Fast, Lightweight Virtual Machine Checkpointing

Michael H. Sun Douglas M. Blough
Georgia Institute of Technology

Abstract

Virtual machine checkpoints provide a clean encapsulation of the full state of an executing system. Due to the large nature of state involved, the process of VM checkpoints can be slow and costly. We describe the implementation of a a fast and lightweight mechanism of VM checkpointing for the Xen virtualization machine monitor that utilizes copy-on-write techniques to reduce the VM's downtime and performance overhead incurred by other forms of VM checkpointing.

1 Introduction

This paper the describes an implementation of a fast and lightweight checkpoint mechanism for virtual machines. A VM checkpoint is a consistent image of the complete state of a running VM at a given point in time. We introduce a copy-on-write (CoW) checkpointing technique for the Xen virtualization platform that is significantly faster and lighter-weight than current VM checkpointing methods.

VM checkpointing requires that a consistent view of a running VM's CPU state, memory, and device state (most notably, disk) be captured at various points in time. The most basic mechanism for checkpointing suspends the running VM, copies its entire state, and finally restarts the VM. Disk schemes that provide copy-on-write snapshot capabilities can eliminate the need to copy entire VM disks [1]. With VM memory sizes easily reaching hundreds of gigabytes, copying the entirety of the suspended VM's memory contents generally results in significant VM downtimes.

VM migration schemes, which can be viewed as a superset of checkpointing, have generally utilized *pre-copying* to drastically reduce the downtime incurred while a VM is migrated between hosts [1, 7]. Pre-copying is a technique that consists of a number of memory copying rounds while a VM is running, before the actual checkpoint/migration point. During each round, memory pages that have not yet been copied or have been recently modified by the running VM are copied out. After a number of copy rounds, the VM is suspended and the remaining *dirty* memory pages are copied out; the VM is then allowed to restart. By performing most of the memory copying concurrently with the running VM, a significantly smaller portion of the memory state needs to be copied while the VM is suspended resulting in substantially shorter downtimes. Pre-copying does introduce substantial resource overheads as copying is performed over a number of rounds; moreover, these overheads and the VM's downtime are sensitive to the write intensity of a VM's execution behavior and can grow to unacceptable levels with particular VM workloads. Currently, Xen, VMware, and KVM all utilize pre-copying in their migration schemes. Though all three only explicitly provide for the most basic VM checkpointing service that does not use pre-copying, pre-copying-based migration mechanisms can easily be adapted for checkpoint use.

We introduce a CoW-based VM checkpointing mechanism capable of tolerating write-intensive VM behaviors more effectively than pre-copy checkpoint mechanisms. Rather than performing any copying of state while a VM is suspended, a consistent view of a VM is taken by write protecting all of a VM's memory through the use of manipulating protection bits in shadow page tables. This constant time operation ensures that VM downtimes remain small even in the face of write intensive VM behavior. Post-copying then takes places in a single round, eliminating the overhead of redundantly copied pages. Unlike other work on CoW, post-copy migration/checkpoint schemes, our CoW-based checkpoint mechanism is completely transparent to the VM, requiring no modification to or assistance from the VM operating system. By enforcing our copy-on-write semantics through hypervisor-level components such as shadow page tables and emulated IO devices, generality of our

checkpoint mechanism is preserved.

2 Related Work

The major system virtualization platforms—Xen, VMware, KVM—currently only provide a primitive *suspend-save-restart* checkpoint mechanism in which significant VM downtimes are incurred. VM migration and VM forking, however, can both be seen as generalized forms of *VM checkpointing* and their bodies of work provide a number techniques adaptable for VM checkpointing.

The majority of VM migration schemes utlize *pre-copying*, a technique in which the bulk of state transfer is done prior to the actual migration point. Xen live migration [1], VMware VMotion [7], and KVM live migrationčitekivity:klv all iteratively pre-copy memory pages for a number of rounds, copying in each round those pages that have been dirtied since the last round until a small *writable working set* (WWS) can be identified or a threshold of rounds is reached. The VM is then suspended, the remaining dirty pages copied along with CPU and device state to the destination where the VM is restarted. By this reducing the state that must be transferred with the VM is suspended, the VM's downtime is greatly reduced.

High availability systems, Remus [3] and Kemari [8], have modified the live migration mechanism of Xen to enable high frequency VM checkpointing between primary and backup hosts. Remus and Kemari checkpoint efficiently by continuously repeating the final copy round of live migration; the VM is suspended or paused and memory dirtied since the last checkpoint is copied to from the primary to the backup. The high frequency of the checkpointing (as often as every 25ms) tends to keep the number of dirty pages copied during checkpoints small, resulting in acceptable VM downtimes. Both systems have also managed to further reduce VM downtime by eschewing the costly suspend/restart signaling of the current Xen implemenation in favor of lightweight event channels. Despite these improvements, checkpointing at such high frequencies incurs significant overhead penalties. Kernel compile benchmarks show more than 50% performance penalty when checkpointed at 50ms intervals and I/O intensive workloads such as SPECweb operate at 25% of their native performance. Reducing the frequency of checkpoints would increase the amount of dirty state that must be copied during a checkpoint, raising the VM downtime; the problem increases in severity with more write intensive workloads.

Post-copy based migration [5] is a technique that attempts to address the major drawbacks of pre-copy based migration: duplicate memory transfers across iterative copy rounds, and less than acceptable VM downtimes for write intensive workloads. It defers memory transfer to after the VM's CPU state has been migrated and the VM restarted at the destination. The memory state is then both actively *pre-paged* to the destination and *demand-paged* when the VM faults over a missing page. This ensures that no memory page is transmitted more than once, and that a VM's downtime depends only on the time to transfer a small portion of CPU state. A pre-paging strategy that actively fetches those pages spatially local to a faulting page access is used to reduce the number of demand page faults. Their implementation however, only supports PV guests as the mechanism of trapping memory accesses utilizes an in-memory pseudo-paging device in the guest. Our approach allows fully virtualized guests to run without modification.

A *VM fork* is an abstraction analogous to a process fork where a running VM spawns child VMs that are clones of itself. Potemkin [9] and the Difference Engine [4] offer VM forks that operate locally within single machine. CoW techniques implemented in a manner similar to our work allow the VM forks to occur quickly by deferring the duplication of memory pages until the contents of the pages actually differ between VMs. Snowflock[6] extends the VM fork abstraction in a distributed fashion, providing the ability for clones to be created across a cluster of machines. In essence, a combination of cloning and migration, Snowflock leverages the same CoW technique used by Potemkin and the Difference Engine to capture an immutable image of the parent VM as well as a demand-paging mechanism similar to Hines et al. to provides missing pages to VM children. Since the goal of VM fork work has been the efficient creation of VM clones that run *simultaneously*, current VM fork implementations are not ideally suited to being checkpoint mechanisms; they do not provide a means of obtaining dependable, persistent checkpoints and require that extra resources be allocated to support a running clone rather than a checkpoint.

Colp et al. [2] have announced and provided patches to Xen that utilize CoW techniques similar to ours in order to provide fast checkpoints. Their effort was performed concurrently, though independently with ours. From a high level the efforts appear congruent, so we therefore provide details of our specific implementation.

3 Design

An effective VM checkpointing solution requires that a consistent view of a large amount of state be captured in an efficient manner. A consistent view entails the procurement of the memory state, CPU state, and I/O device state of an VM at an instantaneous point in time. The checkpoint mechanism should minimally interrupt VM execution, incur a small resource footprint, minimally

degrade the performance of applications in the VM, and complete in a brief time period.

We define the *VM downtime* to be the largest length of time that a VM's execution is suspended during the checkpoint process. [2] The resource footprint is measured by the amount of memory, CPU time, and disk space utilized by the checkpoint process. The impact of checkpointing on VM applications is measured by the differences in benchmark results—both application-level benchmarks and microbenchmarks—between when the benchmark is run with and without checkpointing enabled.

Pre-copying approaches exhibit poor performance under write-intensive VM behavior because the writeable working set of memory pages that must be copied while a VM is suspended becomes so large that the VM incurs a significant (greater than 1 sec) period of downtime. This problem will be exacerbated given the current growth trends of CPU performance greatly outpacing memory bus bandwidth. In contrast, our approach utilizes a copy-on-write mechanism that enables a consistent view of a suspended VM to be captured in an extremely short period of time irrespective of the write intensity of the VM workload. During VM suspension, memory pages are only write protected, deferring memory copying to after the VM has been allowed to restart. This also ensures that each memory page will be copied only once, avoiding the wasteful duplicate copies that can occur during pre-copy rounds.

We explain the process of our CoW checkpoint approach which captures a persistent image of the entire VM's state at each checkpoint.

Step 1: Initiate Checkpoint
The checkpoint management process initiates a checkpoint by suspending the executing VM. This entails descheduling the VM, quiescing I/O devices, and unmapping shared memory regions. This is the start of the VM's downtime.

Step 2: Capture Consistent View
The VM's CPU and device state is copied to persistent storage as part of the checkpoint image. All of the VM's memory pages are then write protected and a snapshot is performed on the VM's disks.

Step 3: VM Resumption & Post-copy
Devices are reconnected, shared memory remapped, and the VM is restarted. Write-protection faults are intercepted and the contents of the faulting memory pages are copied to the checkpoint image, and the memory page is made writable again. Other sources of memory writes such as DMA are intercepted and CoW performed on those memory pages. Concurrently, the checkpoint manager process actively copies out the VM's memory pages, making them writable again after they have been copied to the checkpoint image.

Step 4: Checkpoint Completion
After every memory page of the VM has been copied to the checkpoint image, either by the checkpoint manager process or through a CoW fault, the checkpoint is complete.

4 Implementation

We have implemented our copy-on-write checkpoint mechanism on the Xen 3.2.2 platform. This Xen platform consists primarily of the Xen hypervisor and a privileged VM called dom0 in which libraries, utilities and management components execute. It is capable of running para-virtualized (PV) domains as well as fully virtualized domains (HVM). Our implementation involves modifications to the hypervisor, some library routines in libxc, and the xend management daemon. The implementation was built to support 32-bit non-PAE operation for both PV and HVM guest VMs.

4.1 Background

The Xen live migration code was the building block from which we built our checkpoint mechanism. The migration code consists of code from both the xend management daemon, the libxc library, and some internal features of the hypervisor. In particular, we take advantage of the *log-dirty mode* that live migration uses, in which shadow page tables are used to keep track memory writes. In this mode, a guest VM (PV or HVM) loads shadows copies of what it believes to be its page tables. This allows the protection bits to be modified in a way transparent to the actual guest. In log-dirty mode, all pages are write protected so that all memory writes can be intercepted through faults. We utilize this mode to capture attempted writes to memory, copy the original memory page before modification to copy-on-write (CoW) buffers, and then allow the memory write to occur.

4.2 Special behavior

Log-dirty mode via shadow page tables doesn't catch all writes to memory however. For HVM domains, I/O is emulated through QEMU, which performs DMA without regard to the domain's shadow page tables. Thus these I/O writes must also be intercepted for copy-on-write snapshots to work. We modify the QEMU daemon to make a hypercall to the hypervisor informing it which pages will be modified. Additionally, the Xen hypervisor

itself emulates certain operations that write to memory without regard for page table protections. For our implementation, we had to track down all of these operations in the hypervisor and ensure a copy-on-write occurs before memory is modified if the domain is being checkpointed.

4.3 Checkpoint execution flow

A copy-on-write (CoW) checkpoint is initiated just as a VM migration would be: through the xend control plane daemon running in domain 0, a specialized and privileged management domain. The daemon makes a call to a function in the libxc library to start the checkpoint. At this point, the VM is still running. CoW code in libxen then allocates memory buffers and other state in preparation for handling the CoW memory snapshot. After all preparations have been made, libxen signals to the xend daemon that the domain should be suspended. xend then disconnects devices, records the QEMU daemon's state, and performs a snapshot on the virtual disk used by the domain. This can be achieved in a number of ways, but for our implementation we utilize a ZFS virtual disk and it's snapshotting capabilities.

The xend daemon then signals the CoW code in libxen that the domain has been suspended. At this point a snapshot of the VM will be taken. Log-dirty mode is turned on which has the effect of write protecting the VM's memory through the shadow page tables. Special CPU state and context are also captured. The xend daemon is then signaled to restart the domain.

A virtual snapshot of the domain has then taken place, as any modifications from the time of VM suspension are tracked. A libxc tool now begins to copy all of the memory pages of the running domain and writes them out to a checkpoint file. After the libxc process copies all the memory, the CoW memory buffers that hold the vital unmodified memory pages are written out to the checkpoint file, replacing any memory pages that might have been copied while the VM had already resumed execution.

4.4 Optimizations

We investigated a number of *fault reduction* optimizations that would attempt to copy those memory pages more likely to fault during the checkpoint process as a means of improving performance. We implemented a *last n address space optimization* where pages dirtied from the last *n* address spaces seen in page directory register cr3 would be copied first to avoid faulting. In our evaluation as will seen later, we realized the majority of the cost of checkpoint comes from the copying overhead of producing a entire checkpoint, and that individual faults were insignificant in comparison. This led us to look into incremental checkpoints, where only recently modified pages are written out to a incremental checkpoint file. That work has not been completed at the time of this report.

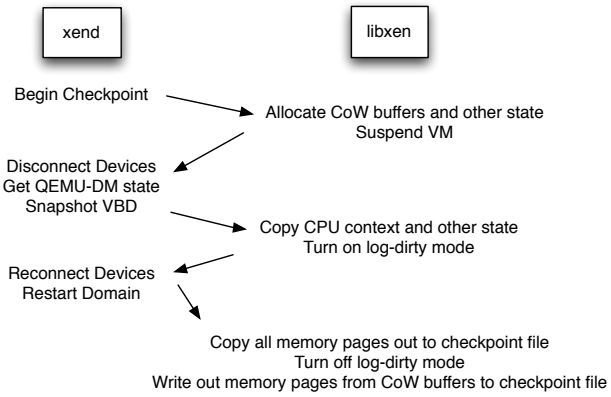

Figure 1: Execution flow of checkpoint mechanism

5 Evaluation

The two most important performance metrics our mechanism strove to improve were downtime and overall performance overhead on the running VM. We evaluate the performance of our checkpoint mechanism against other possible methods of VM checkpointing, namely a standard full stop checkpoint, and one based on pre-copying. Our testbed consists of a machine with a 2.4 Ghz Intel Core Duo E6600 with VT support, 2GB of RAM and a 7200 RPM SATA hard drive.

We began our evaluation with a setup consisting of a Damn Small Linux (DSL) distribution 256 MB HVM VM running idly and measuring the downtime each checkpoint incurred under the standard checkpoint mechanism, our copy-on-write mechanism and a checkpoint mechanism that used pre-copying. We made a simple modification to the Xen live migration code to create checkpoints instead of migrating VMs. The results of the mean of three trials can be seen in Figure 2. Clearly, the use of CoW checkpointing improved the overall downtime by more than 76% compared to the next closest mechanism.

We then evaluate how well our checkpoint mechanism performs on a VM running a kernel compile workload. A 512 MB Arch Linux HVM VM is loaded with a standard Linux kernel compile while checkpoints are taken at different frequencies. Figure 3 shows for all intervals, our CoW-based checkpoint mechanism outperforms the pre-copy based mechanism.

We also compare the overhead the differing checkpointing mechanisms have on our kernel compiling VM. Figure 4 shows that other than at the highest checkpoint-

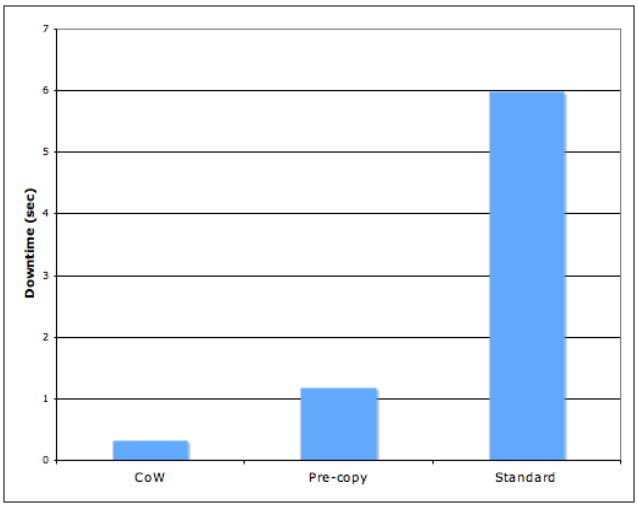

Figure 2: Downtime of Damn Small Linux checkpoint

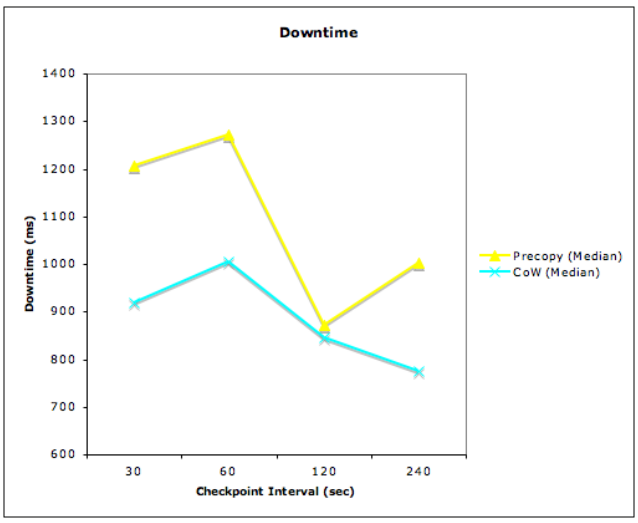

Figure 3: Downtime of precopy-based vs. CoW-based checkpointing

ing frequency (30 sec), our CoW-based solution has a lower overhead than the pre-copy based mechanism.

6 Conclusion

We have designed and implemented a copy-on-write-based checkpointing solution for Xen that supports both PV and HVM guests. It is fast and lightweight, generally outperforming the next best solution, a pre-copy-based checkpoint mechanism.

7 Acknowledgments

Would like to thank Mike Hibler and the Emulab team for their extraordinary help in setting up our testbed environment.

References

[1] CLARK, C., FRASER, K., HAND, S., HANSEN, J., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (2005), vol. 2, USENIX Association Berkeley, CA, USA, pp. 20–20.

[2] COLP, P. VM Snapshots.

[3] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), USENIX Association Berkeley, CA, USA, pp. 161–174.

[4] GUPTA, D., LEE, S., AND VRABLE, M. Difference engine: Harnessing memory redundancy in virtual machines.

[5] HINES, M., AND GOPALAN, K. Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging And Dynamic Self-Ballooning.

[6] LAGAR-CAVILLA, H., WHITNEY, J., SCANNELL, A., RUMBLE, S., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. University of Toronto–Department of Computer Science Technical Report CSRG-TR578 Impromptu Clusters for Near-Interactive Cloud-Based Services.

[7] NELSON, M., LIM, B., AND HUTCHINS, G. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (2005), USENIX Association Berkeley, CA, USA, pp. 25–25.

[8] TAMURA, Y. Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT. In *Xen Summit* (2008).

[9] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A., VOELKER, G., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review 39*, 5 (2005), 148–162.

Notes

[1] LVM, ZFS

[2] A suspended VM has been descheduled by the hypervisor and it's I/O devices quiesced; a VM becomes active again when it is scheduleable and its I/O devices have been reconnected

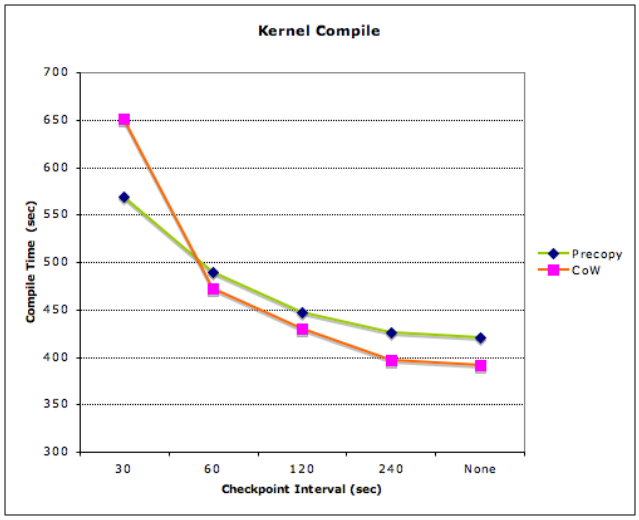

Figure 4: Compile times at various checkpoint intervals