HSG-LM: Hybrid-Copy Speculative Guest OS Live Migration without Hypervisor

Peng Lu, Antonio Barbalace, Binoy Ravindran

Department of Electrical and Computer
Engineering, Virginia Tech, USA
lvpeng, antoniob, binoy@vt.edu

Abstract

Current Virtual Machine (VM) live migration mechanisms only focus on providing a satisfying user experience by offering minimal downtime. In this paper, we present a novel live migration technique called HSG-LM, which also aims to provide a satisfying user experience to whoever is responsible for triggering the VM migration (e.g., the data center administrator). HSG-LM is implemented in the guest OS kernel in order to not rely on the hypervisor throughout the entire migration process. HSG-LM exploits a hybrid strategy that reaps the benefits of both pre-copy and post-copy mechanisms. Furthermore, HSG-LM integrates a speculation mechanism that improves the efficiency of handling post-copy page faults. From our evaluation on different real-world workloads (Sysbench, Apache, etc.), the results show that HSG-LM incurs minimal downtime as well as short total migration time. Moreover, compared with competitors, HSG-LM reduces the downtime by up to 55%, and reduces the total migration time by up to 27%.

Categories and Subject Descriptors   D.4 [*OPERATING SYSTEMS*]: Organization and Design

General Terms   Design, Measurement, Performance

Keywords   virtual machine; hypervisor; migration

1.   Introduction

Transferring the running application between two host machines by process migration has been thoroughly studied [6, 10, 15, 19]. However, in the real world, it is complex to implement an efficient native process migration technique because the running task is intimately bound to the hosting OS (e.g., file descriptors, sockets) as well as the running platform (e.g., device drivers, native compilation). Moreover, considering the shared memory case, as each process may interact with several other running processes, solo process migration is not possible unless the involved processes are completely independent. Otherwise, a special group of processes need to be migrated together [18]. Besides all these process-level migration mechanisms, a system-level strategy that migrates all the running applications together as well as their hosting OS is presented. Now the system-level migration attracts more attention, especially through the virtual machine (VM) technology.

A virtual machine monitor (VMM) mediates all interactions between hardware and software, encapsulates the state of the running VMs, and isolates concurrently running OSes and applications. VMs are increasingly being used by data centers to provide greater flexibility and higher resource utilization, as well as bring more security through isolated environments. A key feature of VM is migration. In traditional stop-and-copy migration, the VM needs to be fully stopped on the source host before migration; its state is then transferred to the target host and resumed there. However, this migration leads to unacceptable downtime for users. The downtime is defined as the waiting time between when the VM is unusable by the users (stopped on the source) and when the VM is working again (resumed on the target).

An alternative is live migration, which refers to migrating VM to the target host while the VM is still running on the source host. The hypervisor (or VMM) is used to manage the entire migration process and to ensure minimal downtime to users. In live migration, the downtime is so short as to be almost unobservable for users (less than 1 second, usually tens of milliseconds), so it does not jeopardize users' interaction. Most state-of-the-art/practice VMMs (e.g., Xen [7], VMware [25], KVM [3]) provide such a mechanism today. VM live migration is also gaining increasing traction in the "Platform as a Service" (PaaS; e.g., Google App Engine [2]) and "Infrastructure as a Service" (IaaS; e.g., Amazon's AWS/EC2 [1]) paradigms in the cloud computing.

However, this hypervisor-based migration method has disadvantages that have been well-studied in the literature during recent years [14, 24]. The main problems are due to the central role that the hypervisor plays throughout the entire migration process. The hypervisor needs to take care of most of the management issues from the beginning to the end of the migration. The central role of the hypervisor is in fact the source of several major security concerns. Szefer *et. al.* [24] show that if the hypervisor is attacked successfully by a malicious party, the attacker can easily inspect the memory, expose confidential information, or even modify the running software in the VM.

In an attempt to resolve these issues, in this paper, we present a live migration mechanism that does not rely on the hypervisor once the migration has been started. Besides addressing the security issues, our research was targeted at finding solutions to use-case scenarios that arise from common VM practice. One representative scenario is VM migrations in a data center that offers an isolated VM for each user. The administrator creates VM for each user with its own fully-customized environment, in order to run its own set of applications. In this scenario, there are several reasons why the administrator might decide to migrate the VMs, such as for load balancing and fault-tolerance purposes. Then in that case, what are the metrics that an administrator should consider in order to provide high availability service?

The answer to this question has driven our research effort in a top-down approach to the solution. Providing the high availability service turns out to be equivalent to providing the best user experience. The administrator does not want to cause the customers to be dissatisfied, or they may leave and choose another provider. Therefore, the administrator has to assure that during VM migration, the downtime perceived by the user, i.e. the time while the user can not access his or her environment, is as short as possible. Today, most hypervisor-based migration mechanisms provide this service with minimal downtime.

In addition to user experience, an administrator also has to consider the total waiting time to the completion of each migration, as it affects the release of resources on both source and target host [12, 13]. For example, for some purposes such as load balancing, hardware upgrade, etc, the administrator needs to shut down a running host. In this case, it needs to migrate all the active VMs to other hosts and wait until the whole migration has ended. From the time the migration starts until the time the VM has been resumed on the target machine, the administrator is not able to release any resource in the VM, and has to wait until the entire migration is completed. Therefore, considering the administrator's patience and its willingness to respect the scheduled maintenance operations, the total migration time is also very important.

As opposed to other works that only consider the user experience by reducing the downtime, in this paper, we introduce a new migration strategy, hybrid and speculative guest OS live migration (or HSG-LM). HSG-LM targets a satisfying experience for both users and administrators by maintaining both minimal downtime and short total migration time. Our contributions include:

1) We implement the migration mechanism inside the guest OS, so migration no longer relies on the hypervisor throughout the entire migration;

2) We design and implement a hybrid-copy migration mechanism, and investigate the pros and cons of the original HSG-LM design;

3) To make improvements, we integrate a speculative strategy into the HSG-LM implementation, and make the final design adaptive to different kinds of workload. It further reduces the downtime and total migration time;

4) We test on different workloads including Sysbench [4], Apache [5], etc. Our results show that HSG-LM incurs minimal downtime and short total migration time. Compared with competitors, HSG-LM reduces the downtime by up to 55%, and reduces the total migration time by up to 27%.

The remainder of the paper is organized as follows. Section 2 discusses past and related work. Section 3 presents the design of HSG-LM as well as the key methodology applied in HSG-LM. Section 4 presents the optimization of HSG-LM by introducing the speculative migration and the workload adaptive design. Section 5 reports our experimental environment, benchmarks, and evaluation results. We conclude in Section 6.

2. Related Work

There are several non-live approaches to VM migration. Internet suspend/resume [22] focuses on saving/restoring computing state on anonymous hardware. Sapuntzakis *et. al.* [21] address user mobility and system administration by encapsulating the computing environment into capsules to be transferred between distinct hosts. Schmidt *et. al.* [23] apply capsules, which are groups of related processes along with their IPC/network states, as the migration units. Similarly, Zap [18] uses process groups (pods) along with their kernel state as migration units. In all these solutions, execution is suspended and the applications within each VM do not make any progress.

Currently, there exist many virtualization-based live migration techniques. Two representative live migration systems are Xen live migration [8] and VMware VMotion [16], which adopt similar pre-copy migration strategies. Remus [9] uses periodic high-frequency checkpointing to handle hardware fail-stop failures on a single host with whole-system migration. It does this by maintaining a completely up-to-date copy of a running VM on the backup server, which automatically activates if the primary server fails. However, Remus incurs a large performance overhead (the performance penalty reported in [9] is up to 50%).

Multi-VM migration mechanisms have also been studied, including VirtCFT [26] and VCCP [17]. VirtCFT provides fault-tolerance for virtual clusters by checkpointing individual VMs to additional backup hosts. VirtCFT adopts a two-phase commit coordinated-blocking algorithm as the global checkpointing algorithm. A checkpoint coordinator broadcasts checkpointing requests to all VMs and waits for two-phase acknowledgements. However, as VirtCFT uses a checkpoint coordinator that communicates (several times) with each VM during a checkpoint period, the downtime is increased due to additional communication delays. VCCP also relies on reliable FIFO transmission to implement a blocking coordinated checkpointing algorithm. Due to its coordination algorithm, VCCP suffers from the overheads of capturing in-transit Ethernet frames and VM coordination before checkpointing.

Besides the pre-copy mechanism, there are also other related works that focus on migration optimization [20, 27]. Post-copy based migration [12, 13] is proposed to address the drawbacks of pre-copy based migration. The experimental evaluation in [12] shows that the migration time using the post-copy method is shorter than the pre-copy method. However, its implementation only supports para-virtualized (PV) guests as the mechanism for trapping memory accesses and utilizes an in-memory pseudo-paging device in the guest. Since the post-copy mechanism requires a modified/patched guest OS, it is not as widely used as the pre-copy mechanism. Hines *et. al.* [12] also introduce the idea to combine pre-copy and post-copy mechanism together but did not discuss the details because it's out of the scope of the paper. Our approach is different from [12]. They propose an adaptive pre-paging method, which keeps track of the access pattern of the application. Although we also apply a kind of pre-paging method, we speculate on locality and highlight how this leads to a series of performance advantages for live migration in Section 5.

3. Design and implementation of HSG-LM

3.1 Migration without hypervisor

In traditional live migration, the hypervisor is responsible for keeping track of the dirty memory pages, as well as creating a new VM on the target host and coordinating the transfer of the VM state between the two hosts. However, one limitation comes from the fact that the migration process totally relies on the hypervisor throughout the entire migration phase. Specifically, in order to track the dirty memory pages, the hypervisor needs to record the dirty map, which is kept inside the host OS, translate each address, find the corresponding pages, and make the transfer. This hypervisor-central model brings a potential security problem: if the hypervisor is attacked successfully by a malicious party, the attacker can easily inspect the memory, expose confidential information, modify the running software in the guest VM or even transfer the guest VM to a remote untrusted site. The

vulnerability of the hypervisor has been well-studied in recent years [14, 24] and we note that the investigation of this problem is not our contribution. However, the potential security problems caused by the hypervisor inspired us to design a mechanism that does not involve the hypervisor during live migration.

An alternative is to perform live migration inside the guest OS. The migration mechanism can still apply the pre-copy procedure (like [11] did in their work) as well as others, and all the tracking and transferring work of memory pages will be done by the guest OS itself, not by the hypervisor. However, this approach is complex to implement in existing operating system software. Difficulties arise in transferring all the consistent states from the running guest OS. Inside a running guest OS, the memory pages can be grouped into three categories while migrating:

1) user-space memory pages;
2) kernel-space memory pages;
3) migration-dependent memory pages (that hold the migration data structures).

For the last type of memory, because the guest OS is still running in order to transfer its final state, it generates new dirty memory pages through the migration mechanism itself. Therefore, the pages that are used by the guest OS to track and transfer other dirty pages are impossible to freeze and migrate. Hansen *et. al.* [11] solve this problem by partitioning the final migration epoch into two stages: the pre-final stage and last stage. In the pre-final stage, they created a shadow buffer that is updated with the current dirty pages, and in the last stage, only the data in the shadow buffer is transferred to the target host by the hypervisor. They verified that a consistent final view of the guest VM may be achieved by performing this resend-on-write followed by a copy-on-write method. However, the new shadow buffer still creates new memory overhead. Moreover, the creation and other operations of the buffer are within the final epoch, which means that these operations increase the downtime as well. Instead of re-implementing this mechanism, and in order to avoid any further overheads, we apply a hybrid-copy migration methodology. After the necessary state of the VM is transferred to the target and the VM is resumed there, the memory pages of the source VM are always transferred per request from the target VM, and only the newest updated copy is fetched and copied during the migration. We present the details of our hybrid-copy in Section 3.2.

We set up a new page fault handler in the guest OS to track the dirty memory inside the OS itself (on the target host) and subsequently generate transfers. Therefore, the migration does not rely on the hypervisor to manage the memory pages anymore. For the new running VM, after resuming on the target host, all memory pages are set as read-only. Thus, if there is any write to a page, it will trigger a page fault. The page fault is then reported to the new page fault handler, and we log the change of this page to the dirty
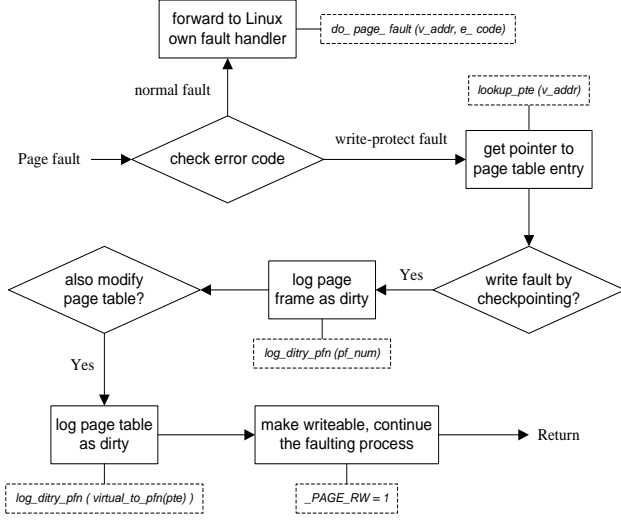
Figure 1: Workflow of the new page fault handler.

bitmap kept by the guest OS. The set of dirty pages can be implemented by using different data structures; here we choose a bit vector because it is easy to estimate the space requirements and memory overhead is minimal.

Note that when a page fault occurs, this memory page is set as writeable, but the page fault handler does not save the modified page immediately, because there may be another new write to the same page during the same interval. Instead, HSG-LM records the address of the faulting page in the dirty bitmap and removes the write protection from the page so that the application can proceed with the write. At the end of each migration, the dirty bitmap contains the address translation info for all the pages that were modified. The workflow of the new page fault handler is shown in Figure 1.

There is another advantage to installing a new page fault handler inside the guest OS. In traditional hypervisor-based migration, the hypervisor does not know what really occurs with the applications running because the isolation feature of the VM prevents all outside access. Therefore, compared with the page fault trap mechanism implemented in the hypervisor, our new handler is able to communicate with both kernel and user space, so as to make a more accurate speculative migration. We discuss the speculative migration in Section 4.

3.2 Hybrid-copy migration

In the Introduction we report a significant use case that highlights two waiting times, both drawn in Figure 2, that are incurred during live migration: downtime and total migration time. As stated there, the downtime reflects the customer satisfaction but the total migration time matters as well, e.g., when dealing with operations management of a data center. Traditional VMM implementations [8, 16] exploits the pre-
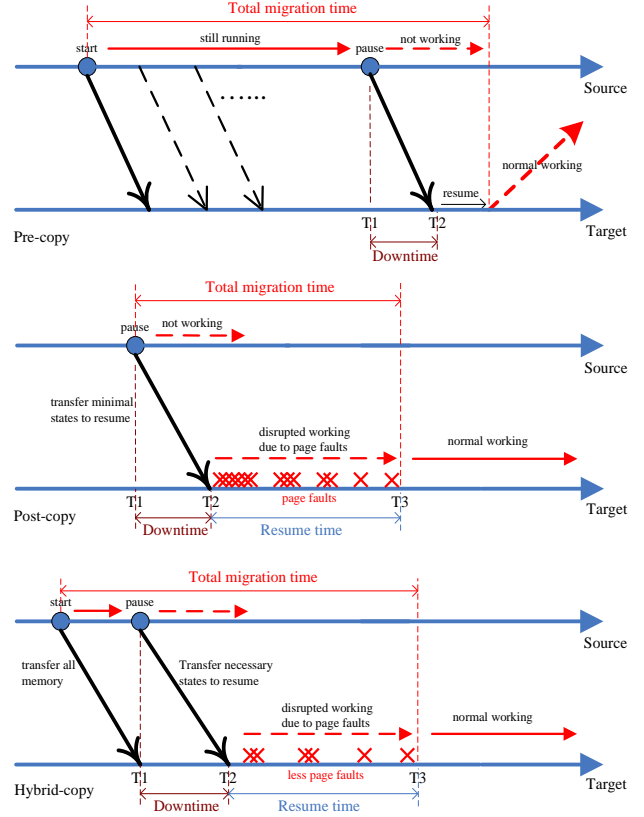
Figure 2: Downtime and total migration time measurements.

copy technique, which is organized in epochs and works as follows:

1) In the first migration epoch, the hypervisor on the source host starts a host thread that pre-copies all of the VM's memory pages to the target host while the VM is running.

2) At the end of each subsequent migration epoch, the hypervisor checks the dirty bitmap to determine which memory pages have been updated in this epoch. The hypervisor transfers the newly updated pages only. Meanwhile, the VM continues to run on source host.

3) When the pre-copy phase is no longer beneficial, the hypervisor suspends the VM on the source host, transfers the remaining data (newly updated pages, CPU register and device states, etc.) to the target host, and prepares to resume the VM there.

The traditional pre-copy migration mechanisms work well regarding the downtime measurement, which is usually less than 1 second. However, the downtime takes only a small fraction of the total migration time. From the pre-copy workflow presented above, we observe that if a memory page is frequently updated by some memory-intensive workload, its updated copy will be transferred every time during each migration epoch. Considering that a normal VM may be as-

signed up to several gigabytes of memory to run the guest OS, this will bring a consistent overhead throughout the entire migration, which leads to unacceptable total migration times. We show this in our evaluation of the original pre-copy mechanism in Section 5.3.

Instead of pre-copy, the post-copy migration mechanism can reduce the total migration time. Hines *et. al.* [12] propose a basic post-copy framework via demand paging as follows:

1) During post-copy migration, the guest VM is first suspended on the source host until a minimal and necessary execution state (or checkpoint) of the VM (including CPU state, registers, and some non-pageable data structures in memory) has been transferred to the target.

2) Although the entire memory data is still on the source host and no memory pages have been transferred to the target, the VM is still resumed on the target host.

3) Whenever the resumed VM tries to access a memory page that has not been fetched, a page fault will be generated and redirected towards the source over the network, referred to as a network fault.

4) The source host will respond to these network faults by fetching the corresponding memory pages and transferring them to the target host.

In the pre-copy migration, the VM on the source host handles user requests throughout the entire migration process. As opposed to it, the post-copy migration mechanism delegates the user services' response to the VM on the target host. We present the downtime and total migration time for post-copy migration in Figure 2 as well. Both downtime and total migration time measurements are the same as that under pre-copy migration. However, with post-copy migration, when the VM starts to resume after a short downtime, it is actually unusable for users for a period of initial time because of the occurrence of too many page faults. This time period is shown as "resume time". In our evaluation, it is still counted in the downtime measurement because during the majority of the resume time, the users' normal activity is impaired.

From Figure 2 and the above discussion, we observe that both pre-copy and post-copy migration mechanisms have their own benefits and limitations: short downtime but long total migration time in pre-copy migration, the contrary for post-copy migration, which leads to acceptable total migration times but incurs longer downtime (including resume time). The difference is shown in our evaluation of both original mechanisms in Section 5. To reap the benefits of both pre- and post-copy mechanisms and to overcome their limitations, our HSG-LM is the first that implements a hybrid-copy method (introduced in [12] as future work but there is no track of such work).

Our hybrid-copy migration performs a single round of classic pre-copy transfer at the beginning, that is, all the memory pages are copied from the source host to the target host while the VM is still running on the source. Then, following the post-copy approach, the VM is paused on the source host, a minimal set of necessary execution states is transferred to the target, and the VM is resumed there. As it performs a pre-copy round first, this hybrid mechanism eliminates a great number of the page faults that usually occur in the following post-copy round. Especially when running read-intensive workloads, HSG-LM incurs minimal downtime, as with the original pre-copy migration. On the other hand, HSG-LM also supports deterministic total migration time as with the original post-copy migration, especially for the write-intensive workloads, because it ensures that a memory page is transferred at most twice during the entire migration (all the numbers are further discussed in Section 5).

4. Speculative migration

4.1 Pros and cons of the hybrid design

In our previous design, after the first pre-copy round, we let the VM resume on the target host after loading only the necessary CPU and device states. Then we restore the updated memory data that was modified by the running VM during the first pre-copy round. Whenever the VM needs to access a memory page which has not been updated, it retrieves the corresponding data from the source host and sets up the pages locally. The benefit of this solution is that the VM starts very quickly, and it always keeps running while restoring the memory data. Moreover, since the VM only needs to restore a small number of CPU and device states in order to start, its performance would not be reduced by large VM memory sizes.

Compared with pre-copy migration, we note that the hybrid approach has some disadvantages. With the pre-copy approach, after the VM starts, the VM works as well as it did before the migration occurred. However, with the hybrid-copy approach, the VM appears to be running almost immediately after restoring the basic CPU and device states, and with workloads with few writes, the penalty induced by page faults that must go out over the network is relatively small. However, write-intensive workloads may modify the memory pages with high frequency so that after the first round, most memory pages received by the target host will need to be updated again, so much of the pre-cached memory data will not actually be usable. This will cause a significant number of network page faults to occur at the beginning, degrading VM performance. Note that the same problem happens in post-copy mechanism too, with an even longer resume time. Our experiments (Figure 6) show that using post-copy migration, for a guest VM with 1GB RAM, during the first 14 seconds, the VM runs too slowly to be useful. Almost all of this time was spent on restoring the updated memory data.

To reduce the total migration time, our goal is to run the VM on the target host as early as possible, but to avoid the performance degradation caused by page faults after the VM

Figure 3: Linux kernel data structures involved in task's virtual to physical memory translation and inverse mapping.

starts. Our approach is to first resume the VM by loading the necessary CPU/device states, and then when a page fault occurs, to determine the memory pages that have a high probability of being accessed in the near future and to restore these pages from the source host. In other words, for each page fault that occurs, we preload several additional memory pages. By doing so, we reduce the possibility of future page faults and thus reduce the overhead to handle them.

We determine the likely-to-be-accessed memory pages by the principle of spatial locality on virtual address: if a page is updated, its neighboring pages are likely to be updated in the near future. We can rely on our knowledge of a page's address to predict the upcoming memory accesses.

4.2 Speculation: choose the likely to be accessed pages

Our HSG-LM runs mostly in the guest OS: it is ready to run in a fully-virtualized environment, and we can directly exploit the kernel data structures of the virtualized guest OS. Knowing the data structure of the guest OS has the big advantage that for each page used by the operating system, we can easily determine which processes are using it. For each process, we can identify the neighboring pages to the faulting one. On top of that, we know the tasks that are currently running on the guest OS.

With these considerations in mind, and the fact that the bitmap of faulty pages is by design maintained in the guest OS, we developed a set of speculative methods aimed at improving the performance of our HSG-LM. The basic idea behind our techniques is to transfer not one page per dirty fault, but a bulk of pages (on-demand pre-paging), in order to decrease the number of network transfers between the source and the target host, shortening the waiting times in the migration. Also, in HPC, the network latencies are still an order of magnitude greater than for a multiprocessor interconnect, so reducing the number of network transfers will increase performance. To improve the user experience, we propose that the pages to be transferred cannot be just chosen randomly or like in [12] relying on the previous access pattern, from the dirty bitmap. Instead, we choose to trans-

fer first the pages related to tasks that the user is interacting with that are not necessarily related to the previous access pattern. This turns out not to be straightforward but instead to require knowledge of the Linux kernel internals (task's memory related data structures) that we briefly summarize in the following for completeness. Figure 3 provides an illustrative sketch.

The Linux kernel, like every other operating system that runs on virtual memory, manages a per-task page table structure. The page table structures allow the processor's MMU to translate the addresses from virtual to physical. In Linux every hardware memory page has a corresponding `struct page` used by the system to "keeps track" of its usage and Linux implements `rmap` (reverse mapping) that makes use of the arrays of `struct page` to translate back from physical to virtual addresses. The Linux kernel maintains for each thread a task control structure (`struct task_struct`). For each process, there are mainly two types of pages that are interesting for our research: anonymous pages and file pages. File pages are memory pages that contain an entire file or part of a file that is normally resident on the hard drive. This file can be either an executable or library, or a data file. Anonymous pages contain the stack and the heap. From the `mm` field in a `struct task_struct`, it is possible to access the process' memory descriptor (`struct mm_struct`) that in the field `mmap` contains a linked list of virtual memory area descriptors (`vm_area_struct`). Each virtual memory area refers to a block of anonymous pages or a block of file pages.

We identify the following speculation techniques:

1) randomly choose a group of r linearly contiguous pages around the faulty page from the dirty bitmap, where r is less than a customizable threshold (RANDOM);

2) bulk copy all the dirty pages that belong to a memory-mapped file; a threshold can be specified to bound the number of pages per transfer (FILE);

3) bulk copy all the dirty pages that belong to an anonymous memory mapping (a single task VMA) (ANON);

4) same as the 3) but involves different VMAs; a threshold can again be specified (ANON_MULTI);

In our preliminary experiments, however, we found that a general VM live migration mechanism does not work well in all cases, for example, when dealing with memory read intensive workload, integrating speculation in the migration actually incurs longer downtime due to the overhead to locate the related pages. Therefore, we finally implement a workload adaptive migration mechanism which could apply different migration methods based on the types of the workload. For such workloads which modify the memory in high frequency, the HSG-LM is triggered with speculation framework. Otherwise, the speculation is disabled in the HSG-LM and it will apply the hybrid-copy approach only. We evaluate the workload adaptive design in Section 5.5

5. Evaluation and Results

5.1 Experimental Environment

Our preliminary experiments were conducted on a set of identical machines equipped with an IA-32 architecture processor (Intel Core 2 Duo Processor E6320, 1.86 GHz) and 4GB of RAM. We set up a 1Gbps network connection between the hosts and shared the file systems among all the machines in this LAN. We built Xen 3.4.0 and ran a modified guest VM with Linux kernel 2.6.18. The guest OS and the host OS (in Domain 0) ran CentOS Linux, with a minimum of services initially executing in the guest OS, e.g., `sshd`. Domain 0 has 1.5 GB of memory allocated, and the remaining memory was left free to be allocated for guest VMs. To ensure that our experiments are statistically significant, each data point is averaged from twenty samples. The standard deviation computed from the samples is less than 3.4% from the mean value.

We refer to our proposed migration design with speculation mechanism as HSG-LM. To evaluate the benefits of the speculative migration, we compare HSG-LM with our initial migration design without speculation, which we refer to as HSG-LM-ns. All our HSG-LM implementations are publicly available at `http://www.ssrg.ece.vt.edu/resilire`. Our competitors include different checkpointing mechanisms including the original design of both pre-copy and post-copy migration mechanisms. To make a more clear comparison, we also implement an improved post-copy migration mechanism (post-copy-i) by integrating the pre-paging idea presented in [12]. In their paper, the pre-paging component is provided that the VM's pages are actively copied from the source to the target, instead of waiting for a fault to occur. They claim this could avoid most faults so as to reduce the waiting time. Furthermore, we compare all these results with a self-migration mechanism [11] which triggers the migration from the guest OS itself by following the similar pre-copy method. As opposed to our design, the self-migration mechanism follows the pre-copy strategy. We choose self-migration as another representative of pre-copy

migration. While the original pre-copy implementation is included in Xen, the other competitors are not publicly available. Thus, we implemented prototypes for each mechanism and used them in our evaluation.

5.2 Downtime

We evaluate the performance for migrating VMs running two types of memory-intensive workloads: read-intensive and write-intensive memory operations. We use the Sysbench [4] online transaction processing benchmark, which consists of a table containing up to 4 million entries. We perform either read or write transactions on the Sysbench database to evaluate the performance of all the migration mechanisms. The experiment is conducted on the guest VM with assigned memory from 128 MB to 1GB, in order to investigate the impact of memory size on downtime, total migration time and other performance characteristics. Although this range of memory may seem moderate given the high quantity of RAM available in today's data centers, we found it to be reasonable for understanding the trends.

We first consider the measurement of the VM downtime. The definition of downtime in the Introduction works well under the pre-copy migration mechanism because after VM resumption, the user could resume normal activity immediately. However, when using the post-copy strategy, after the VM has been resumed on the target host, it is actually unusable by the users for an initial period of time due to excessive page faults, so the resume time should be counted into the downtime measurement as well. When measuring the downtime in our experiment, we start measuring the elapsed time when the VM is stopped on the source host, and stop when the resumed VM is fully usable by the users. Figure 4a shows the downtime results for the Sysbench-read benchmark for six migration mechanisms with four different sizes of assigned guest memory.

We can make observations regarding the downtime measurements. First, the downtime results of the pre-copy migration mechanisms (including self-migration) are short, while the original post-copy migration incurs the longest downtime. This is because memory updates are rare during the Sysbench-read workload runs; there are relatively few dirty memory pages left in its final migration epoch. On the other hand, by using post-copy migration, there are no memory pages restored in the resumed VM, so thousands of page faults occur, and the resumed VM needs to go back to the source host to fetch the corresponding pages, leaving the VM unusable and leading to a much longer downtime. The improved post-copy mechanism could significantly reduce the downtime by fetching the pages actively instead of on-demand, but this additional process does introduce new overhead, which still leads to longer downtime than pre-copy mechanism.

Second, the downtime results under the HSG-LM design (including HSG-LM-ns) are very similar to those under pre-copy mechanisms. Sysbench-read is a memory read inten-

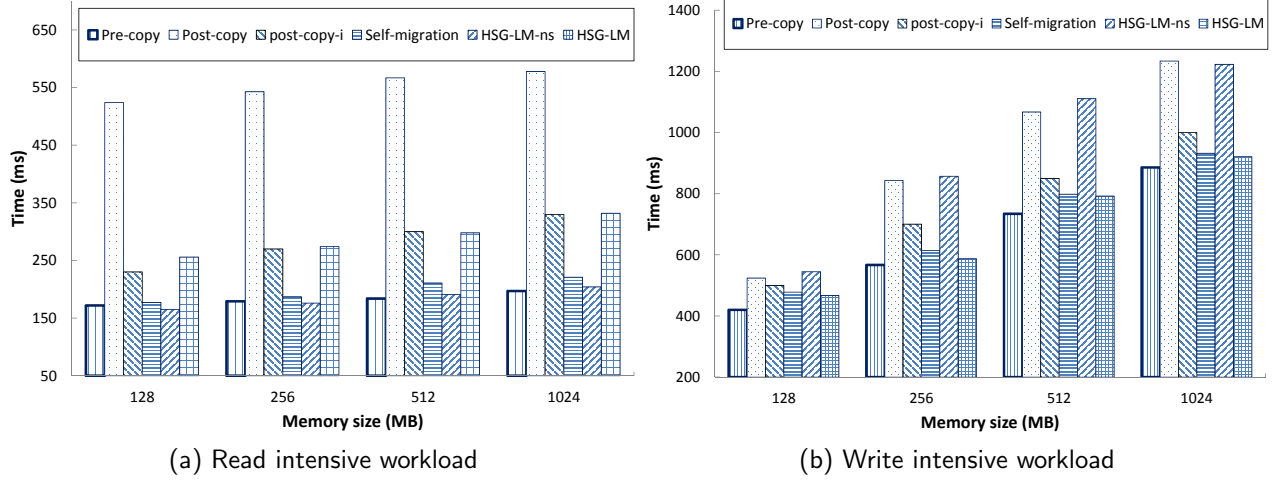(a) Read intensive workload

(b) Write intensive workload

Figure 4: Downtime comparison under different types of workloads.

sive workload. Thus, the guest VM memory is updated at a low frequency. Because we apply a hybrid strategy in the HSG-LM design, it first runs a pre-copy round to transfer all the memory pages and then follows the post-copy strategy to resume the VM on the target host. Memory updates are rare when running this workload, meaning that most memory pages are already restored in the resumed VM after the first round. Therefore, there would not be so many page faults in the HSG-LM mechanism compared with the original post-copy mechanism, and we do see a downtime reduction of roughly 55%.

Finally, compared with our original design without speculation (HSG-LM-ns), HSG-LM incurs longer downtime. As for both HSG-LM-ns and HSG-LM, the downtime depends on the time that is used to handle the page faults after the VM has been resumed on the target host. As the Sysbench-read benchmark only generates minimal dirty memory, page faults rarely occur, so spatial locality does not hold in this case. However, the HSG-LM mechanism will still perform speculative migration, finding neighboring pages of the faulted page and transferring them. This migration is unnecessary and increases the transfer time. Therefore, HSG-LM-ns incurs a smaller downtime than HSG-LM.

Figure 4b shows the downtime results in the same cases as Figure 4a, except running a memory write intensive workload (Sysbench-write benchmark) which updates the guest memory with high frequency. The findings are mostly similar to the previous ones about Figure 4a, but there are two contrary observations in Figure 4b:

The downtime results under the HSG-LM-ns design are very similar to the post-copy mechanism results, meaning that the downtime is not as good as that under the pre-copy migration mechanisms. As the guest VM memory is updated at high frequency, after the first pre-copy round in our original design, most of the memory pages restored in

the resumed VM have been updated again, so there would still be thousands of page faults and the pre-copy round would not do anything to help.

Another observation is that HSG-LM incurs a shorter downtime than HSG-LM-ns. Although in the HSG-LM mechanism the memory pages restored in the resumed VM need to be updated again after the pre-copy round, the hope is that the speculative migration will transfer the bulk of the required memory pages when one page fault occurs, reducing the frequency of network page faults as well as the total time to make the transfer. From Figure 4b, we observe that the HSG-LM mechanism achieves downtime performance comparable to that of pre-copy migration mechanisms.
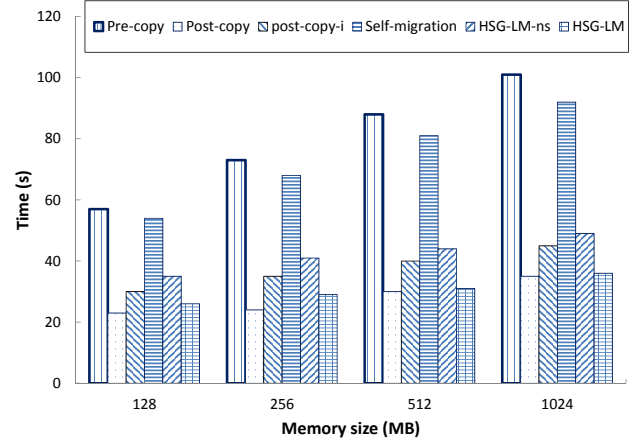
5.3 Total migration time

We also measured the total migration time, which is from when the migration is triggered to when the resumed VM is fully usable by users. Figure 5a shows the total migration time results when running the Sysbench-read workload. We firstly observe that the results of the pre-copy migration mechanisms (including self-migration) are longest, while the original post-copy migration mechanism incurs the shortest downtime. All migration mechanisms need to migrate all the memory pages at least once, either by pre-copying or by page fault handling. However, the pre-copy migration mechanism needs another two rounds, one to determine to stop the pre-copy migration, followed by the final round which transfers the updated memory (although rare) and the necessary CPU and device states. On the other hand, the post-copy migration only needs one round to fetch and transfer all the memory pages based on the page faults, so it achieves a better total migration time.

Second, the total migration time using the HSG-LM design (including HSG-LM-ns) is between that of the pre-copy and the post-copy mechanisms. Because we apply a hybrid
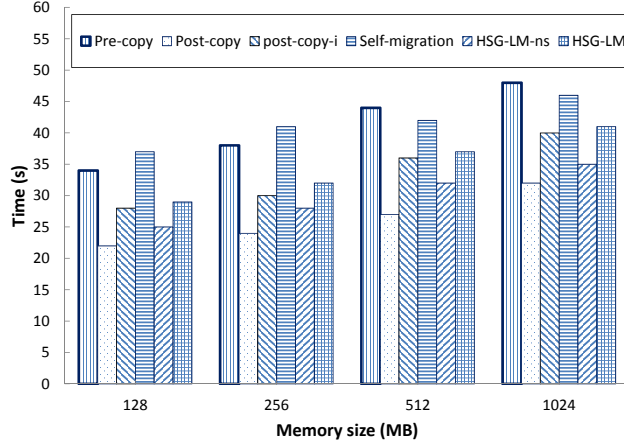
(a) Read intensive workload	(b) Write intensive workload

Figure 5: Total migration time comparison under different types of workloads.

strategy in the HSG-LM design, it first runs a pre-copy round to transfer all the memory pages and then follows the post-copy strategy to resume the VM on the target host. With this strategy, there are two migration-related rounds in total, and compared to the pre-copy migration mechanisms, the total migration time reduction is roughly 27%.

Finally, the comparison between HSG-LM and HSG-LM-ns is similar to that when measuring the downtime. The reason is also the same: since HSG-LM performs speculative migration, it finds neighboring pages of the faulted page and transfers them in bulk. This transfer is unnecessary and incurs longer transfer time. Therefore, as for the total migration time when running memory read intensive workloads, HSG-LM-ns performs better than HSG-LM.

Similarly, Figure 5b shows the total migration time results in the same case as Figure 5a, except running a memory write intensive workload (Sysbench-write benchmark) that updates the guest memory with high frequency. An obvious observation is that the post-copy migration mechanisms and our hybrid migration mechanisms perform much better than the pre-copy migration mechanism. This is because all the memory pages are only transferred once under post-copy migration mechanism, or at most twice under our hybrid migration. However, under pre-copy migration, if one memory page is frequently updated by the workload, it will be transferred in the corresponding round as long as it's marked as dirty. Therefore, a huge number of memory pages may be transferred several times in the pre-copy migration, leading to a long total migration time. Also, we could observe that HSG-LM performs better than HSG-LM-ns when running a memory write intensive workload, the same as when measuring the downtime. This again verifies that our speculative migration works well when the running application updates the memory in some way, which is more like practical workloads.
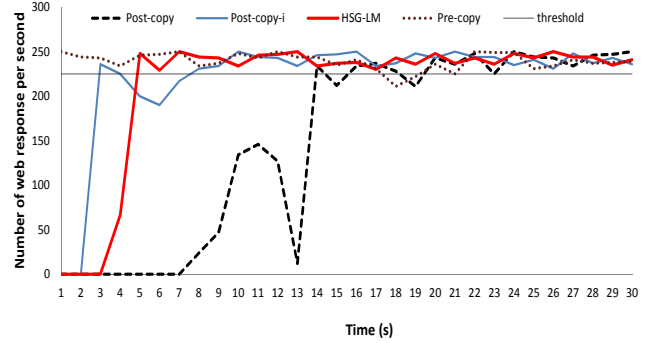

Figure 6: Performance degradation after VM resumes.

5.4 Performance degradation after resumption

Figure 6 shows the performance degradation after the VM starts. We configure the VM with 1GB RAM, and first run the Apache web server [5] on it. We set a client on another server that makes requests to load web pages, one immediately after another, simultaneously via a 1Gbps LAN, under four mechanisms (Pre-copy, Post-copy, post-copy-i, HSG-LM). Note that we did not include Self-migration evaluation in Figure 6, which exhibits very similar results as Pre-copy migration because it also follows the pre-copy approach during migration. Since the same load of web requests are processed in this experiment, the measured throughput (i.e., number of web responses received per second) can be compared for evaluating the impact of page faults on the VM performance during the initial period after the VM resumes. We set a threshold that all the throughput numbers under the threshold are recognized as performance degradation.

We observe that under pre-copy migration mechanism, the VM starts to work immediately after resuming, with no obvious performance degradation. However, the trade-off

Sysbench	100%Read, 0%Write	75%R, 25%W	50%R, 50%W	25%R, 75%W	0%R, 100%W
Downtime (ms)					
Pre-copy	179	331	564	765	866
Post-copy	522	604	789	979	1250
post-copy-i	313	444	647	891	1003
Self-migration	198	392	601	821	923
HSG-LM	**184**	**360**	**591**	**808**	**919**
Total migration time (s)					
Pre-copy	47	62.1	71.7	94,8	103.9
Post-copy	30.7	31.1	32.3	34.3	35.5
post-copy-i	40	41.5	44	46.6	49.2
Self-migration	45.1	58	69.9	88	92.8
HSG-LM	**34.5**	**34.8**	**35.4**	**36.1**	**37.2**

Table 1: Downtime and total migration time comparison under Sysbench with mixed operations.

with the pre-copy migration mechanism is that it takes a long total migration time (tens of seconds) to transfer the memory page, which is especially bad when running memory write intensive workload. On the other hand, with the post-copy migration mechanism, the VM starts the fastest, but it suffers performance degradation for a long time, i.e., it needs to wait for 14 seconds to resume its normal activity. The improved post-copy migration performs better, but it still suffers degradation due to the page faults at the beginning period. Compared with post-copy migration, under HSG-LM, the VM endures performance degradation for only about 4.5 seconds, which reduces the VM's unusable time by as much as 68%.

5.5 Workload adaptive evaluation

Our final experiment evaluates the effectiveness of our workload adaptive migration in Table 1. Using Sysbench [4], we set mixed transactions in the benchmark, e.g., 25% read transactions and 75% write transactions and understand the trends under different migration mechanisms. We still assign 1GB RAM for the guest VM. The key in our workload adaptive design is whether to trigger the speculation or not, when dealing with different types of workloads. In this evaluation, there are two extreme cases regarding the workload, one with totally read transactions and another with totally write transactions. We observe that in the downtime evaluation, the original pre-copy migration shows the best performance since it's designed to provide the minimal downtime. Nevertheless, HSG-LM always has the second shortest downtime in all cases, which means HSG-LM is an adaptive approach for both memory read and write intensive workloads, as well as the workload with mixed read and write transactions. We observe the similar result regarding the total migration downtime, i.e. the original post-copy migration leads to the best performance while HSG-LM is the second best in all cases.

6. Conclusion

We present a hybrid-copy live migration technique, HSG-LM, implemented in the guest OS, without hypervisor involvement in the migration. We evaluated the proposed migration technique with and without speculations on the pages likely to be accessed in the future and we found that this is a good compromise between the two pre- and post-copy mechanisms presented in previous works. Furthermore, HSG-LM is the only technique that provides in the average case short downtime and short total migration time. By speculating on the pages likely to be requested in the future and migrating them together in a single transfer, we improved our original design. Our work opens future research paths on how to adaptively select when to speculate and when not to speculate, as well as selecting which kind of speculation to use based on the application's workloads.

A potential disadvantage of HSG-LM is the lack of legacy support. The traditional hypervisor-based migration techniques have the benefit that they automatically provide migration for unmodified guest OS. Our new migration mechanism must be provided as an OS kernel primitive, since it is implemented as a new page fault handler inside the guest OS kernel. However, we believe that this downside will not obstruct acceptance from the VM community, since we believe — like suspend/resume — that HSG-LM could become a commonly-supported OS kernel feature.

References

[1] Amazon elastic compute cloud (amazon ec2). `http://aws.amazon.com/ec2/`.

[2] Google app engine - google code. `http://code.google.com/appengine/`.

[3] Kvm: Kernel based virtual machine. `www.redhat.com/f/pdf/rhev/DOC-KVM.pdf`.

[4] Sysbench benchmark. `http://sysbench.sourceforge.net`.

[5] The Apache HTTP Server Project. `http://httpd.apache.org/`.

[6] A. Barak and O. La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Gener. Comput. Syst.*, 13(4-5):361–372, Mar. 1998.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[8] 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*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[9] 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*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[10] F. Douglis and J. Ousterhout. Mobility. chapter Transparent process migration: design alternatives and the sprite implementation, pages 56–86. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[11] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.

[12] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09:Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[13] 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*, VTDC '11, pages 11–18, New York, NY, USA, 2011. ACM.

[14] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[15] I. Lyubashevskiy and V. Strumpen. Fault-tolerant file-i/o for portable checkpointing systems. *J. Supercomput.*, 16:69–92, May 2000.

[16] M. Nelson, B. H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[17] H. Ong, N. Saragol, K. Chanchio, and C. Leangsuksun. VCCP: A transparent, coordinated checkpointing system for virtualization-based cluster computing. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.

[18] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36:361–376, December 2002.

[19] D. Pei. Modification operation buffering: A low-overhead approach to checkpoint user files. In *IEEE 29th Symposium on Fault-Tolerant Computing*, pages 36–38, 1999.

[20] A. F. R. Bradford, E. Kotsovinos and H. Schioeberg. Live wide-area migration of virtual machines including local persistent state. In *VEE'07: Proceedings of the third International Conference on Virtual Execution Environments*, pages 169–179, San Diego, CA, USA, 2007. ACM Press.

[21] C. R. Sapuntzakis, C. and B. Pfaff. Optimizing the migration of virtual computers. In *Proceedings of OSDI*, New York, NY, USA, 2002.

[22] M. Satyanarayanan and B. Gilbert. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11(2):16–25, 2007.

[23] B. K. Schmidt. Supporting ubiquitous computing with stateless consoles and computation caches, 2000.

[24] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 401–412, New York, NY, USA, 2011. ACM.

[25] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[26] M. Zhang, H. Jin, X. Shi, and S. Wu. Virtcft: A transparent vm-level fault-tolerant system for virtual clusters. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 147 –154, dec. 2010.

[27] M. Zhao and R. J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *VTDC '07:Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, pages 5:1–5:8, New York, NY, USA, 2007. ACM.