RLC - A Reliable approach to Fast and Efficient Live Migration of Virtual Machines in the Clouds

Sanidhya Kashyap
sanidhya.kashyap@research.iiit.ac.in

Jaspal Singh Dhillon
jaspal.dhillon@research.iiit.ac.in

Suresh Purini
suresh.purini@iiit.ac.in

***Abstract*—Today, IaaS cloud providers are dynamically minimizing the cost of datacenters operations, while maintaining the Service Level Agreement (SLA). Currently, this is achieved by the live migration capability, which is an advanced state-of-the-art technology of Virtualization. However, existing migration techniques suffer from high network bandwidth utilization, large network data transfer, large migration time as well as the destination's VM failure during migration. In this paper, we propose Reliable Lazy Copy (RLC) - a fast, efficient and a reliable migration technique. RLC provides a reasonable solution for high-efficiency and less disruptive migration scheme by utilizing the three phases of the process migration. For effective network bandwidth utilization and reducing the total migration time, we introduce a learning phase to estimate the writable working set (WWS) prior to the migration, resulting in an almost single time transfer of the pages. Our approach decreases the total data transfer by 1.16 x - 12.21x and the total migration time by a factor of 1.42x - 9.84x against the existing approaches, thus providing a fast and an efficient, reliable VM migration of the VMs in the cloud.**

***Keywords*-Virtualization, Live migration, Reliability, Dynamic Resource Management**

I. Introduction

Virtualization has become one of the key technologies in the era of Cloud Computing. Today, data centers are continuously employing the virtualized architecture to run applications inside virtual machines (VMs) that are mapped on to physical machines. This is accomplished by using techniques such as full virtualization, hardware assisted virtualization [1],[2]. Virtualization technology enables efficient utilization of hardware resources by server consolidation [3],[4] and demand based dynamic allocation of resources. The ability to run multiple VMs on a pool of networked physical machines and migrate them from one physical machine to another forms the basis for building *dynamic* Infrastructure-as-a-Service (IaaS) clouds. Live migration of VMs in such IaaS clouds provides significant opportunities for efficient utilization of resources through server consolidation and dynamic resource provisioning while adhering to SLAs and decreasing the cost of datacenter operations.

During the live migration process, an active VM is migrated from one physical host (source) to another physical host (destination) over the network. This requires a VM state transfer which consists of a snapshot of main memory and device states. From a cloud provider perspective, a VM migration should be transparent enough to have an unnoticeable fast migration by neither exposing the latency to the user nor affecting the other collocated VMs. However, existing migration approaches, such as pre-copy [5] and post-copy [6], are inefficient for highly utilized physical machines inside a datacenter. These approaches are quite inefficient in case of workloads with large working set size or even memory intensive workloads. Furthermore, they can incur significant performance overhead by consuming huge amount of network bandwidth. Besides that, the post-copy approach which in a way is efficient than pre-copy [6] but does not provide any destination VM guarantee during the migration period. This becomes another concern from a cloud provider perspective.

In this paper, we propose *reliable lazy copy* (RLC) approach consisting of all the three phases of the process migration, namely - *push phase*, *stop-and-copy phase* and *pull phase* [5]. It is designed to retain the beneficial features of both the pre-copy and post-copy approaches. The disadvantages of both pre-copy and post-copy approach are circumvented by introducing a *learning phase* prior to the push phase. During the learning phase an estimate of *writable working set* (WWS) is obtained. This results in an almost single time transfer of pages. Similarly, the post-copy approach's reliability issue is addressed by modifying the pull-phase, so that the VM can safely resume execution on the source machine even if the destination machine crashes. To the best of our knowledge, RLC approach provides a reliable and efficient migration capability which works productively in highly utilized physical nodes.

We discuss the limitations of other migration schemes in Section II. Then, the proposed RLC approach is presented in detail in Section III followed by the implementation details in Section IV. Later, the relevant experimental results are discussed in Section V. Later, the related work is summarized in Section VI, and Section VII draws the conclusion and future work.

II. Other migration schemes limitation

To motivate our RLC approach, this section discusses the key limitations of the existing approaches that fail to provide effective migration.

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.56

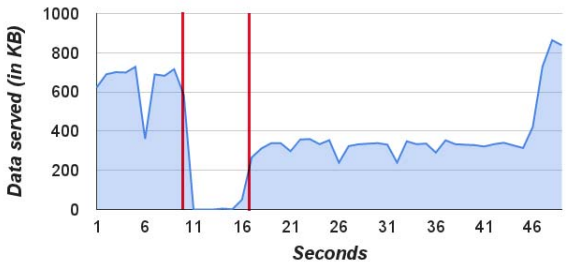

Figure 1. Perceivable downtime is the duration between the two vertical lines. Downtime (60 ms) is negligible when compared to the perceivable downtime (6sec). Apache Bench tool was used to generate the network requests for the post-copy live migration approach.

A. Pre-copy migration and its pitfalls

In the pre-copy approach, the main memory pages of the VM are transferred to the destination without halting the VM. Pages which got dirtied during the last round are retransmitted in the next round. This iterative copying of dirtied pages continues until a small writable working set (WWS) is identified. This phase can also stop after a preset number of iterations. After this, the VM is halted and the remaining dirtied pages along with the device states is transferred to the destination followed by resuming of the VM on the destination.

The problem with this approach is that a large WWS and a high page dirtying rate causes a huge number of page retransmissions. This results in unnecessary CPU and network bandwidth utilization. As the VM downtime is dependent on the WWS, it can be quite high in case of large WWS. Therefore, this approach is not a cost-effective approach from the cloud providers perspective.

B. Post-copy migration and its pitfalls

Post-copy approach works by transferring the VM's device states to the destination once the migration command is issued. Later, when the pages are required by the VM after resuming, those pages are fetched by the destination over the network via a remote page fault handler. We introduce a new performance metric with respect to the post-copy approach as *perceivable downtime*. It is a time period during which the VM is unresponsive after resuming execution on the destination. Figure 1 illustrates the difference between the downtime and perceivable downtime. The perceivable downtime is approximately 6 seconds when compared to the actual downtime which is 60 milliseconds. During that period, the response of the server to the requests is zero even after resuming on the destination.

The major issue with the post-copy approach is the very high perceivable downtime for memory intensive workloads as well as the unreliability of the pull phase of the approach. If the VM fails due to an issue on the destination physical machine during the migration process, we cannot restore it back, as the VM state is distributed across the source and destination machines.

III. Lazy Copy (LC) approach for Live Migration

In this section, we propose the lazy copy approach which tries to minimize the total data transfer and downtime in the pre-copy approach and the perceivable downtime in the post-copy approach, while optimally utilizing the network bandwidth at the same time. Further, our reliable lazy copy (RLC) approach is tolerant to fail-stop behaviour of the destination physical machine. We discuss both of the lazy copy and reliable lazy copy approaches along with other optimizations in the subsections below.

A. Lazy Copy Migration Algorithm

The various phases, earlier described by Clark et al. [5], of the lazy copy algorithm is as follows:

1) *Push phase:* The entire memory is transferred to the destination in a single pass without suspending the VM.
2) *Stop-and-copy phase:* The VM is then suspended and the dirty bitmap along with the device states are transferred to the destination. The dirty bitmap indicates the pages that got dirtied during the push phase.
3) *Pull phase:* The VM is resumed. Whenever the VM accesses a page that got dirtied during the push phase, a major page fault gets generated which results in a network fault. This network fault is resolved by retrieving the corresponding page from the source with the help of dirty bitmap that helps in resolving the page fault either locally or a transfer over the network.

B. Learning Phase

During the stop-and-copy phase, no pages get transmitted. Thus, a page gets transferred in the push phase and optionally in the pull phase. The page is fetched from the source in the pull phase, only if it gets dirtied. Therefore, we introduce a *learning phase* before the push phase to minimize the number of pages that gets transferred twice.

When the VM migration process is initiated, the learning phase starts to estimate the pages present in the writable workable set. These pages will not be transmitted during the push phase as they are most likely to be dirtied again and have to be retransmitted during the pull phase. If the migration is initiated at time t, then the learning phase happens during the time interval $[t, t + \delta]$. The parameter δ has an impact on the accuracy of estimated WWS and the overall migration time.

We use an adaptive histogram to estimate the WWS. The learning phase interval $[t, t + \delta]$ is divided into equal sized epochs and at the end of each epoch, Algorithm 1 is invoked. The algorithm estimates the WWS by computing a histogram of page usage with a forgetting factor α. The array

Algorithm 1 Estimating WWS using adaptive histograms.

```
1: sum ← 0.0
2: for i ← 1, nopages do
3:     hist[i] ← αdb[i] + (1 − α)hist[i]
4:     sum ← sum + hist[i]
5: end for
6: average ← sum / nopages
7: for i ← 1, nopages do
8:     if hist[i] ≥ average then
9:         wwsapx[i] ← 1
10:    end if
11: end for
```

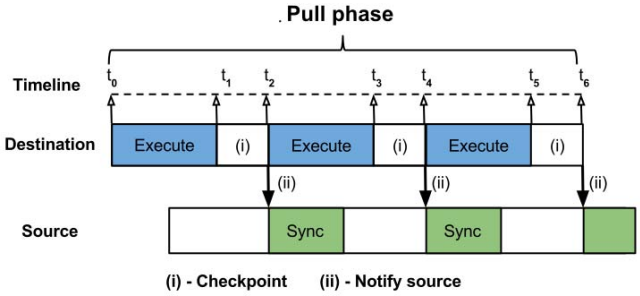

Figure 2. VM is suspended and its state is checkpointed during the time intervals marked (i). The VM state at the source gets asynchronously updated after each checkpoint.

$hist$ holds the histogram of a page usage for every page and the array db holds the dirty bitmap during the last epoch. The bitmap array $wwsapx$ contains the estimated WWS. We set δ to three seconds and α to 0.8 in our experiments.

C. Reliable Pull Phase - RLC

A VM can fail during its service life cycle when either the underlying physical machine or the host operating system running on it fails. We assume a *fail-stop* behaviour of all the components in the subsequent discussion. During the migration process, failure of either the source or the destination machine will have different implications depending on the phase of migration as discussed below.

- *Push phase and stop-and-copy phase:* Failure of the source will result in permanent loss of VM. There will not be any affect if the destination fails as the source still has the current active state of the VM.
- *Pull phase:* The VM is active on the destination machine. Since the VM state transfer from source to destination is still under progress, the source machine may contain the current state of the VM partially. Failure of either the source or the destination results in VM failure.

Thus, pre-copy approach is tolerant to destination failures whereas the post-copy and lazy copy approaches are not, due to the unreliable pull phase. In this work, we modify the pull phase to make our LC migration scheme tolerant to destination machine failures. The pull phase is divided into epochs and at the end of each epoch the incremental changes in the VM state at the destination are checkpointed to a secondary storage device such as NAS. The secondary storage device is accessible to both the source and destination machines through a network. After checkpointing the latest state, the destination notifies the source so that the source can update its state asynchronously. This reduces the VM recovery time on the source in case of a destination failure. The externally visible state of a VM during each epoch is buffered and released only after successfully checkpointing the latest VM state to the disk. Since the VM is paused at the end of each epoch, there will be a performance degradation. And this degradation is only during the pull phase and it is the cost we pay for reliability. If the destination machine fails, then the VM can be restored at the source by using the latest checkpointed state. The hard disk space consumed by the log files generated during checkpointing depends on the duration of the pull phase and is usually small.

1) Checkpointing Memory and Device States: The memory and the device states constitute the dynamic state of a VM. So checkpointing them is equivalent to checkpointing the VM state, which begins at the starting of the pull phase. At the end of each epoch, the VM is suspended; its dirtied pages and the device states are checkpointed and committed. Then the source machine is notified which in turn starts updating the state of the VM asynchronously. After the notification to the source, the VM resumes execution and the next epoch starts. Figure 2 summarizes this procedure.

2) Network buffering: Today, most of the applications rely on TCP connections which provide strong service guarantees. Thus, there is no requirement of the packet replication, since their loss will be accounted as a transient network failure. This fact simplifies the network buffering problem in which the packets get queued for transmission and are only transmitted after the VM state is successfully checkpointed at the end of each epoch. Thus any VM state exposed to the external world can always be recovered from the checkpointed state.

3) Disk State Consistency: During the pull phase, all the disk operations are made synchronous. Before an update is made to a disk during an epoch, the old data is copied to a shared log file and then the new version is synchronously committed to the disk. If the destination VM fails, the disk changes are reverted back with the help of the old disk state that is present on the shared log file between the source and the destination.

D. Other Optimizations

We incorporated some optimizations in our lazy copy VM migration implementation which are discussed in the following subsections.

1) Block Based Paging: Handling page faults involve two kinds of overhead - process invoking the page fault handler and network protocol overhead. These overheads can be reduced by fetching a contiguous group of pages which we call as a *page blocks*, for every page fault. Table I shows

Table I
DATA TRANSFER OVERHEAD WHILE TRANSFERRING PAGE BLOCKS OF DIFFERENT SIZE.

Pages transferred	Size (KB)	Data transferred (KB)	Overhead (%)
1	4	4.53	13.28
4	16	16.72	4.52
16	64	65.49	2.23
64	256	259.84	1.5
128	512	516.47	0.87
256	1024	1028.92	0.48
1024	4096	4115.66	0.48

the data transfer overhead while transferring page blocks of different sizes. If the block size is too large, then the page fault handling time increases, which proportionately hampers the VM progress. So the block size can be neither too large nor too small. Through experimentation, we chose a block size of 128 pages in our implementation. During the page block transfer, only the dirtied pages are transferred. The page block associated with a page i is defined as the pages falling in the interval $[i - \alpha * 128, i + (1 - \alpha) * 128]$. We chose $\alpha = 0.25$ heuristically in our implementation.

2) Active Background Page Prefetching: Instead of fetching page blocks only on a demand basis, a low priority background thread proactively fetches dirtied pages from the source. This decreases the number of major page faults and hence the perceivable downtime. Whenever the page fault handler gets activated, the background page prefetching thread gets suspended. For post-copy and lazy copy migration techniques, page prefetching is important to completely transfer the VM state in deterministic time. Otherwise, some pages could remain on the source for arbitrarily long times until a page fault occurs. This unnecessarily hogs the resources on the source machine.

3) Page Compression: The entire main memory is divided into logical blocks of size 32MB. During the push phase, each block is iteratively sent to the destination machine after applying the LZO compression algorithm [7]. LZO (real-time compression) algorithm requires a buffer of size B bytes to compress and store a block of size B bytes. Due to this, the size of the logical blocks for compression cannot be arbitrarily large. However, page blocks transferred during the pull phase are not compressed.

4) Multi-Threaded Push Phase: If enough compute power is available and the available network bandwidth is not sufficiently utilized, we can deploy more than one thread to carry out the push phase. If there are k threads, then the main memory is divided into k segments of equal size. Each thread compresses a segment of memory and transfers it to the destination on a separate socket connection.

IV. IMPLEMENTATION DETAILS

The proposed lazy copy migration algorithm has been implemented on Qemu 0.13 and KVM for Linux 3.6.3 kernel. The source code of Qemu/KVM is modified at specific points related to memory allocation and migration.

A. Page Fault Handler

We use a client-server model to transfer pages from the source to destination. The server resides on the source machine, while the client is on the destination. During the push phase, the main memory pages are transferred to the destination reside in the virtual address space of client. Whenever the VM accesses a page whose page table entry (PTE) is not yet created in the page table, a page fault is generated. The page fault handler refers to its page bitmap to check if the client contains a latest version. The presence of latest version results in creation of a new PTE, otherwise the page is fetched over the network by communicating with the server through the client and then the PTE is created.

B. Reliable Pull Phase

In the following subsections, we present various implementation details with respect to the reliable pull phase.

1) Memory and other device states: The dirty bitmap of every epoch for checkpointing is obtained by tracking the guest writes to memory which is the similar feature used by the pre-copy approach through *shadow page tables*. The identified dirtied pages, which are obtained at every epoch and the device states are directly dumped to the epoch based shared log file that is directly accessible to the source over the network. The source updates its state using these log files asynchronously and deletes them.

2) Network buffering: The network buffering mechanism has been implemented as a Linux queuing discipline. The inbound traffic is directly delivered to the guest but the outbound traffic is queued until the current VM state has been checkpointed. This is achieved by using the libnl library[8] which provides a *plug* module to buffer and release the packets through *intermediate functional block* driver[9] which redirects the output to the bridge.

3) Disk commit: In QEMU, all the reads and writes to the disk are asynchronous through asynchronous I/O (AIO). We synchronously write the data to the shared log file shared between source and destination (same as the disk block) before committing the write request. This process continues till the pull phase is over.

V. EXPERIMENTAL WORK

In this section, we present a detailed evaluation of the lazy copy migration technique against the existing pre-copy and post-copy approaches. Our test environment consists of 2 Dell Workstations with 12GB RAM. The source and the destination machines are connected via a Gigabit Ethernet switch. Network bandwidth is available for complete utilization by the source and the destination machines. Each VM is allocated 2GB of memory in all the experiments.

We chose applications and workloads, both real and synthetic, with diverse characteristics for the evaluation process. From SPEC CPU2006 [10], a memory intensive benchmark (`429.mcf`) and a cpu intensive benchmark (`401.bzip2`)

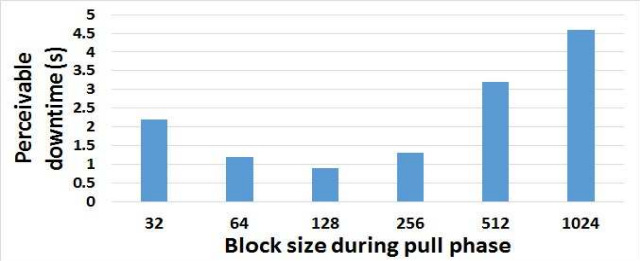

Figure 3. Perceivable downtime measured for various block sizes using Apache Bench. lazy copy algorithm is used for VM migration.

are chosen. We have used Linux Kernel Compile (`LKC`), and `memcached` [11] server which caches multiple key/value pairs in the main memory. For load generation, we use `memaslap` as client and it resides on a different machine. Finally, we used a synthetic benchmark `memtester` (with WWS as 1GB), which is a highly memory write intensive program for finding faults in RAM.

A. Block Size Selection for the Pull Phase

We use perceivable downtime as the criteria for selecting the block size during the pull phase. For block size selection, we use the naive pull phase which does not support reliability. Figure 3 shows the perceivable downtime for different block sizes. Network requests to the Apache server running on the VM are generated by running the Apache client program (Apache Bench) [12] on a machine different from both the source and destination. Figure 3 shows that the perceivable downtime is minimal when the block size is 128 pages. The perceivable downtime is the time period in which the server does not respond to the client at the start of pull phase. As the block size increases, the thread which handles the demand paging requests will end up waiting for the prepaging thread running in the background, as only one of them will be active at the same time. This results in increased freeze time of the VM process, thus exacerbating the perceivable downtime.

B. Pre-copy, Post-copy Implementations and Lazy Copy Variants

In our experiments, we used the existing implementation of the pre-copy VM migration in the KVM hypervisor. The KVM pre-copy implementation applies page compression to the pages with same content. These are generally zero pages and only single byte is transferred along with the page offset information.

We implemented our own post-copy VM migration algorithm in KVM to compare its performance against the proposed approach. The post-copy implementation does a block based prefetching of pages both in the background and on an on-demand basis. It is identical to the LC pull phase. Since the implementation is for hardware assisted VMs, the ballooning based approach [13] cannot be used.

We implemented the following four variants of the LC migration algorithm to measure the impact of various optimizations.

1) *lazy copy (LC):* The lazy copy migration algorithm without the learning phase. Niether page compression nor multi-threading is used during the push phase. However, pages with the same value throughout are compressed by sending the corresponding value alone.
2) *lazy copy learning (L^2C):* lazy copy implementation from the above with a learning phase before the push phase.
3) *lazy copy learning-compression (L^2C^2):* L^2C implementation with LZO algorithm based page compression during the push phase. Since block based compression is done, single byte page compression used in LC and L^2C does not apply.
4) *lazy copy learning-compression -parallel (L^2C^2P):* L^2C^2 implementation wherein the push phase is parallelized.

We use the acronyms PR and PO for pre-copy and post-copy respectively. Henceforth, we use acronyms to refer to various migration schemes.

C. Pre-copy vs Post-copy vs lazy-copy Variants

In this section, we compare the basic lazy-copy approach and its optimized variants against the existing pre-copy and post-copy algorithms. Table II show how pre-copy, post-copy, lazy copy and lazy-copy learning migration approaches compare against each other on various workloads. The performance metrics used are application degradation, data transferred, migration time and downtime.

1) Total Data Transferred: The data transferred during the post-copy migration of a VM is around 2 GB (Table II). This is as expected since the VM memory size is of 2 GB. In the case of pre-copy approach, the total data transferred depends on the working set size and the page dirtying rate. The data transferred for write intensive applications like `mcf` and `memcached` is substantially higher than their VM size due to multiple page retransmissions. For `bzip2`, the data transferred is less than the VM size. This is due to the compression of zero pages. While for lazy copy, it can be observed from the Table II that the lazy copy approach outperforms pre-copy on all workloads. When the learning phase is invoked in the lazy copy approach, it can be noticed that even for write intensive applications with working set size as large as the allocated VM size (`mcf` and `memcached`), the data transferred in the lazy copy approach almost matches that of post-copy. This shows that the learning phase is effective in estimating the writable working set which will not be transmitted during the push phase. With the invocation of the learning phase, the total data transfer gets decreased by a factor of 1.04x to 1.65x against lazy copy and 1.08x to 5.92x against pre-copy.

Table II
PERFORMANCE COMPARISON OF PRE-COPY (P), POST-COPY (PO), LAZY COPY(LC) AND LAZY COPY LEARNING (L^2C) MIGRATION SCHEMES.

	Total data transferred (MB)				% Degradation				Total migration time (s)				Downtime (ms)			
Workload	PR	PO	LC	L^2C	PR	PO	LC	L^2C	PR	PO	LC	L^2C	PR	PO	LC	L^2C
memtester	2206	2064	2014	1658	24.7	3.5	6.6	2.5	58.7	60.9	23.7	22.4	245	99	143	145
bzip2	892	2064	645	576	3.2	3.9	2.6	1.8	12.5	70.3	11.1	9.6	255	79	120	126
mcf	7234	2064	3489	2103	18.7	12.3	12.2	5.9	197.6	60.3	51.9	37.7	712	137	220	230
LKC	536	2064	519	498	18.8	12.6	5	4.3	8.7	75.7	8.5	7.9	212	41	55	63
memcached	14373	2064	3884	2429	64.8	9.2	8.9	5.2	302.3	74.5	57.4	35.8	8448	200	446	427

2) Total migration time: Total migration time for lazy copy learning decreases by a factor of 1.3x - 8.4x for pre-copy and 1.6x - 9.6x for post-copy. For the post-copy approach, it can be noted from Table II that the migration times are different for different workloads. This is due to the variation in the number of major page faults that get generated for different workloads, even though the same amount of data is being transferred. For pre-copy approach, the migration times are proportional to the total data transferred which is naturally high for write intensive workloads. lazy copy approach outperforms the pre-copy and the post-copy approaches, and lazy copy learning outperforms the lazy copy approach on all the workloads. lazy copy outperforms post-copy even on workloads `mcf` and `memcached` for which it transfers more data than post-copy. This is due to the presence of read-only pages in the working set resulting in fewer network faults which leads to less network bandwidth contention during the pull phase.

3) Application Performance Degradation: From Table II, we can infer that the L^2C algorithm outperforms all the migration algorithms on every workload with a maximum performance degradation under 6 percent. The application performance degradation computed for lazy copy learning includes the learning phase overhead. lazy copy algorithm outperforms post-copy (1.01x) and pre-copy (1.01x to 1.57x) approaches on all workloads, except on `memtester` where post-copy does better than lazy copy. This is due to the load on the source and destination machines while transferring pages during the push and pull phases respectively. All the migration algorithms show a relative performance degradation on write intensive workloads. However, the performance degradation is substantial for the pre-copy approach, like in the case of benchmarks `memcached` and `memtester`. Even for a mixed workload like `LKC`, the performance degradation for pre-copy is substantial when compared with post-copy and lazy copy approaches.

4) Downtime: The general trend is that the downtime is higher for write intensive workloads (refer Table II). This is due to the excessive resource (especially pages) utilization of the VM prior to the stop-and-copy phase. The performance of the thread that performs the stop-and-copy phase is effected due to that. Post-copy migration approach performs consistently better than the rest of the techniques. This is due to the minimal amount of data transfer that happens during the stop-and-copy phase. Pre-copy approach performs the worst due to the large number of dirtied pages that need to be transferred. lazy copy and lazy copy learning does far better than pre-copy but do not outperform post-copy.

Table III
EFFECT OF PAGE COMPRESSION DURING PUSH PHASE ON LAZY COPY (LC) AND LAZY COPY LEARNING (L^2C) MIGRATION SCHEMES.

	% Degradation		Total data transferred (MB)		Total migration time (s)	
Workload	L^2C	L^2C^2	L^2C	L^2C^2	L^2C	L^2C^2
memtester	2.5	2.4	1658	1199	22.4	16.4
bzip2	1.8	1.0	576	457	9.6	8.2
mcf	5.9	3.4	2103	1784	37.7	23.1
LKC	4.3	1.1	498	169	7.9	6.1
memcached	5.2	4	2429	1923	35.8	30.7

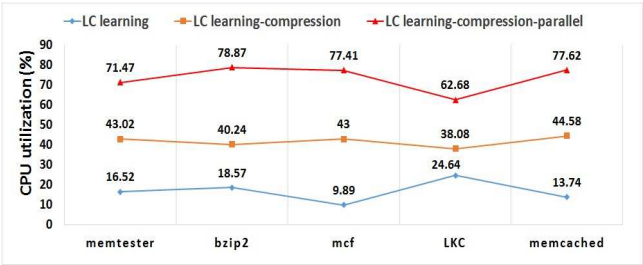

Figure 4. % CPU utilization during the push phase for lazy copy learning (L^2C), lazy copy learning-compression (L^2C^2) and parallel lazy copy learning-compression (L^2C^2P) approach.

D. Push Phase Page Compression

In this section, we present the impact of compressing pages using the real time LZO algorithm in the push phase. Pages are not compressed in the pull phase as the goal there is to serve the pages as quickly as possible. Table III shows the impact of page compression on different migration metrics. The total data transfer gets decreased by a factor of 1.83x to 7.47x for pre-copy, 1.16x to 12.21x for post-copy and 1.26x to 2.94x for LC. Whereas the total data transferred in the push phase of the naive lazy copy, gets decreased by a factor of 1.1x to 5x. The migration time denoted in the table also includes the compression time. The total migration time gets decreased by a factor of 2.43x - 8.57x for post-copy and 1.42x - 9.84x for pre-copy. Figure 4 shows the CPU utilization by various migration schemes. With the introduction of compression, the CPU utilization increases by 16.78% to 30.13% in push phase, while the

Table IV
EFFECT OF PARALLELIZING THE PUSH PHASE ON LAZY COPY LEARNING-COMPRESSION (L^2C^2) MIGRATION SCHEME.

	% Degradation		Total migration time (s)	
Workload	L^2C^2	L^2C^2P	L^2C^2	L^2C^2P
memtester	2.4	1.7	16.4	12.9
bzip2	1.0	0.9	9.6	7.4
mcf	3.4	1.8	32.7	26.7
LKC	1.1	0.8	7.9	5.7
memcached	4	7.4	35.8	27.4

Table V
AVERAGE PAGE BLOCK SIZE FETCHED BY BOTH THE THREADS ALONG WITH THE RATIO OF BACKGROUND VS ON-DEMAND THREAD.

Workload	Ratio	Effective Block size
memtester	1.6	80
bzip2	99	18.6
mcf	2.1	52.9
LKC	0.9	4.7
memcached	99	23

push phase time period decreases by an average factor of 1.5x (not shown in the Figure).

E. Push Phase Parallel Data Transfer

Table IV compares the performance of the parallel version of lazy copy compression when two threads are employed during the push phase with the single threaded version. It can be observed that the application performance degradation and migration time consistently improves in the parallel version. The performance degradation of `memcached` increases as the VM and the migration mechanism share the same network link. Due to parallelization, the migration mechanism demands more network bandwidth from the shared link. Figure 4) shows the increase in the CPU utilization due to push phase parallelization. If the source machine has under utilized compute power in the form of idle cores, this is actually an ideal usage. Depending upon the criteria, the cloud providers can use this for VMs with only memory and CPU intensive workloads.

F. Block Based Paging & Proactive Background Prepaging

Table V shows the average block size during the pull phase for various workloads. For memory intensive applications, the average block is large, as there is more likelihood of having continuous pages to constitute blocks. As the block size increases, the number of major page faults and network faults decrease. The network faults reduced by an average of 35.8 times with minimum being 4.7 times for `LKC` and the maximum being 80 for `memtester`. Table V also shows the ratio of pages fetched by the background thread versus the on-demand thread. Fetching pages in the background before they are faulted helps in reducing the perceivable downtime. It can be observed that more pages are fetched by the background thread when compared with the on-demand thread for all workloads except for `LKC`. This is because of irregular write access patterns.

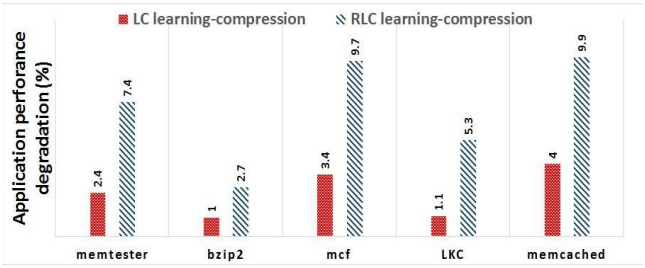

Figure 5. Application degradation (%) when reliable pull phase is enabled.

G. Effect of Reliability

When we use the reliable pull phase in the lazy copy learning-compression migration scheme, the application performance degradation increases. This is due to synchronized disk writes and periodic suspension of VM for checkpointing. Figure 5 shows the impact of reliability on the application degradation. The performance degradation is substantial for memory intensive workloads such as `memtester`, `mcf` and `memcached`, since the time taken for checkpointing a VM state is proportional to its WWS size per epoch. Since `bzip2` is compute intensive with small WWS, the increase in the performance degradation is small. Due to multiple writes, the performance degradation of `LKC` is also substantial.

Correctness verification: In order to verify the working of the designed reliable approach, we deliberately induced network failure / process kill during the following phases: (1) before the resumption of the VM at destination when all the data has been transferred to the destination and it is about to resume, (2) during the transient state of a VM between two checkpoints (3) during the checkpointing phase, when the VM's state is written to the disk. The checkpoints were taken every 50 milliseconds throughout the pull phase of the lazy copy learning-compression approach. For every workload and at every failure point, the source successfully took over the execution. All the workloads continued to run till successful completion.

VI. RELATED WORK

Pre-copy approach for live migration of VMs, derived from process migration[14], has been a widely studied topic during the last decade [5]. Clark et al. [5] proposed a pre-copy approach on top of the Xen VMM with dynamic rate limiting to increase the available network bandwidth utilization and reduce large downtimes due to high page dirtying rates. Svard et al. [15] proposed a technique wherein, the incremental changes in dirtied pages are computed and the incremental changes are transmitted to the destination after applying a run-length encoding (RLE) compression algorithm. Jin et al. [16] proposed a migration scheme which uses an adaptive compression technique. Ibrahim et al. [17]

proposed an online adaptive pre-copy algorithm for VMs running HPC applications. Zhang et al. [18] tried to reduce the total data transferred during the migration process by using hash based fingerprints to find identical and similar pages.

Hines et al. [13] proposed the post-copy live migration approach for para-virtualized VMs and implemented it on Xen hypervisor. They used dynamic self-ballooning technique and pre-paging technique to reduce the total data transferred and network faults. We consider the HVMs from the Cloud perspective where only HVMs are used to guarantee SLAs. Cavilla et al. Hirofuchi et al. [19; 20] showed that post-copy migration is suitable for VM consolidation when the VM workloads are suddenly changed. They also used the post-copy approach for instantaneous consolidation of the VMs.

The closest work to our approach is that of Peng et al. [21] . Peng et al. [21] provides a hybrid self-migration technique for guests without the intervention of the hypervisor. Whereas our approach is guest agnostic and hence requires no modifications to the guest operating system. Other related work on hybrid migration include Luo et al. [22], Nocalae and Cappello [23] and Zheng et al. [24] which only talk about *disk* block based live migration, whereas our technique concentrates on live migration where the disk is already shared and memory becomes the critical state to be transferred to the destination inside data centers.

Our reliable pull phase is similar to that of Remus [25]. Remus provides fault tolerance to fail-stop failures of a physical host by asynchronously propagating its state to a backup host at high frequency. We adapt the synchronous disk writes in spite of its overhead due to the short pull phase period. This is not applicable in the context of high availability where checkpointing has to go on continuous basis.

VII. Conclusions

We have designed and implemented a reliable lazy copy live migration technique which provides fast and efficient migration at a minimum cost. Our novel learning phase as introduced prior to the push phase, which is used to estimate the WWS, helps us in transferring the memory only once while adhering to the SLAs. This leads to minimal source's resource utilization and an unnoticeable latency to the user. Thus, our RLC approach provides a reliable and an efficient migration capability, thus making it a promising technique for the future data centers.

References

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

[2] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XII.

[3] H. Lv, Y. Dong, J. Duan, and K. Tian, "Virtualization challenges: a view from server consolidation perspective," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, ser. VEE '12.

[4] H. Lv, X. Zheng, Z. Huang, and J. Duan, "Tackling the challenges of server consolidation on multi-core systems," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10.

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

[6] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43.

[7] "Lempel-ziv-oberhumer (lzo) real time data compression library." [Online]. Available: http://www.oberhumer.com/opensource/lzo/

[8] "Netlink protocol library suite (libnl)." [Online]. Available: http://www.infradead.org/~tgr/libnl/

[9] "Intermediate functional block device (ifb)." [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb

[10] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34.

[11] "Distributed caching with memcached."

[12] "Apache bench." [Online]. Available: http://httpd.apache.org/

[13] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09.

[14] F. Douglis and F. Douglis, "Transparent process migration in the sprite operating system."

[15] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11.

[16] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*.

[17] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman, "Optimized pre-copy live migration for memory intensive applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11.

[18] X. Zhang, Z. Huo, J. Ma, and D. Meng, "Exploiting data deduplication to accelerate live virtual machine migration," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*.

[19] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Reactive consolidation of virtual machines enabled by postcopy live migration," in *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, ser. VTDC '11.

[20] ——, "Enabling instantaneous relocation of virtual machines with a lightweight vmm extension," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*.

[21] P. Lu, A. Barbalace, and B. Ravindran, "Hsg_lm, hybrid-copy speculative gues os live migratoin without hypervisor," ser. Systor '13.

[22] Y. Luo, B. Zhang, X. Wang, Z. Wang, Y. Sun, and H. Chen, "Live and incremental whole-system migration of virtual machines using block-bitmap," in *Cluster Computing, 2008 IEEE International Conference on*.

[23] B. Nicolae and F. Cappello, "A hybrid local storage transfer scheme for live migration of i/o intensive workloads," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12.

[24] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-aware live storage migration for clouds," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11.

[25] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08.