Fast Restore of Checkpointed Memory using Working Set Estimation

Irene Zhang   Alex Garthwaite   Yury Baskakov   Kenneth C. Barr
VMware, Inc.
{izhang, alextg, ybaskako, kbarr}@vmware.com

Abstract

In order to make save and restore features practical, saved virtual machines (VMs) must be able to quickly restore to normal operation. Unfortunately, fetching a saved memory image from persistent storage can be slow, especially as VMs grow in memory size. One possible solution for reducing this time is to lazily restore memory after the VM starts. However, accesses to unrestored memory after the VM starts can degrade performance, sometimes rendering the VM unusable for even longer. Existing performance metrics do not account for performance degradation after the VM starts, making it difficult to compare lazily restoring memory against other approaches. In this paper, we propose both a better metric for evaluating the performance of different restore techniques and a better scheme for restoring saved VMs.

Existing performance metrics do not reflect what is really important to the user—the time until the VM returns to normal operation. We introduce the time-to-responsiveness metric, which better characterizes user experience while restoring a saved VM by measuring the time until there is no longer a noticeable performance impact on the restoring VM. We propose a new lazy restore technique, called working set restore, that minimizes performance degradation after the VM starts by prefetching the working set. We also introduce a novel working set estimator based on memory tracing that we use to test working set restore, along with an estimator that uses access-bit scanning. We show that working set restore can improve the performance of restoring a saved VM by more than 89% for some workloads.

Categories and Subject Descriptors D.4.5 [*Reliability*]: Checkpoint/restart

General Terms Measurement, Performance

Keywords Checkpoint/restore, Performance

1. Overview

Virtual machines (VMs) have become one of the primary computing environments in the cloud. One benefit of virtualization is the ability to save and restore the state of a running VM. This ability enables users to *suspend* an idle VM, pausing its execution, which can free up data center resources and reduce power usage. It also allows users to *checkpoint* a VM, freezing the VM at a single point in time, which is used for backup and fault tolerance. Outside of the cloud, VMs have become ubiquitous on desktops and laptops as well. These users save VMs for similar reasons, suspending VMs to allow them to easily return to previous work while freeing up resources for other applications, and checkpointing VMs to save their operating system and applications in a known good state.

In order to make suspending and checkpointing VMs practical, the hypervisor must be able to quickly restart the VM from a saved state. Users are more inclined to suspend an idle VM if it takes seconds to resume rather than minutes. The ability to restore quickly from a saved image can also enable many other useful features. For example, cloud service providers would like to be able to quickly boot stateless VMs, allowing them to dynamically allocate VMs as needed. This quick-boot feature could be simulated by restoring to a checkpoint taken right after boot if the restore process is very fast.

It is challenging to restore a saved VM quickly because the VM's saved state must be retrieved from slow persistent storage. This state includes some CPU and device state, and the contents of the VM's memory. Restoring the saved memory image, which is generally an order of magnitude or two larger than device state, takes the bulk of the time. The time to restore a VM continues to grow as the memory requirements of virtual machines increase while the speed of storage does not keep pace. Reflecting the growth in memory size for PCs, VMs 10 years ago typically had less than 256 MB of memory, while today 1 GB VMs are common on laptops and PCs and even larger VMs are used in datacenters.

The simplest approach to resuming a VM's execution is to restore all memory contents at once, along with the device state. We call this method *eager restore* because the hypervisor eagerly retrieves and sets up all guest memory before starting the guest. The time to restore a VM this way increases linearly with the size of the VM. This method worked well when VMs used relatively small amounts of memory, but cannot be sustained as the size of VM memory grows. It takes tens of seconds to retrieve a few hundred megabytes of saved memory contents from disk, but this time increases to minutes as the saved memory image becomes larger than a gigabyte.

An alternate approach is to load only the CPU and device state before starting execution, and restore memory contents in the background while the VM runs. Any time the VM touches memory that has not yet been restored, the hypervisor pauses the execution of the VM and retrieves that memory from disk. Because the memory contents are retrieved when the VM accesses that memory, we call this approach *lazy restore*. This approach is appealing because the VM starts much faster, after retrieving only a small amount of device state, and the cost does not grow with the VM's memory size. However, whenever the VM accesses a page of memory that has not yet been restored, the contents of that page must be faulted in from

VEE'11, March 9–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03. . . $10.00

disk on-demand, and the execution of the VM cannot be resumed until the page has been retrieved.

Handling these on-demand page faults can have a significant performance impact. Previous research has shown that users notice if the response time to a user action exceeds 100 milliseconds, and become frustrated if it exceeds 1 second [17, 23]. With disk seek times on the order of 10 milliseconds, this allows only around 10 accesses to unrestored memory before the user notices performance degradation and around 100 accesses before the user becomes frustrated. Unfortunately, even simple user actions like moving the mouse can require hundreds of memory accesses, giving some idea of the performance impact if all of those accesses cause a page fault. Our experiments and previous user reports reflect this—even though the VM is running during the lazy restore, it is so slow that it is essentially unusable. In fact, we found that the VM's performance can be impacted severely for so long during a lazy restore that eager restore can actually seem to perform better from the user's point of view. [1]

Neither the eager approach nor the lazy approach to restoring a saved VM offer ideal performance for the user. Instead, we propose using a hybrid approach that prefetches the working set of the VM's memory, then starts the VM and restores the rest of memory lazily. We call this approach *working set restore*. By prefetching the working set, we minimize the performance degradation after the VM starts because most of the memory that the VM accesses will have already been restored. Working set prefetching is commonly used for applications [11]; our contribution is to apply it to the problem of optimizing the restore of a saved VM.

The success of working set restore depends on the accuracy of the working set estimator. We discuss two estimators in this paper. The first is a working set estimator that uses access-bit scanning. While the access-bit scanning technique is simple, it still does a good job of finding the working set. Our experiments show that, for some workloads, the access-bit working set estimator was able to predict up to 98.6% of the VM's accesses while the VM is being restored. We also implemented a novel technique for checkpointed VMs that traces memory access during checkpointing to gather the working set. Our insight is that when a checkpointed VM is restored, it returns back to the point where checkpointing began. If the VM is deterministic, it should re-execute in the same way after it restores as when it was checkpointed. This means that we can almost perfectly predict the memory accesses of the VM after it starts by tracing memory accesses during the checkpointing. Our experiments show this to be true; we found that we could predict up to 99.2% of memory accesses using this technique.

It is difficult to measure the performance benefit of working set restore using existing metrics because they do not account for performance degradation after the VM starts. The two most commonly used metrics are the total time to restore the VM and the time until the VM starts. The total time to restore the VM does not account for the fact that, during a lazy restore, the VM starts (and the user may be able to use the VM) long before the entire VM is restored. Using the total time until the VM starts avoids this problem, but does not account for the fact that the VM might not be usable after it starts. This paper presents a new metric, called *time-to-responsiveness* (TTR), that better measures the time until the VM appears restored to the user (i.e. the time until the VM returns to normal operation). Time-to-responsiveness builds on the idea of a minimum mutator utilization (MMU) [3, 4, 14], a metric previously used to measure the ongoing overhead of a service like garbage-collection techniques. We extend MMU to give us the time until the overhead of the restore process has fallen to an acceptable level for the VM to appear restored.

The rest of this paper is organized as follows. In Section 2 we examine the problem with the two existing metrics for comparing restore techniques and explain our new time-to-responsiveness metric. Section 3 gives background on the checkpoint and suspend mechanism in VMware Workstation that working set restore is built on. In Section 4, we present the design and implementation of working set restore for VMware Workstation. This section also details our simple working set estimator using access-bit scanning and our new working set estimator that uses memory tracing. Section 5 evaluates the performance benefit of working set restore and how the accuracy of the working set estimator affects performance using our time-to-responsiveness metric. Section 6 discusses related work, and Section 7 concludes.

2. Measuring Restore Performance

One reason designing a better restore scheme is difficult is that there are no effective metrics to compare the performance of different restore schemes. Using the total time to restore or the time until the VM starts to measure performance can make eager restore or lazy restore appear optimal. However, neither metric reflects what is really important to the user, which is the time until the VM and its applications return to normal performance. With this observation, we propose a new time-to-responsiveness metric that better characterizes the user experience while restoring a saved VM.

2.1 Common Metrics

There are two commonly used metrics for measuring restore performance: the total time to restore the VM and the time until the VM starts. Both are appealing metrics because they are easy to measure and understand. The total time to restore the VM measures the amount of time it takes for all of the VM's memory to be copied from persistent storage to host memory. For eager restore, the total time to restore depends on the size of the saved VM and the speed of the persistent store. The performance of eager restore using this metric scales linearly with the size of the VM's memory. For this metric, eager restore is optimal because it allows memory to be linearly fetched from the disk as fast as possible, maximizing disk throughput. Lazy restore can take quite some time to completely finish because memory is restored only when it is accessed by a VM or touched by a slow-running background thread. Therefore, the performance is bounded by the memory size of the saved VM and the speed at which the background thread retrieves memory. If the VM is not using most of its memory, the background process could take a long time to restore all of memory because it is not important to quickly restore memory that is not being used. However, this metric does not account for the fact that the VM has started and the user might be able to use the VM long before the lazy restore completely finishes.

The time until the VM starts measures the time until the VM is able to begin execution of the guest. This metric better accounts for the fact that the VM might start before it is completely restored. For eager restore, this metric is the same as the total time to restore the VM because the VM does not start until it is completely restored. The performance of a lazy restore using this metric depends on the size of the device state of the saved VM and disk speed. Unlike eager restore, the performance of lazy restore for this metric does not decrease with the size of memory. Using this metric, it appears that lazy restore is optimal because the device state is the minimal state that must be restored before the VM can start, so the VM starts as soon as possible. However, we observed that the VM is not immediately usable during a lazy restore because of performance degradation caused by accesses to unrestored memory. Depending on the severity of the performance degradation, the user cannot

[1] For this reason, VMware Workstation 7.0 switched from lazy restore back to eager restore.

necessarily use the VM and its applications although the VM has started. While the time until the VM starts better accounts for restore performance than the total time to restore, it still does not measure when the VM becomes responsive to the user.

2.2 Mutator Utilization

Instead of measuring the total time to restore or start the VM, a better metric for restore performance should measure the time until the VM *appears* to be restored, which is when the performance degradation is no longer apparent to the user. In order to pinpoint that time, we must be able to measure the performance degradation of different restore schemes after the VM starts.

Measuring the performance degradation of a service has been well studied. In particular, work on the impact of pause times caused by garbage-collection makes use of the notion of a *minimum mutator utilization* (MMU) to evaluate and compare different garbage-collection techniques [3, 4, 14]. Pause times are periods in which an application—the *mutator*—is inactive while the garbage collector (GC) performs some amount of work. In the context of restore, the VM and its applications can be considered the mutator and the pauses caused by accesses to unrestored memory can be considered similar to pause times. The MMU is the lowest utilization achieved by the application during its execution for some application-specific window of time; the window reflects both the scheduling needs of the application and the fact that GC pauses may not be uniformly distributed or of equal duration.

Unlike a service like garbage collection that is on-going, a restore is limited by the size of the VM's memory. When the VM first starts it may access a lot of unrestored memory, but as more of its memory is restored, either by accesses from the VM or by the background thread, the accesses to unrestored memory will slow and eventually stop. It is important to measure this change over time, so a single minimum mutator utilization is not sufficient. But we can apply the idea of a mutator utilization as a way of measuring performance degradation over time. We can track the performance degradation caused by the ongoing restore process by measuring how much time was spent restoring memory and how much time was available to the guest OS and other "normal" processes, the mutator, during a particular window of time after the VM starts. For that window, if most of the time is spent restoring memory, then the VM's performance must suffer. If most of that window was spent running the guest OS or doing other normal work, then the VM must be running normally.

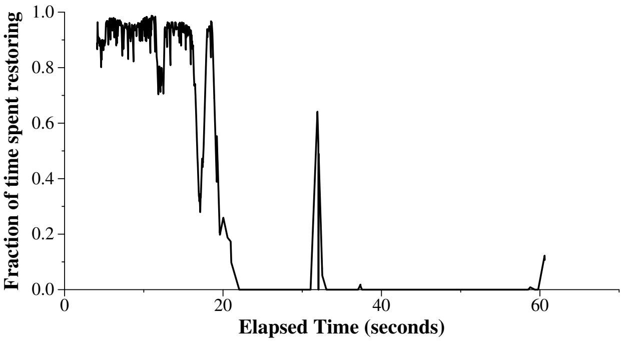

Figure 1. Fraction of each second restoring pages accessed by the VM during the lazy restore of a 1 GB VM running Red Hat Enterprise Linux 4. For the first 20 seconds, almost 0% of the time is available to the VM, which will lead to severe performance degradation. After the first 20 seconds, the VM will see minimal overhead from memory being lazily restored except for a few spikes, causing small lags.

As an example, Figure 1 shows the fraction of time spent restoring memory during the lazy restore of a typical Linux VM. It is clear from Figure 1 that there must be significant performance degradation for the first 20 seconds, as nearly all time is devoted to restoring memory. After the first 20 seconds, the VM is probably running normally because there is very little overhead (other than a few spikes) imposed by restoring memory and most of the time is spent doing normal work in the guest or elsewhere.

2.3 Time-to-responsiveness (TTR)

The utilization graph shows how the restore process affects the VM after it starts. In order to pinpoint the time when the performance degradation of the restore process is no longer apparent to the user, we extend the notion of MMU to calculate the time-to-responsiveness (TTR) metric for restoring a saved VM. We define TTR as:

$$\mathrm{TTR}(w, u) = \min\{t | \mathrm{MMU}_{\forall t' \geq t}(w) \geq u\}$$

where the time-window w and desired utilization u are application-specific parameters. TTR is the earliest time such that from that point forward in the execution of the VM, we achieve a minimum level of utilization for the VM. The level of performance is entirely application-dependent because some applications can tolerate more performance degradation than others, so we look at TTR as a curve instead of a single number. In general, we choose to fix w and then plot TTR as minimum utilization over time, which shows, for each point in time t, the minimum mutator utilization from t until the end of the restore. To find the actual time until the performance degradation is no longer apparent from such a graph, we need to determine the minimum utilization that the VM can tolerate. For example, for VMs and applications that can operate normally with at least 90% of the execution time each second, the minimum utilization for TTR would be 90% with a one second window.

Figure 2 shows the TTR graph for the example from Figure 1 with window sizes of 0.1 seconds, 1 second and 10 seconds. From such a graph, one can choose a suitable level of performance and identify the earliest point in the restore process where the minimum utilization is achieved. Such a point is the TTR for the corresponding level of performance. One can see that when the minimum utilization is 0, the TTR is also 0, so a VM that can tolerate any degree of performance degradation and still operate normally can be considered restored right away. When the level is 100%, the TTR is the time that the last page was restored on-demand. VMs that require all of the system time (i.e. cannot tolerate any performance degradation) cannot be considered fully restored until there are no more pages to be restored on-demand. Section 5 discusses how the TTR metric correlates to actual application performance.

To compare restore schemes using TTR, we can compare TTR for each scheme for a particular window and minimum utilization. Alternatively, we could plot the TTR curves for each scheme, giving a comparison over a range of applications. The TTR for an eager restore is always the same for all window sizes and minimum utilizations because eager restore does not impact the performance of a VM after it starts. We show the TTR of a lazy restore in Figure 2; the TTR of eager restore for the same VM is 32 seconds. Using TTR to compare, it becomes less obvious whether eager restore or lazy restore are optimal. For the example VM, the TTR of both eager and lazy restore is about 32 seconds, if the VM can tolerate a minimum utilization of 80% for a window size of 1 second. With TTR, one can see that both eager and lazy restore are optimal for certain classes of applications. In this example, if the VM requires a minimum utilization of more than 80% for a window size of 0.1 seconds, which would indicate that it is running a very time-sensitive application, then eager restore would be the better choice. If the VM can tolerate a minimum utilization of 80% for a window

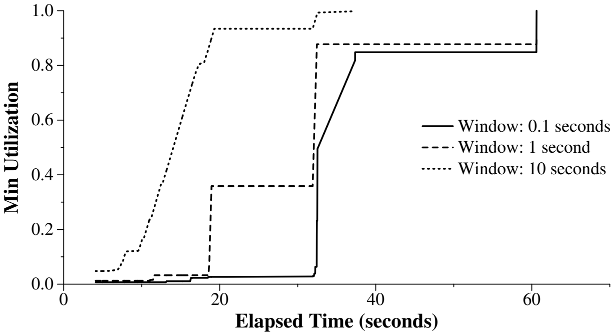

Figure 2. Minimum utilization of an idle RHEL VM during a lazy restore as shown in Figure 1. For each level of performance, the x-axis shows the corresponding TTR. Ideally, the percentage rises to 1 as quickly as possible, returning the system to 100% utilization.

size of 10 seconds, then lazy restore would be the better solution. This conclusion makes sense intuitively: lazy restore is better for applications that can tolerate more performance degradation and eager restore is better for applications that cannot.

3. Saving and Restoring in VMware Workstation

We implemented a prototype of working set restore using the suspend and checkpoint features in VMware Workstation. Workstation is a hosted hypervisor that runs guest OSes on top of a host OS, in contrast to a native hypervisor, which runs directly on hardware [22]. This section describes how Workstation saves and restores VM memory for suspend and checkpoint, providing context for the design of working set restore discussed in the next section. Both suspend and checkpoint use the same mechanism, so working set restore improves the performance of both.

3.1 Memory Model

In a hosted hypervisor, the hypervisor must depend on the host to allocate memory for the VM. Workstation backs guest memory using a memory-mapped file, called the paging file. In addition to being used for backing guest memory, Workstation directly uses the paging file as the VM's saved memory image for a suspend. Workstation uses a copy of the paging file as the saved memory image for a checkpoint to keep the format of the memory image identical for suspended and checkpointed VMs, so that one restore mechanism can be used for restoring both.

Workstation uses the memory pages allocated by the host for the paging file to store pages of guest memory. Workstation sets up page tables to allow the guest to directly access this memory. The paging file is a raw data file and stores the VM's memory sequentially by guest physical address. For example, the first 4096 bytes of the paging file are the first page of guest physical memory and so on. The advantage of keeping the paging file unformatted and memory-mapped is that the host OS can do almost all of the work in managing the memory pages backing guest memory; the hypervisor only needs to manage memory mappings for the guest.

3.2 Suspend and Checkpoint Mechanisms

Workstation supports both suspending and checkpointing VMs. An important difference between the two features is that a checkpointed VM continues to run after the checkpoint. We will discuss the process of suspending a VM first as it is simpler. Since the VM stops running after the suspend, Workstation can directly use the VM's current paging file as the saved memory image. This makes the process of saving a VM's memory for suspend relatively easy. Workstation just unmaps the paging file, being sure to mark pages that are dirty. The host OS will handle writing dirty pages to disk.

Checkpointing a VM is more complicated because Workstation must make a copy of the paging file. Workstation cannot directly use the VM's paging file in this case because the VM continues running and using its paging file after the checkpoint. To minimize the impact on the running VM, Workstation stops the VM, saves the device state, and then restarts the VM and copies the paging file lazily while the VM continues running. A background thread linearly scans pages of memory, copying them to the saved paging file. Since the VM continues to modify memory, Workstation must ensure that no memory is overwritten before the background thread copies it to the saved paging file. It does this by using memory *write traces*. Write traces notify the hypervisor before the guest OS writes to a traced page. For checkpointing, Workstation puts write traces on all guest memory pages. When the guest tries to write to one of those pages, the write trace will trigger. Workstation will copy the page to the saved paging file and remove the write trace on the page, then allow the guest to write to the page. The background thread removes write traces as it scans through memory because the guest is free to write any page that the background thread has already copied.

3.3 Restore Mechanisms

VMware Workstation can restore suspended and checkpointed VMs. Workstation must preserve the state of a checkpointed VM when restoring it, so that the VM can be restored back to that checkpoint again, which is not necessary with a suspended VM. Since we do not have to worry about preserving the state of a suspended VM, we can directly use the saved paging file as the paging file for the running VM. The paging file is memory-mapped into the hypervisor's address space and used to back guest memory. Workstation touches each page of the paging file to force the host to allocate a page and bring the memory contents off of disk, and to set up meta-data for the page. Again, because the file is memory-mapped, the host will take care of allocating memory and reading the file from disk.

Workstation cannot directly use the saved paging file as the paging file for a checkpointed VM, so it must first make a copy of the old paging file. Workstation allocates a memory-mapped file, then copies the saved paging file into that file. This process has the same effect as touching each page when restoring a suspended VM; it causes the host to allocate a page, then Workstation copies the page from the old paging file into the page and sets up the meta-data for the page. At this point, Workstation can start execution of the VM.

The process just described is eager restore for suspended or checkpointed VMs. Workstation also supports lazy restore for both suspended and checkpointed VMs. Instead of touching or copying each page of the paging file before starting the VM, Workstation starts the VM first. While the VM runs, Workstation executes a separate thread in the background that touches, and copies if necessary, the pages of the paging file. To keep track of which pages have been touched, Workstation keeps a bitmap of restored pages. When the guest accesses an unallocated page, the hypervisor checks if the page is unrestored. If so, the hypervisor restores the page on-demand by touching it, which causes the host OS to allocate a page and fetch the contents from disk, and copying the page if necessary.

4. Working Set Restore

Working set restore is built on the features of VMware Workstation discussed in the last section. First, we discuss techniques for estimating the working set of the VM. Then, we explain the details of implementing working set restore.

4.1 Working Set Estimation

We implemented two working set estimators: one using access-bit scanning and another using memory traces to track memory accesses. Estimating the working set by access-bit scanning works for both suspend and checkpoint, while estimating the working set using memory tracing only works for checkpoint. Using access-bit scanning is more general-purpose, but imposes a constant overhead while the VM runs. Our implementation requires a hardware MMU [2], which most modern machines support. Access-bit scanning can also be done with a software MMU, but it imposes a greater overhead and misses some accesses. Using memory tracing only works for checkpoints, but it only imposes an overhead during the checkpointing process, which has an overhead anyway. Our experiments found using memory tracing to be more accurate at estimating the working set.

4.1.1 Access-bit Scanning

Our implementation of working set estimation using access-bit scanning works like a simplified CLOCK algorithm [8]. We use the flags in the page table entries to monitor the access patterns of the guest OS. We constantly loop over the page tables, scanning and clearing access-bits in the background while the VM is running. Any page whose access-bit has been set during the scan loop is marked as being part of the working set. The speed at which the scanner runs depends on the time it takes to restore the VM. In particular, we would like to scan memory at about the same speed at which it is restored by the background thread. This is simple because the speed of the background thread is set by a configuration option, so the speed of the scanner can be set using the same configuration option. By making the two rates equal, the working set we get is approximately the number of pages that the guest OS touches during the amount of time that it takes for a restore. If the page scanning rate is too fast or slow, then we will underestimate or overestimate the number of pages that we need to prefetch. We would like to avoid overestimating because a larger working set takes longer to copy when saving the VM and prefetch when the VM is being restored. Underestimating the working set, even by a small amount, can degrade performance during the lazy restore, as we show in Section 5.4.2.

4.1.2 Memory Tracing

The insight that motivates using memory traces for working set estimation is that when a checkpointed VM is restored, it returns to the point where lazy checkpointing of memory began. Thus, if the VM is deterministic, it will re-execute the same code that it executed during the lazy checkpoint period. Most VMs are not perfectly deterministic, as timing can change, so the prediction will not be perfect. However, our experiments show that this working set estimator comes close to having an oracle predict the memory accesses of the VM during the lazy restore.

To capture the memory accesses during the lazy checkpointing process, we add read traces to the write traces that Workstation uses already. Like write traces, read traces notify the hypervisor before the guest reads a page of memory. When notified of a read or a write during the lazy checkpoint, we note that the guest accessed that page and add it to our working set. If it is a write trace that triggered, we copy the page as usual and remove both traces. There is no need to leave the traces in place once the page has been added to the working set, so we remove them to reduce the performance overhead. If it is a read trace that triggered, we add the page to our working set and remove the read trace, leaving the write trace. It is not necessary to copy the page for reads, so we do not. This reduces the cost of handling a read trace since reads are more common than writes. We need to leave the write trace intact because the guest might still write to the page, at which point we would have to copy the page. For pages that are copied by the background thread, we leave the read and write traces in place. Leaving both traces in place ensures that we do not miss any page accesses, but adds some overhead to the lazy checkpointing process. However, the overhead is minimal because we only need to set a bit when those traces trigger, rather than copying a page to disk.

4.2 Saving and Restoring the Working Set

For a checkpoint or suspend, the working set must be saved with the VM after it is collected. Depending on the working set estimator, the working set is either collected continuously, for the access-bit scanning estimator, or during the checkpointing process, for the memory tracing estimator. After the working set has been collected, the pages in the working set are copied to a separate file, so that they are stored sequentially on disk. This copying incurs some cost when saving the VM. We save the working set separately from the memory image because the saved memory image shares the same format as the paging file. It is possible to avoid the cost of copying the working set by reorganizing the saved memory image to group the pages in the working set together, but that would negate the advantages of keeping the format of the paging file and the saved memory image of a checkpointed VM the same. We also save a bitmap of which pages in memory are in the working set, which is an efficient way of saving the guest physical address of each page in the working set.

Working set restore depends on both lazy and eager restore mechanisms in VMware Workstation. Working set restore first pre-restores the working set by copying pages from the saved working set file into the paging file before starting the VM. It marks those pages as restored in the bitmap that tracks restored pages. Workstation then starts the VM and restores the rest of memory in the background while the VM runs. Like lazy restore, if the VM accesses unrestored memory, it causes a page fault and Workstation restores (by either touching or copying) the page on-demand.

5. Evaluation

This section evaluates the performance of working set restore. We answer the following questions:

- Does working set restore improve overall performance?
- Does time-to-responsiveness correlate to user experience?
- How do eager restore, lazy restore and working set restore compare using TTR?
- Does working set size affect TTR for different restore techniques?
- How accurate are our working set estimators?
- Does the accuracy of the working set estimator affect the performance of working set restore?

5.1 Test Setup

We evaluated the performance of working set restore using a mechanical hard disk. All of the experiments in this paper were run with dual 2.3 GHz AMD Opteron 2376 Quad-Core processors, 4 GB of memory, and a Seagate model ST3750330AS hard disk with the specifications shown in Table 1. The AMD Shanghai processor supports hardware virtualization, allowing us to use hardware page tables for access-bit scanning. The host OS was 64-bit Ubuntu 9.10.

In general, our testing procedure consisted of the following steps:

1. Run the VM until the VM achieves steady state. Since the steady state is different for each workload, we define the steady state separately for each experiment in the following sections.

Table 1. Disk Specifications for Seagate ST3750330AS [20]

Interface	SATA 3.0 Gb/s
Capacity	750 GB
RPM	7200 RPM
Cache	32 MB
Average Latency	4.16 ms
Random read seek	< 8.5 ms
Random write seek	< 9.5 ms

2. Save the VM along with the working set.
3. Ensure that none of the saved VM state is cached on the host by clearing out the host buffer cache using `sync` and `echo 3 > /proc/sys/vm/drop_caches` to drop all file caches.
4. Restore the saved VM image with one of the restore techniques.

To ensure fair comparisons between restore techniques, we restored the same checkpointed VM for each technique. The simple eager and lazy schemes ignored the working set information saved in the checkpoint. We saved the working set as estimated by both access-bit scanning and memory tracing for each checkpoint, so that we can compare the two techniques with the same checkpointed VM. Working set restore also works for suspended VMs, but we use checkpointed VMs because the experiments are easier to replicate with a checkpoint. The performance for restoring both are similar because they share the same restore mechanism.

For all of the experiments, we fix the window size for TTR at 1 second and only draw the minimum utilization curve for that window size. We believe that 1 second is an appropriate window size for the applications that we use and is a small enough window to provide good user responsiveness.

Since the performance of eager restore is always about 32 seconds for our 1 GB VM, we do not include it the graphs. For our application tests, we plot lazy restore against working set restore using memory tracing estimator. The performance of working set restore using both estimators that we implemented is similar, so we do not include both in our graphs. Section 5.4.1 compares the two working set estimators that we implemented and shows the effects of the accuracy of the working set estimator on working set restore.

5.2 Simple Linux Test

First, we revisit the motivating example from Section 2. The VM in Figure 1 was running 32-bit Red Hat Enterprise Linux 4 Update 4 (Linux 2.6.9-42.0.3ELsmp). We consider the VM to have reached steady state 20 minutes after booting without any applications running, so we save the VM after running for 20 mins. In this experiment, the working set was about 3% of the 1 GB of memory. The `free` command reported 36 MB used and 230 MB used for buffer caches.

We compared the performance of lazy restore and working set restore by restoring the same saved VM image using both methods. Figure 3 shows the location in the guest's physical address space of prefetched pages and pages restored on-demand. For lazy restore, there were a large number of accesses to unrestored memory after the VM starts. There is some locality to the accesses. The cluster of accesses in the very low physical address space is from the kernel. The cluster of accesses in the higher physical address space is from applications. The locality of application accesses is mostly because the VM is newly booted and has not run many applications yet, so the physical address space is not yet fragmented.

Working set restore prefetches exactly those regions of memory where most of the accesses to unrestored memory were during the lazy restore. Once the VM starts, there are less accesses to unrestored memory for working set restore compared to lazy restore. The next section shows how this reduction in accesses to unrestored memory leads to less performance degradation after the VM starts.

5.3 Application Testing

We evaluated the end-to-end application performance of working set restore using two sample applications: SPECjbb and MPlayer. Our goal was to determine the performance improvement of working set restore for applications and show how TTR correlates with application performance. We selected these applications because they are representative of a range of workloads and have a metric that clearly shows the effects of performance degradation during a restore. We are interested in MPlayer because it has a small, active working set (primarily its frame buffer) and it cannot tolerate lag. SPECjbb is interesting because it has a larger working set, with some parts of the working set being more active than others, and its performance is not time-sensitive.

Application performance was measured as the instantaneous performance of the application over time for a relevant metric. We found the steady state for the application by running the application by itself in a VM until the chosen metric stabilized. To determine when the application returned to normal operation on restore, we measured the time between the start of the restore and when instantaneous application performance reverted to one standard deviation of the steady state.

5.3.1 SPECjbb Setup

The Standard Performance Evaluation Corporation's Java Business Benchmark (SPECjbb®2005) implements a Java application server in a simulated three-tier system [21]. We ran the benchmark in a virtual machine with 1 GB of memory and 1 virtual CPU. The guest operating system was a 32-bit Ubuntu 8.10 Server (Linux 2.6.27-11-server) running a Java virtual machine (Sun's JVM 1.5.0_17) configured with a 768 MB heap. We chose this benchmark because it exhibits little spatial locality, making it a challenge for simplistic working set estimators and a good stress of working set restore.

The standard implementation of SPECjbb runs for a fixed amount of time (several minutes) and outputs the number of *business operations* achieved during that amount of time as BOPS: business operations per second. To capture the instantaneous performance of SPECjbb, we measured the number of business operations completed during each second of the test and plotted the BOPS over time.

When running in a VM, the BOPS metric is relative to guest time. Unfortunately, such performance metrics can be inaccurate because of different timing behavior in virtualized environments [24]; these timing effects are exacerbated while the VM is being restored. Thus, we modified the benchmark to respond to periodic requests from an external monitoring computer. That computer sent a periodic, timestamped status request to the virtual machine over a UDP channel. The modified workload replied with the number of elapsed business operations. This allowed us to monitor the performance of SPECjbb in the VM over time using wallclock time.

We instantiate two warehouses, the recommended set up for a single CPU VM, which have a working set of 190 MB. The steady state number of BOPS per second for SPECjbb depends on the speed of the host machine running the VM. In this case, the steady state performance was around 8307 BOPS, with a standard deviation of 440 BOPS.

5.3.2 MPlayer Setup

Enjoyment of multimedia playback relies on good interrupt service latency and low jitter. Thus, we chose a video playback benchmark as another stress test of working set restore; any slowdown during

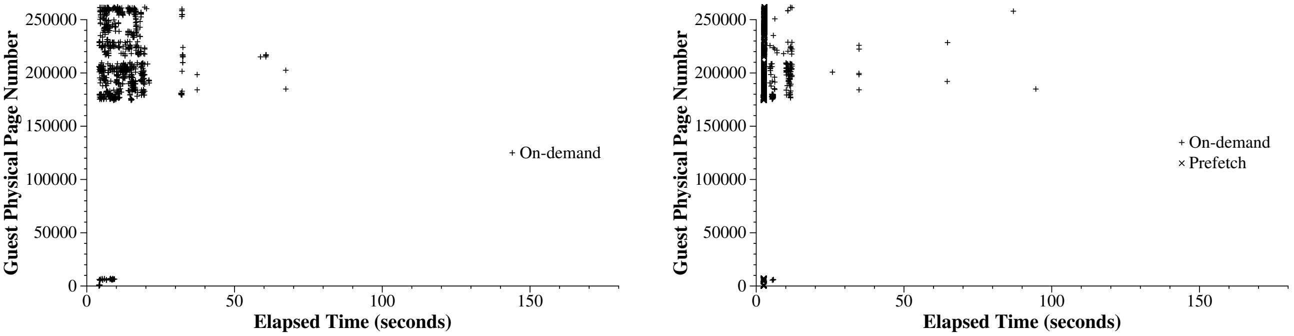

Figure 3. Comparison of location of page requests for a lazy restore (left) and working set restore (right). On-demand pages are unrestored pages accessed by the VM after it starts. Prefetch pages are pages in the working set that were prefetched by working set restore. Working set restore prefetches the pages that were restored on-demand during the lazy restore, reducing the number of accesses to unrestored memory after the VM starts.

the restore will be evident in the frames-per-second metric. We used MPlayer version 1.0rc2-4.2.4 in a 64b Ubuntu 8.04.3 LTS virtual machine (Linux 2.6.24-24-generic SMP). We played a 320x240 24bpp video clip encoded with AVC1 at 29.970 frames-per-second (fps). Thus, the steady state is ∼30 fps.

MPlayer was modified to augment its existing continuous progress report with timestamps. After each frame was decoded, the program logged the current time and the number of decoded video frames. This allowed us to compute a continuous fps metric. When playback drops below ∼30 fps, we consider the performance to be degraded.

5.3.3 Performance Evaluation

We first discuss the end-to-end performance impact of working set restore for our two applications. Figure 4 shows the instantaneous throughput of SPECjbb during a lazy restore and a working set restore. For the lazy restore, the throughput dropped to almost zero for the first 12 seconds of the restore. It does not return to steady state performance until 20 seconds after execution begins. For working set restore, the throughput returns to steady state performance within 3 seconds. However, the graph does not include the time required to prefetch the working set, which must be taken into account to compare with lazy restore. Working set restore estimates the working set to be 192MB, which takes 6 seconds to prefetch. Altogether, working set restore returns the application to steady state performance in 9 seconds. In comparison, eagerly restoring the VM would have taken 32 seconds, so both lazy and working set restore are faster than eager in this case, but working set restore is the fastest of the three.

Figure 5 shows the number of frames rendered by MPlayer during a restore. The VM was saved after 410 seconds of video playback. Performance drops for both restore methods simply because the VM stops in the middle of playback. With working set restore, playback recovers after 3 seconds and MPlayer returns to rendering 30 fps. With lazy restore, MPlayer is completely stopped for the first 20 seconds and only recovers after 35 seconds. Like the previous experiment, these graphs do not include the time to prefetch the working set. The working set size of MPlayer is fairly small because it mostly consists of the video buffer cache; working set restore only prefetched 41 MB of memory, requiring 1 second of prefetching. In this experiment, 1 second of prefetching saved 32 seconds of performance degradation after the VM starts, clearly showing the benefit of working set restore for workloads with a small working set. When we account for both prefetching time and performance degradation, working set restore takes 4 seconds to restore MPlayer back to steady state performance. Using eager restore, MPlayer would have resumed with no performance degradation after 32 seconds. In this case, lazy restore performed worse than eager restore, but working set restore offers much better performance than either eager or lazy restore.

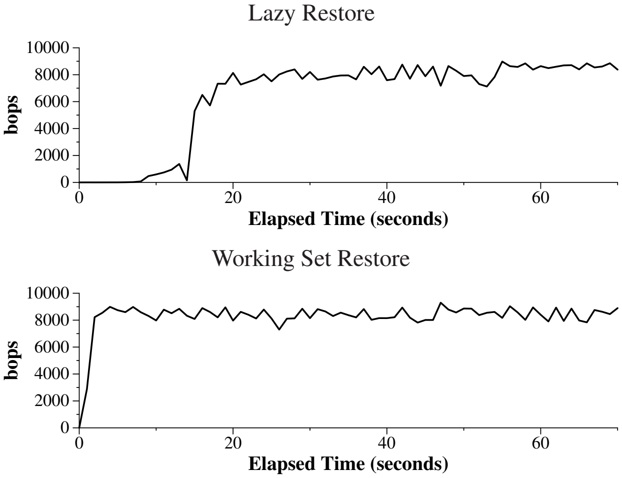

Figure 4. Comparison of instantaneous throughput of SPECjbb during a lazy restore and working set restore. The BOPS drops below steady state performance for the first 20 seconds of the lazy restore, but only for the first 3 seconds of working set restore. Working set restore takes an additional 6 seconds to prefetch the working set, which is 192 MB, so altogether working set restore improves performance by 11 seconds over lazy restore. Eager restore takes 32 seconds, so working set restore reduces restore time by 23 seconds when compared to eager restore.

5.3.4 TTR Evaluation

Next, we'll discuss the time-to-responsiveness of MPlayer and SPECjbb and how TTR correlates with the application performance shown in the last section. Figure 6 shows TTR for the SPECjbb application and Figure 7 shows TTR for the MPlayer application. Unlike the previous section, both graphs in this section begin when Workstation starts and include the time to restore the device state and prefetch the working set.

It is easy to see the advantage of working set restore from the TTR graph. Comparing Figure 6 and Figure 4 from the previous section shows that a minimum utilization for SPECjbb of 70% is required. For the SPECjbb workload, it takes lazy restore 22.8 seconds to gain 70% minimum utilization, which is close to the 20 seconds that it takes SPECjbb to return to steady state performance

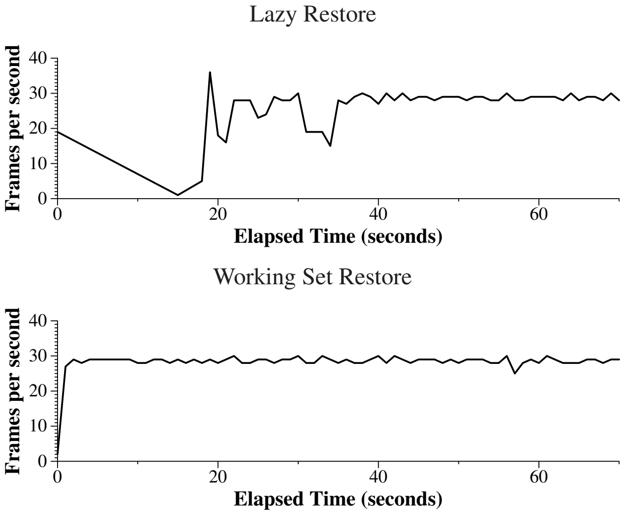

Figure 5. Comparison of the MPlayer frame rate during a lazy restore and a working set restore. During the lazy restore, the frame rate is impacted for 35 seconds. Working set restore only takes 1 second to restore the 41 MB working set. Using working set restore, the frame rate only slows for 3 seconds, giving a 31 second improvement in overall performance.

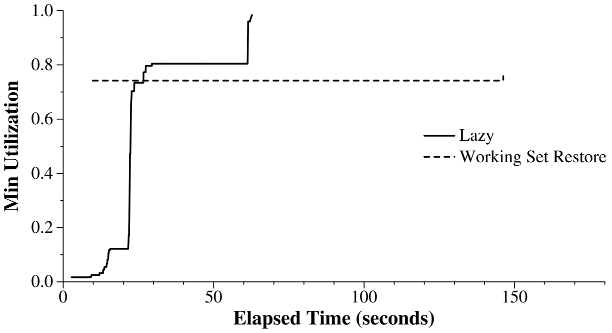

Figure 6. TTR for the SPECjbb application using lazy restore and working set restore. With working set restore, the TTR reaches 70% in 9.8 seconds seconds, including 6 seconds of prefetching. With lazy restore, there is no prefetching, but the TTR does not reach 70% until 22.8 seconds after the VM starts. Comparing with Figure 4 shows that a required minimum utilization of 70% correlates well with performance impact on throughput for SPECjbb.

during a lazy restore. In contrast, working set restore achieves this level of minimum utilization 9.8 seconds after starting. Working set restore impacts performance for 9 seconds total in the previous experiment, including the time to prefetch the working set.

Figure 7 shows the time-to-responsiveness for the MPlayer application. Again, this graph starts from when the hypervisor starts, so it includes the time to restore the device state and prefetch the working set. The MPlayer VM has a 128 MB SVGA frame buffer. The SVGA frame buffer is part of the virtual graphics driver, so it must be restored along with the other devices before the VM starts. This adds around 5 seconds to the time until the VM starts for both lazy and working set restore. Altogether, the total time until the VM starts is 7.4 seconds for lazy restore and 9.3 seconds for working set restore.

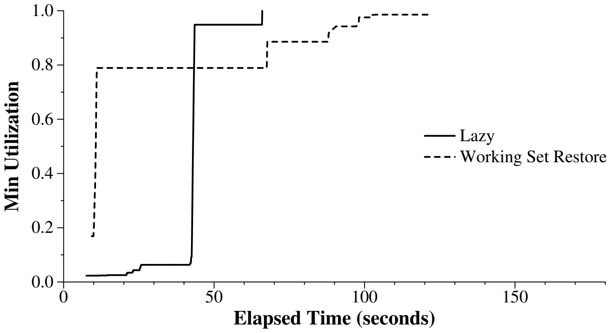

Figure 7. TTR for the MPlayer application using lazy restore and working set restore. By looking at Figure 5, MPlayer seems to require a min utilization of around 80% to maintain its base frame rate. The TTR for MPlayer does not correlate as well with the application metric as SPECjbb.

Comparing the TTR with Figure 5 shows that MPlayer requires a minimum utilization of around 80%. Lazy restore achieves this level of utilization after 43.3 seconds, which is 35.9 seconds after the VM starts. Working set restore gains a minimum utilization of 80% after 11 seconds, which is 1.7 seconds after the VM starts. The TTR for MPlayer does not correlate as well with our chosen performance metric of frames per second as SPECjbb. We speculate that there is less of a direct correlation between TTR and FPS because of buffer effects caused by the frame buffer and MPlayer adjusting its playback in real-time in response to performance degradation.

Using these two example workloads, we have shown how TTR might be used to compare the performance of restore techniques. The window and minimum utilization can be determined experimentally by using an end-to-end metric like throughput, but it does not always correlate directly to end-to-end performance. Nevertheless, TTR better reflects user experience than either total time to restore or total time until the VM starts. In the following sections, we use TTR to compare several aspects of working set restore.

5.4 Working Set Estimation

This section compares the accuracy of the two working set estimators that we implemented and analyzes the impact of imperfect working set estimators on working set restore.

5.4.1 Estimator Comparison

We compared the accuracy of the access-bit estimator and the memory tracing estimator for different workloads by taking several measurements during the restore. We measured the number of prefetched pages to find the size of the estimated working set. We counted the number of accesses to prefetched pages and unrestored pages after the VM starts. These two numbers give some idea of the accuracy of the working set estimator. The number of pages accessed that were prefetched is the number of pages where the restore *avoided* demand-paging. The number of pages accessed that were unrestored represents the number of pages restored *on-demand*. These two numbers also give some sense of whether each working set estimator overestimates, underestimates or completely misses the working set. Table 2 gives a comparison for the MPlayer benchmark, and Table 3 gives a comparison of the two estimators for the SPECjbb benchmark.

Since MPlayer has a well-defined active working set, both estimators do well. The memory trace estimator does slightly better by avoiding the most accesses to unrestored memory, but the access-bit estimator still reduces accesses by 80%, while only prefetching 4% of memory. While MPlayer is very predictable, the working set

Table 2. Measure of the effectiveness of the working set estimator for MPlayer in reducing demand-paging. Both access-bit scanning and trace-based working set estimation reduce the number of accesses to unrestored memory, reducing performance degradation.

MPlayer	Prefetched Pages	Avoided Accesses	On-demand Pages
Lazy	0	0	1,548
A-bit	10,114	8,617	343
Trace	10,113	10,084	84
Eager	262,144	0	0

Table 3. Measure of effectiveness of working set estimators for SPECjbb in reducing demand-paging. The access-bit working set estimator does not perform as well as the memory trace estimator for SPECjbb. SPECjbb is a deterministic workload, so memory tracing is effective, reducing the number of on-demand pages by more than 97%.

SPECjbb	Prefetched Pages	Avoided Accesses	On-demand Pages
Lazy	0	0	1,623
A-bit	83,471	30,501	442
Trace	47,082	47,082	47
Eager	262,144	0	0

of SPECjbb is more difficult to predict for a simple working set estimator. It is larger and not accessed as consistently as the MPlayer frame buffer. It is clear that the access-bit estimator does not work as well. Working set restore with the access-bit estimator prefetches almost twice as much memory as working set restore with the memory trace estimator (83,471 vs. 47,082) but uses less of it (30,501 vs. 47,082). However, SPECjbb is a deterministic workload, so memory tracing works very well, reducing the number of accesses to unrestored memory by more than 97%. Working set restore using the memory tracing estimator only prefetches 193 MB, which is close to the size that we estimated the working set of SPECjbb to be.

5.4.2 Estimator Accuracy

The performance of working set restore is dependent on the accuracy of the working set estimator. With an oracle, working set restore would be optimal, but no estimation techniques are perfect. The goal of this experiment was to measure the decrease in performance of working set restore as the accuracy of the working set estimator decreases. We tested this by restoring the same saved VM several times with a fraction of the working set retained. We randomly dropped a percentage of the pages in the working set for each experiment. Since the pages removed from the working set are randomly chosen, there is some variation in the experiments. We did not test the performance of a working set estimator that overestimates the working set because overestimating does not lead to performance degradation after the VM starts, it simply increases the time to prefetch the working set. If our working set estimators are very accurate and only include the minimal set of pages accessed by the VM after staring in the working set, then the performance of working set restore should drop as soon as some of the working set is not prefetched.

Figure 8 shows the TTR for different percentages of the working set dropped and the TTR for lazy restore for the MPlayer experiment. The more pages missing from the working set estimate, the more pages must be faulted in on-demand. When we drop even 5% of the working set, the performance begins to look like lazy restore. This sharp decrease reflects the cost of paging content from a relatively slow medium like a disk. This decrease also reflects well on our original working set estimate because it shows that our original working set was already the minimal set of pages that must be prefetched to guarantee good performance after the VM starts.

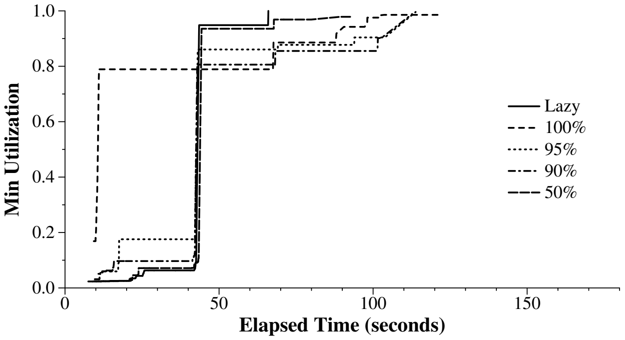

Figure 8. TTR for lazy restore and working set restore with partial working sets. We started with the memory trace estimator, which only missed 84 pages of the working set (see Table 2). We retain a random portion of the working set for each percentage line. Note that the performance drops quickly, meaning that the original working set was already the minimum set of pages that needed to be prefetched to guarantee good performance.

5.5 Working Set Size Microbenchmark

This benchmark artificially generates working sets of differing sizes by allocating a percentage of guest memory, then touching pages at random. The pages are chosen completely randomly, so there is no guarantee that all pages are touched or that all pages are touched once before being touched again. This benchmark runs on top of the basic RHEL VM presented in Section 5.2.

As the working set size becomes a larger fraction of total memory, the performance of both lazy restore and working set restore will decrease. Lazy restore has increased performance degradation because there will be more accesses to unrestored memory, so the VM will be unusable for longer. If the VM is actively using most of its memory then the working set will be almost all of the memory. Working set restore will have to prefetch most of memory, and its performance will approach the performance of eager restore.

Figure 9 shows the performance of lazy restore for a working set that occupies 20% to 80% of memory. As expected, the performance degrades as the VM accesses more memory actively because the VM will touch more unrestored memory during the restore. The performance for all sizes of working sets is worse than eager restore, which takes around 32 seconds, showing that lazy restore is only the better choice if the VM is actively using very little of its memory.

Figure 10 shows the performance with working set restore. The performance of working set restore also suffers as the size of the working set increases. As the benchmark is accessing memory randomly, it is difficult for the working set estimator to capture all of the working set. As the size of the working set grows, it becomes more likely that the working set estimator will miss part of the working set because it was just not accessed before the checkpoint was taken. However, working set restore still offers better performance than lazy restore for all sizes of working sets. For a minimum utilization of 60%, working set restore does well, staying below 35 seconds for all working set sizes except for when the working set is 80% of the memory size.

5.6 Working Set Restore Overhead

Using working set estimation for working set restore imposes an overhead on the VM. The amount of performance impact from

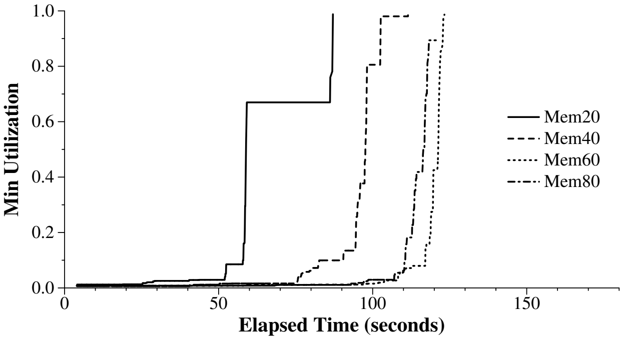

Figure 9. TTR for lazy restore of varying working set sizes. The time to reach 80% minimum utilization degrades quickly with lazy restore. There is more disk thrashing as the size of the working set increases because the microbenchmark is randomly accessing a larger region of memory.

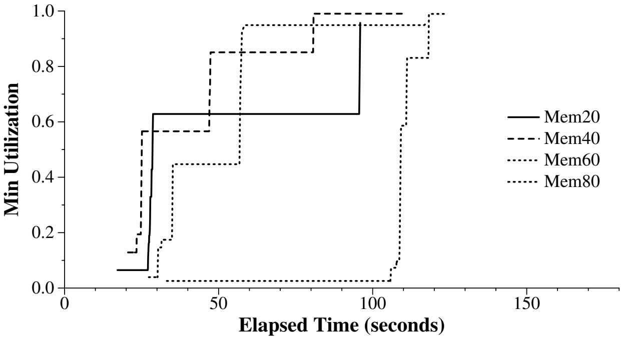

Figure 10. TTR for working set restore of varying working set sizes. Because the microbenchmark is random, the working set estimation techniques are not as successful, but working set restore still performs better than lazy restore for all memory sizes.

working set estimation depends on the estimator and the application. These experiments give the overhead for our two working set estimators for our benchmark applications, MPlayer and SPECjbb.

The overhead of the access-bit scanning estimator is ongoing because the scanning process always runs while the VM is running. However, the overhead is small because we are only scanning 4,096 pages every second. We ran each VM for 10 mins and measured the number of cycles for each scan. The overhead varies slightly for each workload depending on how much memory the application is using and how much memory is mapped in the guest OS. The access-bit scanner is optimized to first scan page directory entries and not scan the page table entries if the page directory entry does not exist or has not been accessed.

Table 4 shows the average number of cycles required to scan the access-bits for 4,096 pages each second and the percentage overhead based on the total number of cycles between scans. For SPECjbb, the access-bit scanning process takes 76,040 cycles on average every second to scan 4,096 pages. This translates to less than a fifth of a percent of overhead for the SPECjbb VM. The MPlayer VM is using less memory, so the access-bit scanner spends even less time scanning page tables on average, only 57,372 cycles on average. This gives only slightly more than a tenth of a percent of overhead for the MPlayer VM.

Table 4. Average number of cycles spent scanning access-bits each second and the estimated percentage overhead for access-bit scanning. The overhead for the access-bit scanning estimator is ongoing because it must constantly scan pages to estimate the working set, but the amount of overhead is small.

	Avg. Cycles	% Overhead
SPECjbb	76,040	.0016%
MPlayer	57,372	.0012%

The memory tracing estimator only imposes an overhead on the VM while the VM is being checkpointed. The estimator requires read traces in addition to write traces and the traces can only be removed once the page has been added to the working set, so more traces are triggered. The overhead imposed by the additional traces depends on the workload because some workloads read from memory more than others.

Each trace causes a page fault, so the VM must pause while the hypervisor handles the trace. It is somewhat difficult to measure the overhead of a page fault because the switch between the VM and the hypervisor pollutes caches as well as using cycles. This experiment attempts to show both the end-to-end and low level effects of additional traces. We measured the increase in the total time to checkpoint as well as the total increase in number of traces triggered. Table 5 shows the results for the SPECjbb workload and Table 6 shows the results for the MPlayer workload.

For SPECjbb, read traces add 30.9% to the number of traces triggered over the lazy checkpoint period, so SPECjbb probably reads and writes many of the same pages. In contrast, we see a 76.8% increase in triggered traces with MPlayer, showing that MPlayer reads more pages that it does not write than SPECjbb. However, an increase in traces does not lead to a significant change in the time to save the VM for either workload. Despite working set restore spending 7.5 seconds copying the working set for SPECjbb, the total time to save increases by 7 seconds. The increase in triggered traces actually decreases the time to lazily save the VM's memory slightly. The lazy save proceeds faster because more pages are being saved when the VM accesses them and fewer are being saved by the slow-running background thread. Similarly, the total time to checkpoint the VM increases by 0.9 seconds for MPlayer, but 5.9 seconds are spent copying the working set. There is also an increase in the total time to checkpoint or suspend a VM for the access-bit scanning estimator, but it simply adds to the total time based on the size of the estimated working set.

Table 5. Overhead of memory tracing for SPECjbb. With the addition of read traces, the number of traces triggered during the checkpointing process increases by 30.9%. The total time to checkpoint the VM increased by 7 seconds, with 7.5 seconds spent copying the working set. The slight decrease in the time to lazily save is caused by more pages being saved when the VM accesses them and fewer pages being saved by the slow-running background thread.

SPECjbb	Traces Triggered	Time to Save
Baseline	42,849	91.7 s
Working Set Restore	56,087	98.7 s

Table 6. Overhead of memory tracing for MPlayer. There was a 76.8% increase in the number of traces triggered with working set restore. The total time to save the VM increased by 0.9 seconds, with 5.9 seconds spent copying the working set.

MPlayer	Traces Triggered	Time to Save
Baseline	5,811	101.7 s
Working Set Restore	10,272	102.6 s

6. Related Work

The time-to-responsiveness performance metric for restoring saved VMs and the working set restore technique build on related work from several different areas of research. We built on work from the HCI and garbage collection communities for evaluating the responsiveness of a VM and work from the OS community on techniques for working set estimation and prefetching.

6.1 Restore and Prefetching Techniques

There is not much previous research in improving the performance of restoring saved VMs, but one related area where there has been more work is improving the performance of restarting an operating system from hibernation. Operating systems cannot take advantage of lazy fetching of memory as easily as the hypervisor because the operating system would have to fetch its own pages lazily, which would be much more complex. Accordingly, most operating systems eagerly fetch saved memory before restarting. Both Linux and Windows use compression to speed up the process of reading saved data from disk [1, 15]. We found that compression is not very effective for saved VMs, unless there is a lot of unused memory that is zeroed. Unless the guest was recently booted, there is generally not much unused memory, as the guest OS will always try to use any unused memory for buffer cache.

Unlike a hypervisor, the operating system does not have to save or restore all of memory; it can dump memory that it knows is not necessary to save, such as the buffer cache. Dumping the buffer cache can hurt performance after the OS restarts, but it also saves a lot of time when hibernating and restarting the OS. In contrast, the hypervisor cannot throw away any of the VM's memory because it does not know what memory the VM is using. We could ask the guest to drop its buffer cache, which only requires a single command in Linux for example, but we would have to paravirtualize the guest to figure out which pages are part of the buffer cache because dropping the buffer cache does not zero those pages. Using working set estimation is a better solution because it gives us an idea of the active memory of the VM without requiring modifications to the guest.

Working set prefetching has been explored for applications. Windows uses a system called SuperFetch [11] that prefetches frequently used files and binaries for commonly used applications. SuperFetch also traces the boot process to predict what files will be needed for the next boot and reorganizes files on disk in the order that they will be needed during the boot process. Some of the ideas are similar to working set restore (i.e. reorganizing data on disk based on when it is needed), but Windows has more information about accesses and dependencies. In contrast, we treat the VM as a black box, so we assume that all accesses are unrelated. A single action in the operating system can look like a series of random memory accesses as different levels of the operating system are accessed, such as the application or the file system stack. Working set estimation works well with this kind of view because it predicts the set of active memory using only memory accesses and does not require additional information.

VM migration faces some of the same problems as restoring a saved VM. In particular, the pull model of migration, where the VM starts at the destination and lazily pulls memory over from the source, is similar in implementation to a lazy restore. One key difference between migration and restore is that the memory comes over the network and is in memory on the source. So, while there is a latency for fetching a page during lazy migration, there is no penalty for random access versus linear access.

Similar to our findings on the performance degradation of lazy restore, the latency of the network makes lazy migration difficult. Sapuntzakis[19] and Clark[7] both dismiss lazy migration due to the performance overhead. Hines[10] propose a scheme similar to working set restore for migration, although they choose all pages that are not dirty, rather than using working set estimation. They propose a "bubbling scheme" for the background restore process where the background thread chooses pages located around the most recent access of unrestored memory. Such a bubbling scheme is not really practical for pulling memory off a mechanical disk where there is a penalty for random access. Recently, the SnowFlock system [16] for VMfork, where a running VM is copied and started on another host, used lazy restore of memory. They use a purely lazy scheme, but paravirtualize the guest to avoid restoring pages that will only be re-allocated. They argue that with this optimization, the performance degradation is minimal because newly forked VMs generally allocate new memory and do not request much memory from the parent VM. We chose not to paravirtualize because running unmodified guests is more general and we cannot expect the VM to allocate new memory when restoring from a suspend or checkpoint.

6.2 Performance Metrics

The HCI community has defined what constitutes good responsiveness for an application. Miller [17] set a general standard for response time more than 30 years ago. A response time of less than 100 milliseconds appears instantaneous to the user, a response time of greater than 1 second requires user feedback, and a response time of greater than 10 seconds will cause the user's attention to wander. Using this standard, we can calculate a rough upper bound for how many times the restore process can go to disk and fetch a page before the VM appears to lag. For each user action, there can only be 10 accesses to unrestored memory. This limit gives us insight into why lazy restore causes such noticeable performance degradation. We found there to be more than a thousand accesses to unrestored memory during a lazy restore, so each user action causes tens or hundreds of disk accesses, causing the VM to slow down, sometimes so much that the user would consider the VM to be unusable during the entire restore process.

In order to compare the performance degradation of various restore schemes, we introduced the time-to-responsiveness metric. TTR quantifies acceptable overhead for a particular service: the restoring of a VM. It answers the question of when the VM is suitably responsive, or alternatively, when it is no longer overly taxed by the process of being restored. Other services have also investigated how one reasons about and reports their overhead. On-going services such as garbage-collection [14] have examined the concept of *minimum mutator utilization* [6] or MMU: given an appropriate time interval or window, it is the least amount of time not consumed by collection activity and so available to the application. Unlike the MMU, time-to-responsiveness differs in that the restoring of a VM is not an on-going activity and we are interested in knowing at which point in time its overhead is sufficiently small.

6.3 Working Set Estimation

The idea of estimating working sets attempts to capture an important notion about locality of accesses [9]. Uses proposed by Denning [9] include not only determining what memory to page out but also what memory to page in. In one sense, restoring a VM is an extreme version of this where the state capturing the working-set for the VM must persist with the saved state. Unlike the working set described by Denning, our working set is simply defined by the set of pages accessed by the VM during a restore, since those are the pages that cause disk accesses during the restore process.

There is much work on improved methods for identifying pages in a working set [5, 12, 13, 18]. This previous work extends the CLOCK algorithm [8] to be resilient to incidental or transient accesses such as those that occur when memory is scanned. Our working set estimators are geared towards predicting accesses during a very specific period of time, which makes them simpler than more general working set estimators. For example, our estimators do not need to ignore transient accesses, since even transient accesses to unrestored memory cause performance degradation. In addition, the rate of our access-bit scanner can be set by the speed of the background restore process and our memory trace estimator only captures accesses to memory during the checkpointing process.

7. Conclusion

We examined existing metrics for comparing techniques for restoring VMs and found that they do not reflect what is most important to the user—when the VM and its applications return to normal performance. We proposed both a better metric and a better restore technique for user experience. Our new time-to-responsiveness metric takes into account the time until the VM starts and the performance degradation of the VM after it starts. Working set restore starts the VM as soon as possible, while minimizing performance degradation after the VM starts. Working set restore accomplishes this by estimating the working set of the VM before the checkpoint, saving the working set along with the checkpointed state and prefetching it before starting the VM. We showed that working set restore works well even with a simple working set estimator that scans access-bits. We introduced a new working set estimator that uses memory tracing during lazy checkpointing to more accurately capture the VM's working set. We found that with the memory tracing estimator, working set restore can anticipate 99% of the VM's memory accesses after starting, avoiding most of the performance degradation of lazy restore with just a few seconds of prefetching.

Working set restore is a predictive technique, so it works best with predictable workloads. The behavior of the VM does not have to be exactly the same before and after the checkpoint, but the VM must use the same set of active memory after it restarts as it did before it was saved. We found that with a random workload working set restore does better than lazy restore, but slightly worse than eager restore. Improving the performance of lazy restore for an unpredictable workload is extremely difficult. Using the guidelines from the HCI community, the VM and its applications must respond to any user action within 100 milliseconds, limiting the VM to around 10 accesses to unrestored memory. Without the ability to use previous memory accesses to predict future accesses, it would be extremely difficult to keep the number of accesses to unrestored memory under that limit and keep the VM responsive. If the VM's workload is truly random, then the technique that is optimal for user experience is not using lazy restore at all, but eagerly restoring the VM's memory image. Fortunately, many applications are predictable because they actively use a limited set of memory pages allocated to them by the OS, so predictive techniques like working set restore can still offer significant performance improvements.

Acknowledgments

Thanks to Dong Ye for modifications to MPlayer and to Lenin Singaravelu for modifications to SPECjbb®2005. Thanks to Kevin Christopher and Jesse Pool for many discussions of ideas and experiences about checkpointing and restore. Thanks to Dan Ports and Karen Zee for their feedback on the many drafts of this paper.

References

[1] μswsusp. http://suspend.sourceforge.net/.

[2] AMD64 virtualization codenamed "pacifica" technology: Secure virtual machine architecture reference manual, May 2005. http://enterprise.amd.com/downloadables/Pacifica_Spec.pdf.

[3] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL '03*, New Orleans, LA, USA, Jan. 2003.

[4] D. F. Bacon, P. Cheng, and V. Rajan. The Metronome, a simpler approach to garbage collection in real-time systems. In *Proc. OTM 2003 Workshops*, 2003.

[5] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *Proc. FAST '04*, 2004.

[6] G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proc. PLDI '99*, Atlanta, GA, USA, May 1999.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI '05*, 2005.

[8] F. J. Corbato. A paging experiment with the Multics system. Technical report, MIT Project MAC, May 1969.

[9] P. J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5), 1968.

[10] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proc. VEE 2009*, Washington, DC, USA, 2009.

[11] T. Holwerda. SuperFetch: How it works & myths, May 2009. http://www.osnews.com/story/21471/SuperFetch_How_it_Works_Myths.

[12] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. SIGMETRICS '02*, Marina del Rey, California, USA, 2002.

[13] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proc. USENIX '05*, 2005.

[14] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. URL http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html.

[15] kernel-enhancements-xp. Kernel enhancements for Windows XP, jan 2003. http://www.microsoft.com/whdc/archive/XP_kernel.mspx.

[16] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proc. Eurosys '09*, Nuremberg, Germany, 2009.

[17] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[18] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In P. Buneman and S. Jajodia, editors, *Proc. SIGMOD '93*, 1993.

[19] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Operating Systems Review*, 36, 2002.

[20] Seagate. Product Manual. Barracuda 7200.11 Serial ATA. http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda7200.11/100452348g.pdf, Jan. 2009.

[21] Standard Performance Evaluation Corporation. SPECjbb2005 User's Guide. http://www.spec.org/jbb2005/docs/UserGuide.html, April 2006.

[22] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, June 2001.

[23] N. Tolia, D. G. Andersen, and M. Satyanarayanan. Quantifying interactive user experience on thin clients. *IEEE Computer*, 39(3), Mar. 2006.

[24] VMware. Timekeeping in VMware virtual machines. http://www.vmware.com/vmtn/resources/238, Aug. 2008.