

Agile Live Migration of Virtual Machines

Umesh Deshpande^a, Danny Chan^b, Ten-Young Guh^b, James Edouard^b, Kartik Gopalan^b, Nilton Bila^c

^aIBM Almaden Research Center, San Jose, CA, USA

^bComputer Science, Binghamton University, Binghamton, NY, USA

^cIBM T.J. Watson Research Center, Yorktown Heights, NY, USA

^audeshpa@us.ibm.com, ^b{dchan20, tguh3, jedouar1, kartik}@binghamton.edu, ^cnilton@us.ibm.com

Abstract—A key attraction of virtual machines (VMs) is live migration – the ability to move their execution state across physical machines even as the VMs continue to run. Unfortunately, the traditional pre-copy and post-copy techniques are not agile in the face of resource pressures at the source host, since it takes a long time to transfer the memory state of a VM. Consequently, the performance suffers for all VMs – those being migrated as well as those being left behind. Prior works have attempted to optimize indirect measures of migration effectiveness such as downtime, total migration time, and network overhead. However, none have treated the performance of VMs impacted by migration as the primary metric of migration effectiveness. We propose an *Agile live migration* technique that quickly recovers the performance of all VMs under resource pressure by eliminating resource pressure faster than traditional live migration. The working set of a VM is typically much smaller than its full memory footprint. Our approach works by transparently tracking the working set of each VM and offloading the non-working set (cold pages) in advance to portable per-VM swap devices. We present a new hybrid pre/post-copy technique that reduces the performance impact on the VM’s workload by transferring only the working set of the VM while enabling destination to remotely access cold pages from the per-VM swap device. We describe the challenges in the design and implementation of Agile live migration in the KVM/QEMU platform without modifying the guest OS in the VM. When live migrating under memory pressure, we demonstrate a reduction in the performance impact on VMs by a up to factor of 2, reduction in migration time by up to factor of 4 besides reduction in memory pressure on both the source and destination hosts.

I. INTRODUCTION

We address the problem of live migrating virtual machines (VMs) quickly in response to resource pressures while reducing the performance impact of migration. Virtualization allows datacenters to better utilize their hardware by consolidating many virtual machines (VMs) on fewer servers during non-peak hours. A VM’s active memory footprint, or working set size, is one of the factors that determines the extent of consolidation. Since less active VMs have small working sets, the consolidation server may swap out infrequently used pages. However, when the VM becomes active again and its working set size increases, retrieving swapped-out pages can degrade the VM’s performance.

To avoid a performance hit, some of the newly active VMs must be migrated to new physical nodes to ensure that

all VMs have enough memory, CPU, and I/O resources to restore their peak performance. Lengthy migration prolongs the resource pressure at the source, which degrades the performance of all VMs, whether being migrated or not.

Unfortunately, existing VM migration techniques such as pre-copy [1, 2] and post-copy [3] lack agility in responding to resource pressure. First, they require that *all* of the VM’s memory be transferred during migration, not just the working set. Secondly, any swapped out memory pages of the migrating VM need to be swapped back in before being transferred, which further increases the migration time. Thirdly, the migration tool itself (such as QEMU in KVM/QEMU or *xend* in Xen) may need to compete with VM’s applications for access to the swap device, which may increase thrashing and further slow down the migration. While some existing virtualization platforms provide the ability to configure per-VM swap devices whose contents need not be migrated with the VM [4], the resident memory of a VM may still contain significant amount of “cold” pages that do not belong to the working set; these cold pages are nevertheless transferred over the network.

In this paper, we present *Agile live migration* which can respond more quickly to resource pressure than traditional approaches. Agile migration is a hybrid pre/post-copy migration technique that eliminates the transfer of a VM’s cold pages during the migration. The key is to retain only the working set of a VM in memory during runtime, transfer only the working set pages at migration time, and make any cold pages remotely available to the destination on-demand. Agile migration is less sensitive to the nature of the workload than pre-copy, while it allows faster ramp up of the migrating VM’s performance at the destination than post-copy. Each VM’s cold pages are stored using a *per-VM remote swap device* which is managed by the host hypervisor, rather than the guest OS. The hypervisor *transparently* tracks the working set of each VM and adjusts each VM’s memory reservation without relying on guest agents and with minimal impact on VM’s performance. The per-VM swap device can be accessed from any host in the cluster, allowing the swap device to be portable as the VM migrates. In contrast, traditional swap devices managed by the hypervisor are either not portable or don’t store all cold

pages, whereas those managed by guest OS don't provide the sufficient control over host memory pressure.

II. BACKGROUND

In this section we review the traditional pre-copy and post-copy migration techniques and their impact on VM agility and performance. Figure 1 shows various components involved in live VM migration in both techniques. The Migration Manager is an external management process associated with each VM that initiates and carries out the migration. The Migration Manager at the source host establishes a TCP connection with the Migration Manager at the destination and transfers the VM's state consisting of virtual CPUs (VCPU), memory contents, and I/O device states. During the migration, any swapped-out pages are brought back into memory and transferred to the destination.

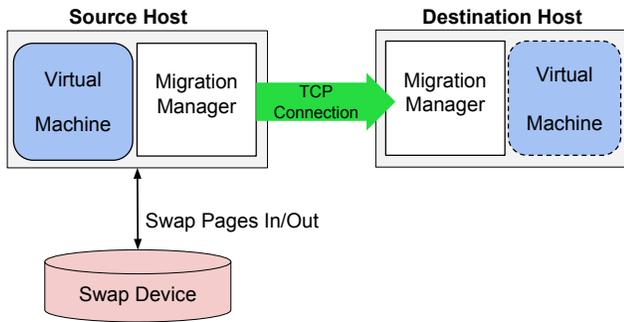


Figure 1. In traditional pre-copy and post-copy, the Migration Manager swaps in all pages swapped out from the source host, then sends them through a direct TCP connection to the destination.

Pre-copy Live VM Migration: In pre-copy live VM migration, the VM's memory is transferred from the source to the destination iteratively while the VM is still running at the source. In the first iteration, the VM's entire memory is transferred, whereas in subsequent iterations, only the pages modified during the preceding iteration are transferred. Upon converging on the VM's writable working set, the VM is suspended at the source and its writable working set and the execution state is transferred to the destination, where the VM resumes. The time during which the VM remains suspended is called its *downtime*, whereas the time taken to complete the entire migration is called its *total migration time*. Since dirtied pages are retransmitted, write-intensive applications increase the VM's total migration time and reduce pre-copy's agility.

Post-copy Live VM Migration: In post-copy, upon beginning the migration, the VM is immediately suspended at the source, its execution state is transferred to the destination, and the VM is resumed there. The VM's memory is both actively pushed from the source and demand paged from the destination. Post-copy has low network overhead since it transfers each page only once in contrast to pre-copy which retransmits dirtied pages. In theory, post-copy is very agile since the VM resumes execution at the destination

quickly after the migration starts. However, immediately upon resumption at the destination, the VM may experience a large performance degradation till all of its working set pages are either demand paged or actively pushed from the source.

III. DESIGN

The goal of Agile migration is to respond to resource pressures by migrating out one or more VMs so as to quickly restore the performance of all VMs – those being migrated as well as those left behind at the source.

Figure 2 illustrates the approach behind Agile migration as a hybrid of pre-copy and post-copy. Each VM is assigned a per-VM swap device which is managed by the hypervisor. This per-VM swap device can be accessed from either the source or the destination hypervisors using block device interface.

Agile migration begins by performing one live pre-copy iteration to transfer the the VM's working set (*hot pages*) and then switches the CPU execution state to the destination. Then onward, any pages dirtied by the VM are either actively pushed to or demand-paged from the destination. The pages residing in the swap device (*cold pages*) are not transferred during the migration. Instead they are demand-paged from the swap device at the destination. In the rest of this section, we describe the operation of Agile migration in greater detail.

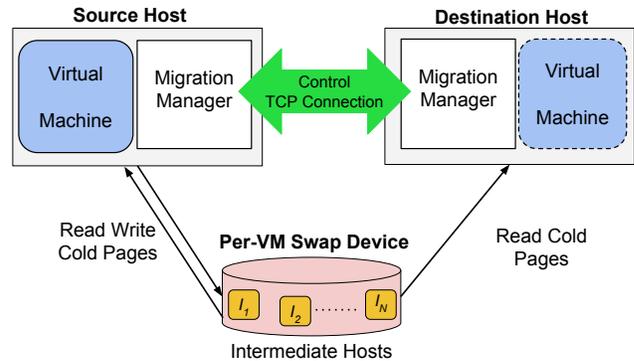


Figure 2. Agile VM migration: The Migration Managers transfer only the working set through the direct TCP connection. Pages in the per-VM swap device are demand-paged by the destination as needed.

1) *Pre-Copy Phase:* When migration begins, the source Migration Manager establishes a TCP connection with the destination Migration Manager. The source Migration Manager transfers the in-memory state of the VM to the destination while the VM still runs at the source. As the Migration Manager at the destination receives the pages over the connection, it copies them into the address space where the VM shall arrive. For each swapped out page, the Migration Manager at the source transfers only the page offset on the swap device. As the source Migration Manager transfers pages, the VM may still modify them, so the source Migration Manager records dirty pages in a dirty bitmap.

Upon completing one live pre-copy round, the Migration Manager at the source suspends the VM and transmits its CPU state and dirty bitmap to the destination.

2) *Post-Copy Phase*: At the destination, the Migration Manager resumes the VM. There are two concurrent mechanisms by which the destination receives the remaining dirtied pages from the source – *demand-paging* and *active push*.

Demand-paging: As the VM resumes at the destination, it faults upon the pages dirtied at the source during the pre-copy round and upon the cold pages on the per-VM swap. Upon receiving a page fault, the Migration Manager at the destination checks if it should request the page from the source by checking the page’s dirty bit in the dirty bitmap. Otherwise, the destination’s Migration Manager finds the swap offset for the required page and reads the page from the per-VM swap. After servicing the page fault, the VM resumes execution.

Active push: If we relied solely on demand-paging, then transferring all dirty pages from the source host would take an unbounded amount of time. Hence, to speed up the transfer of dirty pages, the source Migration Manager actively pushes the remaining pages to the destination besides responding to demand-paging requests. Once all the VM’s memory pages have been transferred, the Migration Manager at the source can terminate the VM and free up its memory.

A. Per-VM Swap Device

Our present design uses a *Virtualized Memory Device* (VMD) that uses remote memory for the per-VM swap devices. VMD is a distributed key-value store that maps page offsets to memory pages of other hosts. We create a VMD by aggregating the cluster-wide free memory of the intermediate hosts $I_1 \dots I_N$. The pages swapped out from the source host are transferred over the network to the intermediate hosts and stored in their memory. Any machine with spare memory in a cluster can contribute to this high-bandwidth low-latency distributed memory pool. Alternatively, a VMD can also use low-power external memory devices or networked storage as a backing store. Although we implemented our own VMD by modifying our prior MemX system [5, 6, 7], one could also adapt other distributed in-memory key-value stores such as Memcached [8] or Redis [9] for use with Agile VM migration.

B. Migration Trigger and VM Selection

We develop a tool to dynamically measure the working set size of each VM and to adjust the per-VM memory reservation to closely match the working set size. If the aggregate working set size of all VMs were to exceed the source host memory size, then the performance of all VMs degrades. We implement a watermark-based mechanism to detect such memory pressures in advance and trigger the migration of VMs so as to free enough memory at the source for the remaining VMs. When the aggregate working

set size of all VMs exceeds a *high watermark*, we start migrating VMs to other hosts. We migrate fewest VMs from the source which reduce aggregate working set size below a *low watermark*. Thus the another migration is not needed till the system reaches the high watermark again.

IV. IMPLEMENTATION

We implemented Agile migration on the KVM/QEMU platform. The Migration Manager is a thread spawned by each KVM/QEMU process to perform the migration of the respective VM. We use VMD as a swap device to store the cold VM pages. In the following section we describe the implementation of VMD and the Migration Manager.

A. Virtualized Memory Device (VMD) Layer

The VMD is a distributed system which enables cluster-wide memory sharing over an Ethernet network. VMD aggregates the free physical memory of intermediate hosts and presents it in the form of a block device. VMD is an extension of our previous work on MemX [5, 10].

The VMD layer is separated into VMD client and VMD server kernel modules. The VMD client module runs on the source and destination hosts. The VMD server module runs on each intermediate host. The clients and intermediate hosts communicate using the TCP protocol. Figure 3 shows the interaction between VMD clients and servers. Upon reception of a read request from the Migration Manager, the VMD client locates the intermediate server storing the requested page, receives the page from the intermediate server, and returns the page to the Migration Manager. Upon reception of a write request, the VMD client determines which intermediate server will store the page using a load-aware algorithm and forwards the page to the chosen intermediate server. On the intermediate servers, memory is only allocated after receiving a page write request. No physical memory is reserved in advance. Each VMD server periodically updates the VMD clients about the availability of memory. The load-aware algorithm works by selecting a VMD server in round-robin order, which reports having any unused memory.

We divide the aggregate memory space into logical partitions, referred to as *namespaces*. To each migrating VM, we assign a separate *namespace* as its per-VM swap device. The VMD client exports the namespace as a block device to the Migration Manager. For instance, on a host running three VMs, the VMD client would export three block devices representing three separate namespaces: /dev/blk1, /dev/blk2, and /dev/blk3. Using the block device interface, the Migration Manager can interact with all intermediate servers without needing to know where a page will be stored and how many intermediate servers are participating in the migration. This abstraction simplifies the implementation of the Migration Manager.

Our implementation of the VMD uses the excess memory at intermediate hosts to stage the VMs' cold pages. However, it is possible to extend the amount of swap space available at the VMD by using excess disk space (HDs and/or SSDs) alongside the excess memory available at the intermediaries.

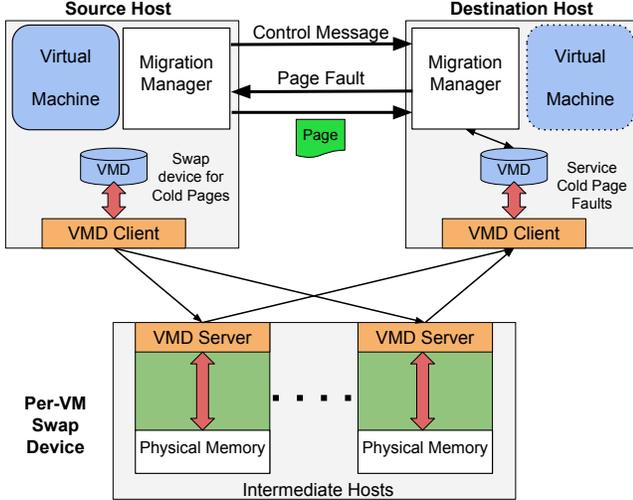


Figure 3. Message exchange and data transfer between the Migration Managers and VMD during Agile migration

B. Per-VM Swap Device

A per-VM swap device is a swap device assigned for a specific VM and is not shared with other VMs. In contrast, system-wide swap devices are shared with all processes and VMs running on the host. We use Linux `cgroups` to assign a VM to its per-VM swap device. `Cgroups` are used to create process groups. The users can manage and track the resources allocated to the group [11]. A few subsystems the user can control for a `cgroup` include memory, CPU, and I/O devices. In the memory subsystem, a user can control how much memory is allocated and limit swap usage for a `cgroup`. `Cgroups` does not allow per-`cgroup` swap file by default so we used a patch to add per-`cgroup` swap file support to the Linux kernel [12]. We assign each KVM/QEMU process into a separate `cgroup` and assign each `cgroup` its own namespace in the VMD. This ensures that all pages belonging to a VM are swapped out to its respective VMD namespace and swapping rate can be tracked per VM.

The motivation for using a per-VM swap device is that the VM's swapped out cold memory state can accompany the VM wherever it is migrated. With a common host-level swap device, the source host discards the cold pages once the VM has migrated out of the host so that the swap space can be reused for other pages. Agile migration requires that the cold memory state of the VM is kept on the swap device even after the migration of in-memory state is complete, so that the cold pages can be requested when they are accessed later. Reusing the space occupied by the cold VM pages can corrupt the VM memory state. Therefore, we disconnect the

per-VM swap device from the source host once the migration of in-memory VM state completes, but the device remains accessible from the destination host.

C. Avoiding Transfer of Swapped Out Pages

On KVM/QEMU platform, a VM runs as a KVM/QEMU process. The process exports part of its virtual address space to the VM as its physical memory. Thus the VM's physical page frames are accessible from the host through the KVM/QEMU's corresponding virtual addresses. The KVM/QEMU process shares the VM's address space with the Migration Manager allowing it to access and transfer the VM pages. During migration, the Migration Manager identifies if the page is swapped out by referring to its page table entry (PTE) from its virtual address. If the page is swapped out the Migration Manager avoids its transfer. To detect if a page has been swapped out, the Migration Manager looks at the corresponding VM's KVM/QEMU process's page table at `/proc/pid/pagemap`, where `pid` is the process's ID. `Pagemap` allows userspace programs to view a process's page table. The `pagemap` is indexed by page numbers and provides information for each PTE including if the page has been swapped out and its offset on the swap device if it has been swapped. During migration, the Migration Manager refers to the `pagemap` before transferring every page. Using a page's host virtual address, the Migration Manager looks for the corresponding PTE in the process's `pagemap` and determines if the page has been swapped out.

D. Transparently Tracking VM's Working Set

Detecting resource pressure and selecting VMs to migrate both depend upon accurate knowledge of the working set size (WSS) of all VMs. We developed a tool to let the hypervisor periodically estimate the WSS of each VM and adjust its memory reservation. Our tool periodically extracts the swapping activity of a VM using the `iostat` utility on the per-VM swap device and computes the number of pages read/written per second. If the swapping rate S goes above a pre-determined threshold τ , then the VM's reservation is increased by a factor β (which is greater than 1), until swapping activity falls below the threshold. Similarly, if the swapping rate S is above the threshold τ , then the VM's reservation is decreased by a factor α (which is less than 1) until the threshold τ is breached. The memory reservation of a VM is changed by modifying specific parameters of the `cgroup` associated with each VM. The memory reservations are initially adjusted every 2 seconds until a stable WSS is reached, at which point, the adjustment interval is increased to every 30 seconds. Our tool can monitor and adjust the reservation for a single VM, as demonstrated in Section V-D. We are currently enhancing this tool to compile the aggregate WSS of all VMs and to trigger migration when the aggregate exceeds a threshold.

E. Operations at the Source Machine

Our implementation of Agile migration is based upon a hybrid pre/post-copy implementation in KVM/QEMU. Upon initiation of migration, the Migration Manager at the source performs one round of pre-copy. In the first pre-copy round, all pages are marked as dirty in the dirty bitmap. The dirty bitmap contains the status of each memory page belonging to the VM. A page is considered clean if it has not been modified in the previous round of pre-copy. During the pre-copy round, the Migration Manager goes through the dirty bitmap and checks if the page's swapped bit has been set in the pagemap. If the page's swapped bit is not set, then the page is transferred in full to the destination and its dirty bit is reset to clean. Otherwise, the page's offset on the swap device is obtained from the pagemap and transferred to the destination along with a SWAPPED flag to indicate the status of the page. After the completion of the first pre-copy round, the dirty bitmap contains bits marked as set for the pages modified during the first round. At this point, the Migration Manager at the source suspends the VM and transfers the CPU state, dirty bitmap to the destination. After the CPU state has been transferred to the destination, the source begins actively pushing the remaining dirtied pages to the destination. The source also services the page faults at the destination host by sending the requested pages.

F. Operations at the Destination Machine

Before migration begins, a KVM/QEMU process is started at the destination to receive the incoming VM. The process allocates memory for the VM where the incoming pages are copied. When an entire page is received, it is directly copied into the allocated VM's memory space. Whereas, when a message with SWAPPED flag is received, the corresponding swap offsets is stored into a table and the corresponding bit is marked as set into the *swapped bitmap*. Upon reception of the CPU state, and dirty bitmap, the VM resumes its execution at the destination. From here onward, the Migration Manager at the destination receives pages actively pushed from the source. These pages are copied into the VM's memory. However, if the VM tries to access a page which is not received from the source, because either the page was dirtied by the VM during the first pre-copy round or the page was swapped out, a page fault is triggered. To handle the fault, the Migration Manager either requests the page from the source or reads it from the swap device.

How faults are handled: The destination Migration Manager consists a UMEM kernel driver and a user-level UMEMD process. The KVM/QEMU process shares the part of its address space assigned as a VM's memory with the UMEMD process. This allows UMEMD to receive pages from the source side and copy them into the VM's memory. The UMEM kernel driver and UMEMD process communicate through a `umem` device (`/dev/umem`). When the VM faults upon a page that is not yet present in its

memory, the fault is trapped by the UMEM driver and forwarded to the UMEMD process. The UMEMD process maintains a dedicated thread to handle the faults which in turn retrieves the faulted page and copies it directly into the VM's memory. It then notifies the UMEM driver, which resumes the VM.

We modified the UMEMD process for Agile migration to service page faults from both the VMD based per-VM swap device and the source host. In Agile migration, the VM generates two types of page faults. 1) Faults for the pages modified by the VM during the first pre-copy round. 2) Cold VM pages located on VMD. When the fault handling thread is notified of the fault, it first determines whether to request the faulting page from the source or the per-VM swap device. For determining the type of the fault, the thread refers to the swapped bitmap. If the corresponding bit is set, then it reads the offset of the page from the swap offset table, reads the page itself from the VMD, and copies the page into the VM's memory. If the swapped bit is not set, the thread requests the page from the source side. Once the page is in place, the thread notifies the UMEMD driver to resume the VM's execution.

V. EVALUATION

In this section, we compare the performance of Agile, pre-copy, and post-copy migration in responding to memory pressure. Our testbed consists of three hosts, each with twelve 2.1GHz Intel Xeon CPUs, 128GB DDR3 DRAM, 128GB Crucial SSD, and 1Gbps Ethernet cards. All hosts are connected to a top of the rack Ethernet switch with 1Gbps interconnect, use Linux kernel 3.14 with the `cgroups` patch applied, and Debian 7 for their operating system. Two of the hosts serve as the source and destination for live migration while the third serves as the intermediate host in the VMD for Agile migration: the performance of the VMD does not depend on the number of intermediate nodes as long as they have enough memory and other resources. We use the pre-copy and post-copy implementations provided by KVM/QEMU. Pre-copy and post-copy both use a 30GB swap partition created from the 128GB SSD at the source host. In all experiments for Agile migration except Section V-D, the VMs `cgroup` reservation is set to a size equal to or smaller than its working set.

A. YCSB/Redis Experiment

In this section, we demonstrate the lack of agility of pre-copy and post-copy migration in responding to memory pressure at the source host. We progressively generate memory pressure at the source host using Yahoo Cloud Serving Benchmark (YCSB) and Redis. Redis is a key-value database and runs in-memory and YCSB is a database benchmark client which interfaces with Redis. The source and destination hosts are configured with 23GB of physical memory using boot-time configuration parameters. The

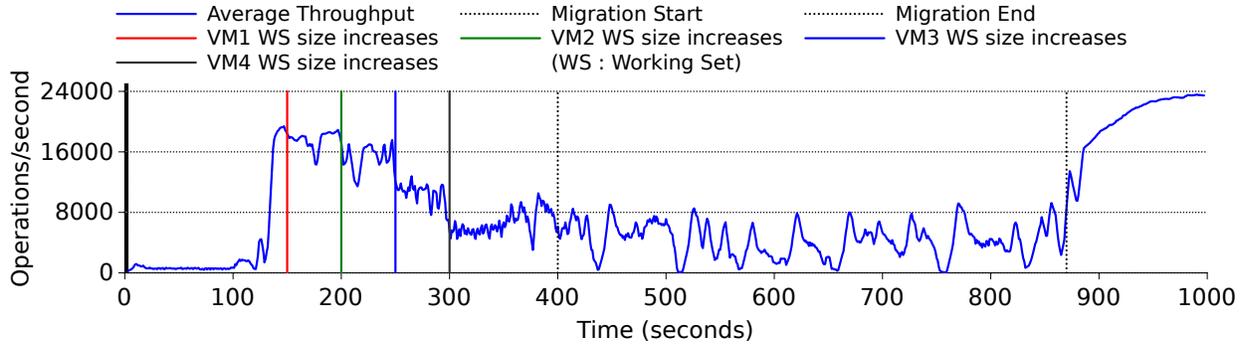


Figure 4. Average throughput of YCSB during pre-copy migration. Four VMs are running on the source host and each VM is running a Redis database server and serving four different YCSB clients. One VM is migrated to relieve the memory pressure at the source host.

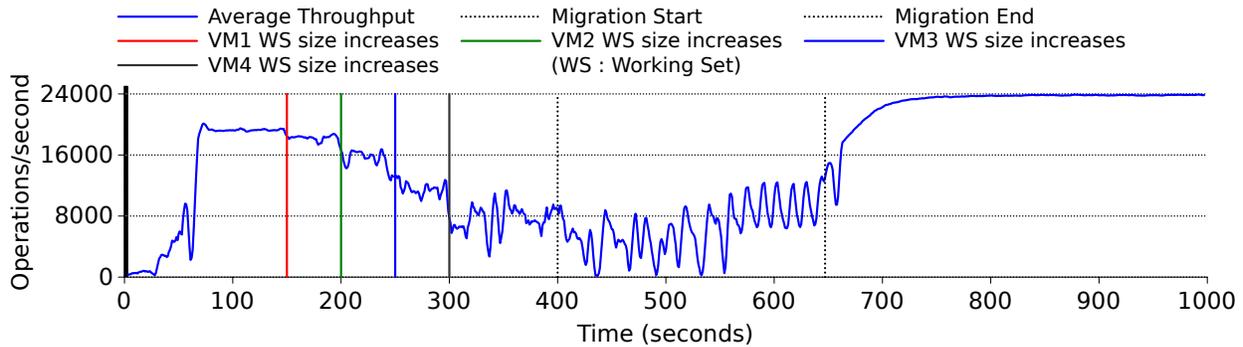


Figure 5. Average throughput of YCSB during post-copy migration. Four VMs are running on the source host and each VM is running a Redis database server and serving four different YCSB clients. One VM is migrated to relieve the memory pressure at the source host.

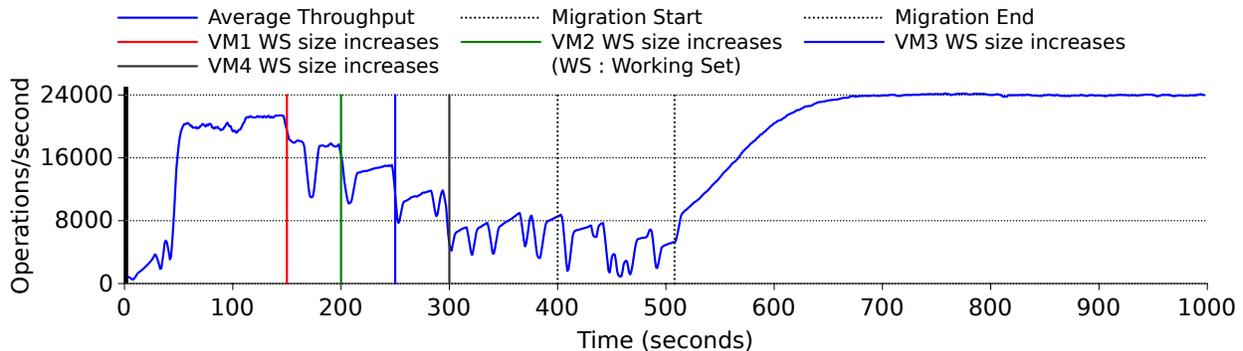


Figure 6. Average throughput of YCSB during Agile migration. Four VMs are running on the source host and each VM is running a Redis database server and serving four different YCSB clients. One VM is migrated to relieve the memory pressure at the source host.

source host runs 4 VMs each with 10GB memory and 2 vCPUs. Each VM is assigned a 5.5GB memory reservation and runs a Redis database server containing a 9GB dataset. Since the aggregate size of the Redis datasets exceeds the VM's memory reservation, loading the dataset results in many VM pages being swapped out to the swap device. 4 YCSB clients are run on an external host to load and query the 4 Redis databases.

We begin the experiment by having each YCSB client query a small fraction (200MB) of the entire dataset using read only operations and a uniform data distribution. Since the amount of data requested by each YCSB client fits in each VM's 5.5GB memory reservation, all requested pages

that reside in the swap device are brought into memory and all YCSB clients show high throughput. This workload represents low VM application intensity and small working set sizes. Such VMs are often consolidated to improve memory utilization.

Next, starting at the 150 seconds mark, we progressively increase the load in each VM by having each YCSB client query a larger fraction (6GB) of the entire dataset one by one every 50 seconds. Once all VMs become active, the host begins thrashing as it cannot accommodate the increased working set of each VM plus the host operating system (around 200MB). Thus, the performance of all YCSB clients degrades. To relieve the memory pressure at the source host,

we migrate one of the VMs to the destination host at the 400 seconds mark. After the migration is complete, the source host can accommodate the remaining three VMs in its memory and the performance of the YCSB client improves. For the following experiment, we manually adjust the VMs' memory reservation to reflect its working set size. Also, since all the VMs run an identical workload, they have nearly identical working set sizes. Therefore a VM is randomly selected for migration.

1) *Migration with Pre-copy*: Figure 4 shows the average performance of YCSB across all VMs during pre-copy migration. Pre-copy migration takes a long time to complete (470 seconds) compared to post-copy and Agile migration as it needs to retransmit all dirtied pages. Also, the migration time of pre-copy is affected by the amount of swapping activity as each page on the swap device needs to be retrieved before it can be sent to the destination.

2) *Migration with Post-copy*: Figure 5 shows the average performance of YCSB across all VMs during post-copy migration. Since post-copy doesn't require retransmission of dirtied pages, the total migration time with post-copy is lower than pre-copy. The migration completes in only 247 seconds and therefore the YCSB performance recovers more quickly. However, all VMs experience severe performance degradation during the migration. Initially, as the migrated VM resumes at the destination host, it faults upon the pages that contain the dataset and still reside at the source host. The degradation is caused by delay in serving the swapped out VM pages that are requested by the VM running at the destination host. However, the performance of migrating VM, and thus the average performance of all 4 VMs, recovers gradually during the migration.

3) *Agile Migration*: Figure 6 shows the average performance of YCSB across all VMs during Agile migration. Since only the in-memory part of the VM's address space is transferred during the migration, the migration completes more quickly. As a result, the performance of the migrating VM and the VMs left back at the source recovers more quickly than with pre-copy and post-copy migration. For comparison, pre-copy and post-copy require 533 seconds and 294 seconds respectively to restore the average performance of YCSB to the 90% of its maximum performance. Whereas with Agile migration, the performance of YCSB recovers only in 215 seconds.

B. Migration of a Single VM

In this section, we increase the memory pressure at the source host and observe its effect on the total migration time and the amount of data transferred for the migration of a single VM. To increase the memory pressure, we keep the host memory size constant at 6GB while increasing the VM memory size from 2GB to 12GB. As the VM memory size is increased more data is swapped out on a disk.

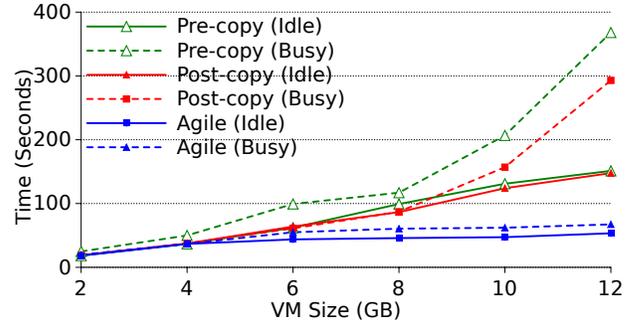


Figure 7. Total migration time for the migration of an idle or busy VM with increasing memory size. Busy VM runs a Redis server with a dataset almost as large as the memory size.

In the following experiment, we migration either an idle or a busy VM. The busy VM runs a Redis server containing a dataset nearly as large as the VM memory size leaving only 500MB of free memory. We query the dataset from a YCSB client residing on an external host and migrate the VM during the test. We measure the total migration time, the amount of data transferred and the performance of YCSB during the test.

1) *Total Migration Time*: When the VM size exceeds the host memory size, the host's memory cannot accommodate the entire VM. Therefore, it swaps out the cold VM pages to disk. During migration with pre-copy or post-copy, these pages are brought back into memory to be transferred to the destination. To retrieve the cold pages from disk into memory, other pages are swapped out to disk to create space. The delay in retrieval of cold pages increases the total migration time of the VM. From Figure 7, we observed that when the VM is idle, the rate of memory transfer gradually reduces once the VM memory size exceeds 6GB. The effect is clearly visible for the migration of a busy VM. When the VM is busy, the VM application and the Migration Manager compete for access to the swapped out pages. Therefore, the host experiences more thrashing in the busy VM setup than in the idle VM setup, causing a sudden increase in the total migration time. With pre-copy, the retransmission of dirtied pages further increases the total migration time. With post-copy, the total migration time for the busy VM is twice as high as for the idle VM, although the same amount of data is transferred in both cases. In contrast, since Agile migration does not access the swapped out pages, the host experiences less thrashing than with pre-copy and post-copy. As a result, the total migration time of the VM remains constant once the VM size goes past 6GB. The busy VM must retransmit more dirty pages, so it transfers more data than the idle VM.

2) *Amount of Data Transferred*: Figure 8 shows the amount of data transferred with pre-copy, post-copy and Agile migration. With pre-copy and post-copy, the entire VM memory state is transferred during the migration. Therefore, the amount of data transferred increases linearly with the VM memory size. In contrast, with Agile migration, only the

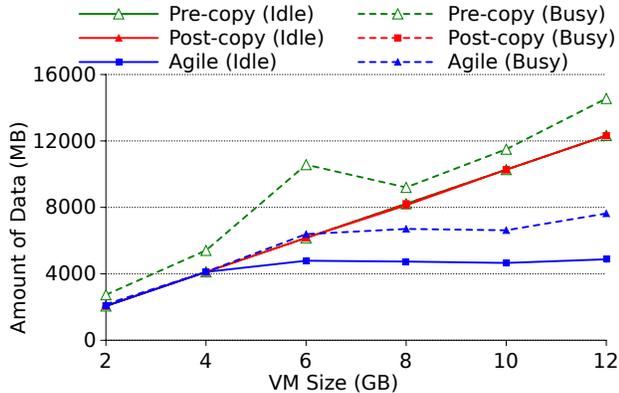


Figure 8. Amount of data transferred for the migration of an idle VM or busy VM. Busy VM runs a Redis server with a dataset almost as large as the memory size

in-memory pages are transferred during the migration. Since the host memory size is 6GB, its memory accommodates about 5.5GB of VM memory state. Therefore, even as the VM varies in memory size, the amount of data transferred with Agile migration remains constant at 5.5GB.

With post-copy, the amount of data transferred from the busy VM increases linearly as that from the idle VM. However, as the VM size increases, the amount of data transferred with pre-copy increases at a higher rate than with post-copy because of the retransmission of dirtied pages. The larger VM requires more time to migrate, which allows the VM to dirty even more pages. When the VM’s memory footprint is smaller than the host’s memory size, the VM is more responsive and the YCSB can modify more pages than when the VM memory is partly swapped out. Therefore, with pre-copy, the amount of data transferred increases more quickly with the VM size up to 6GB. For Agile migration also, all the pages dirtied during the first live round of Agile migration are retransmitted after the VM resumes at the destination. Therefore, Agile migration transfers more data for the migration of a busy VM than for the migration of an idle VM.

C. Application Performance

In this section, we use different applications to demonstrate the effect of pre-copy, post-copy and Agile migration on the application performance. We run 4 VMs on the source host, each with 2 vCPUs and 10GB of memory. The source and the destination hosts are configured with 23GB of memory. Each VM can use up to 5.5GB of host physical memory. A swap device serves additional required memory. We execute the following applications inside the VMs and migrate one VM while the application is in progress.

- 1) YCSB/Redis: We use the same setup as presented in Section V-A. The 4 VMs each contain a 9GB Redis dataset while 4 external YCSB clients query the datasets. We migrate one VM while the application is in progress to relieve the memory pressure at the

	Pre-copy	Post-copy	Agile
YCSB/Redis (Ops/s)	7653	14926	17112
Sysbench (Trans/s)	59.84	74.74	89.55

Table I
AVERAGE APPLICATION PERFORMANCE ACROSS ALL 4 VMs WITH PRE-COPY, POST-COPY AND AGILE MIGRATION. EACH VM HAS 10GB MEMORY. ONE OF THE VM IS MIGRATED DURING THE TEST TO RELIEVE THE MEMORY PRESSURE AT THE SOURCE.

source. We measure the average performance of the application across 4 VMs through the migration.

- 2) Sysbench Online Line Transactions Processing (OLTP) Benchmark: Sysbench [13] is a database benchmarking tool. We run a MySQL server inside 4 VMs, each hosting an 8GB dataset. Since each VM can use up to 5.5GB of memory, the host swaps out the remaining dataset. We query the database using 4 clients from an external host. To relieve the memory pressure at the source, we migrate one VM to the destination host. We measure the performance of 4 clients through the migration over 300 seconds and calculate the average.

Table I shows the average performance of the application across all 4 VMs during the migration. Pre-copy re-transmits the pages dirtied during the migration, which delays the transfer of the VM memory state. Further, the additional network traffic due to retransmission interferes with the traffic of the applications communicating with the VMs over the network. As a result, the applications perform the worst with pre-copy. Post-copy is quicker than pre-copy in eliminating memory pressures, so the VMs at the source recover sooner. However, the performance of the migrating VM suffers severely during migration, since the VM requests the faulted pages from the memory constrained destination. This adversely impacts the overall application performance. Since, Agile migration performs only one live round of memory transfer, it completes the live memory transfer phase more quickly than pre-copy. Therefore, fewer pages are dirtied. Moreover, since Agile migration eliminates the transfer of the swapped out pages, Agile migration least interferes with the application traffic. Consequently, the applications perform the best with Agile migration.

Table II shows the total migration time and Table III shows the amount of data transferred during the migration. It can be observed that pre-copy clearly lacks agility in responding to memory pressure. With YCSB/Redis, pre-copy migration transfers almost 2 times as much data as Agile migration while taking 4 times as long as Agile migration. Even with moderately write intensive applications, Agile migration transfers less data than post-copy, while reducing the total migration time by half.

	Total Migration Time (Seconds)		
	Pre-copy	Post-copy	Agile
YCSB/Redis	470	247	108
Sysbench	182.66	157.56	80.37

Table II

COMPARISON OF TOTAL MIGRATION TIME OF PRE-COPY, POST-COPY AND AGILE MIGRATION. WE RUN 4 10GB VMs AT THE SOURCE HOST AND MIGRATE 1 VM DURING THE TEST TO RELIEVE THE MEMORY PRESSURE AT THE SOURCE.

	Amount of Data Transferred (MB)		
	Pre-copy	Post-copy	Agile
YCSB/Redis	15029	10268	8173
Sysbench	11298	10268	7757

Table III

COMPARISON OF THE AMOUNT OF DATA TRANSFERRED WITH PRE-COPY, POST-COPY AND AGILE MIGRATION. WE RUN 4 10GB VMs AT THE SOURCE HOST AND MIGRATE 1 VM DURING THE TEST TO RELIEVE THE MEMORY PRESSURE AT THE SOURCE.

D. Transparent Working Set Tracking

Here, we demonstrate the accuracy of our approach for transparently tracking the working set of a VM and its impact on the applications running inside the VM. We run a VM having 2 vCPUs and 5GB memory, containing a 1.5GB Redis database. The host is configured with 128GB of memory. The VM's KVM/QEMU process is added in to a cgroup with 5GB memory reservation. We query the database from an external YCSB client. While the Redis database is being queried, our mechanism tracks the VM's working set size and dynamically adjusts its memory reservation. The reservation adjustment parameters are $\alpha = 0.95$, $\beta = 1.03$ $\tau = 4\text{KB/sec}$. Figure 9 shows the accuracy of the working set tracking mechanism, whereas Figure 10 shows the impact of adjusting the VM memory reservation on the performance of the YCSB client. It can be observed that our mechanism quickly adjusts the memory reservation to match the VM's working set size. Further, YCSB quickly recovers from any transient degradation.

VI. RELATED WORK

Different VM migration techniques have different approaches to reducing the amount of data transferred and total migration time. Content optimizations such as deduplication [14, 15, 16, 17, 18] and compression [14, 19, 20] reduce the amount of data transferred by eliminating or reducing the number of uniform and identical pages sent. Ballooning aims to reduce the memory footprint of a VM before migration begins. Post-copy [3] transfers each VM page only once so it works well in write-intensive scenarios. Reactive cloud [21] uses post-copy to quickly react to sudden overloads. Scatter-Gather VM migration [22] allows the fast eviction of a VM when the destination host is resource constrained so that the source host can be re-purposed. Although all approaches mentioned reduce the amount of data transferred and total

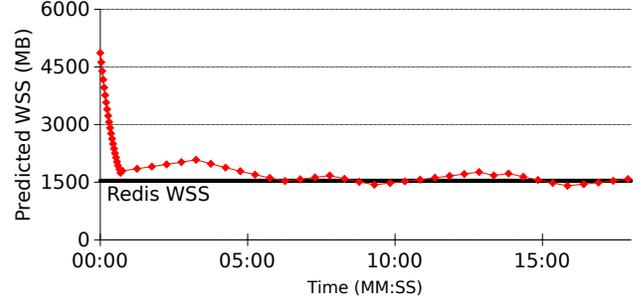


Figure 9. Dynamic WSS tracking for a VM containing 1.5GB Redis dataset. The Redis server is queried by an external YCSB client.

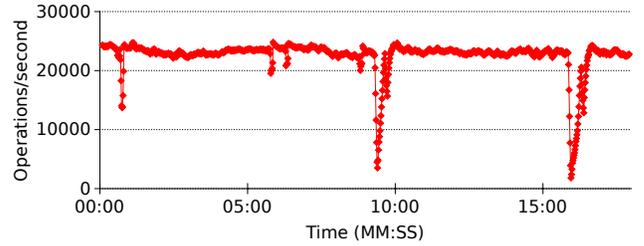


Figure 10. Performance of YCSB client querying the Redis server located inside a VM whose reservation changes dynamically as in Figure 9.

migration time, they still require the transfer of the VM's entire memory state. Jo et.al. [23] eliminate the transfer of cached pages from the source host, which are fetched directly from the network-attached storage. Jettison [24] proposes partial VM migration in which only the working set of an idle VM is migrated to the destination to achieve greater consolidation, while the remaining memory is slowly demand-paged from the source. In contrast, Agile migration eliminates the transfer of cold pages and leaves no residual state at the source.

VMWare uses a per-VM swap device [4] shared between the source and the destination hosts to avoid the transfer of swapped out VM pages during the migration. However, since the memory reservation for a VM remains fixed, the cold pages residing in the main memory are also transferred during the migration. This allows the VM to dirty even more pages, which are re-transferred during the migration. The problem becomes worse when the VM is running a write-intensive workload because pre-copy rounds take longer to converge. Even though an optimization of pre-copy, called SDPS [25], slows down vCPUs to speed up migration of write-intensive VMs, it degrades the application performance further during migration [26]. In contrast, our approach only retains the VM's working set in the memory and evicts all cold pages to the swap device. Further, we use a hybrid of pre/post-copy to transfer the working set so that migration remains agnostic to the workload type.

A number of techniques focus on detecting resource pressures and alleviating them through VM migration, but none adapt the migration technique specifically for agile response. VMWare DRS [27] and SandPiper [28] monitor resource

usage at the host-level to detect hotspots which triggers VM migration. Sandpiper [28] uses gray-box approaches to detect and respond to increased resource usage - a Linux OS daemon and Apache module installed in each VM read information contained in application logs. Zhang et al. [29] use access-bit scanning to estimate the working set of a VM. Chiang et al. [30] propose resizing VMs according to their working set sizes to increase consolidation. Overdriver [31] proposes resolving transient hotspot with network memory swap and sustained hotspots via migration.

VII. CONCLUSION

Traditional live migration approaches do not have the *agility* necessary to quickly respond to resource pressures since they transfer the entire memory of a VM. We presented a new approach, called *Agile* VM migration, that can quickly migrate a VM by transferring only the VM's working set and CPU state. During runtime, the hypervisor ensures that only working set pages of each VM reside in memory and cold (non-working-set) pages are evicted to a remote per-VM swap device. During migration, only the resident working set pages are transferred whereas the cold pages are retrieved on-demand by the destination from the per-VM swap device. In evaluations, Agile migration restores the migrating VM's performance the fastest and impacts the performance of co-located VMs the least compared to pre-copy and post-copy.

ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation through grants 1320689, 1527338, 0845832, 0855204, 1040666, and a US Department of Education GAANN Fellowship.

REFERENCES

- [1] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *NSDI*, 2005.
- [2] M. Nelson, B. H. Lim, and G. Hutchins, "Fast Transparent Migration for Virtual Machines," in *USENIX Annual Technical Conference*, 2005.
- [3] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Operating System Review*, vol. 43, no. 3, pp. 14–26, 2009.
- [4] I. Banerjee, P. Moltmann, K. Tati, and R. Venkatasubramanian, "VMware ESX Memory Resource Management: Swap," in *VMWare Technical Journal*, 2014.
- [5] U. Deshpande, B. Wang, S. Haque, M. Hines, and K. Gopalan, "Memx: Virtualization of cluster-wide memory," in *ICPP*, 2010.
- [6] M. Hines and K. Gopalan, "MemX: Supporting large memory workloads in Xen virtual machines," in *Virtualization Technology in Distributed Computing (VTDC)*, Reno, NV, Nov. 2007.
- [7] M. Hines, J. Wang, and K. Gopalan, "Distributed Anemone: Transparent Low-Latency Access to Remote Memory in Commodity Clusters," in *Proc. of the International Conference on High Performance Computing (HiPC)*, Dec. 2006.
- [8] Memcached, <http://memcached.org>.
- [9] Redis, *Key-value Cache and Store*, <http://redis.io>.
- [10] K. Gopalan, M. Hines, and J. Wang, "Distributed adaptive network memory engine," Mar. 29 2011. [Online]. Available: <http://www.google.com/patents/US7917599>
- [11] Cgroups, "<http://lxr.free-electrons.com/source/documentation/cgroups/cgroups.txt>."
- [12] Y. Zhao, "Per-cgroup swap file," <http://lwn.net/articles/592923>."
- [13] Sysbench, "<http://sysbench.sourceforge.net/index.html>."
- [14] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *HPDC*, 2011.
- [15] U. Deshpande, B. Schlinker, E. Adler, and K. Gopalan, "Gang migration of virtual machines using cluster-wide deduplication," in *CCGrid*, May 2013.
- [16] S. A. Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu, "Vm flock: Virtual machine co-migration for the cloud," in *HPDC*, June 2011.
- [17] P. Riteau, C. Morin, and T. Priol, "Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing," in *Proc. of EURO-PAR*, September 2011.
- [18] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe, "Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines," in *VEE*, 2011.
- [19] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *Proc. of Cluster Computing and Workshops*, August 2009.
- [20] P. Svard, B. Hudzia, J. Tordsson, and E. Elmroth, "Evaluation of delta compression techniques for efficient live migration of large virtual machines," in *VEE*, 2011.
- [21] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, "Reactive cloud: Consolidating virtual machines with postcopy live migration," *IPSI Transactions on Advanced Computing Systems*, pp. 86–98, Mar. 2012.
- [22] U. Deshpande, Y. You, D. Chan, N. Bila, and K. Gopalan, "Fast server deprovisioning through scatter-gather live migration of virtual machine," in *IEEE Cloud*, July 2014.
- [23] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *VEE*, 2013.
- [24] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: Efficient Idle Desktop Consolidation with Partial VM Migration," in *Eurosys*, April 2012.
- [25] VMWare Inc., *VMware vSphere vMotion Architecture, Performance and Best Practices in VMware vSphere 5*, <https://www.vmware.com/files/pdf/vmotion-perf-vsphere5.pdf>.
- [26] VMWare Knowledge Base, *Virtual machine performance degrades while a vMotion is being performed*, <http://kb.vmware.com/kb/2007595>.
- [27] VMWare Inc., "VMware DRS: Dynamic Scheduling of System Resources," http://www.vmware.com/files/pdf/drs_datasheet.pdf."
- [28] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Intl. Journal of Computer and Telecommunications Networking*, vol. 53, no. 17, 2009.
- [29] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *VEE*, 2011.
- [30] J. Chiang, H. Li, and T. Chiueh, "Working Set-based Physical Memory Ballooning," in *ICAC*, June 2013.
- [31] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, "Overdriver: handling memory overload in an oversubscribed cloud," in *VEE*, 2011.