Analysis of live virtual machine migration algorithms used in KVM and Xen
(Pre-copy vs. Post-copy live migration)

Junlin Huang

TU Dresden, Lehrstuhl Rechnernetze,
Nöthnitzer Str. 46, 01187 Dresden, Germany

Abstract. Live migrating operating system between physical machines is very useful for administrators in data centers and clusters. It allows clean seperation between hardware and software, and also provides some powerful features like fault management, load balancing and low-level system maintenance.

In order to decrease the total migration time of pre-copy and post-copy and save network bandwidth during migration, some algorithms can be used to avoid intensive copying of memory pages and to compress the being transferred memory pages. In this paper weintroduce first an algorithms for migrating operating system with memory intensive applications. A mechanism called Dynamic Sell-Ballooning is also suitable for migrating operating systems without degrading the application performance. Due to the resend problem of frequently updated memory pages, hierarchical copy and dynamic pages transfer recording can be used to solve this problem. Dynamic pages transfer recording is combined with a compression algorithms which can save a lot of network banwidth. The statistics also show advantages of each design.

Keywords: Cloud Computing, Live Migration, Xen, KVM, Pre-Copy, Post-Copy

1 Introduction

Operating system virtualization has attracted considerable interest in recent years, particularly from the data center and cluster computing communities. It has previously been shown that virtualization allows many OS instances to run concurrently on a single physical machine with high performance, providing better use of physical resources and isolating individual OS instances.

KVM is a most recent revolution of open source, x86 virtualization technology. It turns the linux kernel into a bare-metal hypervisor using the hardware support built into different processers, like Intel and AMD. That means that KVM can use linux to do what a hypervisor needs to do, like tasks scheduling, managing memory and interacting with hardware devices. What KVM does is

creating virtual machines as linux processes which can then run either linux or windows as a guest operating system. It uses a modified version of another open source module - QEMU - to provide IO device emulation inside KVM is thus able to effciently and effectively run both linux and windows workloads in virtual machines - and also linux applications alongside natively if required. The reason people use KVM is that KVM is able to take advantage of the performance, scalability and security already built into Linux, which has been enterprisely hardened for over 10 years and is trusted by millions of organizations in the heart of their data center to run their mission critical workloads. This gives KVM a significant feature that normal virtualization solutions don't have.

Not like KVM, Xen runs in a more privileged CPU state than any other programms on the machine. It is a hypervisor that using a microkernel design and providing services that allow multiple operating systems to run on the same machine concurrently. Xen manages memory and schedules CPU for all virtual machines. Xen has a special part which is called "dom0", this part is the most privileged domain(we call a virtual machine a "domain" in Xen) and the only domain which can access the hardware directly. The Xen Project was born with the concept that virtualization should be controllable in the manner which later came to be called Cloud Computing. The availability of Xen Cloud Platform and its associated programming interface ensure that you can control your VMs the way you want to, using whatever tool stack you choose[**?**]. This feature makes Xen fitter for live migration of virtual machines.

Pre-copy and post-copy are appraoches that used in live migration techniques. First, pre-copy transfers all memory pages and then copies pages just modified during the last round iteratively. When appcations' writable working set(WWS) becomes small or the maximum number of iterationsis reached, the virtual machine is suspended and only CUP state and dirty pages in the last round are sent out to the destination[CFH+05]. In contrast, Post-copy defers the memory transfer phase until after the virtual machines' CPU states have already been transferred to the target host and resumed there. post-copy first transfers all processor states to the target host, starts the virtual machines at the target host and then actively pushes memory pages from origin to target host. Concurrently, any memory pages that are faulted on the virtual machine at target host and not yet pushed, will be tranferred over the Internet from source.

2 Background

2.1 Operating System Live Migration

Live Operating System Migration means migrating an operating system and its applications as one unit that allows us to avoid many difficulties faced by process-level migration approaches. The narrow interface between a virtual operating system and the virtual machine monitor makes it possible that the original host machine and network remain available in order to service certain system calls or

even memory accesses on behalf of migrated processes[CFH+05]. Live migration at the level of an entire virtual machine allows states in memory to be transferred in a consistent and efficient way. This applies to kernel-internal states as well as application-level states, even when they are shared between many cooperating processes. For Example, an online game server or streaming media server can be migrated without requiring clients to reconnect[CFH+05]. Live migration also allows a separation of concerns between the users and operators of a data center or cluster. Users don't need to provide the operator any OS-level access at all when the operator migrates the software and services that running within the virtual machine. Similarly the operator needs not be concerned with the details of what is happening within the virtual machine, instead they can just simply migrate the entire operating system and its attendant processes as a single unit[CFH+05].

The logical process of live migrating an operating system can be seperated in 5 stages as shown in Figure 1.

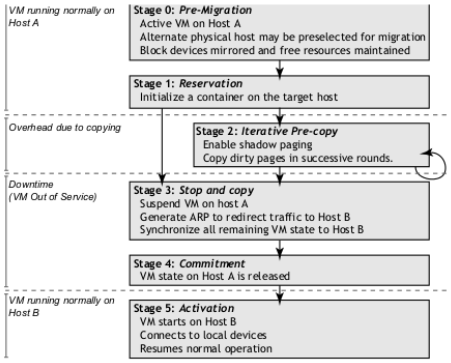

Fig. 1: Live Migration Stages

Stage 0: Pre-Migration. An active VM is running on physical host A. To make any future migration faster, a target host(we say host B in the following text) is needed to reserve the resources for the VM on host A.

Stage 1: Reservation. A request muss be sent to migrate an OS from host A to host B. The necessary resources should be available on B in this step and a VM container of that size should also be reserved. Failure to secure resources here means that the VM simply continues to run on A unaffected.

Stage 2: Iterative Pre-Copy. Now the iterative steps of migration begin. During the first iteration, all pages are transferred from A to B. In the following interations of migration only the dirty pages will be transferred.

Stage 3: Stop-and-Copy. In this step the count of iterations or the percentage of retransmitted pages are lower than a specified number. Then the running OS instance at A will be suspended and its network traffic will be redirected

to B. Also the CPU states and any remaining inconsistent memory pages are then transferred from A to B. At the end of this step there is a short suspend for copying the rest memory pages from Host A to Host B. The copy at A is still considered to be primary and is resumed in case of failure.

Stage 4: **Commitment**. Host B then tells A that it has successfully received a consistent OS image. Host A replys B as a commitment that A may now discard the original VM, and host B becomes the primary host.

Stage 5: **Activation**. Now host B may be activated and its device drivers will be reattached to ensure that it can correctly run on the new machine.

Overall. Live OS migration is a extremely powerful tool for cluster administrators, allowing separation of hardware and softweare considerations, and consolidating clustered hardware into a single coherent management domain.

2.2 Pre-copy and Post-copy

Pre-copy and post-copy are appraoches that are used in live migration techniques. Figure 2 shows the mechanisms and difference between pre-copy and post-copy. First, pre-copy transfers all memory pages and then copies pages just modified during the last round iteratively. As shown in Fugure 2, there are N rounds of copy. When appcations' writable working set(WWS) becomes small or the maximum number of iterations is reached, the virtual machine is suspended and only CPU states and dirty pages in the last round(Downtime in Figure 2: Pre-copy Timleline) are sent out to the destination. In contrast, Post-copy defers the memory transfer phase until after the virtual machines' CPU states have already been transferred to the target host and resumed there. First, post-copy transfers all processor states to the target host, downtime is before copying of all pages. Then it starts the virtual machines at the target host and actively pushes memory pages from origin to target host. Concurrently, any memory pages that are faulted on the virtual machine at target host and not yet pushed(shown as Post-Copy Prepaging in Figure 2), will be tranferred over the Internet from source.

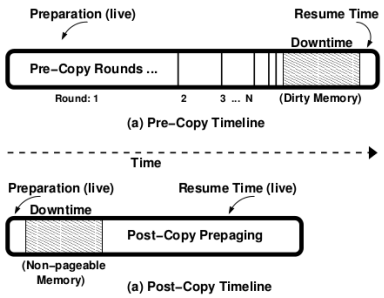

Fig. 2: Timeline for Pre-copy vs. Post-copy

2.3 Xen and KVM

Xen is a high performance resource-managed virtual machine monitor(VMM) which enables applications such as server consolidation, co-located hosting facilities, distributed web services, secure computing platforms and application mobility[BDF+03]. Xen can host many operating systems at the same time and enables user to dynamically instantiate an operating system to execute whatever they desire. Also, Xen can shutdown the operating systems on idle or low balanced servers and migrate the other busy operating systems to a smaller group of servers which are capable to host the migrated operating systems.

KVM(Kernel-based Virtual Machine) is a unique hypervisor. The KVM developers, instead of creating major portions of an operating system kernel themselves, as other hypervisors have done, devised a method that turned the Linux kernel itself into a hypervisor[Hab]. Under this model, every virtual machine is a regular Linux process, scheduled by the standard Linux scheduler. Traditionally, a normal Linux process has two modes of execution: kernel and user. The user mode is the default mode for applications, and an application goes into kernel mode when it requires some service from the kernel, such as writing to the hard disk. KVM adds a third mode, the guest mode. Guest mode processes are processes that are run from within the virtual machine. The guest mode, just like the normal mode (non-virtualized instance), has its own kernel and user-space variations. From the non-virtualized instance, a KVM virtual machine is shown as a normal process and it can be killed just like any other process. KVM makes use of hardware virtualization to virtualize processor states, and memory management for the virtual machine is handled from within the kernel.

3 Memory Migration

3.1 Optimized Precopy Live Migration for Memory Intensive Applications

A study shows that due to a high rate of memory changes, the current KVM rate control and target downtime heuristics do not cope well with HPC applications: statically choosing rate limits and downtimes is infeasible and current mechanisms sometimes provide suboptimal performance. This study presents a novel on-line algorithm that is able to provide minimal downtime and minimal impact on end-to-end application performance. At the core of this algorithm is controlling migration based on the application memory rate of change. The performance of iterative pre-copy migration of KVM is controlled by the bandwidth allocated, as well as a target downtime for the last iteration. These two metrics are chosen by system administrators to both minimize the migration overhead and satisfy service level agreements(SLAs) that require guarantees on the maximum downtime. For HPC applications, downtimes are generally not as important as overall execution time, although downtime can result in application failure due to factors such as network timeouts.

This study shows the relation between down time and migration time, execution time and migration time with statistics which are shown below:

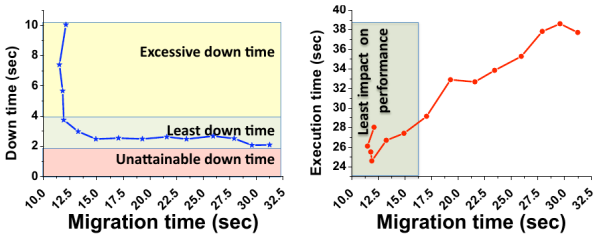

Fig. 3: A typical VM migration plot for the downtime vs. the duration of live migrationA typical VM migration plot for the downtime vs. the duration of live migration.

Several problems are apparent when examining the two plots: i) a static target downtime can be easily requested below the attainable downtime, resulting in increased migration time or failure to migrate the virtual machine; ii) best application performance is obtained with a short migration time, which in turn causes an increased downtime.

They did an experiment for analysing impact between migration and application performance, then had a conclusion that using a static target downtime as a condition for switching from iterative migration to stop-and-copy leads to sub-optimal behavior[IHIR11]. They propose a new technique where switching is decided by matching memory update patterns that indicate that no beneficial progress, reduction in downtime, is achieved by continuing live migration. As a progress measure they use the probability of further reduction in downtime, without requiring any static estimates.

Figure 4 shows the memory update patterns. The first pattern represents the case where the number of modified pages is not reduced by iterative pre-copying. In this case, the applications' memory-activities are active and rate of memory change exceeds the available bandwidth, if the migration goes on, the network will be fulfilled with memory pages and the performance of the live migration programm will degraded without any downtime reduction. The second pattern is when the application's memory-activities stop during execution such that a small downtime can be attained. In applications, this situation happens if the applications execute synchronization and barrier operations. The third pattern occurs when most of the transmitted pages are similar to those transmitted in the previous iteration: in this case the modified pages can be drained but the

application performs an iterative computation on the dataset. That means if an intensive memory activity of applications is detected, this migration goes into a downstate(in this experiment 1 second)

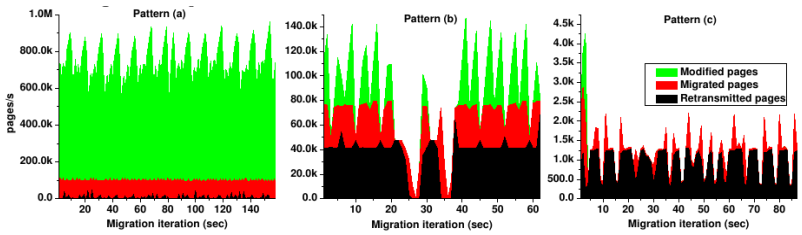

Fig. 4: a) modified pages are stable for a prolonged period of time; b) application activity drops below a certain threshold; and c) the migration activity involves high retransmission of pages.

lgorithm 1 describes the details of this approach. The main idea of this implementation is to estimate the rate of page transmission during migration. This is essential for detecting the three different patterns shown in Figure 4. Detecting pattern a requires monitoring the number of page changes per a constant time interval(For example 1 second). As shown in lines 25-37 of the algorithm, linear regression is used to estimate the number of pages to send in order to make the sampling interval constant. This implementation also computes a moving average in order to handle variation of the monitored data. A simple filtering technique is used to check if the rate of page modification is stable. If this rate is stable in a region, the migration switches then to a stop-and-copy stage(lines 41-43 in Algorithm 1). For pattern b, the implementation detects when all modified pages have been transferred in an interval less than the preset sampling interval (omitted for brevity in Algorithm 1). To track the page retransmission activity required for the detection of pattern c, this algorithm maintains a bitmap corresponding to the pages transmitted in the previous iteration. The number of retransmitted pages will be counted in this iteration and will be compared with the number of retransmitted pages in the last iteration. If the percentage of retransmissions exceeds 90% of the migrated pages the algorithm proceeds to the last stage of migration (lines 41-43 in Algorithm 1). In this algorithm only one history bitmap for the previous iteration is stored. It is not difficult to extend this algorithm to maintain a window of bitmaps more than two iterations. The starting part of migration is the worst period for downtime and does not match any of these patterns: the number of dirty pages monotonically decreases with iterations and the number of page retransmissions is quite small. Thus, in this implementation the whole memory space will be transferred once before enabling the convergence mechanisms[IHIR11].

3.2 Dynamic Self-Ballooning

A study identified a deficiency in both pre-copy and post-copy migration due to which free pages in the VM are also transmitted during migration, increasing the total migration time. To avoid transmitting the free pages, they developed a "dynamic sef-ballooning"(DSB) mechanism[HDG09]. Self-Ballooning is an existing technique that allows a guest kernel to reduce its memory footprint by releasing its free memory pages back to the hypervisor. DSB implements a balloon mechanism which can trigger periodically without degrading application performance and responds dynamically to VM memory pressure by inflating the balloon under low pressure and deflating under increased pressure. Kernel Memory allocation requests can be directly dealed by the DSB implementation without the need for guest kernel modifications. It neither requires external introspection by a co-located VM nor excessive communication with the hypervisor.

The implementation of DSB has three components: i) **Inflate the balloon**: A kernel-level DSB thread in the VM first allocates as much free memory as possible and hands those pages over to the hypervisor. ii) **Detect memory pressure**: Memory pressure indicates that some entity needs to access a page frame right away. In response, the DSB process must partially deflate the balloon depending on the extent of memory pressure. iii) **Deflate the balloon**: Deflation is the reverse of Step 1. The DSB process repopulates it's memory reservation with free pages from the hypervisor and then releases the list of free pages back to the guest kernel's free pool[HDG09].

For detecting the page fault at the target VM in this implementation, a mechanism called Pseudo-Paging is selected, the reasons are that it is the quickest way to implement and costs least, but it requires significant changes to the guest kernel. This mechanism swaps out all pageable memory at the source VM to an in-memory pseudo-paging device within the guest kernel. This can be achieved with minimal overhead and without any disk I/O. The memory reservation for the VM's source copy appears as a pseudo-paging device.

Figure 5 illustrates the details of this mechanism. Two loadable kernel modules, one inside the migrating VM and one inside Domain 0 at the source code, are needed to detect page-fault on the target VM and provide service.

Page-fault detection and servicing is implemented through the use of two loadable kernel modules, one inside the migrating VM and one inside Domain 0 at the source node. These modules leverage the work on a system called MemX[HG07], which provides transparent remote memory access for both Xen VMs and native Linux systems at the kernel level. The memory pages of the migrating VM at the source will be swapped out to the pseudo-paing device mentioned above. A lightweight MFN exchange(machine frame number exchange) mechanism is used to performed this “swap”. This “swap” is achieved without coping any memory pages and the memory pages are only mapped to Domain 0 at the source. CPU states and non-pageable memory pages are then transferred

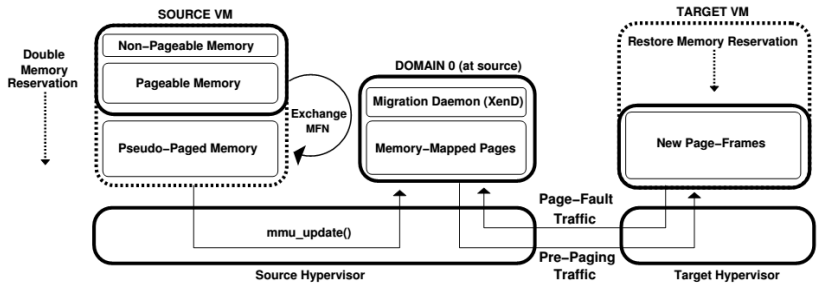

Fig. 5: Pseudo-Paging : Pages are swapped out to a pseudo-paging device within the source VM's memory by exchanging MFN identifiers. Domain 0 at the source maps the swapped pages to its memory with the help of the hypervisor. Prepaging then takes over after downtime.

to the target VM during the downtime of post-copy. Note the pseudo-paging approach implies that a small amount of non-pageable memory, typically small in-kernel caches and pinned pages must be transferred during downtime. This transmission increases the downtime of current post-copy implementation.

MFN exchange (machine frame number exchange) remaps the pseudo-physical address of the pages within the VM with zero copying overhead. The VM's memory reservation is first doubled and all the processes in the VM will be suspended for swapping out the pageable frame. This can be achieved directly through the use of existing software suspend code in the Linux kernel. As a frame is paged, the VM's PFN (pseudo-physical frame number) will be re-written to MFN mapping (called a physmap) and the frame's kernel-level PTE will also be re-written in order to simulate an exchange between the frame's MFN with that of a free page frame. The exchanged MFNs will be memory-mapped into Domain 0 once the exchanges are over. Once the VM resumes at the target, the VM on the target begins to request missing pages. The MemX "client" module in the target VM begins to deal with page faults and perform prepaging by coordinating with the MemX "server" module in Domain 0 at the source. These two modules communicate via a customized and lightweight remote memory access protocol (RMAP) which directly operates above the network device driver.

To make the Sell-Ballooning dynamic, the most important thing is detection of memory pressure. There are two mechanisms in the linux kernel for detecting memory pressure in a completely transparent manner. The first one is the kernel's existing ability to perform physical memory overcommitment. Memory overcommitment within an individual OS allows the virtual memory subsystem to provide the illusion of infinite physical memory. The linux kernel also provides a transparent mechanism to detect memory pressure: through the kernel's filesystem API which is called set_shrinker(). One of the function parameters acts as a callback to some memory-hungry portion of the kernel. This indicates to the virtual memory system that this function can be used to request the dealloca-

tion of a requisite amount of memory that it may have pinned. DSB periodically reclaim free pages that may have been released over time and inflates and deflates the balloon as much as possible. In this implementation only 95% of the memory will be inflated because inflating to 100% will due to a out-of-memory error. If memory pressure is detected during this time, the thread preempts any attempts to do a balloon inflation and will maintain the size of the balloon for a backoff period of about 10 intervals.

3.3 Hierarchical Copy Algorithms for Xen Live Migration

On Xen, The most important part of live migration is memory migration. Live migrating a VM basically consists of transferring its memory image from one host to another to allow workload balancing and the avoidance of downtime[xen].

In Xen, pre-copy transfers in the first interation all memory pages from the source host to the destination one. In the subsequent iteration, it only copies the dirtied pages in the previous iteration. The iterative copiy-approach of Xen is described below.

Xen uses 3 memory page tables to achieve the bitmap for the dirtied pages: to_skip, to_send and to_fix. They are summarized as followed.

- to_skip indicates he pages that will not be transfered in this iteration.
- to_send indicates the page that have been dirtied in the last iteration.
- to_fix indicates the the pages that have to be transferred in the last iteration(at the end of the whole migration).

For example, a page indicated by to_skip is ignored for transferring and a page indicated by to_send will be transferred in the next iteration, while the pages indicated by to_fix will be transferred at the end of the migration(downtime of pre-copy). But this approach is evaluated as wither not effective well for reducing downtime or migration time, or not suitable well for Xen Vms platforms beause of high dirtying rate between the end of the last iteration and the beginning of the next iteration[LQY+10]. In [LQY+10], the migration time of dirtied pages during migration phases will be recalculated by implementing a hierarchical copy algorithms. In this new approach, only to_send and to_fix from the traditional approach are reserved and two new data structures are introduced: tuple dirty_count and layer. The variable dirty_count[i] illustrates the number of modified times that the page i. Its range is from 0 to 255 with an initial value of 0. The variable[i] described more details of page migration behavior.

- layer[i]=0: page i is not dirty and need to be transferred in the next iteration.
- layer[i]=1: page i is without high dirty rate and will be transferred in the next iteration.
- layer[i]=2: page i is with high dirty rate and will not be transferred in the next ietration.

Moreover, three structures as follows for assistance:

- dirtypage_amount: indicates the total amount of dirty pages with initial value 0.
- total_dirtycount: indicates the modification times of dirty page with initial value 0.
- critical: the threshold of hierarchical layer.

total_dirtycount is asigned as below:

$$totaldirty\_count = \sum dirty\_count[i] \tag{1}$$

critical is a ratio of total_dirtycount and dirtypage_amount:

$$critical = \frac{total\_dirtycount}{dirtypage\_amount} \tag{2}$$

The tuple layer is assigned as below:

- `if dirty_count[i] == 0 then layer[i]=0;`

  `if dirty_count[i] < critical then layer[i]=1;`

- `if dirty_count[i] >= critical then layer[i]=2;`

In summary, in this approach only the pages that are not modified and with a modified rate under a specified threshold will be transferred, the pages that are too frequently modified will not be transferred in next iteration.

Then they did a performance analysis with unmodified Xen and hierarchical copy algorithms added Xen in ubuntu-8.10 with the image sizes of 128M, 256M, 512M and 1024M respectively.

TABLE I. MIGRATION OF HIGH DIRTY-PAGE RATE (XEN)

Memory size(MB)	Number of iterations	Downtime (ms)	Total transfer time (ms)
128	30	3970	36823
256	30	3712	51755
512	30	3763	73619
1024	30	3812	79638

Fig. 6: (a)

TABLE II. MIGRATION OF HIGH DIRTY-PAGE RATE (HCA)

Memory size(MB)	Number of iterations	Downtime (ms)	Total transfer time (ms)
128	30	3714	30165
256	30	3567	46214
512	26	3578	70539
1024	25	3371	73426

Fig. 7: (b)

Compare these two tables, we can see that the total transferred time of both Xen and HCA (Hierarchical Copy Algorithms) are increasing with an increase of the memory size. However, compared with Xen, the HCA shortens total migration time significantly. It shows that this approach has shorter downtime than Xen under high dirty-page rate environment. The reason is that the HCA algorithm could effectively control the times of migration iterations. As a result, under High dirtypage rate scenario, the total migration time and downtime are both decreased obviously compared with the traditional Xen scheme[LQY+10].

3.4 Dynamic Page Transfer Recording and Compression

The main problem of live migrating an operating system is that memory pages which are frequently updated are likely to be transferred many times, that leads to long migration time and waste of network bandwidth. An approach called dynamic page transfer reordering, can be used to solve this problem. A part of this approach is similar to last approach(Hierarchical Copy Algorithms) in which less frequently written pages are prioritized over frequently updated ones. In this approach a page weight is for each page based on the number of times a page has been modified during the migration process and all pages are transferred in the order of page weight. A compression approach is also used in order to save more bandwidth. The compression approach consists of two parts: the page-cache which stores the previous versions of pages and the RLE(Run-Length Encoding) compression algorithms[run] which is particularly suitable in the case of transferring the pages consist of sequences of zeros and ones.

To implement the dynamic pages transfer reordering, the standar live migration algorithms of KVM is modified to transfer the pages in the order of pagfe weight. A variable holds the CPW(current page weight) and increases every time the pages scan pointer move from the lower address to a higher address. Only the pages whose page weight is equal to the CPW are considerred to be transferred during each scan. After each iteration, the CPW is reset to be zero and the page weight of each page will not be changed. Pages are stored in a 2-way set associative cache[Dre]. If a previous version of page exists in the cache, a delta page is created by applying a binary XOR operation between the old and new version of this page. This delta page is then compressed with the RLE algorithms before

it is sent instead of the full page is sent.

To evaluate the performance of this algorithms, a test of live migration with different algorithms and working set(WS) sizes has been implemented. Figure 6 shows the result of the experiment.

LIVE MIGRATION DOWNTIME.

WS Size	64 MB	128 MB	256 MB	512 MB	1024 MB
Vanilla	1 s	1.6 s	3 s	5.8 s	9.1 s
XBRLE	0.05 s	0.8 s	0.1 s	0.2 s	0.35 s
PRIO	0.1 s	0.2 s	0.26 s	0.3 s	0.36 s

(a)

MIGRATION TIME/TRANSMITTED DATA.

VM Size	512 MB	1024 MB
XBRLE	32 s / 657 MB	52 s / 1316 MB
PRIO	20 s / 404 MB	36 s / 972 MB

(b)

PAGE RESENDS FOR 1024 MB WORKING SET SIZE.

# Resends	1	2	3	4	5	6
XBRLE	528k	265k	262k	94k	23k	4
PRIO	528k	225k	48k	3	1	0

(c)

Fig. 8: **Vanilla**: the standar algorithms in KVM. **XBRLE**: a algorithms in which only XOR binary run length encoding is implemented. **PRIOR**: the approach drescbed above.

We can that the PRIO algorithm outperforms the Vanilla and XBRLE algorithms in terms of transferred data and migration time while obtaining the same low migration downtime as the XBRLE algorithm.

4 Conclusion

References

BDF+03. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

CFH+05. Christopher Clark, Keir Fraser, Steven Hand, Jacob˜Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration

of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

Dre. Ulrich Drepper. Memory part 2: Cpu caches. `http://lwn.net/Articles/252125/`. 2007.

Hab. Irfan Habib. Virtualization with kvm. `http://dl.acm.org/citation.cfm?id=1344217`. 2008.

HDG09. Michael~R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

HG07. Michael~R Hines and Kartik Gopalan. Memx: supporting large memory workloads in xen virtual machines. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, page~2. ACM, 2007.

IHIR11. Khaled~Z Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page~40. ACM, 2011.

LQY+10. Zhaobin Liu, Wenyu Qu, Tao Yan, Haitao Li, and Keqiu Li. Hierarchical copy algorithm for xen live migration. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2010 International Conference on*, pages 361–364. IEEE, 2010.

run. run-length encoding. `http://en.wikipedia.org/wiki/Run-length_encoding`. Accessed: 2014-06-02.

xen. Xen - wikipedia. `http://en.wikipedia.org/wiki/Xen`. Accessed: 2014-06-02.