determined by applications rather than users, so the fixed size will probably be underestimated or overestimated, thus limiting the performance improvement. [13] uses hidden Markov model (HMM) to predict the dirty probabilities for each memory page but consumes large resources to train an HMM online.

In this paper, we propose a system for Memory/disk operation aware Lightweight VM Live Migration across data-centers with low performance impact (MLLM). MLLM is novel in the following aspects:

- It considers the different adverse performance impacts from different memory operations (i.e., disk write, disk read, memory write, and memory read) of a VM in different VM migration models (i.e., pre-copy and post-copy) and adaptively chooses the models for different memory operations in order to reduce the adverse performance impact as much as possible. It jointly considers memory and disk for disk write operations, so that disk content can be directly fetched from the local memory rather than from the remote source VM.
- To reduce the overhead of finding frequently updated contents online, we first track the disk and memory access sequence by leveraging the new hardware feature called Extend Page Table Access/Dirty Bit [14] (EPT A/D bit). With this feature, MLLM eliminates the large time consumption introduced by frequently trapping into the hypervisor, thus reducing the guest VM overhead.
- MLLM can more accurately estimate the workingsets. Specifically, we predict disk read workingset based on access locality characteristics, while predict memory write workingset by a modified CLOCK algorithm [15].

A workingset is a collection of data which is more frequently accessed during a period of time. Based on the access sequence, we identify different workingsets: memory write workingset and disk read workingset. We identify these workingsets because the required inputs for identifying them are easier to get. We can know memory read workingset and disk write workingset by calculating the complementary set. We then adopt specific migration models for each of workingset. Moreover, we also present two optimizing methods to filter unused blocks and to de-duplicate data content by a hot data cache, thereby greatly decreasing the amount of data to be transferred.

We developed MLLM on the QEMU/KVM [16] platform and conducted real wide-area migration experiment with different workloads from 1) Beihang Xueyuan Road data-center to Beihang Shahe data-center; 2) Beihang Xueyuan Road data-center (in Beijing) to Shanghai Jiaotong University (in Shanghai). The results of the experiment show that MLLM offers negligible downtime and shorter total migration time, while incurring fewer guest penalties in comparison with existing solutions.

The remainder of the paper is organized as follows. We show our motivation in Section II. In Section III, we propose the basic design of our system. Section IV presents the enabling techniques and several optimizations of the system. Then we evaluate the effectiveness and performance in Section V. Section VI introduces the related work on the live migration. Finally, in Section VII, we conclude the work of MLLM.

II. Motivation

The memory operations of a VM's workload can be classified to disk write, disk read, memory write, and memory read. We summarize the performance impacts when conducting different operations in the pre-copy and post-copy models in Table I. Accessing memory is supposed to be very fast (on the order 100 nanoseconds) and it is always in a synchronous way, which means it will block other operations until success. While the disk accesses are usually in an asynchronous way and their time consumption is on the order of 10 milliseconds, which is the same as on-demand-fetching latency. Therefore, on-demand-fetching latency is unacceptable for memory access while it is bearable for disk access. For this reason, memory write and memory read operations will introduce heavy performance impact for post-copy.

TABLE I: Performance impact on different operations.

Models		post-copy	pre-copy
Performance Impact	Disk Write	**Little**	Yes
	Disk Read	Medium	**No**
	Memory Write	Heavy	**Yes**
	Memory Read	Heavy	**No**

On the contrary, disk read operation will only impose medium impact. We notice that disk write operations are special: Disk write operations on old blocks may just modify part of the data rather than totally replace it. So they may need to read old data from disk first (eligible to trigger on-demand-fetchings), edit them in the memory, and then write; While disk write operations on fresh blocks will just

replace the fresh blocks. For this reason, they will copy whole blocks of data from memory buffer cache to disk. Based on our experiment, 68% of the disk write operations are associated with fresh blocks. Therefore, disk write on post-copy will avoid many on-demand-fetchings thus they only introduce a little performance impact. For pre-copy, disk and memory read operations will not cause dirty retransmission, so they don't impose performance impact. In contrast, both disk and memory write operations cause dirty retransmission and impose performance impact.

The above analysis results give us guidance on improving the VM migration methods. That is, we can distinguish the different memory operations of a VM's workload, and adaptively use different VM migration models in order to reduce adverse impact on the VM performance. For disk-write operations, we can use post-copy, while for other memory operations, we can use pre-copy.

III. Basic Idea of the System Design

1) Using Different Migration Models for Different Workingsets: As analyzed in Section II, different kinds of memory operations have different adverse performance impacts in the two different VM migration models. Therefore, we aim to predict operations and use the better migration model accordingly. More specifically, we predict the locality of each memory/disk operation and finally divide all the VM data into four sets: memory write workingset, rest of memory (memory read workingset and cold memory pages), disk read workingset, and rest of disk (disk write workingset and cold disk blocks). After that, we use post-copy for the disk write and disk cold workingset while use pre-copy for other three workingsets. By this means, both the VM performance impact and total network traffic (and hence migration time) will be reduced.

2) Workingset-aware Pre-copy Schedule: Recall that the disk read workingset and all the memory will be migrated by pre-copy model. We also change the migration sequence in the pre-copy phase for performance improvement. Rather than transferring them in a default sequence, we prefer to transfer the memory write workingset at last. And we also copy the disk reading workingset prior to pre-copying all the memory pages, because that memory pages usually have more chances to get dirty than disk blocks. In this way, the total amount of dirty data and total migration time will decrease.

Based on the basic idea discussed above, we show Figure 1 to illustrate the overall work-flow of MLLM. In the first four steps, the VM runs on the source physical host, we refer to these fours steps as the pre-copy phase. And after the short suspension in Step 5, the VM runs on the destination host. So we call step 6 as the post-copy phase in the rest of our paper.

- **Step 1: Workload profiling.** Once the migration request is issued, the online workload profiling steps begin to trace memory/disk access sequence data and predict workingsets based on the data. This is a preparing stage to improve migration efficiency. More frequently updated contents will be migrated later to avoid dirty retransmission. We start to trace disk access sequence data and predict disk read workingset

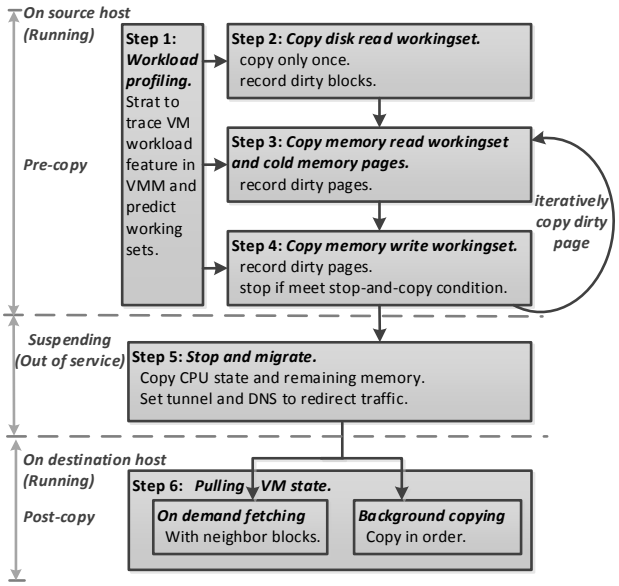

Fig. 1: The migration workflow.

immediately because these data contents will be transferred first (Step 2). When Step 2 is nearly completed, we start to trace memory write access sequence data and predict memory write workingset. By this way, we can get memory workingset information before copying memory (Step 3). It is also worth noting that the workingsets may change since VM is running, so we keep profiling and updating the predicting results until stop and migrate step (Step 5) begins.

- **Step 2: Copy disk read workingset.** This step, also the real beginning of the migration, begins after Step 1 when we get enough data to predict the disk read workingset. In this step, we migrate the disk read workingset. Since the disk read workingset may overlap with disk write workingset, they may get dirty as well. Therefore, we record dirty blocks in a bitmap but do not retransfer the blocks in this step, and then we will transfer them in post-copy phase (Step 6).
- **Step 3: Copy memory read workingset and cold memory pages.** In this step, we copy the memory read workingset and cold memory pages. We also record the pages which get dirty during the migration.
- **Step 4: Copy memory write workingset.** After copying the memory read workingset and memory cold pages, we copy the memory write workingset and record dirty pages in this step. When the copy is finished, we return to Step 3 to copy the dirty memory pages and then go to Step 4 to copy the write workingset dirty memory pages. This process repeats iteratively until any of the following conditions are satisfied, and then we jump to Step 5: i) The remaining data needs to be transferred in the pre-copy model is less than a threshold, e.g., it can be transferred within 30ms [16]; ii) The reserved migration time has been used up; iii) The number of pages sent in the current round is greater than the last two rounds, which means that the dirty rate is bigger than the transfer rate so that the condition 1 will not be satisfied.
- **Step 5: Stop and migrate.** We suspend the VM and transfer the CPU state and remaining memory pages in transience.

Meanwhile, we set a virtual private network (VPN) tunnel and a domain name server (DNS) to redirect traffic and new connections to the destination host. To ensure the data consistency between the source and destination, we also transfer the bitmap of untransferred and dirty disk blocks to the destination host.

• **Step 6: Pulling VM state.** In this step, we resume the VM on the destination side. Since some data is dirty and some data is still not transferred, the remaining state of VM is required to be copied, which includes the dirty blocks in disk read workingset and the rest of the disk. The subsequent procedure is processed by two concurrent threads explained below.

`Background copying`: Post-copy [8] method copies the remaining states stealthily in the background. As analyzed in §II, disk write operations may not need on-demand-fetching and disk read operations incur most of on-demand-fetching. Therefore, in our design, to avoid the large amount of on-demand-fetching, we propose to first copy the dirty blocks in disk read workingset, then the rest of the disk blocks.

`On-Demand-Fetching`: It is possible that some data has not been transferred in the background copying when is needed in the destination side. In this case, on-demand-fetching is conducted. To let the destination side know that some blocks have not been transferred, before resuming the VM on the destination host, we set the remaining blocks unreadable in the hypervisor. If they are being read, we call for the up-to-date content from the source host and fill the corresponding blocks. The whole migration process succeeds when Step 6 is finished. At that point, the destination host notifies the source host to destroy the source VM.

IV. Enabling Techniques

For the above designs, we must address two challenges:

• To predict these two workingsets, we need profiling (i.e. tracking history access sequence of both memory and disk). However, existing memory access trace method is time-consuming. So a challenge is to trace memory access sequence without incurring much overhead (presented in §IV-A).

• The second challenge of our solution is to accurately estimate the workingsets. We predict disk read workingset by access locality characteristics (presented in §IV-B), while predict memory write workingset by a modified CLOCK algorithm [15] (presented in §IV-C).

A. Efficient Access Sequence Tracing

As indicated, to predict the workingsets, MLLM needs to trace both memory write and disk read access sequence.

1) Disk: The hypervisor emulates all disk I/O, disk reads and writes initiated by guests are explicitly visible to the hypervisor. So no further action is required, and we can use an existing method that directly logs read and write access sequence in the emulator.

2) Memory: The traditional way [1], [6] to trace memory write is setting write protection for all the memory pages, so all of the write operations upon these pages will trigger VM-exit, which switches CPU context from the VM to the hypervisor, and then we can log the memory writes in the hypervisor. However, it introduces high performance loss due to a large number of VM-exits. To provide efficient access tracing, MLLM leverages the new Intel hardware virtualization extension features, EPT A/D Flag and Page-Modification Logging [14]. Specifically, MLLM enables accessed and dirty flags for EPT using bit 6 of the extended-page-table pointer (EPTP), and enables page-modification logging by setting bit 17 of the VM-execution control field. Then, whenever there is a write to a guest-physical address, the processor sets the dirty flag to the corresponding EPT entry. Meanwhile, the processor will also record the guest-physical address of this access to a 4-KByte region of memory (the address can be configured at VM-execution control field). When up to 512 addresses have been recorded, a VM-exit will be triggered by log full event. In the VM-exit handler, we will get the memory write sequence record from the 4-KByte memory region, and then we clear all the dirty flags as well as the 4-KByte memory region. In this way, 512 write accesses triggers only one VM-exit, which significantly improves performance compared with the traditional method.

B. Disk Read Workingset Estimation

The disk read workingset should be predicted based on both the temporal and spatial locality. The existing method [11] estimates disk read workingset only at the block-level; that is, the blocks near the accessed block are included in the workingset. However, these blocks may not belong to the same file of the accessed block. Therefore, failing to take file system construction into consideration may adversely affect the accuracy of identifying the workingset. In MLLM, we consider that: i) If a block is read, then it is likely to be read again in the recent future; ii) If a block of a file is read recently, then the subsequent blocks in this file would possibly be read as well; iii) If a small file is read recently, then other small files in the same file directory would probably be read as well. For example, in scenarios such as project compilation, when one .c file is accessed, the other .c files in the same directory will be very likely to be accessed too. Therefore, when a disk read operation occurs, we add the all the blocks associated with target blocks in these three categories into the workingset. Once the workingset is full, we leverage the reuse distance algorithm [17] to evict blocks. The reuse distance here equals the number of other distinct blocks accessed between its last access and its current access. If the block has been accessed only once, then the reuse distance is infinite. In MLLM, a block with a large reuse distance is evicted first even if it is added to the workingset recently.

In order to make sure that the blocks in a workingset are updated, we use a queue to maintain the disk read workingset, as is shown in Figure 2. The first half part of the queue is considered as eviction zone (blocks here are eligible to be deleted from the workingset); while the other half of the queue is the safe zone. The reason why we set a safe zone and eviction zone is to ensure that a new coming block has enough time to gain its real reuse distance. For a block K accessed

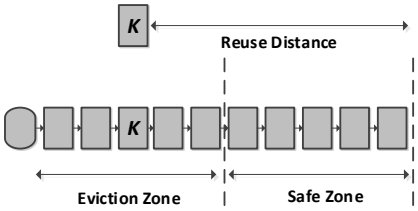

Fig. 2: The reuse distance algorithm.

or associated with a block that is accessed, if K is already in the queue, we move K to the end of the queue and update the reuse distance of this block, which is the previous position of K in the queue to the end of the queue. Otherwise, we need to add K to the tail of the queue and select a block to delete from the queue if the queue is full. Specifically, we select the blocks with the largest reuse distance in the eviction zone. Note that some of the blocks that have not been accessed for a long time may escape from being selected if their previous reuse distances are small. Therefore, the algorithm updates the reuse distance by adding 1 of all blocks that are in front of the selected block when it is deleted. This is because the reuse distance of the blocks after this selected block is incremented by 1 after the block is deleted.

The maxsize of the workingset can be configured by users. If the VM needs to run on the destination host as early as possible, the maxsize should be set relatively small. Otherwise, we should increase the size. In our implementation, we set the maxsize as one tenth of the total disk size. And all the files smaller than 1MB are considered as small files. For many applications with low disk usage, the disk read workingset will generally not reach the maxsize. If so, more disk blocks which do not in the read workingset will be copied in post-copy phase, and it will not affect much of the migration performance.

C. Memory Write Workingset Estimation

Different applications in the VM usually have different memory usage patterns. Therefore, for memory write workingset prediction, we should not arbitrarily set a workingset maxsize as explained in §I, because if the setting is larger or smaller than the real situation, it will affect the improvement performance. Therefore, the reuse distance algorithm (with fixed workingset size) is not suitable in this case. Instead, we slightly modify the CLOCK algorithm [15] (which is simple but effective) for the purpose of predicting workingsets. It can also automatically adapt to the actual workingset size. This is the first work that uses the CLOCK algorithm for this purpose though previous works use it for predicting cache visits.

As shown in Figure 3, in the CLOCK algorithm, all the physical memory pages are regarded as a circular buffer; we constantly loop over and scan the pages, like the hand of a clock. Any page that has been written during the scan loop is marked as being part of the current workingset. More specifically, based on the access frequency, we label each page by hot page, warm page, and cold page. Hot pages and warm pages are included in the workingset while cold pages are not. The hand moves through the pages and checks each page's

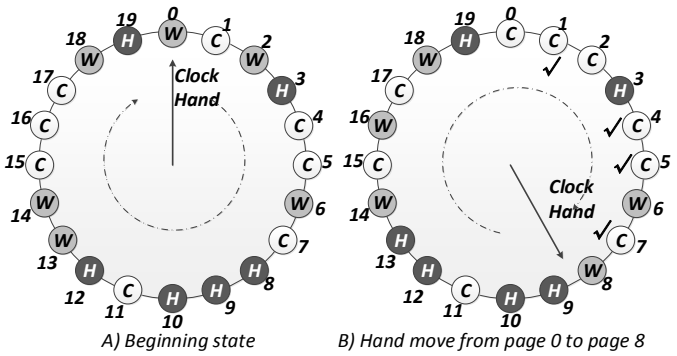

Fig. 3: An example of how the CLOCK algorithm works.

label. If a page is marked as cold, the page is eligible to be copied in this iteration. Otherwise, the algorithm updates the page's status (i.e. changes it to a warm page if it was hot, and changes it to a cold page if it was warm). After that, the hand moves to the next page. Parallel to the scan loop, the algorithm also updates the page's status based on memory write accesses. When a cold page is accessed, it will be promoted to a warm page, and when a warm page is accessed, it will be promoted to a hot page. To make cold/hot statuses accurately reflect their current access behavior, the speed at which the scanner should run depends on the time it takes to migrate the memory. In particular, we move the scan hand at the same speed as the migration pointer (pointing to the page to be migrated).

Figure 3 shows an example of our CLOCK algorithm. The left circle is the initial state of the memory migration. We derived the cold/hot statuses of the pages using initial profiling before the memory migration began. The right circle shows the movement of the clock hand from page 0 to page 8. As we can see, pages 0 and 2, which were warm pages before, became cold pages after the clock hand scanned them; pages 1, 4, 5, and 7 were originally cold pages so they were transferred to the destination; and page 8, which was a hot page, became a warm page. During the time the clock hand moved from page 0 to page 8, pages 3, 6, 13, and 16 were written. For this reason, page 16 became a warm page, page 13 became a hot page, and the statuses of pages 3 and 6 kept the same after being scanned.

D. Filtering Unused Blocks in Disk

There are two common states for disk space, used state and available state (not used state). During migration, only the used disk blocks are necessary to be copied. But existing strategies [7], [11], [18] tend to migrate all the blocks regardless of their states, which introduces longer transmission latency and higher network traffic.

In our system, we propose a VM introspection (VMI) based approach to recognize the unused blocks. Our approach leverages VMI to scan the file system meta-data which records which blocks are currently available. With this information, during *Step 6: Pulling VM State*, we can ignore the blocks that are not used when transferring disk blocks, thereby avoiding a large amount of unnecessary transfers. Specifically, to get this information, we need to scan all the meta-data in the virtual disk's file system, including both in-memory meta-data and

in-disk meta-data. We present the phases of our approach in detail below.

• **Phase 1: Read from Disk.** Firstly, before *Step 5: Stop and migrate*, we read the available block following the file system structure. More specifically, we first read out the disk's superblock (the file system structure) that has the information of the position of each block group (another file system structure). From the description of a block group, we search for the bitmap which records whether each block in the group is used or available. Once acquiring the bitmap blocks for all block groups, we can build a complete bitmap for the entire disk. We conduct this phase before *Step 5: Stop and migrate* (which cannot be long) since scanning a large number of disk blocks would take a relatively long time.

• **Phase 2: Read from Memory.** The meta-data that has been updated in the memory but has not yet written back to disk might break data consistency. To mitigate this problem, in *Step 5: Stop and migrate*, we utilize the VMI technique to check whether there are available meta-data cached in memory. Then, we update the result based on the in-memory meta-data.

E. Redundant Data Elimination

More than 79% primary VM disk data overlap among the same OS distributions just with different upper applications [19]. Leveraging this phenomenon, we adopt the optimization method in [20] to reduce the amount of transferred data. It builds a sufficient cache on each side of the migration, recording the hot blocks/pages that appear frequently. The two caches are synchronized. The source checks the fingerprint of a block/page first. If it is identical to the blocks/pages in the destination's cache, the source only transfers the index instead of the block/page content, which can reduce network traffic usage. From another perspective, this strategy trades extra storage space for both the reduction of network flow and the speedup of transmission, since the outlet bandwidth is much more valuable than data center storage resources.

V. Performance Evaluation

The performance of MLLM is evaluated mainly on four aspects: 1) The hit rate of predicted workingset; 2) The migration performance (e.g. downtime, total migration time); 3) Network condition adaptability; and 4) The impact on guest applications. We leverage our self-implemented **XvMotion** [7] and the combination of two different workload-aware migration methods **Zheng+HMM** (i.e., migrate disk using zheng's optimization [11] while migrate memory using Sun's optimization [13]) as the baseline. Xvmotion is the state-of-the-art approach for cross data-center migration, and it uses IOMirroring method to copy the dirty disk blocks and is adopted in VMware's real scenarios. [11] is a representative approach that uses workload patterns to optimize the disk migration, while [13] optimizes memory migration sequence by calculating memory dirty probability based on HMM.

A. Experiment Setup

The majority of the VM migrations in our experiments are from Beihang Xueyuan Road Cloud Data-center to Beihang Shahe Cloud Data-center. The bandwidth reserved for migration between two data-centers is 50MBps. The hardware platform of Beihang Xueyuan Road Cloud Data-center are Inspur NF5280M4 Blade servers, which are configured with Intel Core Intel Xeon E5-2620-v3, 6 cores, 12 threads processors, 16GB DDR memory, a 300GB disk with 7200 RPM, and Intel I219-LM Gigabit NIC card. We use Dell PowerEdge R630 in Beihang Shahe Cloud Data-center. The physical hosts here are all configured with Xeon E5-2609-v3, 6 cores, 6 threads processors, 8GB DDR memory, 1800GB 7200RPM disk, and Gigabit NIC card. To evaluate the network condition adaptability, in Section V-D, we conducted VM migration experiments from Beihang Xueyuan Road Cloud Data-center to Shanghai Jiaotong University. The host physical machine Shanghai is with a 16GB memory of RAM and Intel Core i7-6700 processors. All the operating system on physical servers are Ubuntu 14.04 with 3.13.0 64bit kernel. The VMs are configured with 2 vcpus, 2GB RAM, and 40GB Disk unless specified otherwise. We evaluate MLLM under five real-world workloads:

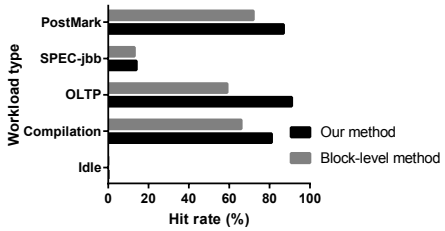

Fig. 4: Hit rates of disk read workingset prediction algorithm.

• **Idle:** The idle workload means that the VM does nothing except the tasks of OS itself after boot-up.

• **Kernel Compilation:** This is a development workload involving memory and disk I/O operations. We compile the Linux 3.13.0 kernel with default compilation configuration.

• **SysBench-OLTP:** The OLTP benchmark [21] is used to measure the database server performance (throughput), and we simulate an OLTP workload with random access.

• **SPEC-jbb:** The Java Business Benchmark of SPEC is to evaluate Java server performance [22] with various tasks.

• **Postmark:** This benchmark is designed to simulate the behavior of file servers [23]. We perform it by simulating 30% write operations and 70% read operations.

B. Workingset Prediction Accuracy

This subsection evaluates the accuracy of our workingset prediction algorithms that are discussed at Section IV. When a VM accesses a disk block or a memory page that is in the workingset, this operation is regarded as a hit; otherwise, a miss. Figure 4 shows the comparison of our disk read workingset prediction algorithm and the block-level algorithm which is adopted in [11]. The Idle and SPEC-jbb has very few read operations, so that there is no measurable disk read workingset. In the other workloads, our disk workingset prediction algorithm reaches 81% to 91% hit rate, while the block-level algorithm has 59% to 72% hit rate.

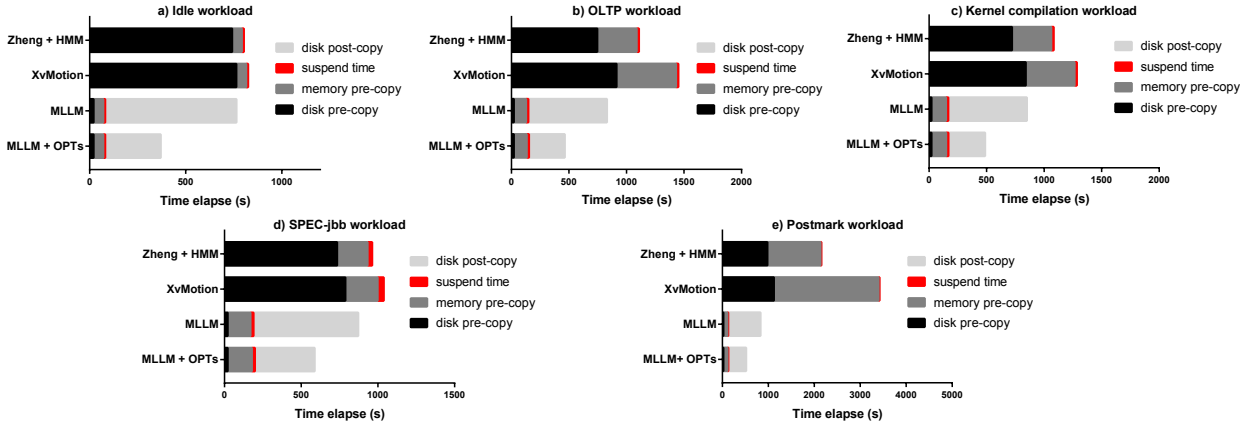

Fig. 5: The experiment result on total migration time.

Table II illustrates the memory write workingset size and the hit rate in the first round pre-copy iteration. As illustrated, the hit rates vary from 71.3% to 100%. SPEC-jbb has the largest write workingset among different workloads (i.e. 167342 pages or 654MB). Also, we can see that the workingset size is quite different in each workload, so the method that configures a fixed workingset size beforehand will not be able to adapt to dynamic workloads.

TABLE II: Measured results in the first round iteration.

Workload	Workingset size (pages)	Hit Rate
Idle	193	100%
Kernel compilation	49712	87.2%
OLTP	64487	81.6%
SPEC-jbb	167342	92.7%
Postmark	17932	71.3%

C. Migration Performance

In this subsection, we measure the migration performance of our system. We carry out the evaluation for the following two modes: 1) **MLLM:** This is our live migration method that only considers the workingset without any further optimization; 2) **MLLM+OPT:** This is our live migration method with unused blocks filtering and redundant data elimination.

We first evaluate MLLM's performance on the migration time (from transferring the first byte to the last byte of VM data). The experimental results are shown in Figure 5. The migration time varies significantly among different workloads, which is mainly due to the varying dirty pattern of blocks and pages. Disk pre-copy time (Step 2 in Figure 1), memory pre-copy time (Step 3 and 4 in Figure 1), service downtime (Step 5 in Figure 1), and disk post-copy time (Step 6 in Figure 1) are all identified in the figures. After the service downtime, the VM will be running in the destination server. For this reason, we can see MLLM can quickly let VM run at the destination server. We can also see that on average, MLLM saves 71.4% and 62.5% total migration time than XvMotion and Zheng+HMM. In addition, we find that XvMotion is sensitive to the workload types. On the contrary, the total migration time of MLLM keeps steady in different workloads. For example, in Postmark workload (Figure 5.e), XvMotion and Zheng+HMM suffer from a long memory migration time because they must retransfer disk dirty blocks and memory dirty pages. While MLLM transfers the disk dirty workingset by post-copy, so it spends much less time. It is also worth noting that in the SPEC-jbb workload (Figure 5.d), the memory dirty rate is higher than the bandwidth, so all the migration approaches enter the stop-and-migrate phase before the remaining state gets small enough (so the service down is larger, and we can see more details in Figure 6). In addition, using the optimization of unused blocks filtering and hot data cache remarkably decreases the total migration time since they reduce the amount of VM state to be transferred.

We demonstrate the experimental result of migration downtime in Figure 6. As we see, for write-intensive workloads (i.e. OLTP, kernel compilation, and SPEC-jbb), XvMotion performs poor on downtime, requiring around 1 second, while Zheng+HMM relatively performs better because it optimizes the data transferring sequence. MLLM outperforms those two methods since it does not need to send the disk dirty blocks in pre-copy. On average, MLLM spends 34.5% and 47.4% less downtime than Zheng+HMM methods and XvMotion correspondingly (calculated by $(x - MLLM)/x$). Its worth noting that since the unused blocks filtering optimization implements VMI during *Step 5: Stop and migrate* stage, it takes a bit longer suspending time than the plain MLLM without the filtering technique.

Figure 7 shows the amount of network traffic created during the migration process. Since the network bandwidth in the experiment environment is reserved and relatively stable, the network traffic is proportional to the VM's migration time. So, the network traffic experiment results are similar as those in Figure 5. It indicates that the MLLM (without optimization) saves 33.8% and 49.6% network traffic than Zheng+HMM methods and XvMotion. Moreover, when additional optimization is used, 24.3% more traffic is saved on average.

D. Network Condition Adaptability

In this subsection, we evaluate MLLM's ability to adapt to poor network conditions. We migrated one VM from

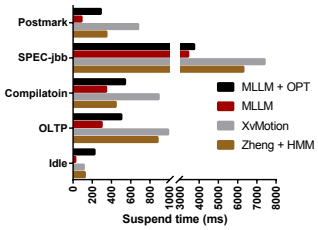

Fig. 6: Experiment results on migration downtime.

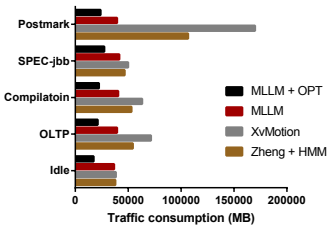

Fig. 7: Experiment results on traffic amount.

Beihang University (Beijing) to Shanghai Jiaotong University (Shanghai) There is no special-purpose network line between so the bandwidth is limited. The bandwidth can only be up to 5MBps, which is an extremely poor condition for VM live migration. We migrate the VM with different workloads and regard it a successful live migration if the migration can be finished within 2 hours and the downtime is less than 1 second. Table III shows that MLLM succeeds to migrate in the Idle and Postmark workloads while others only succeed in Idle workload. Therefore, MLLM can adapt to the more harsh environment than other VM migration methods.

TABLE III: Ability to adapt to poor network condition.

Approaches	MLLM + OPT	XvMotion	Zheng + HMM
Idle	√	√	√
OLTP	X	X	X
Kernel compilation	X	X	X
SPEC-jbb	X	X	X
Postmark	√	X	X

E. Impact on Guest Applications

We also evaluate MLLM's performance by measuring the impact on guest VM applications during the migration process. We ran several benchmarks and get the performance metrics in different scenes: 1) when there is no migration (baseline), 2) in MLLM's pre-copy phase (both memory and disk access trace are on), 3) in MLLM's post-copy phase, 4) in XvMotion's memory pre-copy phase, 5) in Zheng+HMM's memory pre-copy phase, and 6) in original post-copy method.

The results are demonstrated in Table IV. The average impact at row 5 is calculated by averaging the impact of each benchmark (calculated by Avg[(baseline.benchmark(i)-X.benchmark(i))/ baseline.benchmark(i)]). First, we can see that Zheng+HMM introduce most overhead (22.35%) to guest VM. This is because it needs to train the workingset prediction algorithm in real-time and it imposes a large number of time-consuming VM-exits events. MLLM's workingset prediction algorithm is simple and it can trace the memory write operations without introducing lots of VM-exits. For this reason, MLLM, which introduces only 3.77% overhead, outperforms others in the pre-copy phase. Moreover, we can see that in the post-copy phase, MLLM only imposes 4.57% overhead, which is much smaller than the original post-copy (77.38% overhead). The reason is that MLLM introduces only a few on-demand-fetchings since the reading workingset is migrated in the pre-copy phase. For some workloads with few disk read operations such as SEPC-jbb, MLLM imposes almost no overhead in the post-copy phase.

TABLE IV: Comparison of performance impact on guest VM.

Workload	OLTP	SPEC-jbb	Postmark	Avg impact
No migration	4487tps	8764bops	312tps	0
MLLM-Pre	4312tps	8339bops	304tps	3.77%
MLLM-Post	4227tps	8745bops	288tps	4.57%
XvMotion-Pre	4288tps	8104bops	293tps	6.02%
Zheng+HMM-Pre	3356tps	6361bops	267tps	22.35%
Post-copy	1126tps	461bops	117tps	77.38%

VI. Related Work

VM live migration approach was first proposed by Clark et al. [1]. They proposed pre-copy to transfer the VM states, and boot the VM when all the VM states are copied to the target host. In contrast to pre-copy, Hines et al. [8] proposed post-copy, which first boots the VM on the target host and then copies the pages on demand from the source host, thus the memory pages will be transferred only once. Liu et al. [24]adopted the idea of ReVirt [25], which achieves live migration by recording the execution of VM and replaying them at the destination host. The hybrid-copy [9], [10] approaches combine the pre-copy and post-copy methods, which change to post-copy after a certain round of iterative pre-copy, so that it transfers memory pages only for limited times.

These earlier academic research works mainly focus on migrating a VM between two closely related hosts within a cluster. To accommodate the increasing requirement of large scale, some attempts have been proposed to enable cross data-center migration. Bradford et al. [18] use the pre-copy method [1] to migrate all the VM states including disk storage. Meanwhile, it combines dynamic DNS with tunneling techniques to guarantee that the existing network connections can continue transparently while the new ones are redirected to the new location. DBRP [26] and Takahashi et al. [27] combine the storage replication technology with traditional migration technology. They ensure that the disk replica is up-to-date in the destination host by using file synchronizing techniques. However, these solutions are fragile and always suffer from configuration complexity since the synchronizing is done outside the virtualization stack. VMware developed IOMirroring [7], [28], which mirrors all the new disk write operations from the source to the destination and synchronize other states in the background at the same time. It also introduces IO-throttling in order to reduce the dirty rate and accelerate the migrating process. Whereas, it consumes lots of network traffics and only performs well when a very large amount of migration bandwidth is reserved.

Many works also focus on reducing the cost (migration time and migration downtime) of migration. Ibrahim et al. [29] optimized the pre-copy by terminating migration when improvements of the downtime are unlikely to occur. VMScatter [30] and Jin et al. [31] compress VM states by eliminating the same content to reduce the amount of transmitted data and network bandwidth consumption. Clark et al. [1] introduced the concept of a writable workingset and pointed out that the migration sequence may impact on the migration performance. Based on this, Zheng [11] takes VM's storage I/O workload

into consideration to achieve higher efficiency by sending the frequently dirty blocks later. Zaw et al. [12] and Sun et al. [13] use Least Recent Used algorithms (LRU) and HMM to predict memory write workingset. However, their methods of getting historical trace data and predicting workingset introduce much overhead to guest VM. In contrast, our method can solve the aforementioned problems. Nathan et al. [32] reveal that existing models to predict migration time are fundamentally flawed and presents a new model that takes workingset size into account. Zhang et al. [33] analyze how much bandwidth is required to guarantee the migration time and the downtime. MigVisor [6] can accurately predict the completion time of VM migration using workingset model, which enhances the system management efficacy.

VII. Conclusion

In this paper, we present MLLM, a workingset-aware live migration method for cross data-center resource management, which improves the cross data-center migration by reducing the guest impact, migration downtime, total migration time, and saving the network traffic. In MLLM, we leverage the Intel hardware feature EPT A/D bit to efficiently acquire the VM disk and memory access sequence during migration. Based on these sequence data, we proposed two methods to predict disk read workingset and memory write workingset with high accuracy. And then we adjust the data transfer sequence based on the workingset information. We also present two optimizing methods to filter unused blocks and to de-duplicate data content by the hot data cache, thereby greatly decreasing the amount of data to be transferred and improving the migration performance. Our real experimental results show that MLLM reduces 62.5%-71.4% migration time, saves 62.4%-71.5% traffic amount, and reduces 34.5%-47.4% service downtime over existing methods. In the future, we aim to improve the workingset prediction accuracy by using machine learning.

Acknowledgement

We thank Ms. Christian Howard for improving the writing. This work was supported by the Chinese National Key Research and Development Program (2016YFB1000103), Beihang Ph.D. student oversea visiting fund. This research was also supported in part by U.S. NSF grants NSF-1827674, CCF-1822965, OAC-1724845, ACI-1719397 and CNS-1733596, and Microsoft Research Faculty Fellowship 8300751.

References

[1] Christopher Clark, Keir Fraser, Steven Hand, et al. Live migration of virtual machines. In *Proceedings of NSDI 2005*, pages 273–286.

[2] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, et al. VM live migration at scale. In *Proceedings of VEE 2018*, pages 45–56.

[3] Nilton Bila, Eric J Wright, Eyal De Lara, Kaustubh Joshi, et al. Energy-oriented partial desktop virtual machine migration. *ACM Transactions on Computer Systems (TOCS)*, 33(1):2, 2015.

[4] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In *Proceedings of INFOCOM 2011*. IEEE.

[5] Tomáš Kukrál, Miloš Kozák, Tomáš Hégr, and Leoš Boháč. VM migration measurement and failure detection. In *Proceedings of TSP 2015*, pages 285–288.

[6] Jinshi Zhang, Eddie Dong, Jian Li, and Haibing Guan. Migvisor: Accurate prediction of VM live migration behavior using a working-set pattern model. In *Proceedings of VEE 2017*, pages 30–43.

[7] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, et al. Xvmotion: Unified virtual machine migration over long distance. In *Proceedings of ATC 2014*, pages 97–108, June 2014.

[8] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of VEE 2009*, pages 51–60.

[9] Feiran Yin, Weidong Liu, and Jiaxing Song. Live virtual machine migration with optimized three-stage memory copy. In *Proceedings of the 9th International Conference on Future Information Technology*.

[10] S. Sahni and V. Varma. A hybrid approach to live migration of virtual machines. In *Proceedings of the 2012 IEEE International Conference on Cloud Computing in Emerging Markets, CCEM 2012*, pages 1–5.

[11] Jie Zheng, Tze Sing Eugene Ng, and Kunwadee Sripanidkulchai. Workload-aware live storage migration for clouds. In *Proceedings of VEE 2017*, pages 133–144.

[12] Ei Phyu Zaw and Ni Lar Thein. Improved live VM migration using lru and splay tree algorithm. *International Journal of Computer Science and Telecommunications*, 3(3):1–7, 2012.

[13] Sun Mingsong and Ren Wenwen. Improvement on dynamic migration technology of virtual machine based on xen. In *Proceedings of the 8th IEEE International Forum on Strategic Technology, IFOST 2013*.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3*. 2016.

[15] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 323–336, 2005.

[16] Qemu-kvm. /urlhttp://www.qemu-project.org.

[17] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.

[18] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of VEE 2007*, pages 169–179.

[19] Anthony Liguori and Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the 2008 Workshop on I/O Virtualization (WIOV 2018)*.

[20] Lei Yu, Haiying Shen, et al. Core: Cooperative end-to-end traffic redundancy elimination for reducing cloud bandwidth cost. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):446–461, 2017.

[21] Sysbench. https://github.com/akopytov/sysbench.

[22] Specjbb. https://www.spec.org/jbb2005/.

[23] Postmark. http://www.filesystems.org/docs/auto-pilot/Postmark.html.

[24] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of HPDC 2009*, pages 101–110.

[25] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of 5th Symposium on Operating System Design and Implementation OSDI 2002*.

[26] Chapter 13: Using Xen with DRBD. http://www.drbd.org/users-guide/ch-xen.html.

[27] Kazushi Takahashi, Koichi Sasada, and Takahiro Hirofuchi. A fast virtual machine storage migration technique using data deduplication. In *Proceedings of Cloud Computing, 2012*, pages 57–64. Citeseer.

[28] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, Min Cai, et al. The design and evolution of live storage migration in VMware ESX. In *Proceedings of the 2011 USENIX Annual Technical Conference, ATC 2011*.

[29] Khaled Z. Ibrahim et al. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of SC 2011*, pages 40:1–40:11.

[30] Lei Cui, Jianxin Li, Bo Li, Jinpeng Huai, Chunming Hu, Tianyu Wo, Hussain Al-Aqrabi, and Lu Liu. VMScatter: Migrate virtual machines to many hosts. In *Proceedings of VEE 2013*, pages 63–72.

[31] Hai Jin, Li Deng, Song Wu, et al. Live virtual machine migration with adaptive, memory compression. In *Proceedings of the 2009 International Conference on Cluster Computing*, pages 1–10.

[32] Senthil Nathan, Umesh Bellur, and Purushottam Kulkarni. Towards a comprehensive performance model of virtual machine live migration. In *Proceedings of the SoCC 2015*, pages 288–301.

[33] Jiao Zhang, Fengyuan Ren, and Chuang Lin. Delay guaranteed live migration of virtual machines. In *Proceedings of INFOCOM 2014*, pages 574–582. IEEE.