ConSnap: Taking Continuous Snapshots for Running State Protection of Virtual Machines

Jianxin Li, Jingsheng Zheng, Lei Cui, Renyu Yang
State Key Laboratory of Software Development Environment
School of Computer Science and Engineering, Beihang University
Beijing, China 100191
{lijx, zhengjs, cuilei, yangry}@act.buaa.edu.cn

***Abstract*—The reliability of data and services hosted in a virtual machine (VM) is a top concern in cloud computing environment. *Continuous snapshots* reduces the data loss in case of failures, and thus is prevailing for providing protection for long-running systems. However, existing methods suffer from long VM downtime, long snapshot interval and significant performance overhead. In this paper, we present ConSnap, a system designed to enable taking fine-grained *continuous snapshots* of virtual machines without compromising VM performance. First, ConSnap adopts the COW (copy-on-write) manner to save the memory pages in a lazy way, and thus decrease the snapshot interval to dozens of milliseconds. Second, we only save the incremental memory pages on the basis of the last snapshot in each epoch to reduce the snapshot duration, and thus mitigate VM performance loss. Third, we propose a *multi-granularity* space reclamation strategy, which merges the unused snapshot files to achieve storage space saving, as well as fast recovery. We have implemented ConSnap on QEMU/KVM and conducted several experiments to verify its effectiveness. Compared with the *stop-and-copy* based incremental snapshots, ConSnap reduces the performance loss by 71.1% $\sim$ 10.2% under Compilation workload, and 14.5% $\sim$ 4.7% for the Ftp workload, when the interval varies from 1s to 60s.**

***Index Terms*—cloud computing; virtual machine; continuous snapshots; copy-on-write; space reclamation**

I. Introduction

Virtualization, an essential technology for consolidating applications into virtual machines (VM) effectively, is one of the cores of cloud computing systems. To provide continuous cloud services, the reliability and security of the running state as well as the data in virtual machines has become a challenge and hot issue. Statistics reveal that the hardware malfunction occupies 44% of the cause of all data loss, while the value is 49% for software errors, such as user misbehavior, software corruption, and malicious agents (viruses and so forth) [1].

Continuous protection of virtual machines is a prevalent technique to provide protection for long-running systems by capturing and preserving the complete execution history of protected VMs [14] [6]. It allows the user or administrator to restore VM state to any preserved point to minimize the data loss in case of system failures. As a result, this technique is widely used for high availability, forensics, debugging, and system administration [5] [9] [6] [13].

The nowadays approaches for continuous protection of VMs fall into two categories: *log-replay* and *continuous checkpointing*. *Log-replay* records all input and non-deterministic events of the VM so that it can replay them deterministically later when the VM fails [6] [12]. However, deterministically replaying the logging events relies heavily on the virtual machine monitor (VMM), specifically, the architecture of CPU. Besides, for multi-core CPU environment, the deterministic replay depends on the exact order in which CPU cores access the shared memory, which is difficult to implement and suffers super linear performance degradation with increase of the number of virtual CPUs.

Continuous checkpointing, or called *continuous snapshots* technique, captures the entire execution state of the running VM at relatively high frequency and saves them into persistent devices, then the saved state can be utilized for quick recovery upon serious corruptions [14] [5]. A virtual machine snapshot saves a complete copy of the VM state comprising of CPU registers, memory state, and device states, etc. Most of the existing methods for *continuous snapshots* leverage *stop-and-copy* [5] [15] or *pre-copy* [14] mechanism, and employ incremental mechanism [7] which only saves the modified pages since last epoch to reduce the snapshot size. Unfortunately, due to the heavy weight of virtual machines, the snapshot size is still large, and thus leads to long VM downtimes, coarse-grained snapshots, and significant VM performance overhead.

In this paper, we propose ConSnap, a fine-grained *continuous snapshots* system to protect the running state of virtual machines. ConSnap employs *copy-on-write* (COW) mechanism to create snapshots for virtual machine in each epoch which decreases the snapshot intervals to dozens of milliseconds. The system tracks the modified pages during snapshot intervals and saves them only once. Besides, ConSnap saves the modified pages into a buffer temporarily and flushes it into disk in a schedulable way, with the aim to mitigate VM performance loss. We have implemented a prototype system based on QEMU/KVM [10] without modifications of Guest OS or applications, and performed several experiments to evaluate the efficiency of ConSnap. ConSnap outperforms both *stop-and-copy* and *pre-copy* based snapshot approach when saving the *full content* memory of VM. Compared with the *stop-and-copy* based incremental snapshots, ConSnap reduces the performance loss by 71.1% $\sim$10.2% under Compilation workload, and 14.5% $\sim$ 4.7% for the Ftp workload, when the interval varies from 1s to 60s.

In particular, the major contributions of our work are

summarized as follows:

- In each epoch, we overlap the actual memory saving with the VM running period by the COW manner to reduce the VM downtime.
- We propose a VM *continuous snapshots* approach by combining incremental snapshot with the above method, while utilizing an adoptive *flush speed controller*, achieving fine-grained snapshot intervals and insignificant performance degradation.
- We provide a *multi-granularity* strategy to reclaim the storage space.

The rest of this paper is organized as follows. The next section provides some related works. Section III introduces the design and implementation of ConSnap in details. The evaluation results are presented in Section IV. The last section is our conclusion and future work.

II. Related Work

Most of the existing methods for *continuous snapshots* leverage *stop-and-copy* [5] [15] or *pre-copy* [14] mechanism. The mainstream virtualization solutions, e.g. KVM [10], Xen [3], VMware [2], only provide the simple and intuitive *stop-and-copy* snapshot mechanism, which suspends the VM, copies the entire state, and resumes the VM. Therefore, the VM downtime per epoch depends on the size of the entire VM memory, or the pages modified since last snapshot if the incremental mechanism [7] is adopted, which only save the modified pages in each epoch to reduce the memory size. As a result, this method results in substantial VM downtimes, and suffers from significant VM performance degradation.

Pre-copy mechanism, which is initially designed for VM live migration [4] to mitigate the VM downtime, saves memory pages in an iterative way. It only copies the modified part compared with the former round and would stop the VM until the number of remaining memory pages can be saved in a short interval. VNsnap [8] is a system to create the distributed snapshot for a closed network of VMs, leveraging Xen live migration function to minimize system downtime. However, the performance of *pre-copy* approach is highly relevant to the access pattern of the application inside the VM. Firstly, the snapshot will suffer from a long duration to be completed (even fail) particularly for write-intensive workloads. Secondly, the host resource will be occupied for a long time because of the large number of copy rounds, affecting the VM execution a lot. Thirdly, the actually saved VM state is not the state of the specified snapshot point. Due to the long and unpredicted VM downtime and snapshot duration, *pre-copy* based snapshot is less suited to taking fine-grained *continuous snapshots*.

High availability systems, Remus [5] and Kemari [15], have modified the live migration mechanism of Xen to enable high frequency VM checkpoints between the primary and backup hosts. Though based on the live migration method, both Remus and Kemari essentially adopt the *stop-and-copy* based incremental snapshots mechanism, because each epoch is just the final *stop-and-copy* phase of migration, suspending the VM and saving the pages. Remus optimize the *stop-and-copy* method on Xen by reducing the number of inter-process requests required to suspend and resume the guest domain, removing xenstore from the suspend/resume process, copying touched pages to a staging buffer rather than delivering them directly, etc. However, Remus causes heavy runtime overhead to the application in the primary VM.

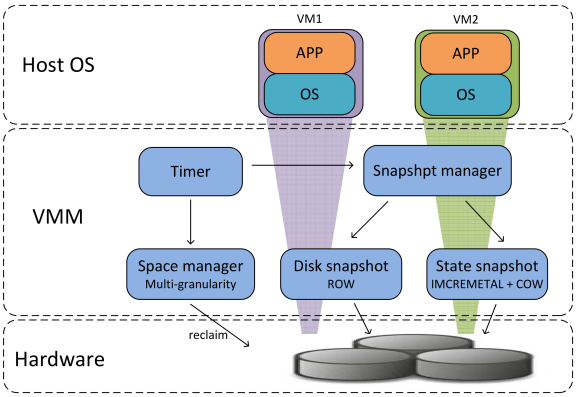


Fig. 1: Design of ConSnap

Ta-Shma et al. [14] presented an approach for virtual machine time travel using Continuous Data Protection and checkpointing which is based on Xen's live migration. Due to lack of experimental results, we cannot evaluate the protection granularity and the performance overhead, but the granularity cannot be much finer because of the long-duration *pre-copy* scheme. Besides, their system performs space reclamation by simply defining an *accessibility window* which specifies the period of time allowing reverting to, and data outside it is to be reclaimed. However, this strategy discards all the saved VM states out of the *accessibility window*, resulting that these states cannot be reverted to in case that it is necessary.

III. Design and Implementation

This section begins with an overview of ConSnap, followed by the design of *single-epoch snapshot*, *continuous snapshots*, and space reclamation respectively, with some implementation details in the final part.

A. Overview

Figure 1 illustrates the design of ConSnap. ConSnap mainly consists of a Timer Controller, a Snapshot Manager and a Space Manager. The Timer Controller can specify time periods to trigger the Snapshot Manager to do snapshots and control the Space Manager to reclaim storage space.

Snapshot manager manages each *single-epoch snapshot* which consists of disk snapshot and state snapshot. We choose *iROW* disk format [11] to realize the disk snapshot, which achieves insignificant and stable VM downtime when taking disk snapshot by improving the traditional *redirect-on-write* method.

For state snapshot, ConSnap only saves the pages modified since the last snapshot to reduce the snapshot size, and puts off the actual memory saving to the VM running period to reduce

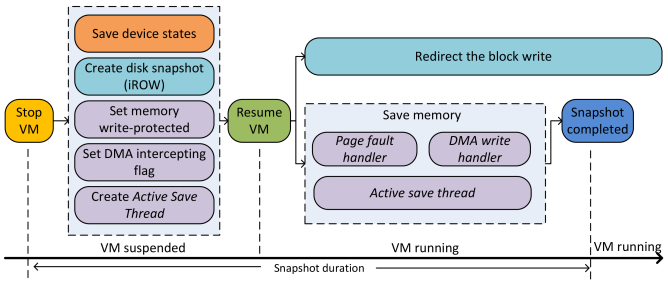


Fig. 2: The workflow of a *single-epoch snapshot*

the VM downtime. Then we use three save manners to save memory pages and design an adaptive *flush speed controller* to mitigate the VM execution performance degradation.

The Space Manger is designed to reclaim the space occupied by the unneeded metadata and snapshot data, and a *multi-granularity* strategy is performed.

B. Single-epoch Snapshot

A VM snapshot is a consistent view of the VM state at an instantaneous point in time, consisting of the memory state, disk state, and device states such as the CPU state, network state, etc. The *single-epoch snapshot* of our ConSnap is a two-stage process, corresponding to VM suspended and VM running respectively, shown in Figure 2.

Once triggered to take a snapshot, ConSnap suspends the executing VM immediately, followed by saving the device states, creating disk snapshot provided by the *iROW* block driver, setting all memory pages write-protected, setting the flag to enable intercepting DMA operations, and creating a background thread. These operations are lightweight enough so that the VM suspension stage can be completed in a few dozens of milliseconds.

Once the suspend stage is completed, we resume the VM. During the VM running, a disk write operation to a specified block will be redirected to a new block by the *iROW* block driver. At the same time, we save the memory pages by following three components collaboratively and concurrently. To save each page only once, a *save_bitmap* is used to record whether a page has been saved.

Page Fault Handler. During the VM running, the write operation on a write-protected memory page will result in a page fault, which can be intercepted in the VMM layer. Then we handle the page fault by saving the corresponding page and its neighbors, and removing their write-protection flags. By saving neighbor pages, we can benefit from the feature of memory locality to reduce the frequency of page fault.

DMA Write Handler. Most virtualization solutions simulate the DMA schema for I/O related devices. This schema dirties the memory pages without triggering the page fault, so we intercept the DMA write operations with another handler and save the corresponding pages.

Active Save Thread. The thread is running in the background to traverse all the guest memory pages in sequence and save the pages which have not been saved. A *single-epoch snapshot* is completed when this thread traverses all the pages one pass.

All of the above three manners copy the memory pages to a buffer temporarily, and then flush it into the snapshot file when it is filled up. However, the VM execution may be affected if the buffer is flushed at a high rate by the *active save thread* due to the I/O contention between the snapshot manager and the VM applications. Therefore, we design a *flush speed controller* to mitigate this effect. To control the flush speed, we divide the buffer into several slices, and suspend the *active save thread* for a period after flushing each slice into snapshot file. The control procedure is illustrated by the Pseudo-code 1. We assume each VM on a host machine has a maximum I/O bandwidth reserved by administrator (R_{IO}). Since applications in the VM occupy a part of the bandwidth (R_{VM}), so only the left bandwidth ($R_{available}$) should be used to flush the slices. Therefore, to guarantee the bandwidth occupied by the flush operation is less than the $R_{available}$, the flush period should be equal to T_{total} at least, calculated by the size of the slice and the available bandwidth. If the actual time to flush the slice (T_{flush}) is less than T_{total}, we suspend the thread for a period, denoted by $T_{suspend}$. Obviously, we design the *flush speed controller* by a predicted way, assuming that the VM I/O rate in this flush period is same as that in last period. With the controller, we reduce the impact on VM performance while achieving a relatively short snapshot duration simultaneously.

Pseudo-code 1 *Flush speed control*

```
1:  for each slice in Buffer do
2:      R_VM = S_sectors / T_interval
3:      R_available = R_IO - R_VM
4:      T_total = S_slice / R_available
5:
6:      flush this buffer slice into snapshot file
7:
8:      if T_flush < T_total then
9:          T_suspend = T_total - T_flush
10:         suspend snapshot procedure for T_suspend
11:     end if
12: end for
```

C. Continuous Snapshots

The *continuous snapshots* utilizes the single-epoch method described above in each epoch, and only saves the memory pages modified since the last snapshot. Figure 3 illustrates the workflow of *continuous snapshots*.

To record the pages modified since the last snapshot, we utilize another bitmap *dirty_bitmap*. We take a *full-content* snapshot firstly, saving the entire memory pages into the snapshot file, and then take a series of incremental snapshots, only saving the dirty pages. To take a snapshot, we first suspend the running VM. Compared with the *single-epoch snapshot*, a extra step during the VM suspended period is updating the *save_bitmap*. The bitmap should be set as all 1 for the *full-content* snapshot, indicating that all the memory pages

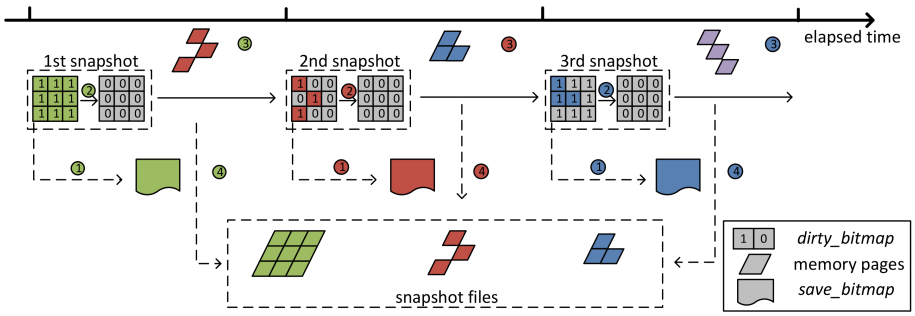


Fig. 3: The workflow of *continuous snapshots*. (1) Copy *dirty_bitmap* (records the pages modified since last snapshot) to *save_bitmap* (records whether a page has been saved in one epoch) when a snapshot is to be taken; (2) Set memory pages write-protected, and clean up *dirty_bitmap*; (3) Set corresponding bit of *dirty_bitmap* when a page fault is intercepted; (4) Save pages marked in *save_bitmap* into snapshot files by three save manners.

should be saved in this epoch; for the following incremental snapshots, the *dirty_bitmap* is copied to the *save_bitmap*, meaning that only the changed pages need to be copied (Figure 3, step 1). Then, to mark the dirty pages, we clean up the *dirty_bitmap* and set all memory pages write-protected (Figure 3, step 2), followed by the VM restarted. During the VM running, we intercept page faults and set the corresponding bit in *dirty_bitmap* to mark that this page has been modified and should be saved in next epoch, then remove the protection flag (Figure 3, step 3). Besides, we check the *save_bitmap* and save the corresponding page into snapshot files if this page has not been saved, then reset the corresponding bit in *save_bitmap* (Figure 3, step 4). This epoch snapshot is completed actually when the modified pages are all saved, indicated by the emptied *save_bitmap*. Obviously, we only save a subset of the entire memory pages in each epoch, so the snapshot size can be greatly reduced, and the snapshot duration can be reduced at the same time. In addition, our method puts off the page saving to the VM running period based on the COW manner, and thus achieves only a few dozens of milliseconds VM downtime. Therefore, both the reduced snapshot duration and the lightweight VM downtime contribute to the fine-grained *continuous snapshots* without significant VM performance degradation.

D. Space reclamation

After long-term running of the VM and *continuous snapshots* with fine granularity, large amount of the storage space will be occupied by the snapshot files. To reclaim the storage space, existing systems typically define an *accessibility window* to specify the period of time allowing reverting to, and data outside the *accessibility window* is to be reclaimed, such as in Ta-Shma's system [14]. However, this method discards all the saved VM states out of the *accessibility window*, resulting that these states cannot be reverted to in case that it is necessary. So we provide a *multi-granularity* strategy, which enables specifying a group of granularities to reclaim the snapshot files created in different time periods. The snapshots taken in different time periods are reclaimed into snapshots with different granularities. The earlier the snapshots are taken,

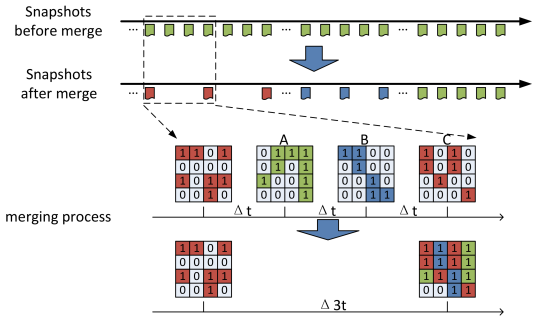


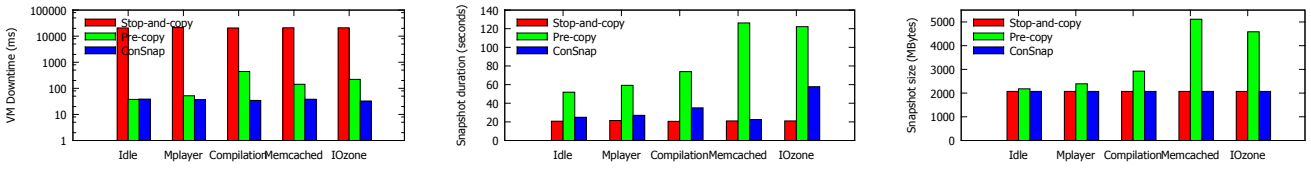
Fig. 4: An example of space reclamation

the coarser the granularity should be. By this way, we can reclaim the storage space without losing all the earlier states. Figure 4 shows a simple example of the space reclamation, reclaiming 1t-granularity snapshots to 2t-granularity and 3t-granularity snapshots respectively.

Note that we save memory pages in an incremental manner, a complete memory state of the VM in one point is also based on the earlier snapshots. Therefore, before reclaiming a snapshot file (target snapshot), we should merge some parts of its data to its child snapshot file first. To do this, we scan the bitmaps of the child snapshot and the target snapshot, determining which pages should be merged based on the condition: the page is not present in the child's bitmap but in the bitmap of the target snapshot. Then we copy the page from the target snapshot to the corresponding position of the child snapshot file. Figure 4 also gives an example which reclaims two snapshot files (A & B) by merging them to the child snapshot (C). We first merge snapshot file C with B, followed by merging the merged file with A. Then the storage space occupied by the two ancestor snapshots (A & B) can be reclaimed and used by the following snapshots.

E. Implementation details

1) Completeness and Consistency of Pages: Since the memory pages can be saved by aforementioned three manners concurrently, guaranteeing the completeness and consistency of the saved pages should be focused in the implementation of



(a) downtime (b) duration (c) size

Fig. 5: Snapshot metrics of *single-epoch snapshot with full content*

ConSnap. The completeness means that we have saved all the memory pages which should be saved, and the consistency indicates that the contents of the saved pages are all same as the contents of the memory pages at the snapshot point. Therefore, we use a bitmap named *save_bitmap* to indicate whether a page has been saved in a snapshot process. When each of the save manners prepares to save one page, we consults *save_bitmap* first and save this page only if the corresponding bit has been set, then reset the bit. An emptied *save_bitmap* means the completion of a *single-epoch snapshot*, and thus we guarantee the completeness of the pages. In general, to ensure the consistency, *save_bitmap* should be locked before each access of the three save manners. However, we give our following analyze to prove that the lock is unnecessary. For the *page fault handler* and the *DMA write handler*, they are actually not executed at the same time since that they occupy different time slices, so they cannot cause the inconsistency. The *page fault handler* and the *active save thread* also cannot lead to the inconsistency because the *active save thread* is read-only for the memory pages, and the same is true for the *DMA write handler* and the *active save thread*. A worst-case scenario may be that the *page fault handler* intercepts a page fault and starts to save the page, while the *active save thread* prepares to save the same page. If the *page fault handler* finishes saving the page (followed by resuming the VM) before the *active save thread*, the Guest OS may dirty the page, and thus the *active save thread* saves a inconsistency page version. Therefore, to guarantee the consistency of the saved memory state, the saved pages are subject to the *page fault handler* version or *DMA write handler* version when we flush the buffer to the snapshot file. Without the lock operation, we avoid the time overhead resulted by the queue for the lock, which may affect the VM execution significantly.

2) Write-protection and Handling page fault: ConSnap calls *cpu_physical_memory_set_dirty_tracking* in QEMU, which finally calls *kvm_mmu_slot_remove_write_access* in KVM to set each memory page write-protected. Once a VM-Exit occurs because of the page fault, *handle_ept_violation* in KVM would be called. So we can handle the page fault in this function by setting the corresponding bit in *dirty_bimap* to record the modified page, and checking whether the page has been saved by the other save manners by *save_bitmap*. If the bit is 1, we exit to QEMU with the flag *EXIT_REASON_CONSNAP* and the *guest frame number* (*gfn*) of this page. Then QEMU handles the exception by saving the corresponding memory page into the snapshot file and removing the write-protection flag. Finally, QEMU resumes the VM by the *ioctl* operation with *KVM_RUN* flag.

3) Flush Speed Control: The key concern of the *flush speed controller* is calculating the VM I/O rate. QEMU implements the VM block I/O in function *bdrv_aio_writev* and *bdrv_aio_readv* in *block.c*, and one of their arguments is *nb_sectors*, indicating the number of sectors in this I/O operation. A sector occupies 512 bytes in QEMU by default. So we accumulate the numbers when the functions are called, and use the sum to calculate the parameter $S_{sectors}$ when a flush operation is triggered.

IV. Evaluation

A. Experimental Setup

We conduct the experiments on a DELL Precision T1500 workstation with Intel Core i7-860 2.8GHz CPU, 4GB DDR3 memory, 500G SATAII hard disk. We configure 2GB memory for the virtual machine unless specified otherwise. The operating system on physical server and virtual machine is debian6.0 with 2.6.32-amd64 kernel.

We apply following application benchmarks as the VM workloads in our evaluation. 1) **Idle** workload means the VM does nothing except the tasks of OS self after boot up. 2) **Media Player** workload means a video is played in the VM with nothing else. 3) **Compilation** represents a development workload involves memory and disk I/O operations, and we compile the qemu-kvm-0.12.5 in our experiments; 4) **Memcached** is an in-memory key-value store for small chunks of data, and the server replies a corresponding value for a request containing the key. We set memcached server in one VM, and configure mcblaster as client in another VM to randomly request the data. 5) **IOzone** is a file system benchmark utility, which is used by us to simulate the workload with a large number of file read and write operations; 6) **Ftp** is a standard network protocol used to transfer computer files from one host to another host.

Our experiments include two parts: *single-epoch snapshot* and *continuous snapshots*. In single-epoch experiments, we compare the following three snapshot methods, all saving the entire memory pages into the snapshot file in the local host. 1) ***Stop-and-copy*** based snapshot, the default snapshot method used in QEMU/KVM, suspends the VM while creating snapshots. 2) ***Pre-copy*** based snapshot, which is implemented by modifying the live migration mechanism in QEMU/KVM, saves memory pages in a iterative way while VM is running,

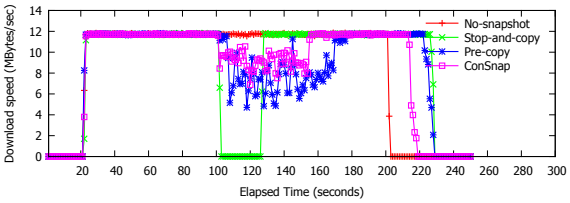


Fig. 6: Ftp download speed

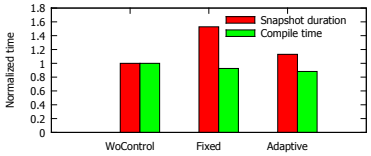


Fig. 7: The effect of *flush speed controller*

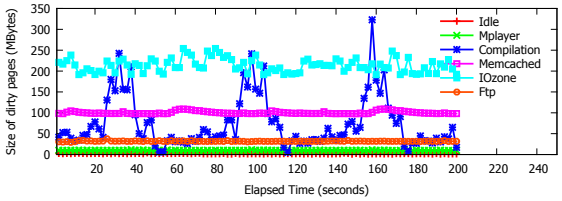
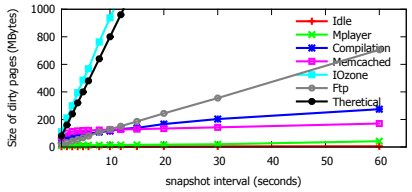
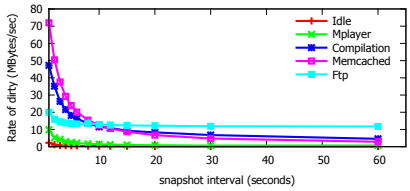


Fig. 8: Size of ditry pages with the elapsed time



(a)



(b)

Fig. 9: Size and average rate of dirty pages with different intervals

and suspends the VM when the remaining pages can be saved in a short interval. 3) **ConSnap** snapshot, our method introduced in this paper, suspends the VM for a lightweight time, and saves memory pages during the VM running period.

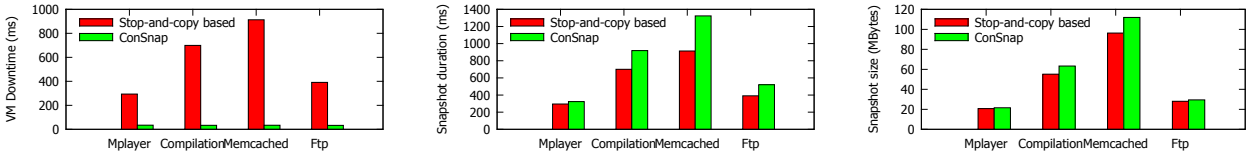
In *continuous snapshots* experiments, we only compare the *stop-and-copy* based incremental snapshot and our ConSnap, because the duration of *pre-copy* based snapshot is long and unstable, which can be shown by the experimental results of the *single-epoch snapshot*.

B. Single-epoch Snapshot with full content

1) Snapshot metrics: We create a series of *full-content* snapshots at interval of 150 seconds when workloads are running in the VM, and record the VM downtime, snapshot duration and snapshot size of each snapshot, which is shown in Figure 5. From Figure 5(a), we can see that the VM downtime of ConSnap is only about 36 milliseconds, short and stable; and the *stop-and-copy* method has a significant downtime, about 20 seconds; while the *pre-copy* based approach results in various downtimes from the workloads, e.g., 37.8ms for Idle situation and 443ms for the Compilation workload, mainly because of the different sizes of the remaining pages in last copy round. For snapshot duration, shown in Figure 5(b), the *stop-and-copy* method is the shortest, while the *pre-copy* based method has the longest time and is various with the workloads, this is because that it includes several rounds to save memory pages. Our ConSnap's duration also is dependent of the specific workload because of the *flush speed controller* which is designed to mitigate the performance loss of VM execution. In spite of this, the duration of ConSnap is up to half of the *pre-copy* based approach's. Owing to saving only one copy for each memory page, the snapshot size of the *stop-and-copy* method and ConSnap are basically the same as the VM memory size, while the *pre-copy* based method results in larger and various sizes for different workloads, as illustrated in Figure 5(c). One thing should be explained is the results related with the IOzone workload. Though IOzone only involves file operations, it dirties a large number of memory pages by DMA write operations, resulting in long snapshot duration and large snapshot size in *pre-copy* based experiments.

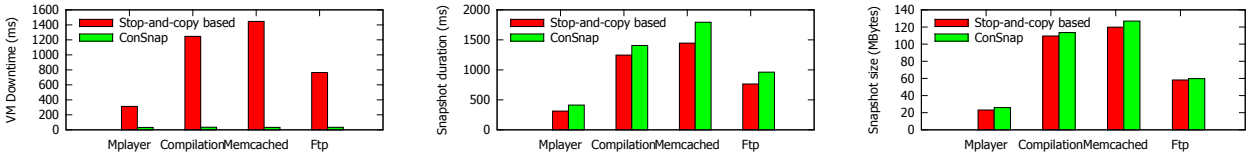
2) Performance impact on VM: To evaluate the impact of snapshot on VM execution, we conduct experiments with a VM as Ftp server and another VM as Ftp client. We run a script to record the data receiving rate of the client VM per second firstly; start downloading a file from the server at the 21st second; then take snapshot with three methods at the 101st second. The results are shown in Figure 6. Without snapshot, the file transfer can be finished in 180 seconds, while the transfer time increases to 208s, 204s, and 195s with once *stop-and-copy* snapshot, *pre-copy* based snapshot and ConSnap snapshot respectively. The results demonstrate that our ConSnap has the minimal overall impact on the VM execution.

3) Flush speed control: We conduct a set of experiments in this section to evaluate the effect of the *flush speed controller*. We take the Compilation workload as example to compare the snapshot duration and VM performance loss of our ConSnap with those of other two strategies, as shown in Figure 7. In the figure, "WoControl" denotes ConSnap without speed control; "Fixed" means the strategy that the *active save thread* is suspended for a fixed period after flushing each buffer slice, and it is 5 milliseconds in this experiment; "Adaptive" denotes the strategy introduced in this paper, in which the suspending time after each flush depends on the I/O rate of the VM, and we set S_{slice} as 1M bytes, R_{IO} as 60 Mbytes/s



(a) downtime (b) duration (c) size

Fig. 10: Snapshot metrics of *continuous snapshots* with 2s interval



(a) downtime (b) duration (c) size

Fig. 11: Snapshot metrics of *continuous snapshots* with 5s interval

respectively. The duration and compilation time are normalized based on that of "WoControl" situation respectively. Though snapshot method without *flush speed controller* obtains the minimum duration, it affects the VM execution most, about more 10% loss than others. The other two methods have a similar performance loss, while our strategy achieves the shorter duration.

C. Continuous Snapshots

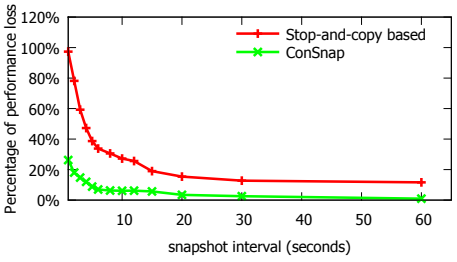
1) Dirty page rate: In *continuous snapshots*, the size of the memory to be saved in each epoch, which depends on the number of the pages modified during two adjacent snapshots, is one of the most critical factors on the snapshot performance. Thus, we conduct a series of experiments to show the number of dirty pages and the average dirty rate of each aforementioned workload with different time periods.

We start each workload in a VM, and record the number of dirty pages every 2 seconds for 100 times in the first group experiments. Figure 8 shows the results, demonstrating that different workloads have different memory access patterns. The Idle situation, Mplayer workload, Memcached workload and Ftp workload all have a relatively stable number of dirty pages, while the Memcached has more dirty pages relatively. The number of dirty pages of IOzone workload fluctuates in a small scope with a high level absolute value, while the Compilation has a significant fluctuation.

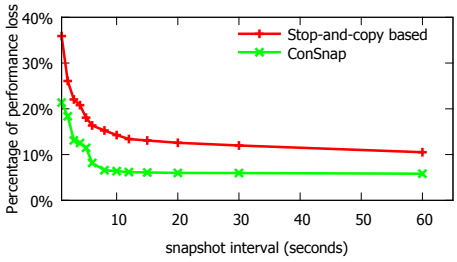
To obtain a more comprehensive result of the dirty rate with various time periods for all representative workloads, we continue conducting several experiments with periods from 1s to 60s. Figure 9 illustrates the dirty size and average dirty rate of each aforementioned workloads, corresponding to different intervals. In Figure 9(a), "theretical" denotes the maximal speed of file writing of the host machine in theory, and is set as 80Mbytes/s. We can see that the dirty rate of IOzone is faster than the maximal speed, and its dirty size increase linearly if the VM memory is enough. Therefore, VMs with similar workloads is unsuitable to do incremental snapshots, and we suggest taking *full-content* snapshots with a coarser granularity. For the other five workloads, Idle and Mplayer situations have insignificant dirty rates; Ftp has a basically linear dirty size; Compilation and Memcached have a high dirty rate when the interval is less than 10s, then the rate levels off for the following coarser intervals. The above results imply that taking *continuous snapshots* would obtain different results depending on the specific workload and the snapshot interval.

2) Snapshot metrics: We take 2s and 5s as the interval representatives to show the snapshot metrics of the *continuous snapshots*. Due to the long and unstable duration, *pre-copy* based method is unsuitable for the *continuous snapshots*, so we only compare our ConSnap with the *stop-and-copy* based incremental snapshots. We take a *full-content* snapshot firstly, and then start the application in the VM, followed by a series of incremental snapshots at the fixed interval. The results are shown in Figures 10 & 11. We can see from the figures that the largest advantage of our ConSnap is the insignificant VM downtime. For the 2s interval, the downtime of the *stop-and-copy* based approach occupies about 15% $\sim$ 35% of the snapshot interval, and the percentage is 6% $\sim$ 25% for the 5s interval. Oppositely, ConSnap has a stable and short downtime, independent of the workload and the interval. The snapshot duration of ConSnap is a little longer than that of the *stop-and-copy* based method because that ConSnap has taken the VM performance impact into account by speed control. However, the impact on the VM execution led by a little longer duration is far less than that by many multiples of downtime, which will be proved in the next section. For the snapshot size, the snapshot size of ConSnap is also a little larger than the *stop-and-copy* based method, which can be explained by the reason that the VM running time of ConSnap is longer than that of the latter, resulting in more modified pages.

3) Performance impact on VM: We evaluate the VM performance impact resulted by the *continuous snapshots* in this section, using Compilation and Ftp workload. We take a *full-content* snapshot at first, followed by a series of



(a) Compilation



(b) Ftp

Fig. 12: Pencentage of VM performance loss with different intervals

incremental snapshots using our method and the *stop-and-copy* based incremental method. During the incremental snapshots, we make the Compilation for several times, or download a large size file from the VM, respectively. We measure the VM performance in terms of the Compilation time and the average download speed respectively. Figure 12 shows the percentage of the VM performance loss based on the performance without snapshots. The results are various with the snapshot intervals. ConSnap reduces the performance loss by about 71.1% $\sim$10.2% compared with the *stop-and-copy* based approach in the Compilation experiments, while the values are 14.5% $\sim$ 4.7% for the Ftp experiments. Besides, the maximal performance loss percentage in Compilation experiments of our ConSnap is about 26.3%, and most of the values are below 20%, which can be considered reasonable for a general-purpose system. Overall, the performance loss decreases with the increase of snapshot interval, and our ConSnap achieves a less performance loss over the *stop-and-copy* based method in every case, both at different intervals and with different workloads.

V. Conclusion and Future Work

This paper presents a fine-grained *continuous snapshots* system ConSnap to protect the running state of virtual machines. In each epoch, ConSnap puts off the actual memory saving to the VM running period to reduce the VM downtime, and proposes an adaptive flush speed mechanism to eliminate the I/O contention with the aim to mitigate VM execution performance degradation. For *continuous snapshots*, ConSnap employs incremental snapshots to only saving the pages modified since the last snapshot. Combined with the above *single-epoch snapshot* method, ConSnap achieves both fine-grained snapshots and low performance overhead. Besides, we introduce a *multi-granularity* strategy to reclaim the space storage. We have implemented ConSnap on QEMU/KVM platform, and conducted comprehensive experiments. The results show that ConSnap achieves sub-second interval *continuous snapshots* without significant performance loss for a variety of workloads.

In future, we will focus on the strategy to revert the VM state from a series of memory image files with insignificant time and resource overheads.

Acknowledgment

This work is supported by the 973 Program (No.2011CB302602), NSFCPrograms (Nos. 91118008, 61202424), China MOST grant (No. 2012BAH46B04), 863 project (No.2013AA01A213), NewCentury Excellent Talents in University 2010 and SKLSDE-2012ZX-21.

References

[1] Statistics about leading causes of data loss. http://www.protect-data.com/information/statistics.html.

[2] Vmware. http://www.vmware.com.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[5] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[7] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9. IEEE Computer Society, 2005.

[8] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 524–533. IEEE, 2009.

[9] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, 2005.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[11] J. Li, H. Liu, L. Cui, B. Li, and T. Wo. irow: An efficient live snapshot system for virtual machine disk. In *ICPADS*, pages 376–383, 2012.

[12] J. Li, S. Si, B. Li, L. Cui, and J. Zheng. Lore: Supporting non-deterministic events logging and replay for kvm virtual machines. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 442–449. IEEE, 2013.

[13] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 87–96. ACM, 2008.

[14] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor. Virtual machine time travel using continuous data protection and checkpointing. *ACM SIGOPS Operating Systems Review*, 42(1):127–134, 2008.

[15] Y. Tamura. Kemari: Virtual machine synchronization for fault tolerance using domt. *Xen Summit*, 2008, 2008.