Future Generation Computer Systems 95 (2019) 126–139

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient live virtual machine migration for memory write-intensive workloads

Chunguang Li, Dan Feng *, Yu Hua, Leihua Qin

Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, China

HIGHLIGHTS

- We discover and analyze the fake dirty pages during VM migration.
- We leverage the secure hash method to avoid transferring the fake dirty pages.
- We propose the intelligent hybrid migration to switch from pre-copy into post-copy at the near-optimal moment.

ARTICLE INFO

Article history:
Received 26 August 2018
Received in revised form 19 November 2018
Accepted 19 December 2018
Available online 8 January 2019

Keywords:
VM migration
Write-intensive workloads
Fake dirty pages
Intelligent hybrid migration

ABSTRACT

Live migration of virtual machines (VMs) is an indispensable feature in cloud computing systems due to the use of load balancing, system maintenance, power management, etc. However, it is usually an intractable problem to migrate the VMs running memory write-intensive workloads, because they rapidly generate plenty of dirty pages, which need to be resent many times. Migration failures or compromises with large downtimes usually result in severe consequences such as the violation of service level agreements, abortion of TCP connections, etc. To solve this challenging problem, we first demonstrate a novel observation: during the pre-copy migrations, some workloads have a large portion of "fake dirty" pages, which are unnecessary and wasteful to be resent. After exploring how they are generated, we propose to leverage the secure hash method to avoid resending them. Besides, to guarantee the successful accomplishment of migration, we further propose the intelligent hybrid migration. Our scheme leverages heuristic and automatically switches from pre-copy to post-copy at the near-optimal moment to obtain a short post-copy duration, thus alleviating post-copy's inherent weaknesses. Evaluations show that our proposed scheme gets a significant performance improvement for VM migration. The workloads which fail to be migrated with pre-copy now accomplish migration quickly, with a total migration time from 27 s to 98 s. Besides, the intelligent hybrid migration remarkably shortens the post-copy duration by an extent from 43% to 60%, compared with the traditional hybrid method.

1. Introduction

As one of the most important techniques in cloud computing systems, virtualization [1] has many superiorities, such as abstraction from heterogeneous hardware, security isolation, convenience for management, etc. A key characteristic of the virtualization technology is live virtual machine (VM) migration [2,3], which refers to moving a running VM from one physical host to another. Live VM migration is powerful in data centers due to the use of load balancing [4,5], online system maintenance [6], fault tolerance [7,8] and power management [9], etc.

Inside data centers, the source and target nodes for migration often share the same storage through SAN (Storage Area Network) or NAS (Network-Attached Storage). So only the content of VM memory and the states of the other virtual devices (for example, the virtual CPU state) need to be transferred. Pre-copy [2,3] is the prevailing approach for VM migration. In a pre-copy migration, the memory content is sent from the source to the target node in several sequential iterations. In the nth iteration, only the pages that are written dirty in the $(n-1)$th iteration are transferred. Thus, pre-copy tries to obtain the goal of a short downtime, during which the VM is temporarily suspended.

However, it is usually an intractable problem to migrate VMs running memory write-intensive workloads [10]. These workloads often persistently occupy a large fraction of system memory, while writing the memory pages dirty very fast. This leads to the rapid generation of plenty of dirty pages, which need to be resent many times during pre-copy. So the total data transferred (TDT) is huge and the total migration time (TMT) is prolonged. A long TMT usually results in the miss of the best migration opportunity, and

* Corresponding author.
E-mail addresses: lichunguang@hust.edu.cn (C. Li), dfeng@hust.edu.cn (D. Feng), csyhua@hust.edu.cn (Y. Hua), lhqin@hust.edu.cn (L. Qin).

https://doi.org/10.1016/j.future.2018.12.048
0167-739X/

makes the activities of system maintenance, load balancing, etc. less effective. Besides, it is not an exception that the speed to generate dirty pages exceeds the network bandwidth for migration. In such a circumstance, pre-copy would not converge to *stop-copy* phase, so migration fails to be completed with an expected downtime [11]. Facing this difficult situation, the administrators in data centers have to force the completion of migration, resulting in an unacceptable downtime as long as several seconds, sometimes even tens of seconds. This long downtime often violates the service level agreements (SLAs). Besides, it also leads to some other serious problems, such as the abortion of TCP connections, consistency problems, etc. [12]. However, without this long downtime, the cancels [13] or failures of migration would lead to severer loss for the data centers. Remedies, such as decreasing the VCPU's running frequency [14] and imposing delays to page writes like the Stun During Page Send (SDPS) in XvMotion [15], also usually violate the SLAs, QoS, etc.

This paper tries to solve the challenging problem of migrating the memory write-intensive workloads. First, our analysis demonstrates a novel observation: during migration, some of these workloads have a large portion of "fake dirty" pages. That is, many dirty pages transferred to the target node are exactly the same in content as their former copies which have already been there. This means that resending them is just a waste of time and network resources. Then we explore the causes for the generation of these fake dirty pages and discover two reasons: the "write-not-dirty" requests issued to memory pages, caused by the silent store instructions [16–19], and the defect of the dirty page tracking mechanism used by migration. Furthermore, we propose to leverage the computation and comparison of the secure hashes for memory pages to avoid transferring these fake dirty pages. We also analyze the hash collision probability and demonstrate that it is reasonable to use secure hash comparison instead of byte-by-byte comparison. Besides, to store the secure hashes of memory pages consumes negligible memory overhead. This is a big advantage of our method against the byte-by-byte comparison and other schemes.

As explained above, pre-copy may result in the failures of migration in some cases. To guarantee the successful accomplishment of migration, we further propose the intelligent hybrid migration. A hybrid migration consists of several pre-copy iterations and a post-copy [20,21] phase. However, post-copy has inherent weaknesses in terms of VM reliability and application performance. So it is critical to shorten the post-copy duration in a hybrid migration design. The key design of our intelligent hybrid method is that, it automatically switches from pre-copy to post-copy, when taking more pre-copy iterations would not further reduce the amount of dirty pages. Thus we do not perform more useless pre-copy iterations, while obtaining a short post-copy duration.

The main contributions of this paper are listed as follows:

(1) We demonstrate a novel observation that a large portion of fake dirty pages exist during pre-copy migration. Experiments are conducted to evaluate the proportions of fake dirty pages for various memory write-intensive workloads. Besides, we explore how they are generated in detail. Furthermore, the secure hashes method is proposed to avoid transferring these fake dirty pages.

(2) To guarantee the success of migration, we further propose the intelligent hybrid migration. Leveraging a heuristic, it automatically switches from pre-copy to post-copy at a near-optimal moment, thus effectively shortening the duration of the post-copy phase.

(3) Evaluation results with the KVM/QEMU platform show that our scheme obtains significant performance improvement for VM migration. With our secure hash method, the workloads, which originally fail to be migrated with pre-copy, now complete migration with a short TMT from 27 s to 98 s. Besides, compared with the traditional hybrid migration, our intelligent hybrid method remarkably shortens the post-copy duration by an extent from 43% to 60%.

The rest of this paper is organized as follows: Section 2 shows an overview of the background of live VM migration. Our observation and motivation are demonstrated in Section 3. Section 4 details the design of our proposed scheme. In Section 5, we show the evaluation results. Section 6 lists the related work and Section 7 draws the conclusions.

2. Background

In this section, we introduce the background for live VM migration, including the key metrics for migration, and the various migration algorithms.

Generally speaking, there are three key metrics for live migration: total migration time (TMT), total data transferred (TDT), and downtime. TMT is the elapsed time from the beginning of the migration to the moment when the VM is able to run on the target host independently. It is important to shorten the TMT. Because long TMT may lead to the misses of the best migration opportunities, and make the activities of system maintenance, load balancing, etc. less effective. Furthermore, TDT refers to the amount of data transferred from the source to the target node during the migration. It indicates the network resources consumed by the migration. Because the migration thread and applications running inside the VMs often share the same network infrastructure, less TDT would leave more bandwidth to applications during the migration. At last, downtime is the period from the moment the migrated VM is suspended at the source node, to the moment it is resumed at the target node. The downtime should be short enough, so that it is not perceived distinctly by users. Besides, it should not have obvious impact on the applications running inside the VMs. The default setting of the maximum expected downtime in KVM/QEMU [22,23] platform is 300 ms.

Pre-copy [2,3] is the prevailing algorithm for migration and has been widely used in many virtualization platforms, such as VMware [24], KVM [22], and Xen [1]. It works as follows. The bulks of the VM's memory pages are transferred to the target node while the VM is still running at the source node. The pages which are written dirty during the transmission are resent to the target node in the next iteration. This iterative process comes to the end, when the remaining dirty pages are so little that the expected time to transfer them is shorter than the maximum acceptable downtime. Then pre-copy steps into the *stop-copy* phase and the VM is suspended at the source node. During this transitory suspension, the remaining dirty pages along with the states of the virtual devices are transferred to the target node. After that, the VM is resumed at the target node. Pre-copy's overriding goal is to keep the downtime short. However, as pre-copy needs to transfer part of the memory pages many times, it often causes long TMT and large TDT. Even worse, when VMs are running memory write-intensive workloads, pre-copy may fail to complete migration with an expected downtime.

To solve the problems of pre-copy, another live migration algorithm, post-copy [20,21], has been proposed. At the beginning of migration, post-copy first suspends the VM at the source node. Then it transfers the states of the virtual devices to the target node. After that, the VM is resumed at the target node without any memory pages. Memory pages are fetched on-demand from the source node when the VM incurs page-faults by accessing non-exist pages at the target node. Concurrently, post-copy actively pushes the pages that have not been accessed from the source to the target node. This background push thread helps to accelerate the process of post-copy. Because each page is transferred only once in post-copy, the TMT is shortened, and the TDT is reduced,

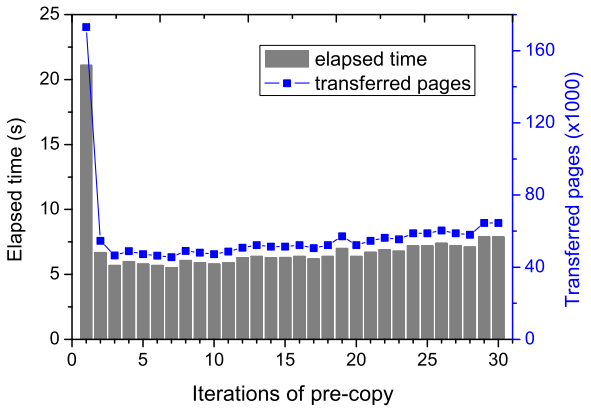


Fig. 1. The elapsed time and amount of transferred pages for each pre-copy iteration when we migrate a VM running zeusmp. The default migration bandwidth of KVM/QEMU is used, which is 32 MB/s. We do not count zero pages when recording the amount of transferred pages.[1]

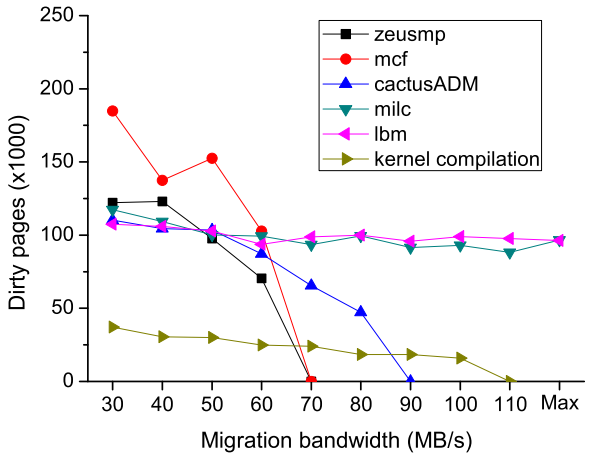


Fig. 2. The average amount of dirty pages with varying migration bandwidth after the pre-copy process does not converge any more. A value of zero means that the workload is migrated successfully with high enough bandwidth. Some of the dirty pages may be zero pages.

compared with pre-copy. Besides, post-copy guarantees the accomplishment of migration with an estimable TMT. However, post-copy has the following two inherent weaknesses. Firstly, post-copy is less robust than pre-copy. During the post-copy migration, both source and target nodes have part of the latest memory status, so malfunctions of the migration network or the target node would cause VM failure. In contrast, the same malfunctions during the pre-copy migration only cause migration failure, while the VM would still run at the source node. Secondly, during the post-copy migration, the continuous demand-fetching of fault pages increases the memory access latencies, thus declines the performance of applications running inside VMs.

Furthermore, the hybrid migration method [25–27] has also been proposed to combine the pre-copy and post-copy algorithms. Consisting of several pre-copy iterations and a post-copy phase, it tries to transfer the bulks of memory pages during the pre-copy period, so that less pages need to be transferred during the post-copy phase. Thus on one hand, hybrid migration guarantees that the migration is completed in finite time; On the other hand, it shortens the duration of the post-copy phase. However, none of the previous works has explored when the best moment is to switch from pre-copy to post-copy. They usually perform fixed one or two pre-copy iterations, and then switch into post-copy.

3. Motivation

3.1. Challenge of migrating memory write-intensive workloads

It is usually a difficult problem to migrate VMs running the memory write-intensive workloads. This is because these workloads generate dirty pages very fast. These dirty pages usually need to be resent many times during the pre-copy migration, resulting in long total migration time (TMT) and huge total data transferred (TDT). Besides, it is a common case that the pre-copy could not converge into the *stop-copy* phase, when the speed to generate dirty pages exceeds the network bandwidth for migration. This situation leads to the failures of migration with the expected downtime.

The following experiment illustrates this problem in detail. We migrate a VM running the zeusmp workload from the CPU2006 benchmark suit [28]. This is a memory write-intensive workload. During the pre-copy process, we record the elapsed time and the amount of transferred pages for each iteration. As is shown in Fig. 1, zeusmp writes the memory pages dirty very fast. From the third iteration of pre-copy, the amount of transferred pages does not decrease as more iterations are performed. This leads to the failure of the migration, unless with an unacceptable downtime as long as 6∼8 s. This is an example of the intractable workloads for VM migration. Prolonging the migration process does not shorten the downtime, while wasting plenty of network resources. Thus, to successfully migrate the VMs running these memory write-intensive workloads is a big challenge in data centers. The goal of this paper is to solve this issue.

In this paper, we use eight memory write-intensive workloads running inside the VMs. The details of these workloads are shown in Table 1. Except Memcached and kernel compilation, all the workloads are from the SPEC CPU2006 benchmark suit. We migrate these VMs using the default setting of the migration bandwidth in QEMU, which is 32 MB/s, except Memcached. For Memcached, we use the maximum bandwidth of our Gigabit Ethernet (about 117 MB/s), because it has larger guest memory size and faster memory dirtying speed. As a result, except bzip2, all of these workloads fail to complete the pre-copy migration with the default setting of maximum downtime in QEMU, which is 300 ms.

Note that the workloads except Memcached are migrated with limited 32 MB/s bandwidth, which could be further increased if there are abundant network resources in data centers. Allocating a higher migration bandwidth may make the migrations of these workloads easier, because less pages become dirty during one iteration. In order to obtain a comprehensive knowledge of the impact of migration bandwidth, we conduct the following experiments. We migrate these workloads except Memcached and bzip2 (because the 32 MB/s bandwidth has already completed the migration of bzip2), allocating different bandwidth varying from 30 MB/s to the maximum network throughput, by the step of 10 MB/s. During the migrations, we record the dirty pages generated in each iteration. We expect to obtain the average amount of dirty pages for one iteration, after the pre-copy process does not converge any more, as shown in Fig. 1. The results are shown in Fig. 2. We plot the average amount of dirty pages for ten iterations, from the 11th to the 20th iteration. Note that the value may decrease

[1] In a zero page, the content of every byte is zero. For some workloads such as zeusmp, zero pages account for a large proportion within all the transferred pages during migration. However, the existing implementation of migration in QEMU employs a standard optimization to quickly recognize zero pages. Besides, it uses one byte (plus a flag to distinguish) to represent one entire 4 KB page when transferring a zero page. So zero pages scarcely consume time or network resources during migration. Due to this reason, when talking about the amount of pages, the results of all the experiments in this paper do not count zero pages.

Table 1
The eight memory write-intensive workloads we use in this paper.

Workloads	Description	Approximate working set size (% of guest memory size)
Memcached	An in-memory key–value store, Memcached-1.4.25, is running inside the VM. The VM is allocated with 6 cores and 6 GB memory. Memcahced is configured with 4 threads and 4 GB cache. A benchmark, memaslap from libmemcached-1.0.18, is executed in another client machine. Memaslap first runs against Memcached to fill up the 4 GB cache. During the migration, memaslap is executed with concurrency level of 6, and performs all set requests against Memcached.	67% of 6 GB
zeusmp	This is a floating point benchmark, which performs the simulation of astrophysical phenomena using computational fluid dynamics. The VM is allocated with 1 core and 1 GB memory (same as the following five workloads).	50% of 1 GB
mcf	This is an integer benchmark, which uses a network simplex algorithm to schedule public transport.	80% of 1 GB
bzip2	This is an integer benchmark, which performs Seward's bzip2 version 1.0.3, modified to do most work in memory.	80% of 1 GB
cactusADM	This is a floating point benchmark, which solves the Einstein evolution equations using a staggered-leapfrog numerical method.	60% of 1 GB
milc	This is a floating point benchmark, which performs a gauge field generating program for lattice gauge theory programs with dynamical quarks.	60% of 1 GB
lbm	This is a floating point benchmark, which implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D.	40% of 1 GB
kernel compilation	The Linux 3.7.9 kernel is compiled with 4 threads in the VM. This is a system-call intensive workload, which is expensive to virtualization. The VM is allocated with 4 cores and 1 GB memory.	15% of 1 GB

to zero for some workloads. This means that these workloads are migrated successfully when the bandwidth is increased to a particular value. It is shown that for zeusmp, mcf, cactusADM and kernel compilation, the amount of dirty pages gradually decreases with the increase of migration bandwidth. At last, they are migrated successfully with high enough bandwidth. This demonstrates that the problem of migrating these workloads is alleviated by a higher bandwidth. However, for milc and lbm, they generate dirty pages so fast, that the amount does not decrease obviously with a higher bandwidth. So they still fail to be migrated, even with the maximum bandwidth of our network.

Our testbed is equipped with Gigabit Ethernet (1 GbE). Faster network infrastructures such as 10 GbE and InfiniBand, with new technologies like SR-IOV (Single Root I/O Virtualization) [29,30], are being deployed in modern data centers, and the migration bandwidth could be further improved. This would further alleviate the problem of migrating memory write-intensive workloads. It could be deduced that, with such faster networks, our VM instances of milc and lbm could be migrated successfully by pre-copy, because their 1 GB memory size is quite small for these fast networks. However, in the next paragraph, we would discuss the impact of larger VM memory size, which on the other hand exacerbates the migration problem. Here we first indicate a side effect of a faster network. With a better network infrastructure, network-intensive workloads, such as Memcached, are able to handle the requests of more clients concurrently. This may make the VM memory get dirtied faster. We conduct an experiment to verify this consideration. For the Memcached workload described in Table 1, we increase the concurrency level of memaslap from 6 to 24, which corresponds to the simulation of more clients requesting the Memcached server concurrently. As a result, during migration, the amount of dirty pages generated in the first iteration is increased from 584,429 to 959,025.

The VM instances used in this paper is relatively small, with several GB memory. However, VMs with much larger memory size, such as the Amazon EC2 X1e instances with memory size from 122 GB to 4TB [31], are being adopted in modern data centers. This trend is making the migration problem more severe, because the time spent on one iteration becomes longer, which makes more pages get dirtied. The impact of a larger memory size is evaluated by the following experiments. We enlarge the VM memory for Memcached from 6 GB to 26 GB, by the step of 4 GB. The cache configured for Memcached is always 2 GB less than the memory size, e.g., 24 GB cache for the 26 GB VM. The other configurations are the same as described in Table 1. When migrating these VMs, we record the time spent and the amount of dirty pages during the first iteration. As is demonstrated in Table 2, both the time spent and dirty pages increase nearly linearly with the memory size. This verifies that a larger memory size does deteriorate the migration problem.

Table 2
The time spent and the amount of dirty pages for the first iteration when the Memcached VMs with varying memory sizes are migrated.

Memory size	6 GB	10 GB	14 GB	18 GB	22 GB	26 GB
Time (s)	42	84	117	158	195	235
Dirty pages (K)	584	1124	1806	2459	3,072	3517

After having discussed the above several factors that affect the migrations of memory write-intensive workloads, we conclude that migrating these workloads is still a big challenge. First, in the production environment, the bandwidth allocated to VM migration could be limited, because the migration process and the applications in the data centers often compete for the same network resources. Besides, when VMs are migrated across networks with low-speed link, e.g., for a WAN migration, the problem would be especially severe. At last, even though faster network infrastructures, such as 10 GbE, could alleviate this problem, the earnings are counteracted by the trend in larger and larger VM memory size.

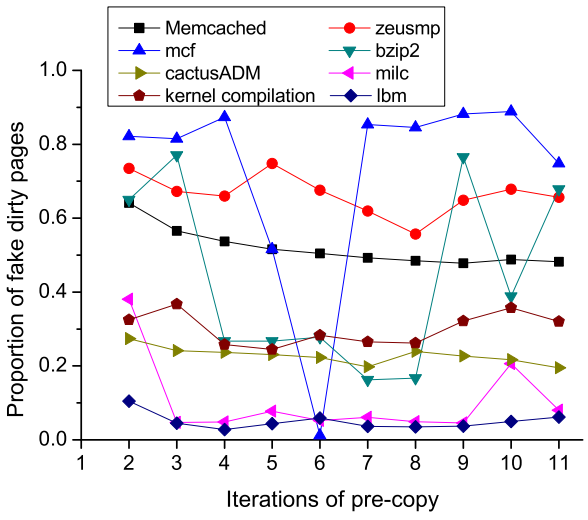


Fig. 3. The proportion of the fake dirty pages within all the transferred pages for each iteration. Note that fake dirty pages begin to appear from the second iteration of pre-copy.

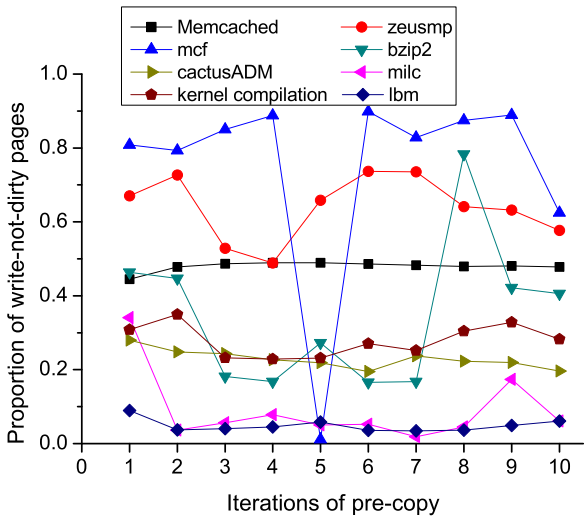


Fig. 4. The proportion of the write-not-dirty pages within all the pages marked as dirty for each iteration.

3.2. Observation: the "Fake Dirty" pages

The main difficulty of migrating the memory write-intensive workloads is the fast generation of dirty pages. Therefore, we first analyze the dirty pages in detail. However, we find that during the pre-copy migration, many dirty pages are in fact "not dirty". This means that even though marked as dirty, they are in fact unaltered in memory content compared with the copies transferred in the former iteration. We name these pages as "fake dirty" pages. Transferring these fake dirty pages is a significant waste of time and network resources, because the same data is copied to the target node while it is already there.

To illustrate the fake dirty pages in detail, we conduct the following experiments. When migrating the VMs running workloads described in Table 1, at the source node, we maintain a buffer in the host memory as large as the VM memory size. During the entire migration time, we keep the content of this buffer the same as the VM memory content in the target node. Specifically, in the first iteration of pre-copy, when sent to the target node, each page is also copied into the buffer according to its guest physical address. From the second iteration, when each dirty page is transferred, its content is compared byte-by-byte with its former copy inside the buffer. If the contents are the same, this page is recorded as "fake dirty"; Otherwise, the new content replaces its former copy for the potential comparison in the future. Thus we obtain the amount of fake dirty pages and compute their proportion in each iteration as shown in Fig. 3.

It is demonstrated that, four of these workloads (Memcached, zeusmp, mcf and bzip2) have remarkably high proportions of fake dirty pages. For example, in some iterations of mcf, the proportion is higher than 80%. For these workloads, if we could avoid transferring these fake dirty pages, the TMT and TDT would be reduced significantly. However, the other four workloads (cactusADM, milc, lbm and kernel compilation) have relatively low proportions of fake dirty pages. The same optimization may obtain limited improvements for them.

3.3. Reasons for the generation of the fake dirty pages

Given the big amount of fake dirty pages, the first question to ask is: why do so many fake dirty pages exist at all? Next, we will explain the two reasons for the generation of fake dirty pages. First, there are many "write-not-dirty" requests issued to the memory pages, caused by the widely existing silent store instructions [16–19]. Second, the dirty page tracking mechanism used for the migration has defects in design. We find that the former is the primary reason, while the latter is the secondary.

1. The "write-not-dirty" requests to memory pages

Virtualization platforms such as KVM and XEN leverage similar mechanisms to track dirty pages during migration, using the dirty bitmap. From the beginning of the pre-copy migration, all of the pages are write-protected so that a write request on a page leads to a soft page fault. While the page fault is handled, the dirty flag of this page is set in the dirty bitmap. The pages marked as dirty in the bitmap are resent in the next iteration.

However, does one page marked as dirty really change in memory content? In other words, does a write request issued to one page really modify the content of this page? In fact, the answer is "not always". This is verified by the following experiment. Again, we migrate VMs running the workloads shown in Table 1. During the setup phase of the migration, first we suspend the VM. Then we copy all the VM memory pages to a buffer as large as the VM memory size. After this, we begin to trace the dirty pages and resume the execution of the VM. Besides, at the end of each iteration, we also suspend the VM temporarily. During the suspension, we compare all the pages marked as dirty in this iteration byte-by-byte with their former copies inside the buffer. If the content of one page is the same as its former copy, it is recorded as a "write-not-dirty" page. Otherwise, this page in the buffer is replaced with the new content, for the possible comparison in the future. After the reset of the dirty bitmap, we resume the VM. Thus, we obtain the proportion of the write-not-dirty pages within all the pages marked as dirty for each pre-copy iteration.

As is demonstrated in Fig. 4, write-not-dirty pages exist in all of the workloads. Besides, the proportions for some workloads, such as Memcached, zeusmp and mcf, are extremely high. Especially, the proportion for mcf is even higher than 80% in most iterations. Compared with the results in Fig. 3, it is deduced that, the primary reason to generate the fake dirty pages is the write-not-dirty requests issued to memory pages.

In fact, the root cause of the write-not-dirty requests is the widely existing silent store instructions [16–19]. A silent store instruction does not change the state of the system, because it writes a value that exactly matches the value which has already been stored at the memory address being written. Results reported

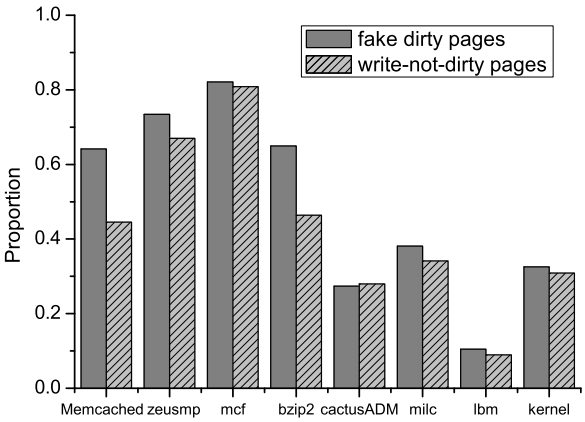


Fig. 5. The proportion of write-not-dirty pages generated in the first iteration compared with the proportion of fake dirty pages transferred in the second iteration.

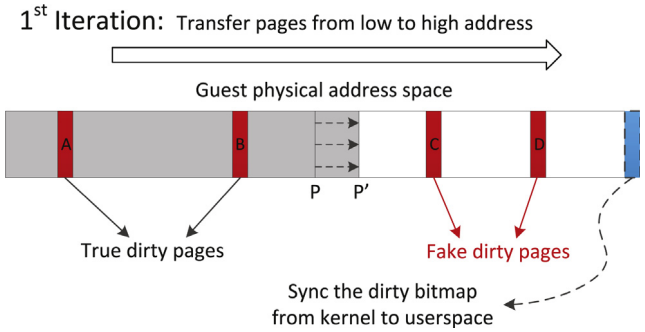


Fig. 6. The KVM dirty page tracking mechanism, which also results in the generation of the fake dirty pages.

in [16] demonstrate that 20%–68% of all store instructions are silent, which explains the high proportion of write-not-dirty pages. Analysis in the context of source code presented in [17] indicates that many silent store instructions are algorithmically in nature, and that they occur in all levels of program execution and compiler optimization. This is consistent with our finding that the write-not-dirty requests exist widely in various workloads.

However, the write-not-dirty request is not the only reason that generates the fake dirty pages. This is demonstrated by Fig. 5. We compare the proportion of the write-not-dirty pages generated in the first iteration with the proportion of the fake dirty pages transferred in the second iteration. If the write-not-dirty request is the only reason for the fake dirty pages, the proportion of the fake dirty pages should not be larger than that of the write-not-dirty pages. This is because in the second iteration, some of the write-not-dirty pages may be written again and become really dirty before they are transferred. However, Fig. 5 shows that the contrary is the case for most workloads, especially for Memcached and bzip2. So there must be some other reasons that also lead to the generation of the fake dirty pages. In fact, we discover that the defect in the dirty page tracking mechanism used by the migration thread is another reason for this. Next, we will explain it in detail.

2. The defect of the dirty page tracking mechanism

As described above, both of KVM and XEN use the dirty bitmap to track the dirty pages during the migration. In the following, we will explore the dirty page tracking mechanism in more detail, and show how it contributes to the generation of the fake dirty pages.

(1) KVM

The KVM/QEMU platform consists of a KVM [22] kernel module and a userspace tool—QEMU [23]. During the migration, both of the kernel and the userspace maintain a dirty bitmap. The kernel bitmap is responsible for tracing the write requests to memory pages and setting the corresponding dirty bits. It records all the pages that are written in one whole iteration. At the end of one iteration, the dirty bitmap is synchronized from the kernel to the userspace through the "ioctl" API offered by KVM. At the same time, the kernel bitmap is reset to start a new round of tracking for the next iteration. In the next iteration, the migration thread leverages the userspace bitmap updated from the kernel to transfer dirty pages.

So the granularity of the dirty page tracking mechanism is as coarse as one whole iteration of pre-copy. It is just this coarse granularity that also results in the generation of the fake dirty pages. We use Fig. 6 to illustrate the reason in detail. We take the first iteration of pre-copy as example, while the other iterations are similar. In the first iteration, the migration thread transfers every page within the guest address space sequentially from the low to the high address. We assume that during the period for the transferring cursor to move from address P to address P', 4 pages, which are pages A, B, C and D, receive write requests. So these 4 pages are marked as dirty in the kernel bitmap. We assume these 4 pages are not the write-not-dirty pages we have explained above. Even so, in this example, only pages A and B are really dirty, while pages C and D are "fake dirty". The reason is, unlike A and B, C and D are marked as dirty before being transferred in this iteration. So when the migration thread sends C and D to the target node, it is the new dirty content that is transferred. Unless C and D are written dirty again after being transferred, when they are resent in the next iteration, it is just the same content that is transferred.

(2) XEN

Unlike KVM, the migration thread of XEN uses three bitmaps to indicate the pages to be transferred.

to_send: It indicates the pages written dirty during the former iteration. Like KVM, at the end of each iteration, the migration thread update the to_send bitmap according to the kernel bitmap and then reset the kernel bitmap.

to_skip: Every now and then during one iteration, migration thread peeks the kernel bitmap to update the to_skip bitmap. However, different from the update of the to_send bitmap, this operation would not reset the kernel bitmap. Pages in to_skip bitmap would not be transferred in current iteration, because they still need to be transferred in next iteration.

to_fix: It indicates the pages that should be transferred during the last *stop-copy* phase.

In terms of the reduction in the generation of the fake dirty pages, the tracking mechanism of XEN performs much better than KVM. This is because, the *to_skip* bitmap avoids many pages written dirty in the current iteration from being transferred. However, during one iteration, if a page to be sent is written dirty after the latest update of the *to_skip* bitmap (so it will not be skipped in this iteration), and just before being transferred, it is still a fake dirty page.

4. Design

According to the finding in the former chapter, if we can avoid transferring the fake dirty pages, the TMT will be shortened, and the TDT will be reduced. So in this chapter, we propose to leverage the secure hashes to realize this objective. Besides, to guarantee the successful completion of the migration process, we further combine our secure hash method with the intelligent hybrid migration. Fig. 7 shows the overall workflow of our design. In the following, we will illustrate our design in detail.

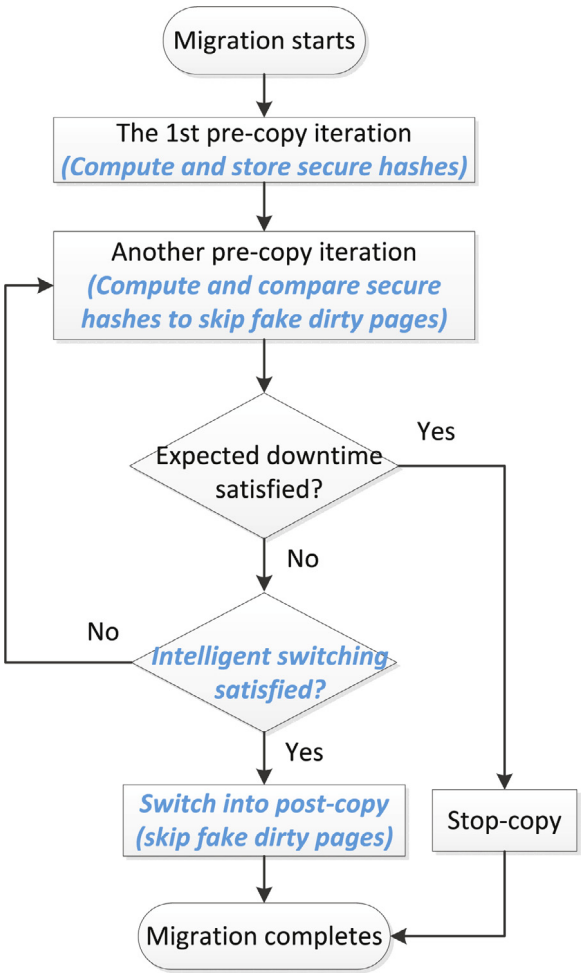


Fig. 7. The overall workflow of our design. The italic blue parts indicate our design added on the original pre-copy algorithm.

4.1. Leverage secure hashes to avoid transferring fake dirty pages

To avoid transferring the fake dirty pages, we propose to compute and compare the secure hash for each page to be transferred. The concrete process is as follows. During the first iteration of pre-copy, we compute and store the secure hash for the 4 KB content of each transferred page. In the following iterations, before a page is sent, its secure hash is first computed. Then this hash is compared with its former version. If the hashes are the same, the content of this page is considered as not changed (we will discuss the hash collision probability in the next paragraph). This indicates that it is a fake dirty page, so we just skip it without transferring. Otherwise, it is a really dirty page which needs to be transferred. Besides, for a really dirty page, the old version hash needs to be replaced with the new one, for the possible comparison in the following iterations.

In our design, the same secure hashes indicate that the content of one page stays unchanged. Now we explain the rationality of this consideration, even though theoretically, there may be hash collisions, which means two different pages may have the same hash. According to the "birthday paradox" [32,33], we are able to calculate the collision probability of a given secure hash pair as follows:

$$\textit{Hash collision probability}: \ p \leq \frac{n(n-1)}{2} \times \frac{1}{2^m} \tag{1}$$

n : *the number of chunks* (*memory pages here*)
m : *the length of the hash in bits*

Table 3
The hash collision probability for SHAx hashes with different amounts of transferred pages during migration.

Amount of transferred pages	SHA1 160 bits	SHA256 256 bits	SHA512 512 bits
256 K (1 GB)	10^{-37}	10^{-66}	10^{-143}
256 M (1 TB)	10^{-31}	10^{-60}	10^{-137}

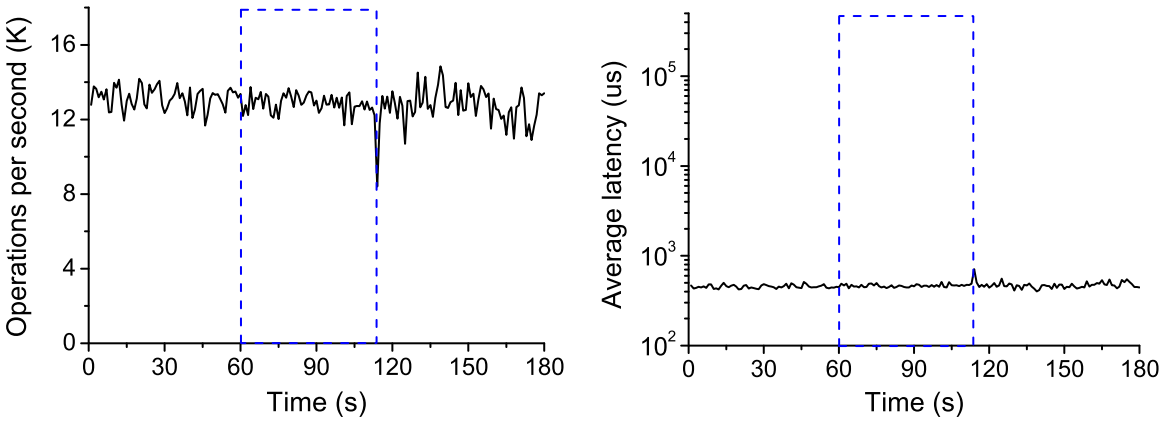
Table 3 shows the hash collision probability for SHAx hashes according to formula (1) with different amounts of transferred pages during the migration. Assume the data size of the transferred memory is as huge as 1 TB, which is quite impossible, the probability for a SHA1 collision is less than 10^{-31}. In fact, this probability is much lower than the probability of a DRAM error in computer systems [34,35]. So the match of secure hashes is sufficient to demonstrate that the content of one page is unchanged. There is no need for a further byte-by-byte comparison.

Leveraging secure hashes, we successfully avoid transferring the fake dirty pages. Besides, there is an obvious advantage to use secure hashes, compared with the byte-by-byte comparison. That is, storing the secure hashes only consumes very little memory overhead. However, the byte-by-byte comparison needs a buffer in memory as large as the VM memory size to hold the copies of all the pages. Take the SHA1 hash as example, for a 4 KB page, its SHA1 hash is only 20 bytes. So the memory overhead to store the SHA1 hashes is only less than 1/200 of the VM memory size, which is negligible.

In fact, a feature called XBZRLE [12,36] offered by KVM/QEMU could also avoid transferring the fake dirty pages, even though this is not its objective in design. However, it consumes very large memory overhead to obtain a good effect. Next we will explain this in more detail. XBZRLE is short for "Xor Binary Zero Run-Length-Encoding". Users could choose to activate this feature during the migration to obtain a possible migration performance improvement. Instead of sending a whole dirty page, XBZRLE sends the compressed content of the update against the old version of this page. So it reduces the amount of data transferred. In order to calculate the update, the old copies of the pages need to be stored at the source node, so it needs a cache in memory. From its principle, it is deduced that XBZRLE could also avoid transferring the fake dirty pages, if their old copies are in the cache. In fact, it does more than this because it also performs delta compression for the really dirty pages. However, it needs a memory cache quite large to hold enough pages in order to get a satisfying performance improvement. Otherwise, there would be a lot of cache misses, which restricts the effect of XBZRLE. The evaluation results in Section 5 demonstrate this in detail.

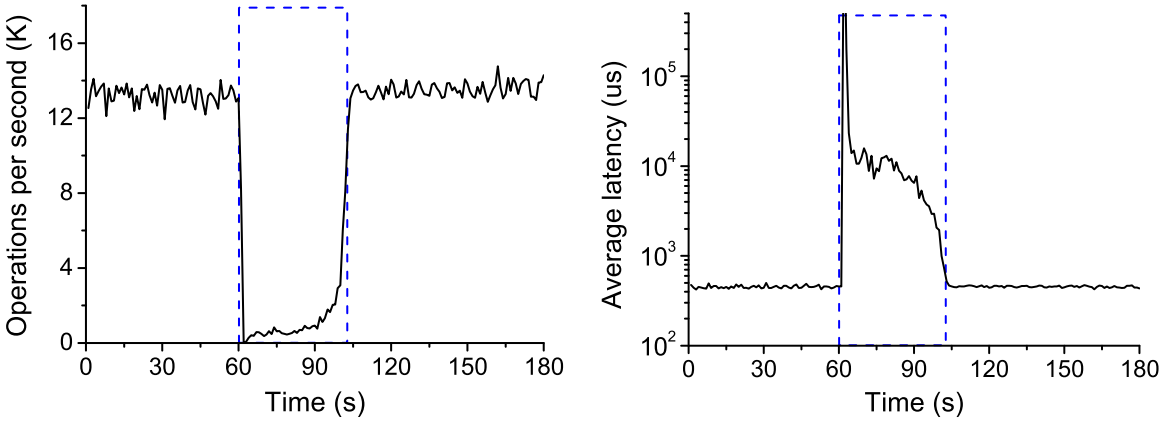
4.2. Intelligent hybrid migration

As explained above, memory write-intensive workloads often result in migration failures with an expected downtime. Our secure hash method would improve the migration performance. However, it does not guarantee the successful completion of the pre-copy migration, especially for the workloads with low proportion of fake dirty pages. Therefore, to guarantee the success of migration, we further propose the intelligent hybrid migration. As introduced in Section 2, hybrid migration combines the pre-copy and post-copy algorithms. A traditional hybrid method usually fixedly performs one or two pre-copy iterations before shifting into post-copy, or is switched manually by administrators in data centers. In contrast, our design is "intelligent", because it leverages heuristic and automatically switches from pre-copy to post-copy at the near-optimal moment. Therefore, our intelligent hybrid migration combines the advantages of pre-copy and post-copy more



(a) Throughput (b) Average latency

Fig. 8. The performance of the Memcached server during the pre-copy migration. The dashed box indicates the duration of the migration.



(a) Throughput (b) Average latency

Fig. 9. The performance of the Memcached server during the post-copy migration. The dashed box indicates the duration of the migration.

efficiently than the traditional ways. Next, we will explain this design in detail.

The post-copy phase in a hybrid migration guarantees that the migration is completed in finite time. However, as explained in Section 2, post-copy has inherent weaknesses in VM reliability and application performance. To verify the weakness of the performance degradation caused by post-copy, we migrate the VM running the Memcached with pre-copy and post-copy respectively. The details of this experiment is almost the same as described in Table 1, except that during migration, memaslap is executed with set-get ratio of 1:9. This set-get ratio makes the VM memory pages get dirty much more slowly than the all-set configuration. Thus the pre-copy migration is able to be completed. We leverage the real-time throughput and the average latency measured by memaslap to represent the performance of the Memcached server. The results are shown in Figs. 8 and 9. Both of the measurements last for 180 s, while the migrations begin at the 60 s. The dashed boxes indicate the durations of the migrations. It is demonstrated that, pre-copy has no obvious impact on the performance of the workload. However, during almost the whole period of post-copy, the throughput and average latency of Memcached are both degraded dramatically. Besides, the performance recovers very slowly as pages gradually arrive at the target node. This result verifies that, compared with the pre-copy, post-copy does have severe impact on the performance of applications running inside the VMs.

Due to post-copy's weaknesses in reliability and performance, in our hybrid method, we try to shorten the duration of post-copy

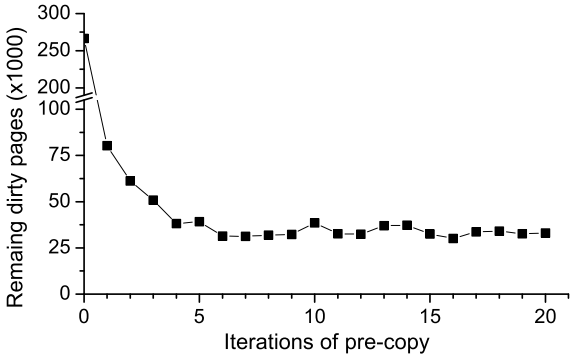


Fig. 10. A typical condition for the amount of remaining dirty pages for memory write-intensive workloads. The VM running the kernel compilation is migrated.

as much as possible. So in our hybrid migration design, it is crucial to pick a suitable moment to switch from pre-copy to post-copy. On one hand, we would like to obtain the shortest duration of post-copy; On the other hand, we do not expect to perform too many useless pre-copy iterations, because they waste time and network resources. Fig. 10 shows the amount of the remaining dirty pages at the end of each iteration when a VM running the kernel compilation is migrated with pre-copy. This represents a typical condition when the memory write-intensive workloads are

migrated. Usually, the first several iterations gradually decrease the amount of the remaining dirty pages, until a turning point occurs. After this turning point, the amount of the remaining dirty pages is not further reduced obviously, and fluctuates within a range.

Based on this observation, we propose the design of our intelligent hybrid migration as follows. First, at the end of each iteration, our hybrid method compares the numbers of pages dirtied and sent during this iteration. The turning point mentioned above is detected when the pages dirtied are not less than the pages sent. The migration could be switched into post-copy just at this moment. However, the random fluctuation in the amount of the remaining dirty pages indicates that, waiting a few more iterations may lead to a shorter post-copy duration, with acceptable increase in the total migration time. So after this turning point, our intelligent hybrid method continues the pre-copy process and records the numbers of the pages dirtied in each iteration to search for a local minimum. At the end of one iteration, if the number of the remaining dirty pages is the minimum of the recent iterations in a sliding window (e.g. three iterations), the migration is switched into the post-copy phase.

In fact, in the above design, we assume that the workload does not change during VM migration. However, if the workload changes, the above design may not obtain good effect. For example, the amount of dirty pages could increase after the workload changes. Thus the above design may cause both the TMT and the duration of post-copy to be longer than a simple hybrid migration with fixed pre-copy iterations. We leave a better design for such a circumstance as future work. So far we still adopt the above design, because the migration time is usually less than several minutes. We consider the possibility that the workload changes during such a short period as small.

5. Evaluation

We have implemented our proposed scheme in the KVM/QEMU platform. We modify the code of QEMU-2.5.1 to realize the secure hash method and the intelligent hybrid migration. We leverage SHA1 to compute the secure hashes, and adopt three iterations as the sliding window size used for the intelligent hybrid migration.

This section evaluates the performance characteristics of our proposed scheme with the workloads shown in Table 1. The results show that our secure hash method obtains significant migration performance improvement for the workloads with high proportion of fake dirty pages. The workloads which fail to be migrated with the pre-copy now accomplish migration quickly. Furthermore, our intelligent hybrid migration guarantees successes of the migrations for all the workloads. More importantly, compared with the traditional hybrid migrations with fixed switching points, our design remarkably shortens the duration of the post-copy phase, thus alleviate the inherent weaknesses of post-copy effectively.

5.1. Experiment setup

As shown in Fig. 11, our testbed consists of four machines. Both the source and target nodes for the migration are equipped with two quad-core Xeon E5620 2.4 GHz CPUs, 32 GB RAM, and two Intel 82574L Gigabit network interface cards (NICs). They share the storage from another machine through the NFS (Network File System) method. The last machine acts as the client of the Memcached workload to execute the memaslap benchmark. All of the four machines are connected via a Gigabit Ethernet switch. Besides, a separate Gigabit Ethernet exists between the source and target nodes, and is used exclusively for the migration process. The OS of the host machines is Ubuntu-16.04, while the guest OS is Ubuntu-12.04. As mentioned above, Memcached is migrated with the max

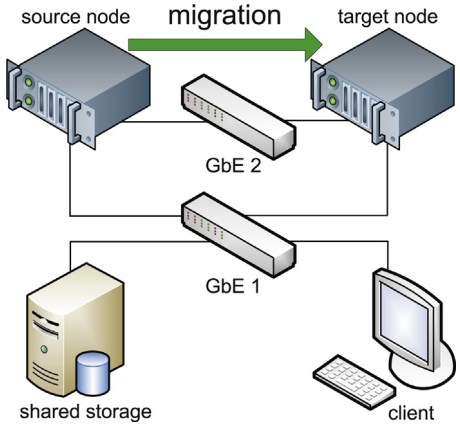


Fig. 11. The testbed of our evaluation. Note that the one client machine runs the memaslap benchmark with concurrency, so it simulates many clients executed against Memcached simultaneously.

bandwidth of our Gigabit Ethernet, which is about 117 MB/s. The other workloads are migrated with the default bandwidth setting in QEMU, which is 32 MB/s. Besides, the maximum expected downtime is 300 ms, as the default setting in QEMU.

5.2. Effects of the secure hash method

Firstly, we conduct experiments to verify the effects of using secure hashes to avoid transferring the fake dirty pages. Our scheme is compared with the original pre-copy and pre-copy with the XBZRLE[2] feature. The XBZRLE cache size for Memcached is 1 GB, while those for the other workloads are all 256 MB.

With our secure hash method, four workloads (Memcached, zeusmp, mcf and bzip2) successfully complete the pre-copy migration. However, the other four workloads (cactusADM, milc, lbm and kernel compilation) still fail to be migrated, no matter using secure hashes or the XBZRLE feature. This result conforms to the proportions of the fake dirty pages shown in Fig. 3. Memcached, zeusmp, mcf and bzip2 have higher proportion than the other four workloads, so they obtain more performance improvements. Fig. 12 shows the results of the TMT, TDT and downtime for Memcached, zeusmp, mcf and bzip2. Compared with the original pre-copy, our secure hash method obtains significant performance improvement for all the three metrics. Three workloads (Memcached, zeusmp and mcf), which fail to be migrated with the original pre-copy, accomplish migration with a TMT of 58 s, 27 s and 98 s respectively. Bzip2 also obtains great benefits, as the TMT and TDT are both reduced a lot. Besides, the XBZRLE feature also gets performance improvement, compared with the original pre-copy, except that zeusmp still does not complete migration. However, our secure hash method performs much better than the XBZRLE feature with the given cache size. Besides, our method consumes negligible memory overhead, compared with XBZRLE. So we conclude that our method that leverages the secure hashes to avoid transferring the fake dirty pages is effective.

Different cache sizes of XBZRLE would lead to various migration performance. So we further repeat the experiments using XBZRLE with different cache sizes. As is shown in Table 4, a larger cache size does obtain more benefits for migration, while also consuming more memory overhead. Through this table, we compare our

[2] The current implementation of the XBZRLE feature in QEMU starts to insert pages into the cache since the second iteration of pre-copy. This results in that the dirty pages start to be delta-compressed from the third iteration. We modify the code to start inserting from the first iteration. Thus the second iteration also obtains the benefits of the XBZRLE.

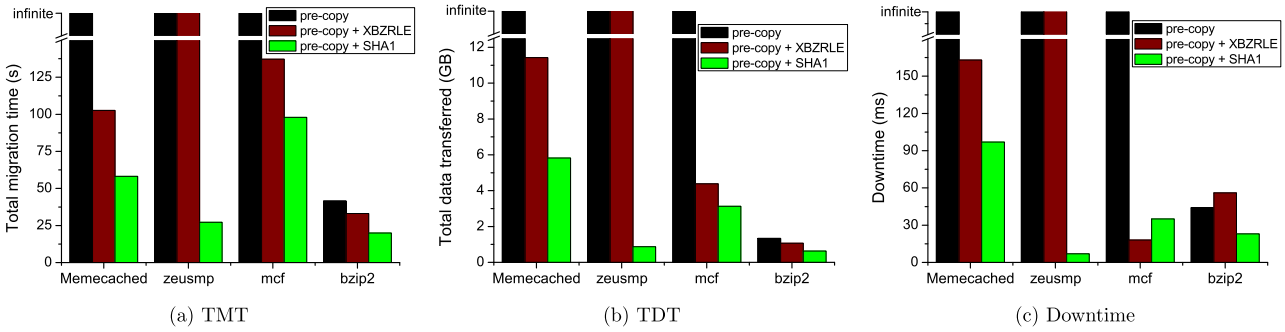


Fig. 12. Migration performance for different methods. The results for cactusADM, milc, lbm and kernel compilation are not shown, because they still fail to be migrated, no matter using secure hashes or the XBZRLE feature.

Table 4
Migration performance for XBZRLE with different cache sizes. The dash lines indicate that migration fails to be completed.

	TMT (s)					TDT (MB)					Downtime (ms)				
	XBZRLE				hash	XBZRLE				hash	XBZRLE				hash
(cache size)	1 G	2 G	4 G	8 G[a]		1 G	2 G	4 G	8 G		1 G	2 G	4 G	8 G	
Memcached	102.7	83.6	64.3	47.3	58.2	11427	9209	6909	4961	5821	163	101	61	41	97
(cache size)	128 M	256 M	512 M	1 G		128 M	256 M	512 M	1 G		128 M	256 M	512 M	1 G	
zeusmp	–	–	24.9	22.6	27.2	–	–	795	722	869	–	–	31	5	7
mcf	–	137.2	76.1	48.5	97.9	–	4386	2425	1543	3128	–	18	120	51	35
bzip2	34.6	33.1	22.7	18.4	19.9	1109	1065	731	590	631	173	56	28	9	23

[a]QEMU requires that the XBZRLE cache size must be a power of 2, and that it must not be larger than the guest memory size. We modify the code to cancel the upper limit of the cache size. Thus the 8 GB cache is able to hold all the 6 GB VM memory pages for Memcached.

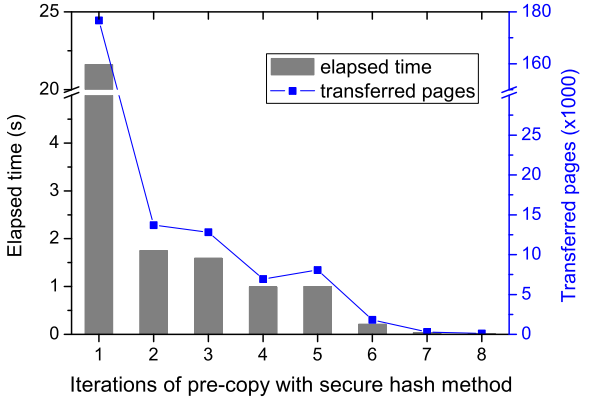


Fig. 13. The elapsed time and amount of the transferred pages for each pre-copy iteration with our secure hash method. The VM running zeusmp is migrated.

secure hash method with XBZRLE in more detail. Take Memcached as example, it is shown that our secure hash method outperforms XBZRLE even with a 4 GB cache. However, when the cache size is expanded as large as the whole guest memory size, XBZRLE performs better. This is because XBZRLE with this cache size not only skips the fake dirty pages, but also performs delta compression on every page to be sent against their old copies, from the second iteration. In conclusion, considering both the performance improvement and the overheads consumed, our secure hash method outperforms XBZRLE significantly.

Furthermore, we record the elapsed time and amount of the transferred pages for each iteration with our secure hash method, for the zeusmp workload. The results are shown in Fig. 13. Compared with the results in Fig. 1, it shows how the secure hash method obtains the benefits in detail. In the second iteration, the amount of the transferred pages is reduced from 54,639 to 13,713, because approximately 75% of the 54,639 pages are fake dirty. Due to the significant reduction in pages to be transferred, the second iteration is shortened from 6.70 s to 1.75 s. An additional effect to shorten one iteration is that, the amount of the dirty pages generated during this iteration is also reduced. This additional effect speeds up the convergence of the pre-copy process.

5.3. Computation overhead of the secure hash method

The computation speed for the SHAx hashes is related with the length of the messages to generate hashes. Fig. 14 shows the throughput to compute the SHAx hashes for the 4 KB memory pages, using one core of the quad-core Xeon E5620 2.4 GHz CPU. It is demonstrated that the throughput to compute anyone of the SHA1, SHA256 or SHA512 hash is much higher than the maximum bandwidth of our Gigabit Ethernet (<125 MB/s). So to use anyone of the SHAx functions is fast enough for our migration system.

However, with faster networks in recent data centers, e.g., 10 GbE, the throughput shown in Fig. 14 is relatively slow. Considering that the Xeon E5620 CPU in our testbed is a little outdated, we also compute the SHA1 throughput on a server with more advanced Xeon E3-1225 v6 3.3 GHz CPU, and get a much faster result of 1.02 GB/s. Besides, with the availability of multi-core or many-core, CPU resources tend to be abundant in current data centers. So the calculation of secure hashes could be speeded up with multi threads. Furthermore, processor vendors are offering some hardware acceleration and new instruction supports (e.g. the Intel QuickAssist Technology [37] and the Intel SHA Extensions [38]) for more efficient SHA computing performance. These techniques would keep computing secure hashes from being a bottleneck in faster networks.

Besides the throughput, we are more interested in how much extra CPU consumption our secure hash method incurs, compared with the original pre-copy migration. In the source code of KVM/QEMU, the migration routine is implemented as an individual thread, separated from the main virtualization thread. So after obtaining the PID (process/thread ID in Linux system) of the migration thread, we are able to monitor its CPU utilization through the Linux "top" command. As is shown in Fig. 15, for the VM running zeusmp, our secure hash method increases the average CPU utilization of the migration thread from 3.7% to 14.9%, which is

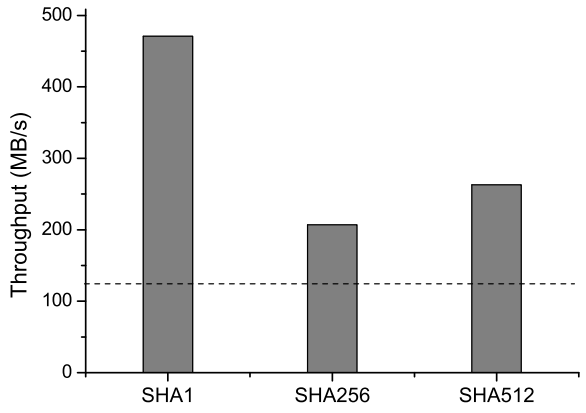


Fig. 14. The throughput to compute the SHAx hashes for the 4 KB memory pages. The dashed line indicates the theoretical maximum bandwidth of our Gigabit Ethernet. The machine for this test is equipped with quad-core Xeon E5620 2.4 GHz CPUs. Note that only one core is used to compute the SHAx hashes.

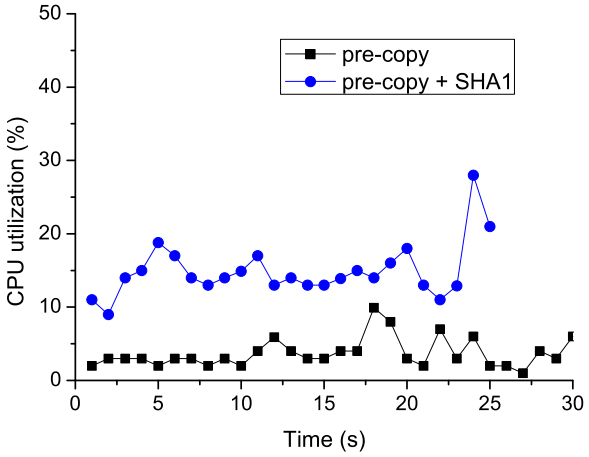


Fig. 15. The CPU consumption of the migration thread with and without the secure hash method. The VM running zeusmp is migrated. The CPU utilization is obtained once per second from the host Linux system.

acceptable. So we consider the CPU overhead spent on computing the secure hashes as insignificant.

5.4. Effects of the intelligent hybrid migration

Next, we further conduct experiments to evaluate the effects of our intelligent hybrid migration. The four workloads with high proportion of fake dirty pages (Memcached, zeusmp, mcf and bzip2) have already finished migration quickly with the secure hash method, so they would not step into the post-copy phase in our intelligent hybrid design. In contrast, the other four workloads (cactusADM, milc, lbm and kernel compilation) step into post-copy at the end of the 4th, 6th, 7th and 7th pre-copy iteration respectively. So we only show the results of these four workloads in this subsection. Besides, as baseline, we also migrate them using hybrid migrations with fixed switching points (at the end of the 1st/2nd pre-copy iteration). Fig. 16 shows the results of the duration of the post-copy phase. The result of a pure post-copy without preliminary pre-copy iterations is also shown in this figure. It is demonstrated that our intelligent hybrid method remarkably shortens the post-copy duration compared with the baseline, except cactusADM. For example, compared with the hybrid migration with one iteration, our method shortens the post-copy duration by an extent of 44%, 43%, and 60% respectively for

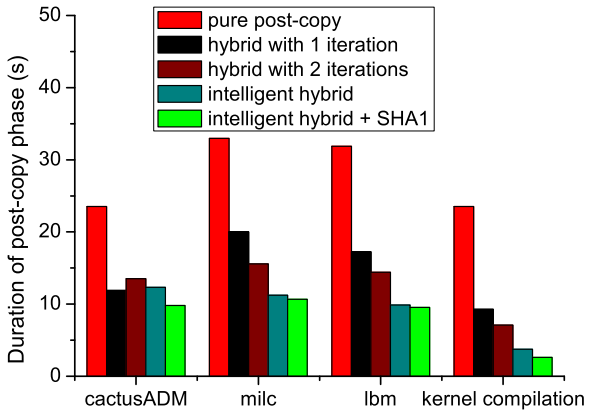


Fig. 16. The post-copy duration for pure post-copy and different hybrid methods.

Table 5
The amounts of the page faults during post-copy phase for the baseline (hybrid migration with one iteration) and our intelligent hybrid migration.

	cactusADM	milc	lbm	kernel
Baseline	207	6031	3678	364
Intelligent hybrid	216	3512	2241	183

milc, lbm and kernel compilation. However, our method has no better effect for cactusADM. This is because the remaining dirty pages of cactusADM cannot be further reduced and maintain in a relatively constant range from just the second iteration. Thus to perform more pre-copy iterations would not further shorten the post-copy duration. Besides, it is shown that the secure hash method further reduces the post-copy duration, because fake dirty pages, although low proportion, are kept from being resent in post-copy phase.

As shown in Figs. 8 and 9, post-copy damages the VM application performance severely, while pre-copy has no obvious impact. So our intelligent hybrid migration reduces the impact of migration on VM application performance, given that the duration of post-copy is shortened remarkably, even though the pre-copy period could be prolonged. Besides, page faults are the immediate causes to damage application performance during post-copy. So we record the amount of page faults for different hybrid methods, to obtain a more comprehensive knowledge of the impact of migration on VM application performance. As is shown in Table 5, compared with the baseline, our scheme reduces the amount of page faults remarkably by an extent from 39% to 50%, except cactusADM. This result further verifies the efficiency of our intelligent hybrid migration to reduce the impact on VM application performance. For cactusADM, as we have explained in the above paragraph, our scheme could not further shorten the post-copy duration. So the amount of page faults is not reduced either.

However, if the network-bound applications are migrated, our method may cause more performance degradation, especially when the application and migration traffics compete for the same network infrastructure. This is because, unlike what is shown in Fig. 8, the network contention would degrade the application performance during pre-copy. Besides, the pre-copy phase is also prolonged by the contention. For this situation, it is better to finish the migration quickly, so simple hybrid migration with fixed iterations could outperform our design.

As our intelligent hybrid method performs more pre-copy iterations to search for a near-optimal switching moment, the TMT and TDT are increased compared with the baseline. This is demonstrated by Figs. 17 and 18. For example, compared with hybrid migration with one iteration, our scheme transfers 58% more data for

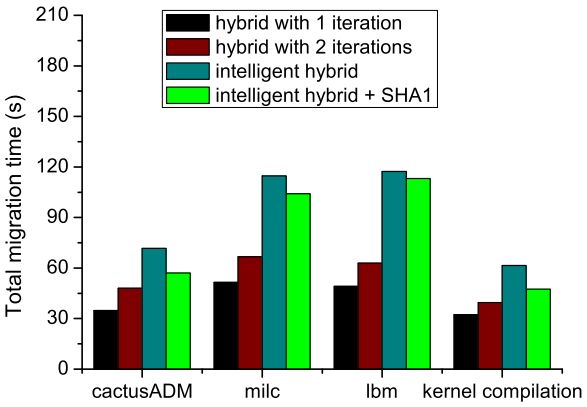


Fig. 17. The TMT for different hybrid methods.

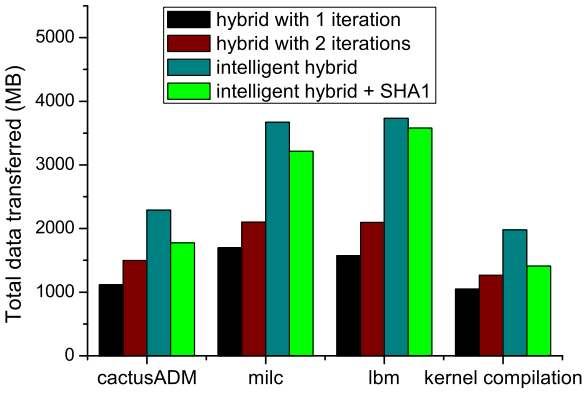


Fig. 18. The TDT for different hybrid methods.

cactusADM. The primary objective of our design is to shorten the post-copy duration, and thus to alleviate the inherent weaknesses of post-copy. So we consider the increase in the TMT and TDT as an acceptable trade-off. However, if the primary goal is to shorten the migration time or to reduce the amount of data transferred, our design may not be preferable. In such a circumstance, hybrid migration with fixed pre-copy iterations or even the pure post-copy is more suitable, even though the inherent weaknesses of post-copy would be more severe.

6. Related work

Optimizations for Pre-copy Migration. Due to its importance for the data center managements, live VM migration has been widely researched for many years since first proposed by Clark et al. [2] and Nelson et al. [3]. A number of techniques have been proposed to optimize the pre-copy migration. Jin et al. [39] improve the pre-copy process by compressing the memory data transferred from the source to the target node. Liu et al. [40] adopt checkpointing/recovery and trace/replay to speed up the convergence of the pre-copy migration. Since the second iteration, rather than the memory data, they send the log files, which record the non-deterministic system events for the previous iteration, to the target node. Then the target node replays with the received log files. Because the log file size is much smaller than the memory data size, their approach leads to great performance improvement for migration. However, this approach is not suitable to migrate the VMs with SMP OS in the multi-processor environment, because expensive memory race among different VCPUs must be recorded and replayed. Jo et al. [41] observe that modern operating systems often use part of the physical memory to cache data from the secondary storage. So in the scenes of migrations with shared storage, they propose to track the VM's I/O operations and maintain a mapping of the memory pages that are in identical form with the data on the storage device. During the migration, the target node directly fetches this part of identical data from the shared storage device. Thus the TMT is shortened remarkably. Song et al. [42] study the parallelization opportunities of live migration. They propose the PMigrate, which leverages data parallelism and pipeline parallelism to parallelize the migration operation. Their prototypes show that PMigrate accelerates the live migration significantly, while consuming more CPU cores and NIC ports for the parallelization. Ibrahim et al. [10] optimize the migration for HPC applications, which are usually memory write-intensive. To guarantee the convergence of pre-copy, they propose to adjust the migration bandwidth limit and downtime setting according to the memory dirtying rate of the applications. However, their approach may lead to a long downtime, while our scheme in this paper completes the migration while keeping a short downtime.

WAN Migration. Besides the live migration in a LAN scope, the WAN migration without a shared storage has also been studied. In the scenes of WAN migration, the VM storage states along with the memory pages need to be transferred. CloudNet [43] proposes a cloud framework that supports efficient WAN migration with VPN based network infrastructure. It presents a set of effective optimizations for migration. They reduce the cost of transferring VM storage and memory data over low bandwidth and high latency network links. Zheng et al. [44] propose to record a history of the VM disk I/O operations, and use this history to predict the I/O locality characteristics of the migrated workload. With this locality characteristics, they design a storage migration scheduling algorithm. This algorithm efficiently alleviates the VM I/O performance degradation during migration. XvMotion [15] is a WAN migration system of VMware. It builds on the live migration and IO mirroring mechanisms in ESX [13]. The migration process of XvMotion first copies the disk data with a single pass. After that, it iteratively copies the memory data with pre-copy. During the whole migration period, the asynchronous IO mirroring is performed, which reflects any additional changes of the source disk to the destination. Besides, XvMotion has been in active use by customers for several years with good effects. SnapMig [45] proposes a VM storage migration design with the snapshot backup servers. During the storage migration, the task of transferring VM base image and snapshots is outsourced to the backup servers. Thus only little data needs to be copied from the source node. This scheme mitigates the interference between VM I/O requests and migration I/O requests.

Memory Deduplication in Migration. It seems that the existing works employing memory deduplication to accelerate VM migration [46–49] cover the problem of fake dirty pages. In fact, most of these works perform deduplication while simultaneously migrating multiple VMs co-located on the same host or within a cluster [46–49]. Deduplication is effective in these scenarios, because these VMs often have some common memory content, given that they may execute the same operating system, libraries, and applications. However, within a single VM, excluding the fake dirty pages discussed in this paper, the amount of duplicate pages is very small. This is verified by the experiment results shown in Table 6. We dump the memory content (excluding zero pages) of VMs running different workloads info files. Then we analyze these memory dump files and compute the percentage of duplicate pages. It is demonstrated that, for most workloads except cactusADM, the percentage of duplicate pages is less than 10%. So we deduce that, without the fake dirty pages, we can only obtain very little performance improvement by applying memory deduplication into single VM migration.

Zhang et al. [50] leverage delta compression to accelerate VM migration. They use the HashSimilarityDetector [51] to find memory pages similar in content, including the identical pages. Then

Table 6
The percentage of duplicate pages for single VM running different workloads.

	zeusmp	mcf	bzip2	cactusADM	milc	lbm	kernel compilation
Duplicate pages	7.77%	1.12%	9.36%	19.65%	2.07%	6.57%	6.09%

they perform XBZRLE on these pages against their reference pages. So at the source node, they need to cache the reference pages in memory, while our scheme does not have this memory overhead.

According to a study by IBM [52], a VM typically moves among a very limited set of hosts. Even it is often migrated back and forth between just two hosts. Aiming at these scenes, VeCycle [53] proposes that the migration source stores a checkpoint of the outgoing VM. When the VM is migrated back, the incoming migration could be speeded up with the old checkpoint. Particularly, VeCycle computes the MD5 checksum of one page, and does not send it if its checksum is within the checksum set of the old checkpoint. As a result, VeCycle reduces the migration time and traffic remarkably, especially when there is high similarity between the VM's current state and old checkpoint. Although the checksum method in VeCycle is similar to our secure hash method, they are proposed for different purposes. VeCycle aims at identifying duplicate pages already existing in the checkpoint, which are mainly pages not updated between the two migrations, while our purpose is to recognize the fake dirty pages, which are updated in fact. Besides, VeCycle identifies duplicate pages when the same contents exist anywhere in the checkpoint, while we only recognize fake dirty pages with the same page frame number. So in fact VeCycle could find more duplicate pages than our method. However, VeCycle only performs checksum comparisons during the first iteration, because they consider it unlikely that a page updated between iterations matches a page already present at the destination. In fact, our analysis indicates that this is possible because of the widely existing fake dirty pages, so we perform the secure hash method throughout the whole migration process.

7. Conclusion

In this paper, we try to solve the challenging problem of migrating the VMs running the memory write-intensive workloads. Our analysis first demonstrates an important fact: during the pre-copy migrations, some of these workloads have a large proportion of fake dirty pages. In fact, these pages are unnecessary and wasteful to be resent. Then we explore how these fake dirty pages are generated. After that, we propose to leverage the computation and comparison of the secure hashes to avoid transferring these fake dirty pages. Besides, to guarantee the successful accomplishment of migration, we further propose an intelligent hybrid migration design. It automatically switches from pre-copy to post-copy at a near-optimal moment, when taking more pre-copy iterations would not further reduce the amount of the dirty pages. Our evaluations show that the proposed scheme gets a significant performance improvement for VM migration. The workloads which fail to be migrated with pre-copy now complete migration quickly, with a TMT from 27 s to 98 s. Besides, the intelligent hybrid migration remarkably shortens the post-copy duration, thus effectively alleviate the inherent weaknesses of post-copy.

Acknowledgments

The authors are grateful to the anonymous reviewers for their valuable comments and feedback. This work was partly supported by the National Natural Science Foundation of China No. 61821003, No. U1705261, No. 61832007, No. 61772222 and No. 61772212; National Key R&D Program of China No. 2018YFB10033005; Shenzhen Research Funding of Science and Technology, China JCYJ20170307172447622.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: Proc. the 19th ACM Symposium on Operating Systems Principles (SOSP), Oct. 2003, pp. 164–177.
[2] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines. In Proc. the 2nd conference on Symposium on Networked Systems Design and Implementation (NSDI), May 2005, pp. 273–286.
[3] M. Nelson, B. Lim, G. Hutchins, Fast transparent migration for virtual machines, in: Proc. USENIX Annual Technical Conference, Apr. 2005, pp. 391–394.
[4] P. Padala, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, Adaptive control of virtualized resources in utility computing environments, in: Proc. the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), Mar. 2007, pp. 289–302.
[5] N. Bobroff, A. Kochut, K. Beaty, Dynamic placement of virtual machines for managing sla violations, in: Proc. the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM), 2007, pp. 119–128.
[6] Z. Zheng, M. Li, X. Xiao, J. Wang, Coordinated resource provisioning and maintenance scheduling in cloud data centers, in: Proc. IEEE International Conference on Computer Communications, Apr. 2013, pp. 345–349.
[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, A. Warfield, Remus: High availability via asynchronous virtual machine replication, in: Proc. the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Apr. 2008, pp. 161–174.
[8] A.B. Nagarajan, F. Mueller, C. Engelmann, S.L. Scott, Proactive fault tolerance for HPC with Xen virtualization, in: Proc. the 21st annual International Conference on Supercomputing (ICS), Jun. 2007, pp. 23–32.
[9] R. Nathuji, K. Schwan, VirtualPower: Coordinated power management in virtualized enterprise systems, in: Proc. the 21st ACM Symposium on Operating Systems Principles (SOSP), Oct. 2007, pp. 265–278.
[10] K.Z. Ibrahim, S. Hofmeyr, C. Iancu, E. Roman, Optimized pre-copy live migration for memory intensive applications, in: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Nov. 2011, pp. 1–11.
[11] J. Zhang, E. Dong, J. Li, H. Guan, MigVisor: Accurate prediction of VM live migration behavior using a working-set pattern model, in: Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Apr. 2017, pp. 30–43.
[12] P. Svärd, B. Hudzia, J. Tordsson, E. Elmroth, Evaluation of delta compression techniques for efficient live migration of large virtual machines, in: Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2011, pp. 111–120.
[13] A. Mashtizadeh, E. Celebi, T. Garfinkel, M. Cai, The design and evolution of live storage migration in VMware ESX, in: Proc. USENIX Annual Technical Conference (ATC), Jun. 2011, pp. 1–14.
[14] H. Jin, W. Gao, S. Wu, X. Shi, X. Wu, F. Zhou, Optimizing the live migration of virtual machine by CPU scheduling, J. Netw. Comput. Appl. 34 (4) (2011) 1088–1096.
[15] A.J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, S. Setty, Xvmotion: Unified virtual machine migration over long distance, in: Proc. USENIX Annual Technical Conference (ATC), Jun. 2014, pp. 97–108.
[16] K.M. Lepak, M.H. Lipasti, On the value locality of store instructions, in: Proc. the 27th International Symposium on Computer Architecture (ISCA), Jun. 2000, pp. 182–191.
[17] G.B. Bell, K.M. Lepak, M.H. Lipasti, Characterization of silent stores, in: Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 2000, pp. 133–142.
[18] K.M. Lepak, Lipasti M.H., Silent stores for free, in: Proc. International Symposium on Microarchitecture (MICRO), Dec. 2000, pp. 22–31.
[19] C. Molina, A. Gonzlez, J. Tubella, Reducing memory traffic via redundant store instructions, in: Proc. International Conference on High Performance Computing and Networking, Apr. 1999, pp. 1246–1249.
[20] M.R. Hines, U. Deshpande, K. Gopalan, Post-Copy live migration of virtual machines, Oper. Syst. Rev. 43 (3) (2009) 14–26.
[21] Y. Abe, R. Geambasu, K. Joshi, M. Satyanarayanan, Urgent virtual machine eviction with enlightened post-copy, in: Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Apr. 2016, pp. 51–64.
[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: The Linux virtual machine monitor, in: Proc. Linux symposium, Jun. 2007, pp. 225–230.
[23] Fabrice B. Qemu, A fast and portable dynamic translator, in: Proc. USENIX Annual Technical Conference, Apr. 2005, pp. 41–46.

[24] VMware, 2017, http://www.vmware.com, accessed July 2017.
[25] P. Lu, A. Barbalace, B. Ravindran, HSG-LM: Hybrid-Copy Speculative Guest OS Live Migration without Hypervisor, in: Proc. the 6th International Systems and Storage Conference (SYSTOR), Jun. 2013, pp. 1–11.
[26] W. Zhang, K.T. Lam, C. Wang, Adaptive live VM migration over a WAN: Modeling and implementation, in: Proc. the 7th IEEE International Conference on Cloud Computing, Jun. 2014, pp. 368–375.
[27] U. Deshpande, D. Chan, T. Guh, J. Edouard, K. Gopalan, N. Bila, Agile live migration of virtual machines, in: Proc. the IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2016, pp. 1061–1070.
[28] SPEC CPU 2006 benchmark, 2017, https://www.spec.org/cpu2006/, accessed July 2017.
[29] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, D.K. Panda, High Performance MPI Library over SR-IOV Enabled InfiniBand Clusters, in: Proc. the 21st International Conference on High Performance Computing (HiPC), Dec. 2014, pp. 1–10.
[30] J. Zhang, X. Lu, J. Jose, R. Shi, D.K. Panda, Can Inter-VM Shmem Benefit MPI Applications on SR-IOV based Virtualized InfiniBand Clusters? in: Proc. European Conference on Parallel Processing (Euro-Par), Aug. 2014, pp. 342–353.
[31] Amazon EC2 X1e Instances, 2018, https://aws.amazon.com/cn/ec2/instance-types/x1e/, accessed Oct. 2018.
[32] S. Quinlan, S. Dorward, Venti: A new approach to archival storage, in: Proc. Conference on File and Storage Technologies (FAST), Jan. 2002, pp. 89–101.
[33] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, J. Kunkel, A study on data deduplication in HPC storage systems, in: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Nov. 2012, pp. 1–11.
[34] B. Schroeder, E. Pinheiro, W.D. Weber, DRAM errors in the wild: A large-scale field study, in: Proc. the 11th International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), Jun. 2009, pp. 193–204.
[35] A.A. Hwang, I. Stefanovici, B. Schroeder, Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design, in: Proc. the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Mar. 2012, pp. 111–122.
[36] XBZRLE (Xor Based Zero Run Length Encoding), 2017, https://github.com/qemu/qemu/blob/master/docs/xbzrle.txt, accessed July 2017.
[37] Intel QuickAssist Technology, 2018, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html, accessed April 2018.
[38] Intel SHA Extensions, 2018, https://software.intel.com/en-us/articles/intel-sha-extensions, accessed April 2018.
[39] H. Jin, L. Deng, S. Wu, X. Shi, X. Pan, Live virtual machine migration with adaptive memory compression, in: Proc. the IEEE International Conference on Cluster Computing and Workshops (CLUSTER), Sept. 2009, pp. 1–10.
[40] H. Liu, H. Jin, X. Liao, L. Hu, C. Yu, Live migration of virtual machine based on full system trace and replay, in: Proc. the 18th ACM International Symposium on High Performance Distributed Computing (HPDC), Jun. 2009, pp. 101–110.
[41] C. Jo, E. Gustafsson, J. Son, B. Egger, Efficient live migration of virtual machines using shared storage, in: Proc. the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2013, pp. 41–50.
[42] X. Song, J. Shi, R. Liu, J. Yang, H. Chen, Parallelizing live migration of virtual machines, in: Proc. the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2013, pp. 85–95.
[43] T. Wood, P. Shenoy, K. Ramakrishnan, J. Merwe, CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines, in: Proc. the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2011, pp. 121–132.
[44] J. Zheng, Ng T. Eugene, K. Sripanidkulchai, Workload-aware live storage migration for clouds, in: Proc. the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Mar. 2011, pp. 133–144.
[45] Y. Yang, B. Mao, H. Jiang, Y. Yang, H. Luo, S. Wu, SnapMig: Accelerating VM live storage migration by leveraging the existing VM snapshots in the cloud, IEEE Trans. Parallel Distrib. Syst. (2018).
[46] U. Deshpande, X. Wang, K. Gopalan, Live gang migration of virtual machines, in: Proc. the 20th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Jun. 2011, pp. 135–146.
[47] P. Riteau, C. Morin, T. Priol, Shrinker: Improving live migration of virtual clusters over WANs with distributed data deduplication and content-based addressing, in: Proc. the 17th International European Conference on Parallel and Distributed Computing (Euro-Par), Aug. 2011, pp. 431–442.
[48] U. Deshpande, B. Schlinker, E. Adler, K. Gopalan, Gang migration of virtual machines using cluster-wide deduplication, in: Proc. the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID), May 2013, pp. 394–401.
[49] U. Deshpande, D. Chan, S. Chan, K. Gopalan, N. Bila, Scatter-gather live migration of virtual machines, IEEE Trans. Cloud Comput. 6 (1) (2018) 196–208.
[50] X. Zhang, Z. Huo, J. Ma, D. Meng, Exploiting data deduplication to accelerate live virtual machine migration, in: Proc. the IEEE International Conference on Cluster Computing (CLUSTER), Sept. 2010, pp. 88–96.
[51] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, A. Vahdat, Difference engine: Harnessing memory redundancy in virtual machines, in: Proc. the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Dec. 2008, pp. 309–322.
[52] R. Birke, A. Podzimek, L.Y. Chen, E. Smirni, State-of-the-Practice in data center virtualization: Toward a Better Understanding of VM Usage, in: Proc. the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Jun. 2013, pp. 1–12.
[53] T. Knauth, C. Fetzer, VeCycle: Recycling VM checkpoints for faster migrations, in: Proc. the 16th Annual Middleware Conference (Middleware), Dec. 2015, pp. 210–221.

Chunguang Li is currently working toward the Ph.D. degree majoring in computer architecture at Huazhong University of Science and Technology, Wuhan, China. His research interests include virtualization, cloud computing, etc. He has published several papers in conferences including INFOCOM, HotStorage, etc.

Dan Feng received the BE, ME, and Ph.D. degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), Wuhan, China. She is a professor and dean of School of Computer, HUST. Her research interests include computer architecture, massive storage systems, parallel file systems, etc. She has many publications in major journals and international conferences, including IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, the ACM Transactions on Storage, FAST, USENIX ATC, HPDC, SC, ICS, IPDPS, etc.

Yu Hua received the BE and Ph.D. degrees in Computer Science and Technology in 2001 and 2005, respectively, from Wuhan University, Wuhan, China. He is a professor now at Huazhong University of Science and Technology. His research interests include computer architecture, cloud computing, network storage, etc. He has many papers published in major journals and international conferences including IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, USENIX ATC, USENIX FAST, SC, INFOCOM, ICDCS, ICPP, etc.

Leihua Qin received the BE degree in Physics in 1991 from HuBei University and the ME and Ph.D. degrees in Computer Science and Technology in 2003 and 2007, respectively, from Huazhong University of Science and Technology. He is a professor and vice dean of School of Computer, HUST. His research interests include computer architecture and network simulations. He has publications in major journals including Journal of Supercomputing, Journal of Internet Technology, etc.