PARS: A Page-Aware Replication System for Efficiently Storing Virtual Machine Snapshots

Lei Cui, Tianyu Wo, Bo Li, Jianxin Li, Bin Shi, Jinpeng Huai
SKLSDE lab, Beihang University, China.
{cuilei, woty, libo, lijx, shibin, huaijp}@act.buaa.edu.cn

Abstract

Virtual machine (VM) snapshot enhances the system availability by saving the running state into stable storage during failure-free execution and rolling back to the snapshot point upon failures. Unfortunately, the snapshot state may be lost due to disk failures, so that the VM fails to be recovered. The popular distributed file systems employ replication technique to tolerate disk failures by placing redundant copies across disperse disks. However, unless user-specific personalization is provided, these systems consider the data in the file as of same importance and create identical copies of the entire file, leading to non-trivial additional storage overhead.

This paper proposes a page-aware replication system (PARS) to store VM snapshots efficiently. PARS employs VM introspection technique to explore how a page is used by guest, and classifies the pages by their importance to system execution. If a page is critical, PARS replicates it multiple copies to ensure high availability and long-term durability. Otherwise, the loss of this page causes no harm for system to work properly, PARS therefore saves only one copy of the page. Consequently, PARS improves storage efficiency without compromising availability. We have implemented PARS to justify its practicality. The experimental results demonstrate that PARS achieves 53.9% space saving compared to the native replication approach in HDFS which replicates the whole snapshot file fully and identically.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Reliability, availability, and serviceability; D.4.5 [*Organization and Design*]: Checkpoint/restart

Keywords Replication; Virtual machine snapshot; Availability; Storage space saving; Introspection

VEE '15, March 14–15, 2015, Istanbul, Turkey..
Copyright © 2015 ACM 978-1-4503-3450-1/15/03. . . $15.00.
http://dx.doi.org/10.1145/2731186.2731190

1. Introduction

Virtualization has become the cornerstone of cloud infrastructures. The applications now run inside the virtual machine (VM) which provides an isolated and scaled computing paradigm, rather than physical machine [9, 12, 28, 29]. The VM snapshot is an essential part to support high availability in cloud infrastructures, it saves the running state of VM into persistent storage and recovers the VM from the saved state upon failures. The main advantage is that the VM can be rolled back to an intermediate state rather than the initial state, thereby reducing the computation loss [20, 32, 43].

However, storing snapshots in persistent storage faces several challenges. First, unlike process snapshot which only saves the memory footprint of the process, VM snapshot involves the entire memory state and thus is heavy. Consider that the virtual machines nowadays are always configured with several GB RAM, the snapshot size is up to GB accordingly. Second, the cloud infrastructure provides thousands of VMs to end-users, and each user may create dozens of snapshots, which will cost hundreds of TB storage totally. Third, disk failures are common nowadays [24, 46, 47], they lead to the loss of snapshot files resident in the disk so that the VM cannot be successfully recovered. This promotes the administrators to employ redundancy mechanism [26, 39, 45, 51] to provide high availability as well as long-term durability, however, at the cost of extra storage space. Overall, storing large amount of virtual machine snapshots in a space-efficient way without compromising availability is critical in large scale cloud infrastructures.

Several works have recognized this problem and proposed approaches which fall into two main categories. One category focuses on reducing the snapshot size when the VM snapshot is being taken. For example, they compress the memory state [30], save each memory page only once in post-copy manner rather than save iteratively [20, 49], or remove the data that exist in both memory pages and disk blocks[14, 43]. De-duplication technique is employed to save only one copy of the identical pages across multiple VM snapshots [21, 22, 42]. However, these works pay no attention to the availability of snapshot files, or in other words, they expect the underlying file system or storage device to provide the availability. Another category of approach, in contrast, attempts to improve availability to prevent data loss from disk failures. They either replicate multiple copies of the full file [26, 55], or utilize erasure code [19, 36, 38] to enhance availability with less storage space. Unfortunately, these methods lack the exploitation of file semantics, and consider that the data in the file have the same importance, so that they replicate the files identically and blindly.

In this paper, we investigate the problem of storing virtual machine snapshots in a distributed file system with the aim to achieve high availability with low storage usage, and propose a Page Aware Replication System, named as "PARS". The key insight behind PARS is the different page importance to system execution. Modern operating systems allocate most of the memory to a cache which stores data that has been recently read (or is to be written back), so that future requests for that data can be served faster than accessing external devices directly. It suggests that these (or at least a part of) cache pages are unnecessary for the guest OS and applications to work properly. In other words, the loss of them after recovery will not cause system crash, but only imposes performance loss. This inspires us to explore the page semantics related to system execution, classify pages by their importance, and then save multiple copies for critical pages while saving less copies for non-critical one, rather than replicate them fully and identically. To summarize, we make the following contributions:

- An analysis of practical memory usage in Windows desktop and Linux server cluster (§3).
- Categorization of memory pages by exploring the importance to system execution, and a page-aware policy to determine the replica count for different categories (§4).
- A mechanism to handle the loss of unnecessary cache pages after rollback recovery (§5).
- Implementation of PARS on QEMU/KVM platform and HDFS (§6), and evaluation under several workloads to prove its effectiveness (§7). The evaluation results show that PARS can reduce storage usage by 53.9% compared to native replication technique in HDFS [3] which replicates three full copies. In addition, PARS reduces 36.4% of snapshot duration without significant performance loss, and could reduce rollback latency by 65.8% for scenarios where performance requirement is not critical.

2. Background and Related Work

2.1 Virtual Machine Snapshot

VM snapshot is a file-based record of the memory state, disk data, and configuration of a virtual machine at a specific time point. It always records the full image of the in-memory state and saves the disk content in an incremental manner since the disk data updates much slowly. VM snapshot is essential for cloud management [10, 20, 32]. For virtual desktop usage, the users can backup the running system and resume the work whenever necessary, or perform intrusion analysis by continuously taking snapshots of the VM under attack. For scientific applications, the computation upon failure can continue from the previously recorded state instead of restarting from the initial state, so that the task can be completed within a limited period in the face of failures.

Although the snapshot mechanism can help the system to survive a majority of failures, the disk failures however may lead to data loss so that the virtual machine fails to be recovered. Distributed file system is an appealing way to provide high availability, it stripes redundant data across multiple disks through techniques such as replication [26, 51, 55] or erasure code [19, 36, 38], so that the data can be properly recovered from failure-free disks. Another benefit is that it employs a large amount of commodity disks to provide higher capacity cheaply. As a result, distributed file systems, such as Google FS [26], HDFS [3], are widely used in cloud infrastructures to store VM images and snapshots.

2.2 Related Work

2.2.1 Snapshot Size Optimization

There exist several optimization approaches which focus on reducing the VM snapshot size when snapshot is being taken. QEMU/KVM uses one byte to represent the entire zero page whose bytes are identical [33]. Jin el al. [30] find that a majority of memory pages have more than 75% similarity, they therefore compress the memory based on strong data regularities. Removing page cache from memory state is one compelling approach to reduce snapshot size effectively, and has been well studied [14, 17, 37, 43, 44]. However, these works still suffer from several drawbacks. Hines et al. [44] leverage balloon mechanism to reclaim temporarily unused memory pages, their method requires the modification and cooperation of guest OS and is not always accepted by users [15]. In addition, it is difficult to determine how many pages should be reclaimed [15]. Chiang et al. [17] remove free pages from snapshot state, but they ignore the removal of cache pages which take a large part of memory state. Park et al. [43] discard the duplicate pages that are resident in both memory and disk blocks, the main disadvantage is that they need to track IO operation to find the unchanged cache pages since guest OS boot up. Consider that snapshot creation is less frequent, their method incur non-trivial IO performance penalty. Koto [37] and Akiyama [14] insert a module into guest kernel to identify the cache pages and then eliminate them during memory state transfer. The modification of guest OS, as mentioned, may be refused by some users.

PARS is inspired by these works but is different. On snapshot, PARS identifies the cache pages and free pages via introspection which requires neither modification of guest OS nor IO tracking [31, 44]. Upon rollback, unlike the above works that reproduce all the cache pages from disk to build a full memory state [14, 37, 43], PARS is able to inform the loss to guest kernel. Thus, the kernel reads the data from disk until they are requested. This ability enables users to load no cache page upon rollback, so that fewer data transfer and faster rollback recovery can be achieved. Similarly, PARS achieves this feature of rollback in a transparent way.

2.2.2 Data Availability

Replication is one well known technique to enhance availability through placing multiple replicas of data in disperse disks. This mechanism has been adopted in many distributed file systems such as Google FS [26], Deep Store [51] and HDFS [3]. However, because the entire file needs to be replicated identically, large amount of storage space will be occupied to store extra replicas. For example, the distributed file systems always set the default replica count to 3, which will cost 2X extra storage. Erasure code splits the file into m stripes, and then codes the stripes to n fragments. The file can be reconstructed from any m of the n fragments. Compared to the replication approach which requires k replicas to survives $k-1$ failures, erasure code can tolerate $n-m$ failures with n/m storage usage. Obviously, appropriate selection of n and m would save storage space for achieving the same availability. Spanner [19], OceanStore [38], Azure Storage [16] and SafeStore [36] are popular distributed file systems which employ erasure code to improve data availability.

Although erasure code is an appealing way to achieve high availability with less storage, it incurs longer latency when recovering the data due to the decoding cost [45]. This implies that it is inappropriate for coding snapshots, because fast VM recovery from snapshot is essential upon failures. Moreover, it tends to reconstruct the entire fragment even if only one block is acquired. This feature is impractical when demand-paging based rollback technique is adopted [52, 53].

PARS employs replication technique to enhance data availability without sacrificing recovery latency. Instead of replicating the entire file, PARS explores the page semantics and saves multiple replicas for a fraction of snapshot state that is critical to system execution, it therefore provides the same availability without unnecessary storage cost.

2.2.3 Storage Space Saving

The existing approaches for space saving in storage system fall into two categories: intra-file compression [40, 41, 51] and inter-file compression [23, 54, 55]. Intra-file compression adopts the compression algorithms such as Lempel-Ziv to express the original data with less data based on statistical characteristics [51]. It can achieve 40% to 70% size reduction, but the drawback is the long access latency as a result of de-compression cost, which is similar to erasure code.

The inter-file compression, also known as de-duplication, merges several identical pages into one copy and several references, thereby reducing the storage usage [21, 27, 42]. The results from Zhang [54] and Nath [42] show that about 40% pages can be merged when large amount of virtual machine snapshots exist. However, inter-file compression does not come for free, the loss of the merged page will lead to the loss of all pages referring to it and consequentially deteriorate data availability [39].

PARS achieves space saving by reducing the replicas for certain pages that are unnecessary for the system to work properly. Without doubt, we believe that PARS is orthogonal to the above methods for saving storage space further.

3. Motivation

3.1 OS Memory Usage

The OS allocates memory for kernel usage and user space usage. The kernel uses memory to store kernel images and kernel objects such as *inode* in file system, *sk_buffer* in TCP/IP stack, *page* in memory management, etc. In user space, the memory are allocated as BSS, Text or Data segments to store the static variables and application codes, or as heap and stack for dynamic objects.

Typically, the OS kernel and user applications always occupy a small part of the entire memory space and leave the remaining memory pages unused which are named as ”free pages”. Free pages mainly consists of two parts: i) the pages that are never allocated since system boot-up and ii) the pages that are ever used but now are freed. Linux kernel always fills the former one with zero, but leaves the freed pages unchanged with its old content and delays clearing the page until subsequent allocation. Windows adopts a different manner, it attempts to clear the freed pages with ”zero page thread”. Unfortunately, the priority of this thread is the lowest, the thread therefore has less opportunity to execute, so that most of the freed pages are unable to be cleared timely. The amount of zero pages gradually decreases due to memory allocation and finally becomes only a few after long-term program execution. This makes the snapshot optimizations such as zero page compression meaningless. We will show the results in §3.2.

Modern operating systems always utilize Buffered I/O to speed up data access to external devices. They allocate the free pages to cache the data that is read recently from (or is dirtied but not written back to) storage device to reduce the access latency, and name these pages as ”cache pages”. For example, Windows introduce SuperFetch [13] to improve interactive responsiveness by preloading applications or data into memory; Linux also employs a similar approach named Readahead [11] to reduce IO latency. Although OS provides reclamation mechanism to release the cache pages, the cache pages are still resident in memory for long. This is because reclamation is only invoked when the amount of free memory is below a specific threshold, or the allocation fails due to the memory shortage. Consequently, the amount of cache pages is large especially after long time execution.

3.2 Measurements

In this subsection, we collect the memory usage of two Windows VMs and twelve Linux VMs in our private cloud platform, and report the amounts of zero pages (whose bytes are all 0s or 1s), used pages (used by kernel or applications) and cache pages resident in the memory.

The Windows VMs are used as virtual desktop for daily office by research students. The Linux VMs are connected and served as a Elasticsearch cluster [2](a distributed, decentralized server used to search documents), which stores over 50 million microblogs from Weibo [7]. To acquire the number of used pages and cache pages, we call the *GlobalMemoryStatusEx* function in Windows and record the result in 15 minutes interval during four days of execution, and we leverage Ganglia monitor to acquire the results in Linux VMs over one month. Meanwhile, we create VM snapshots periodically to count zero pages.

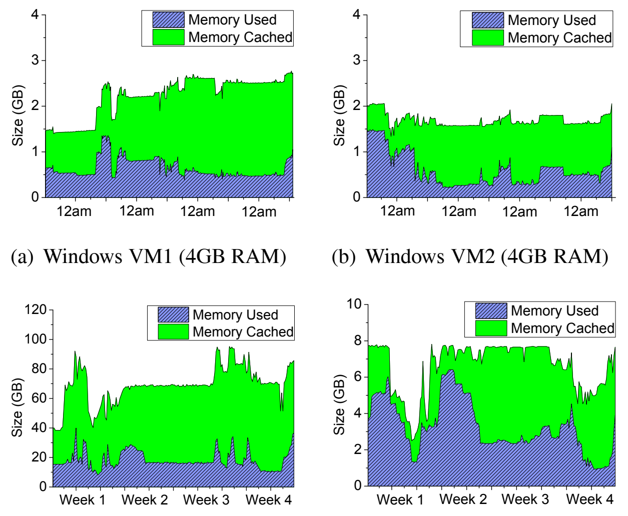
Figure 1 reports the size of cache pages and used pages. As we can see, the cache pages take a large fraction of memory, for example, they take 48.3% and 32.2% for two Windows VMs after four days of execution, and take 53% on average for the Linux VM cluster. The used pages are much less, they take 17.2%, 16.9%, 23.8% and 40.5% of the memory respectively. The free pages (neither used nor cached) also takes a large part in Windows, they take about 50% of the memory. However, the free pages don't amount to zero pages. Table 1 compares the average amount of zero pages and free pages from ten random epoches, it can be found that the amount of zero pages is much less. This is because the page that has ever been allocated but now is freed will be left with the content uncleared, it therefore is a free page but not a zero page. The similar results can be found in other works, for example, Park et al. [43] demonstrate that the free pages and cache pages on average take 17% and 66% of the memory respectively.

	VM1	VM2	VM3	VM cluster
Free pages	1.8	2.2	1.1	22.8
Zero pages	0.31	0.61	0.23	1.37

Table 1. Amount of free pages and zero pages (GB).

3.3 Summary

The virtual machine snapshot size is always large, especially after long time execution since most of the memory pages are allocated and then polluted. The zero pages take much less than free pages, implying that the potential to reduce snapshot size through zero page compression is less than expected. Moreover, a large amount of cache pages are resident in memory. Note that the cache pages are always used to improve performance, this suggests that the loss of cache pages would not crash the guest OS or applications (§5.4). Therefore, storing the cache pages and free pages in snapshot, especially replicating multiple copies of them, plays no role in improving availability but imposes extra storage overhead, and thus is unnecessary. Consequentially, we need a page aware replication mechanism to reduce the storage cost without compromising availability.



(a) Windows VM1 (4GB RAM) (b) Windows VM2 (4GB RAM)

(c) Linux VM Cluster (96GB RAM) (d) Linux VM3 (8GM RAM)

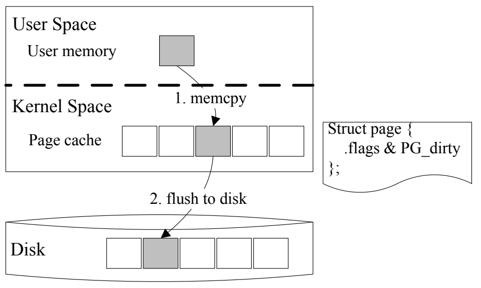
Figure 1. Amount of cache pages and used pages. For Linux VMs, we report the results of the entire cluster (c) and one randomly selected VM (d).

4. Page-Aware Replication Policy

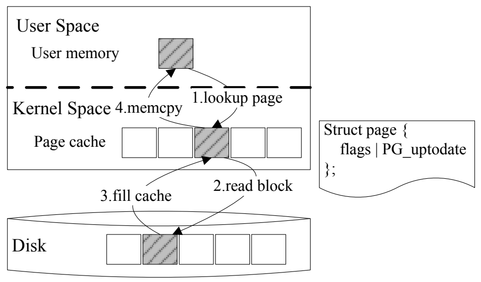
PARS explores the page semantics and categorizes the page according to its importance to system execution, it then saves different number of replicas for different page types. Specifically, PARS replicates multiple copies of critical pages to survive disk failures for proper VM recovery, and saves at most one copy of non-critical pages to save storage usage. In the following subsections, we firstly describe how to categorize the memory pages, then introduce the determination of the replica count taking availability and space saving into account, and finally discuss the potential space saving.

4.1 Analysing Page Type

The memory pages can be divided into several mutually exclusive categories corresponding to how operating system uses the pages [34]. In Linux, we can acquire five categories by the page semantics. (1) Free pages, which are not used by any process or kernel. They are either never allocated since boot-up, or are ever used but now freed and left as is with old contents. (2) Cache pages, which are not currently used by any process. They are allocated to cache data to be read or written back to eliminate the access latency between application and external devices. (3) Inode pages, which are used by user process. These pages contain data from disk and are mapped into the virtual address space of process. (4) Anonymous pages, which are accessed frequently by user process. These pages are allocated to heap or stack through functions such as *malloc*. (5) Kernel pages, which are only used by kernel. These pages are used to store the kernel binaries or internal data structures.



(a) Write system call



(b) Read system call

Figure 2. The procedure of filling page cache by read and write system calls.

The loss of the memory page produces different effects on system behavior, it may make no harm, or incur performance degradation, or even crash the system. Based on the effects, a memory page can be classified into one of the three categories: i) *unmeaningful pages*, ii) *performance related pages* and iii) *runtime critical pages*. The following paragraphs will explain the classification in detail.

The Free pages are currently not used by kernel or process, they are not linked in any way in the Linux kernel, and their loss makes no effects on the system execution. Thus, they are regarded as *unmeaningful pages*.

The Kernel pages, Inode pages and Anonymous pages are necessary for the application or kernel to work properly, missing such page will lead to misbehavior or even system crash, therefore they belong to *runtime critical pages*.

The Cache pages may lie in Page Cache which is used to cache the pages related to file, or in Buffer Cache which is related to block device. The Cache pages are difficult to be categorized directly, because they may be used to store the data either copied from user memory by *write* system call, or read from the underlying disk device by *read* system call.

The *write* system call is issued by user process, it uses the file descriptor, user buffer and associated size as the parameters. The write operation firstly calculates the page index through the file descriptor, and then attempts to lookup the *page*[1] in *radix tree* which is used in Page Cache to arrange *pages* efficiently. If the *page* is not present in *radix tree*, OS kernel will allocate a new one and append it into *radix tree*, mark the *page* as dirty (PG_dirty), and finally copy the content from user memory to the page, as illustrated in Figure 2(a). The kernel forks *pdflush* thread to flush the dirtied pages into disk asynchronously. The write operation related to Buffer Cache is similar except that it uses BH_dirty flag to indicate whether the *page* is dirtied. These dirtied cache pages containing the data to be written back are critical to the system execution, since their loss would lead to inconsistency between the application and disk blocks. Therefore, they are regarded as *runtime critical pages*.

[1] The italic *page* here is a kernel object of "struct page" to describe one physical page.

The procedure of *read* system call, as illustrated in Figure 2(b), is a bit similar to *write*. It firstly lookups the *page* in *radix tree*, if the *page* is present and its flag is marked as up-to-date (PG_uptodate is set), the page content can be copied to user memory directly. Otherwise, the kernel allocates a new *page* into *radix tree*, reads the requested data from disk blocks, fills data into the page, and finally copies the page content to user memory. The read operation related to Buffer Cache is similar except it uses BH_uptodate flag to indicate that the *page* is up-to-date. This kind of cache page is just an identical copy of data resident in the block, so that the page loss only incurs longer latency when subsequent read operations access the same block. As a result, we treat these clean cache pages as *performance related pages*.

4.2 Replica Count Policy

Replica count plays an important role in availability and space saving. A large count can improve availability by tolerating more disk failures, but at the cost of more storage usage. On the other hand, a small count can save storage space, but sacrifices the availability. In many systems [3, 26, 51], the default replica count is 3, which achieves a well tradeoff between storage space and availability. We determine the replica count of pages according to their importance to system execution. Table 2 illustrates the page categories as well as the associated replica count in PARS.

Unmeaningful pages are irrelevant to system execution. Moreover, their page content are all 0s or 1s, or left with the old content when it is freed, and thus are meaningless. This suggests that saving these pages is unnecessary, therefore the replica count of this page type is set to 0.

Performance related pages can be discarded from snapshot state to save storage space [43, 44]. However, after rollback recovery, their loss will force the system to access the data from external device rather than the cache, resulting in longer latency and consequentially performance degradation. The degradation may be unacceptable for some workloads desiring low latency [48, 50], as a result, we by default save one copy of these pages for the sake of application performance. Meanwhile, the users can set the replica count to 0 if they prefer storage space saving to performance guarantee.

Page category	Page semantics	Replica count
unmeaningful pages	Free pages	0
performance related pages	Clean cache pages	1
runtime critical pages	Dirtied cache pages; Inode pages; Kernel pages; Anonymous pages	k(default is 3)
metadata	Record the page address and page category	k

Table 2. Page categories and associated replica count.

Runtime critical pages are necessary for the system to perform properly, missing one such page may lead to runtime misbehavior or even system crash. As a result, we save k (default is 3) identical replicas of these pages to survive up to $k-1$ simultaneous disk failures. The users can specify a larger value if the underlying hardware suffers high failure rate or higher availability is required.

Metadata. Apart from the memory pages, the metadata, which records the page address and page type, should also be saved as part of the snapshot file. The metadata is undoubtedly critical, thus it is highly desired to save k copies to ensure the availability. Fortunately, the size of metadata is small. We use 8 bytes to represent the address and 1 byte to denote the page type. For a VM configured with 4GB RAM (1M pages), the metadata size is $k*1\text{M}*(8+1)\text{B}=9k\text{MB}$.

4.3 Expected Storage Saving

In this part, we analyze the expected storage space saving yielded by PARS compared to the conventional replication technique when achieving the same availability. We suppose that the percentages of *runtime critical pages*, *performance related pages* and *unmeaningful pages* in the snapshot file are α, β, γ respectively, where $\alpha+\beta+\gamma=1$. The size of snapshot file is denoted by S. The conventional storage system creates k identical replicas of the full file, therefore, the storage usage W is $k*S$. PARS saves no *unmeaningful pages*, and saves one copy of *performance related pages*, therefore the storage usage of PARS denoted by W' is determined by:

$$W' = k*\alpha*S+\beta*S \tag{1}$$

And then the storage reduction r can be calculated by:

$$\begin{aligned} r &= 1-\frac{W'}{W} = 1-\frac{k*\alpha*S+\beta*S}{k*S} = 1-\frac{k*\alpha+\beta}{k} \\ &= 1-\alpha-\frac{\beta}{k} = \beta+\gamma-\frac{\beta}{k} = \frac{k-1}{k}*\beta+\gamma \end{aligned} \tag{2}$$

It can be concluded from (2) that the increase of *performance related pages* and *unmeaningful pages* has potential to reduce the storage usage. In practical cases as illustrated in Figure 1(c), where β is 0.53 and γ is 0.238, PARS can achieve 59.1% space saving when k is 3. In addition, if higher availability is required, PARS would save more storage space, for example, the reduction is 63.5% when k is 4 and 66.2% when k is 5. This also suggests that PARS can achieve much higher availability using the same storage by saving more replicas of *runtime critical pages*.

5. Data Loss Handling upon Recovery

One challenging task in PARS is to handle the page loss due to disk failures, especially for the *performance related pages* that are vulnerable to failures as a result of only one data copy. We assume that disks follow a simple fail-stop model rather than suffer silent errors [35], so that data loss can be detected easily. The following sections will describe how to handle the loss of snapshot state.

5.1 Loss of Metadata

We save k copies of metadata file. If there still exists one full copy of metadata, we can fetch the page address and the associated type, load the page content from snapshot into the guest memory space (supposing that the page content are available), and finally resume the virtual machine. Once all the copies of metadata are lost, the snapshot would be obsolete. One possible approach is to find an earlier available snapshot and then roll back the VM from it.

5.2 Loss of Unmeaningful Pages

The *unmeaningful pages* are useless and are not saved in the snapshot, their loss therefore is only considered as the result of metadata loss. Provided that the metadata exists, we identify the *unmeaningful pages* and simply fill zero into the associated memory pages upon rollback recovery.

5.3 Loss of Runtime Critical Pages

The *runtime critical pages* are replicated k times across disperse disks. Unless the k copies are lost simultaneously, the snapshot state could be reconstructed completely from any failure-free replica. Once the worst case happens, i.e., no replica is available, the rollback recovery from this snapshot would fail. Similarly, our advice is to roll back the VM from an earlier created snapshot or allow the user to specify a snapshot file for recovery.

5.4 Loss of Performance Related Pages

Handling the page loss of *performance related pages* is more challenging. First, they are saved only one copy and hence are more vulnerable to frequent disk failures. Second, the page loss without informing guest kernel would lead to data inconsistency which further results in application misbehavior. Specifically, if a page has been allocated to cache the data read from disk, the kernel would be aware of the existence of the page, that is, the kernel is able to index the *page* in *radix tree* and then accesses the proper page content. Supposing that the page now is lost due to disk failure, the memory content associated with this page are missing accordingly after recovery. However, the *radix tree* remains unchanged since it belongs to *runtime critical pages*. This implies that the kernel after recovery can still index the *page* in *radix tree* but cannot access the proper content because they are lost. Consequently, inconsistency arises and may result in application misbehavior or crash.

One possible solution for maintaining consistency is to remove the *page* from *radix tree* for the lost page through VM introspection. However, this method will cause chaos of kernel internal variables and is extremely complicated. For example, the *radix tree* reserves a variable *count* to record the number of *pages* in the tree, if a *page* is removed, the *count* should decrease accordingly. Moreover, the *page* is always linked to *active_list* or *inactive_list* for page reclamation, removing the *page* involves operations on these lists.

Our solution is simple and effective. Observed that when user application accesses data, the kernel firstly acquires the associated *page* from *radix tree*, and then tests if the page content is up-to-date (i.e., PG_uptodate flag is set). If it is not, the kernel will skip the cache and read the data from external device to update the page content. The related codes in kernel-2.6.32 are listed in Listing 1, some details are removed for clarity. Inspired by this, our approach is to clear the PG_uptodate flag of the lost pages upon recovery. Once the kernel after recovery requests the data which once existed in page cache but now is lost, it would realize that the page content is out of date. Therefore, it reads the data from the disk device, fills the proper data into the memory page, and finally copies the content to user. In this way, we hide the page loss to kernel. The procedure is similar for the Buffer Cache, the difference is that it removes the BH_uptodate flag.

Listing 1. Key codes of read operation.

```
1  do_generic_file_read(mapping, offset)
2  {
3    tree = mapping->page_tree;
4    page = radix_tree_lookup_slot(tree,offset)
5    if(page != NULL) {
6      if(!PageUptodate(page)) // PG_uptodate
7        // read data from external device
8        mapping->a_ops->readpage(page)
9    }
10   else {
11     page = page_cache_alloc_cold(mapping)
12     read_pages(page)
13   }
14   copy_to_user(page)
15 }
```

5.4.1 Safe Removal of PG_uptodate Flag

The OS kernel may not always perceive the removal of PG_uptodate flag after rollback. For example, in Listing 1, if the page content is up-to-date, then the kernel would execute the following steps: 1) get page from the radix tree (line 4), 2) find that the content is up-to-date (line 6), 3) copy the data to user (line 14). Supposing that the VM state is checkpointed after step 2 but before step 3, the kernel after rollback recovery will continue to execute step 3 without checking PG_uptodate flag again. As a result, even if the PG_uptodate flag is removed upon rollback due to page loss, the kernel still copies the non-existent data to user space. This issue is attributed to that the *read* operation is not atomic, so that the snapshot procedure running concurrently may interfere with the kernel execution. Our solution is to delay the snapshot procedure so that the kernel is able to perceive all the removals of PG_update flag. Specifically, when snapshot operation is required, we start to intercept the *read* system call through VM introspection [25], and checkpoint the VM before the *read* execution. In this way, the *read* operation is considered to be atomic in the view of snapshot time point. One issue is that if *read* operation is rarely invoked, the snapshot procedure would be blocked for an unpredictable long time. To avoid this, we fork the snapshot procedure if no *read* is intercepted within 1 seconds. It is worth noting that the performance penalty due to interception is insignificant, because the interception lasts at most 1 seconds.

6. Implementation

6.1 Overall Architecture

PARS is implemented in the VMM layer using version qemu-kvm-0.12.5. PARS adopts introspection technology to acquire the page type, and thus requires no modification of guest OS. In addition, the QEMU snapshot mechanism is modified to save the page content following our page-aware policy. The architecture of PARS is illustrated in Figure 3, it consists of two key components: PageClassifier and SnapDaemon. The two components are outlined below:

PageClassifier employs VM introspection to parse the internal data structure of guest kernel, explores the page semantics and then categorizes the pages. It is implemented in KVM module, and is transparent to the upper operating systems. Additionally, it supports multiple kernel versions, as will be described in §6.3.

SnapDaemon is responsible for taking snapshots of VM and rolling back the VM upon failures. SnapDaemon utilizes *libhdfs* [8], which provides C APIs to interact with HDFS, to write (read) data to (from) HDFS. In PARS, SnapDaemon only specifies the replica count when creating the file, and leaves the responsibility of replication to HDFS.

SnapDaemon provides two snapshot approaches: stop-and-copy snapshot [43] and copy-on-write (COW) based live snapshot [20]. In stop-and-copy snapshot, SnapDaemon involves three steps. First, it traverses the memory pages, for each page, it acquires the page category from PageClassifier and then saves the pair {page address, category} into the Metadata File which is replicated three copies. Second, it fetches the *runtime critical pages*, and saves them into Critical File which is replicated three copies. Finally, SnapDaemon obtains the *performance related pages*, and writes

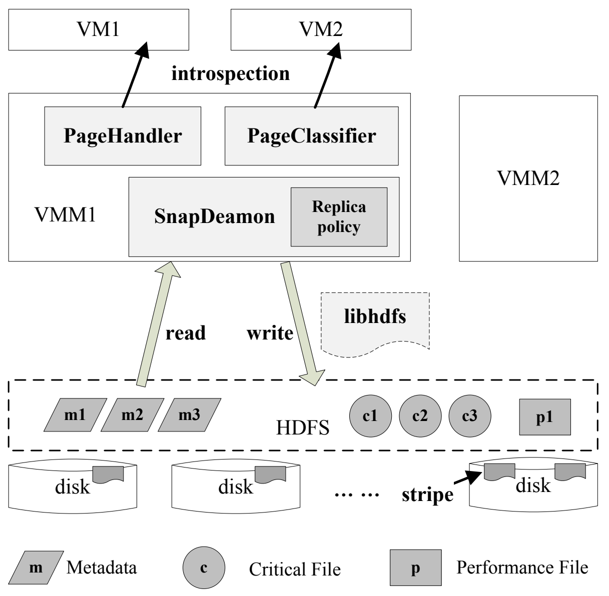


Figure 3. PARS architecture.

the content into Performance File. The procedures are similar for COW snapshot which saves the memory pages in demand-paging manner. If one page requires to be saved, SnapDaemon firstly explores the page type and then writes the content into corresponding file. Upon recovery, SnapDaemon firstly reads the metadata, and then fetches the page content from Critical File and Performance File. SnapDaemon considers the data to be lost if file *open* or *read* function returns invalid value. If the data are available, SnapDaemon fills them into the guest memory. Otherwise, SnapDaemon hides the page loss by employing PageHandler which clears the flag of lost pages as described in §5.

6.2 The Use of Introspection

VM introspection helps PARS to classify the memory pages (PageClassifier) and handle the page loss (PageHandler). It is specific to guest OS. We first describe the implementation details for Linux kernel 2.6.32.5, and then discuss the implementation on other Linux kernel versions in §6.3.

To explore the page semantics, the key is to acquire the object *page* which describes the information of a physical page. Fortunately, in Linux kernel, all the *page*s lie in *mem_map* which is a kernel array used to manage all the memory pages. *Mem_map* is located in the physical memory started from 0xffffea0000000000, it uses the page's physical frame number (pfn) as the index. Thus, we can locate the *page* by *mem_map* and pfn, and further read the value of *page* through *kvm_read_guest()* which is provided by KVM for reading data from a specified guest physical address.

Once the value of *page* is known, we can analyze its variables to determine the page category. Specifically, if the usage count (*_count*) of *page* is zero, the associated page therefore is not used by any process or the kernel, implying that the page belongs to *unmeaningful pages*. If the page is in use but the map count (*_mapcount*) is zero, the page would be in cache. A page is in Page Cache if variable *private* is zero, otherwise it is in Buffer Cache. For a page in Page Cache, it belongs to *runtime critical pages* if one of the following flags in variable *flags* is set: i) PG_dirty which means that the page is dirtied, ii) PG_writeback which means that it will be written to disk and iii) PG_lock which means that the page is prepared for I/O. Otherwise, it is regarded as *performance related pages*. For a page in Buffer Cache, we acquire *buffer_head* of the *page* and categorize it into *runtime critical pages* if either of two flags in *b_state* is set: i) BH_dirty denoting a dirtied page and ii) BH_lock implying a locked page. Otherwise, the page belongs to *performance related pages*. The remaining pages are neither in cache nor are unmeaningful, and thus are *runtime critical pages*.

The procedure to handle page loss upon recovery using introspection is similar to how we categorize the page. We clear the PG_update flag of *page* or BH_uptodate flag of *buffer_head*, and then write the value of *page* back to *mem_map* through *kvm_write_guest()* which is used to write data to a specified guest physical address.

6.3 Multiple Platforms Support

We detect the guest kernel versions by distinguishing the interrupt descriptor table and system call table entry address which vary significantly across OS versions [18].

Our solution to categorize pages and handle page loss only involves one key kernel object, i.e., *page*, and thus is easy to support other Linux kernel versions. The difference across different versions lie in three aspects: i) the address of *mem_map*, ii) variables of *page* and iii) variables of *buffer_head*. The address of *mem_map* is denoted by VMEMMAP_START, it is 0xffffea0000000000 and is unchanged in Linux kernels from 2.6.32 to latest 3.14. The variables of *page* or *buffer_head* across different versions, however, are different. In a newer version, new variables may be added, variables may be renamed, and the offset of variable may be changed. Table 3 illustrates such cases in kernel 3.14 that are different from 2.6.32. Observed that only a few variables of *page* and *buffer_head* are used for introspection, we reserve an array to record these variables and their offsets along with the kernel version. Upon introspection, we firstly identify the kernel version, locate the *mem_map* by VMEMMAP_START, read the value of *page* and *buffer_head*, access the variables from the recorded offset, and finally categorize pages or handle page loss.

version	**2.6.32**	**3.14**
New variable		counters
Renamed variable	kmem_cache * slab	kmem_cache * slab_cache
Different offset	mapping=16	mapping=4

Table 3. Some changes of *page* between two kernels.

6.4 Storing Snapshots in HDFS

HDFS is designed to store large size files that are up to hundreds of MB or GB, and is suitable to store write-once, read-many-times data. The virtual machine snapshots have several features that are consistent with HDFS. First, the snapshot file are created and written only once, the subsequent operations are either reading the data for recovery or deleting the file for space reclamation. Second, the snapshot file is important and may be read multiple times, e.g., the snapshot contains important data, or new configured environment for software development. Third, the size of snapshot files is always large and up to several GB.

We leverage *libhdfs* to interact with HDFS. The *hdfsOpenFile* function treats replica count as one parameter. Thus, PARS focuses on the determination of file to be replicated and the replica count, yet leaves the responsibility of replication to HDFS. Implementation on HDFS involves only 70 lines of code. It is supposed to easily port to other distributed file systems due to the simplicity of the interfaces.

7. Evaluation

7.1 Experiment Setup

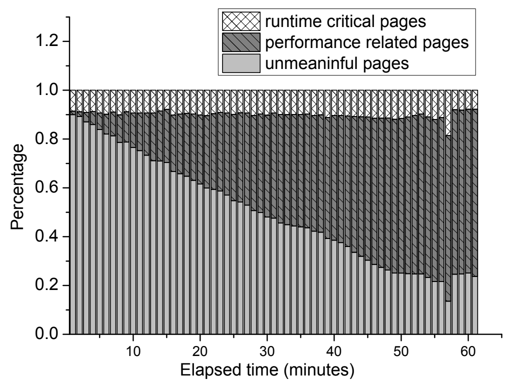
Our testbed consists of six physical severs, each is configured with 8-way quad-core Intel Xeon 2.4GHz processors, 48GB DDR memory, a 500GB WD disk, and Intel 82576 Gigabit NIC card. The servers are connected with switched Gigabit Ethernet. The virtual machines are configured with 4GB RAM unless specified otherwise. Both the physical servers and VMs are installed with Debian Linux 6.0. We setup HDFS on top of five physical servers, and use a separate server to run virtual machines. The snapshot files are stored into (or loaded from) HDFS via LAN rather than the local disk.

We evaluate PARS under several workloads. (1) Compilation, we compile Linux kernel 2.6.32.5 along with all the modules. (2) Mummer is a bioinformatics program for sequence alignment, it is CPU and memory intensive [4]. We align genome fragments obtained from NCBI [1]. (3) MPlayer is a video player, it prefetches a large fraction of movie file for performance requirements. (4) MySQL is a database management system [5], we employ SysBench tool [6] to read (write) data from (to) the database that stores 500,000 records. (5) Elasticsearch [2] is a file search server, it attempts to load as much data as possible into memory. The Elasticsearch server manages about 50 million microblogs, and the client continually requests the micorblogs with random generated user id.

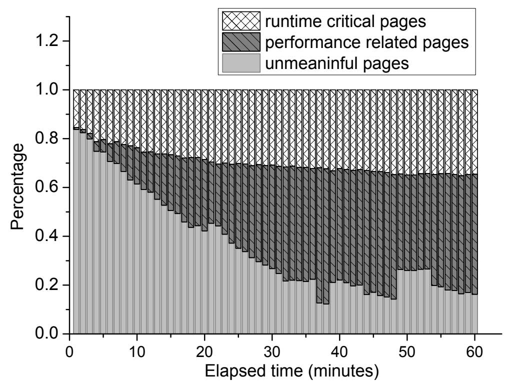
7.2 Distribution of Different Page Types

We first report the percentages of three page categories in different epoches and under various workloads, and then compute the expected space saving.

Different times. We explore the memory usage in 60 seconds interval for over one hour running under Compilation and Elasticsearch. The percentages of three page categories are illustrated in Figure 4. As we can see from Figure 4(a), the percentage of *unmeaningful pages* decreases as the VM executes, for example, it is 90% at the start and gradually decreases to 21.6% in the 57th minute when the compilation completes. This is because more and more free pages are allocated to cache data. The percentage of *performance related pages*, as expected, increases with the time and takes a large fraction of memory pages. For example, it exceeds 50% since the 40th minute, and reaches 67.8% at the end of compilation. The percentage of *runtime critical pages* keeps almost steady, it is around 10% during the compilation. Figure 4(b) depicts the results under Elasticsearch. Different from Compilation, the *runtime critical pages* here occupies a large fraction of memory, e.g., its percentage exceeds 30% since the 30th minute. This is because Elasticsearch allocates its own buffer to fill data, and this buffer lies in the heap region which is regarded as *runtime critical pages* from the view of PARS. The percentage of *performance related pages*, as expected, gradually increases with the execution, and it finally takes about 50% of the memory pages at the end of the trace.



(a) Percentages under Compilation



(b) Percentages under Elasticsearch

Figure 4. Percentages of different page categories.

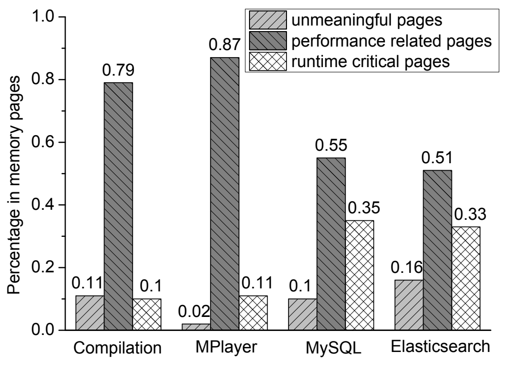


Figure 5. Percentages under various workloads.

Different workloads. The page count of different categories is specific to the workload and its temporal activity. We therefore count the pages on ten random epoches under various workloads after long time running, compute the percentage of page count to the whole memory size, and report the average percentage in Figure 5. As expected, *performance related pages* occupy the majority of memory state, they take 79%, 87%, 55% and 51% for Compilation, MPlayer, MySQL and Elastisearch respectively. The percentages for MySQL and Elasticsearch are a bit lower because the two applications allocate a number of memory pages as buffer to ensure low request latency. Consequently, we can compute the expected space saving by the equation (2). Compared to the conventional approach which replicates the file fully and identically, PARS is expected to reduce 63.7%, 60%, 46.7%, and 50% storage space when the replica count is 3.

7.3 Practical Storage Space Saving

In this subsection, we evaluate the practical storage saving yielded by PARS. We randomly collect 100 snapshots that are created by users of our private cloud platform during daily operations. The operations may be Linux kernel studying, software development and testing, or office usage. The associated guest kernel of these snapshots include CentOS 6.4 (2.6.32 kernel), Ubuntu 11.10 (3.0.3 kernel) and Debian 6.0 (2.6.32.5 kernel). We compare PARS with two approaches: 1) Native HDFS approach which stores 3 replicas of each file, 2) Compression approach, which is used widely in both storage system and snapshot system [30, 40, 41, 51], it compresses the snapshot first and then stores the files into HDFS. We employ *zlib* to compress data.

We first present detailed results to show the potential saving for a single snapshot. We load the snapshot state into guest memory and determine the page categories, and calculate the maximum, minimum, average and medium percentage of *runtime critical pages* in these snapshots. Meanwhile, we compress these snapshots and calculate the compression ratio. As we can see in Table 4, the percentage of *runtime critical pages* varies widely, e.g, the maximum percentage is 56.4% while the minimum value is only 13.7%. Smaller percentage implies less storage usage, because *runtime critical pages* are replicated three copies. The results of compression ratio are also varied, and they suggest that about half (52.7%) of the space can be saved with Compression approach.

We then re-create these snapshots with our PARS approach, compute the file size, and report the results in Table 5. The total size of 100 snapshots are 251GB. With Native HDFS approach, 753GB storage space will be occupied. The Compression approach reduces the usage to 356GB due to the benefit of compression. Our PARS approach uses 347GB, which achieves a 53.9% reduction compared to the Native HDFS approach. These results strongly support our work on achieving storage efficiency. It's worth noting that the storage cost of PARS is almost identical to Compression approach, however, the time to create snapshot and roll back the VM is much less, which will be described in §7.4.

	max.	min.	avg.	med.
Compression ratio	72.7%	29.1%	47.3%	48.5%
Percentage of *runtime critical pages*	56.4%	13.7%	28.2%	26.1%

Table 4. Compression ratio and percentage of *runtime critical pages*. The smaller the better.

Modes	Native HDFS	Compression	PARS
Storage usage	753G	356G	347G
Space saving	N/A	52.7%	53.9%

Table 5. Storage usage of three approaches.

7.4 PARS Overhead

The overhead of PARS falls into two main aspects. One is the time overhead that involves introspecting the memory pages (introspection time), saving the memory pages to HDFS (snapshot duration), and loading the pages upon recovery (rollback latency). Another is the performance loss due to the introspection cost during live snapshot when saving the memory page on demand. In this section, we compare PARS with both Native snapshot approach in QEMU and Compression based snapshot approach which compresses the memory pages first and then writes them into HDFS.

7.4.1 Snapshot Duration

The snapshot duration of PARS using the stop-and-copy approach mainly consists of the time to introspect the guest memory, save Critical File and save Performance File. We also classify the pages in Native approach, to explore how much time can be saved by writing one replica of Performance File compared to three replicas. Figure 6 compares the detailed duration of PARS against Compression approach and Native approach. It can be seen that the introspection time is minor, it is less than 2 seconds for 4GB

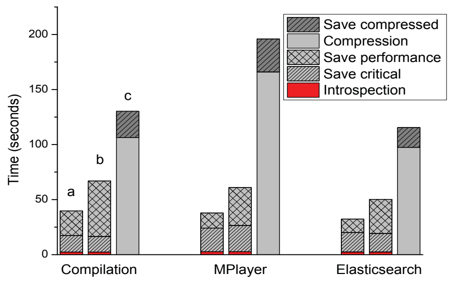


Figure 6. Time overhead on snapshot creation. a) PARS approach, b) Native snapshot, c) Compression based snapshot.

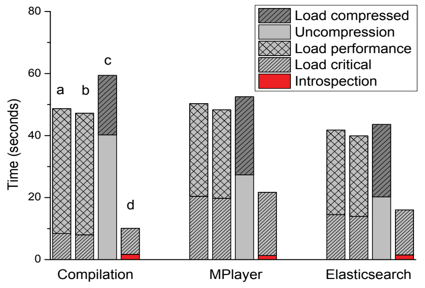


Figure 7. Details of rollback latency. a) PARS, b) Native snapshot, c) Compression based snapshot, d) PARS without loading *performance related pages*.

RAM (Table 6 summaries the introspection time for varying memory sizes on snapshot and rollback.). This is because our introspection method involves only a few bit operations. The time to save Critical File are almost identical between PARS and Native snapshot, because the two approaches both write three copies. However, PARS takes much less time to save Performance File than Native snapshot, for example, it spends 22.4 seconds while Native snapshot takes 50.4 seconds under Compilation. The reason is that PARS only saves one copy, while Native snapshot saves three copies which contend for network and disk I/O bandwidth. On average, PARS reduces the duration of Native snapshot by 36.4%. The Compression approach, as expected, spends much more time, mainly due to the long computation time which takes more than 80% of the full snapshot duration.

RAM size	1GB	2GB	4GB	8GB	16GB
snapshot	1.24	1.51	1.84	2.33	3.16
rollback	1.38	1.63	2.05	2.56	3.32

Table 6. Introspection time overhead (seconds).

7.4.2 Rollback Latency

The latency to roll back the virtual machine with PARS mainly involves the time to read Performance File and Critical File. Moreover, once the data of Performance File is lost, we need to handle the page loss. Therefore, apart from the rollback latency of the above three approaches, we also present the result of PARS without loading *performance related pages*. The results from Figure 7 show that the rollback latency are almost identical between PARS and Native snapshot, this is because the underlying HDFS supports parallel data loading from disperse disks when acquiring the file. On rollback recovery without loading *performance related pages*, PARS only reads the Metadata and Critical File, modifies the internal data of guest kernel to hide the page loss, and then resumes the VM. Consequentially, it results in less rollback latency. On average, PARS without loading *performance related pages* achieves a 65.8% latency reduction compared to PARS with loading all pages. The latency of Compression approach is a bit larger than the other approaches, similarly, the cost is mainly attributed to the decompression time.

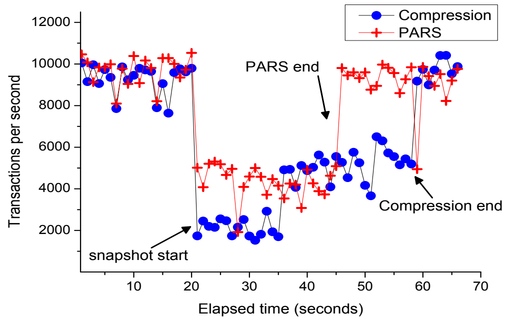


Figure 8. TPS of Elasticsearch during live snapshot.

7.4.3 Performance Degradation during Live Snapshot

During live snapshot, PARS explores the page type for the page to be saved on demand. From Table 6, we can conclude that the introspection imposes negligible performance loss on native copy-on-write snapshot. Thus, the applications perform similar during PARS COW and native COW live snapshot. In this experiment, we compare PARS COW live snapshot (PARS) against Compression based live snapshot (Compression). We setup Elasticsearch server in a VM, and record the transactions per second (TPS) during normal execution and live snapshot.

Figure 8 compares the results of the two approaches. The TPS during PARS live snapshot drops from 10531 to 5015 since snapshot starts at the 20th second, and returns to normal at the 46th second when snapshot completes. The Compression approach performs much worse, the Elasticsearch server handles only half of the transactions compared to PARS. Specifically, the TPS on average is 2170 during Compression snapshot while it is 4463 during PARS snapshot from the 20th second to the 36th second. The TPS during Compression snapshot increases since the 36th second, we at-

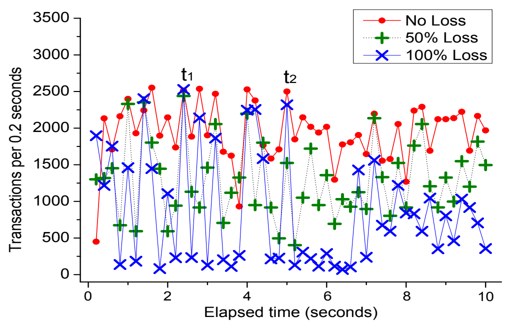


Figure 9. TPS with varying degrees of page loss.

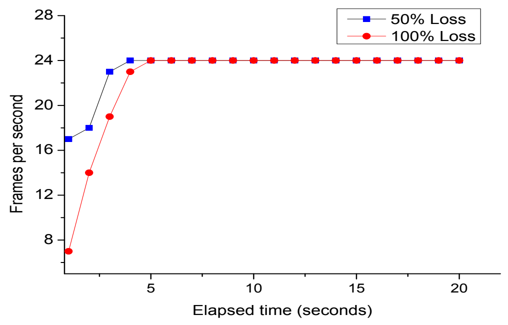


Figure 10. FPS with varying degrees of page loss.

tribute this to the benefit of access locality because the subsequent access to the page that has been saved will not trigger page fault again. It should be pointed out that the snapshot duration of PARS is shorter than that of Compression, implying that the Elasitcsearch server regains the full capacity early and thus is able to handle more transactions.

7.5 Performance Loss on Disk Failures

In this part, we evaluate PARS in the face of loss of *performance related pages* due to disk failures. HDFS stripes a file into disperse disks, so it is difficult to quantify the loss of file for real disk failures. Therefore, we randomly select pages and mark them as lost when reading them from HDFS. We consider three degrees of page loss: No Loss, 50% Loss and 100% Loss (equal to zero replica). We measure the application performance in terms of transactions per second (TPS, the TPS here actually refers to transactions per 0.2 second) of Elasticsearch and frames per second (FPS) of MPlayer.

Figure 9 compares the TPS of three degrees. It can be seen that TPS decreases seriously in 100% Loss case. Although some required data would hit in the private cache owned by Elasticsearch, making TPS in t_1 and t_2 be as high as that of No Loss, most of the desired data have to be fetched from disk rather than memory cache. On average, the server without loading *performance related pages* handles 884 transactions each epoch, a 54.9% loss compared to the No Loss case which is able to handle 1961 transactions. The server in 50% Loss case handles 1321 transactions on average, which is better than 100% Loss. This is attributed to the low access latency since lots of data are loaded into the cache. Note that although loading no *performance related pages* could reduce the rollback latency by over 20 seconds as shown in Figure 7, the application here suffers serious performance degradation after recovery. The degradation may be intolerable for low-latency high-throughput scenarios [48, 50]. One solution is to trace working set cache pages during snapshot and only load these cache pages upon rollback [52], to offer a tradeoff between rollback latency and performance. We leave this as our future work.

The results of FPS are depicted in Figure 10. For the first three seconds, the FPS are 7, 14, 19 for the 100% Loss case, and are 17, 18, 23 if half of the data are lost. Note that 14 frames per second is sufficient to play the video fluently, this implies that the *performance related pages* might be eliminated from snapshot for some scenarios where applications are not time critical, such as virtual desktop, so that the storage usage can be further reduced.

Another point should be noted is that the FPS regains the full capacity from the 3rd second, while the TPS keeps low for a longer duration in the face of page loss. We owe the difference to the variety of mechanisms on how the application accesses the pages in cache. Specifically, MPlayer sequentially accesses the video file, so that the subsequent read operations will always hit the page prefetched in cache. In contrast, Elasticsearch client randomly requests the data which may be scattered in the cache. Thus, the hit rate is low.

8. Conclusions and Future Work

This paper proposes PARS, a page-aware replication system for storing VM snapshots efficiently. PARS explores the in-memory pages and categorizes them according to their importance to system execution. It enhances the data availability by replicating the critical pages, and reduces the storage space by saving one copy of performance related pages which take a large fraction of memory state. The advantage of PARS over naive replication is that it requires much lower storage for achieving the same availability, or is able to provide higher availability with the same storage cost.

For future work, we plan to identify the working set cache pages during snapshot and only save these active cache pages instead of the whole cache pages, to further reduce storage usage without compromising application performance after recovery. In addition, we plan to explore Windows kernels which are widely used for daily office work.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments and help in improving this paper. This work is supported by China 863 Program (2013AA01A213), China 973 Program (2011CB302602), China HGJ Program (2013ZX01039-002-001) and NSFC Program (91118008).

References

[1] National center for biotechnology information. `ftp://ftp.ncbi.nih.gov`.

[2] Elasticsearch. `http://www.elasticsearch.org/`.

[3] Hdfs. `http://hadoop.apache.org/`.

[4] Mummer. `http://mummer.sourceforge.net/`.

[5] Mysql. `http://www.mysql.com/`.

[6] Sysbench. `http://sysbench.sourceforge.net/`.

[7] Weibo. `http://weibo.com`.

[8] libhdfs. `http://hadoop.apache.org/docs/r1.2.1/libhdfs.html`.

[9] Salesforce, 1999. `http://www.salesforce.com`.

[10] Using the snapshot, 2003. `https://www.vmware.com/support/ws4/doc/preserve\_snapshot\_ws.html`.

[11] readahead, 2005. `https://lwn.net/Articles/155510/`.

[12] Amazon ec2, 2006. `http://aws.amazon.com/ec2/`.

[13] Superfetch, 2007. `http://en.wikipedia.org/wiki/\\Windows\_Vista\_I/O\_technologies`.

[14] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden. Fast wide area live migration with a low overhead through page cache teleportation. In *Proceedings of CCGrid*, pages 78–82, 2013.

[15] N. Amit, D. Tsafrir, and A. Schuster. Vswapper: A memory swapper for virtualized environments. In *Proceedings of ASPLOS*, pages 349–366, 2014.

[16] B. Calder, J. Wang, A. Ogus, and N. N. et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of SOSP*, pages 143–157, 2011.

[17] J.-H. Chiang, H.-L. Li, and T. cker Chiueh. Introspection-based memory de-duplication and migration. In *Proceedings of VEE*, pages 51–62, 2013.

[18] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security: A short paper. In *Proceedings of the ACM Workshop on Cloud Computing Security*, pages 97–102, 2009.

[19] J. C. Corbett, J. Dean, and M. E. et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI*, pages 251–264, 2012.

[20] L. Cui, B. Li, Y. Zhang, and J. Li. Hotsnap: A hot distributed snapshot system for virtual machine cluster. In *Proceedings of USENIX LISA*, pages 59–73, 2013.

[21] L. Cui, J. Li, B. Li, and et al. Vmscatter: Migrate virtual machines to many hosts. In *Proceedings of VEE*, pages 63–72, 2013.

[22] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proceedings of HPDC*, pages 135–146, 2011.

[23] C. Dubnicki, L. Gryz, L. Heldt, and M. Kaczmarczyk. Hydrastor: A scalable secondary storage. In *Proceedings of FAST*, pages 197–210, 2009.

[24] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of OSDI*, pages 1–14, 2010.

[25] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of NDSS*, pages 191–206, 2003.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of SOSP*, pages 29–43, 2003.

[27] D. Harnik, O. Margalit, D. Naor, D. Sotnikov, and G. Vernik. Estimation of deduplication ratios in large data sets. In *Proceedings of MSST*, pages 1–11, 2012.

[28] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX Annual Technical Conference*, pages 113–128, 2008.

[29] X. Jiang and D. Xu. Violin: Virtual internetworking on overlay infrastructure. In *Parallel and Distributed Processing and Applications*, pages 937–946, 2005.

[30] H. Jin, L. Deng, and S. Wu. Live virtual machine migration with adaptive memory compression. In *Proceedings of CLUSTER*, pages 1–10, 2009.

[31] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of ASPLOS*, pages 14–24, 2006.

[32] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimmal downtime. In *Proceedings of DSN*, pages 87–98, 2011.

[33] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[34] J. F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. Technical report, Aalborg University, 2007.

[35] G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. In *Proceedings of Euro-Par*, pages 442–453, 2005.

[36] R. Kotla, L. Alvisi, , and M. Dahlin. Safestore: A durable and practical storage system. In *Proceedings of ATC*, pages 127–142, 2007.

[37] A. Koto, H. Yamada, K. Ohmura, and K. Kono. Towards unobtrusive vm live migration for cloud computing platforms. In *Proceedings of APSys*, pages 1–6, 2012.

[38] J. Kubiatowicz, D. Bindel, Y. Chen, and S. C. et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*, pages 190–201, 2000.

[39] X. Li, M. Lillibridge, and M. Uysal. Reliability analysis of deduplicated and erasure-coded storage. *Proceedings of SIGMETRICS Performance Evaluation Review*, 38(3):4–9, 2010.

[40] C. Liu, D. Ju, Y. Gu, Y. Zhang, D. Wang, and D. H. Du. Semantic data de-duplication for archival storage systems. In *Proceedings of ACSAC*, pages 1–9, 2008.

[41] C. Marshall. Efficient and safe data backup with arrow. Technical report, 2008.

[42] P. Nath, M. A. Kozuch, D. R. OHallaron, and J. Harkes. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of ATC*, pages 71–84, 2006.

[43] E. Park, B. Egger, and J. Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of VEE*, pages 75–85, 2011.

[44] M. R, Hines, and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of VEE*, pages 51–60, 2009.

[45] R. Rodrigues and B. Liskov. High availability in dhts erasure-erasure code vs replication. In *Proceedings of IPTPS*, pages 226–239, 2005.

[46] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics*, 78:1–11, 2007.

[47] B. Schroeder and G. A. Gibson. Disk failures in the real world what does an mttf of 1,000,000 hours mean to you. In *Proceedings of FAST*, pages 1–16, 2007.

[48] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of ACM SoCC*, 2013.

[49] M. H. Sun and D. M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, 2010.

[50] Y. Xu, M. Bailey, B. Noble, and F. Jahanian. Small is better: Avoiding latency traps in virtualized data. In *Proceedings of ACM SoCC*, 2013.

[51] L. L. You, K. T. Pollack, and D. D. E. Long. Deep store: An archival storage system architecture. In *Proceedings of ICDE*, pages 804–815, 2005.

[52] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of VEE*, pages 534–533, 2009.

[53] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite. Optimizing vm checkpointing for restore performance in vmware esxi. In *Proceedings of USENIX ATC*, pages 1–12, 2013.

[54] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng. Multi-level selective deduplication for vm snapshots in cloud storage. In *Proceedings of Cloud*, pages 550–557, 2012.

[55] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of FAST*, pages 1–14, 2008.