Fast Restore of Checkpointed Memory using Working Set Estimation

Irene Zhang Ken Barr Alex Garthwaite Yury Baskakov Jesse Pool Kevin Christopher
VMware, Inc.

Overview

What is the goal of fast restore?

Reduce the time to restore a saved system to an interactively usable state.

Why do we need fast restore?

The ability to save and restore the state of running systems can enable a variety of useful features like suspend-to-disk, system checkpointing, system migration, and many others. Unfortunately, restoring a saved system is time-consuming, discouraging the use of save and restore features, especially in cases where a user is waiting to interact with the saved system.

How do we make restore faster?

Use *partial lazy restore* to hide part of the latency of fetching the saved system state. The most expensive part of restore is fetching the saved memory image. Partial lazy restore prefetches the working set of memory pages before the system starts and then lazily restores the rest of memory.

Background

Why partial lazy restore?

Lazily restoring memory is effective for hiding the time to read a large memory image. But lazy restore can degrade system performance because each memory access becomes a read request. When a large fraction of system time is spent restoring memory, the system is essentially unusable.

Using an eager scheme avoids performance degradation, but the user must wait until all of memory has been fetched, which could be tens of seconds if the memory image is being read from a disk or over the network. Eager restore scales linearly with the size of memory, so the performance will worsen as systems require more memory, and disk and network bandwidth do not keep pace.

By prefetching the working set before the saved system is started, partial lazy restore avoids the performance degradation of lazy restore, while still effectively hiding most of the latency of reading the saved memory image.

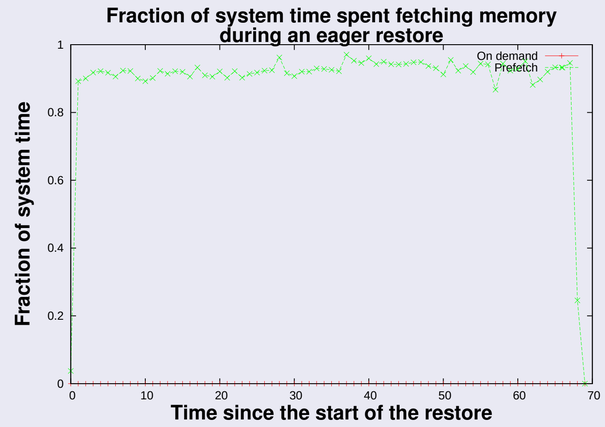


Figure: Full eager restore of RHEL running Firefox, Evolution and OpenOffice. The restore process spends 70 seconds restoring memory from disk before RHEL is started and the user is able to use the system.

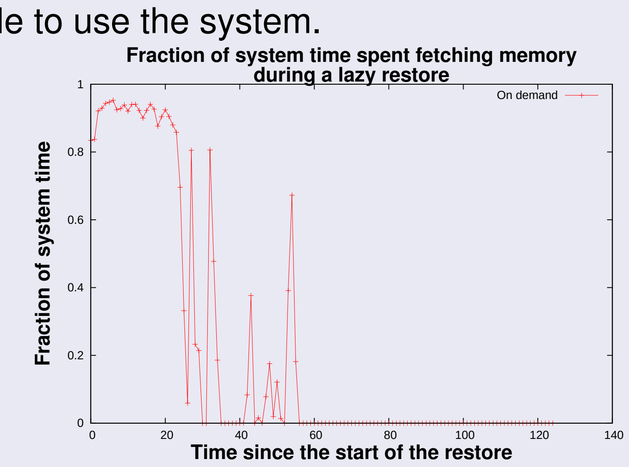
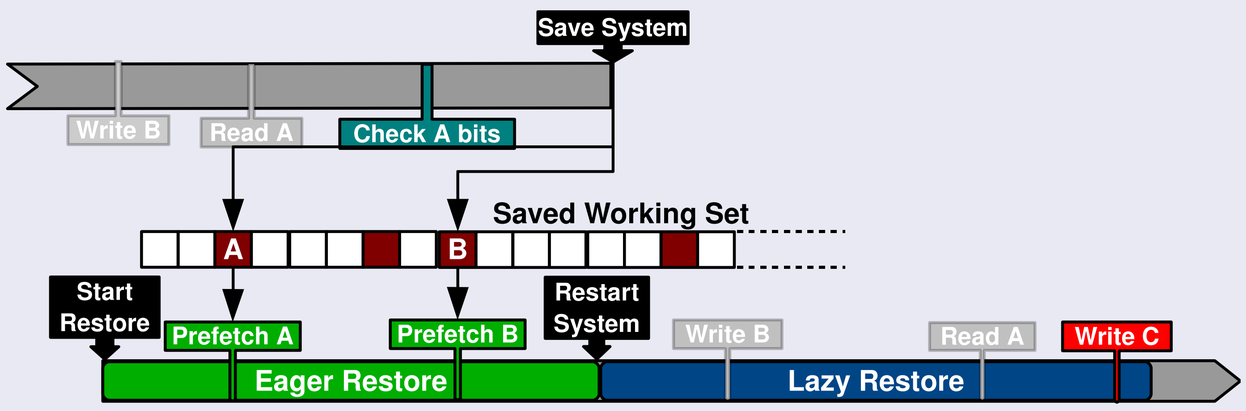


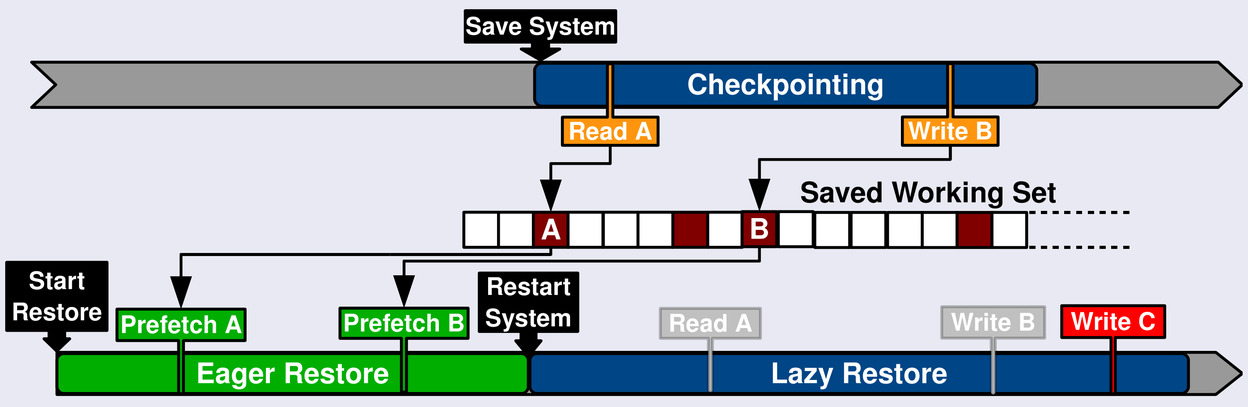
Figure: Full lazy restore of the same RHEL system from hard disk. For the first 25 seconds, the system spends almost all of its time restoring memory, so the user sees serious performance degradation during this period.

Working Set Estimation with Access-Bit Scanning



Assuming that past memory accesses predict future accesses, the set of pages accessed recently before saving can predict which pages will be accessed after the system starts. We track the set of recently accessed pages by periodically scanning and clearing the access bits in the system's page tables.

Working Set Estimation with Memory Access Tracing



If the system continues to run after saving, we can get a precise prediction of memory accesses by tracing memory accesses immediately after saving. Assuming the sytem is somewhat deterministic, the system should make almost the same memory accesses after it starts as it did immediately after the save. This method is only useful if the system continues running after the save point such as in checkpointing.

Performance Results

Partial Lazy Restore with A-bit Scanning

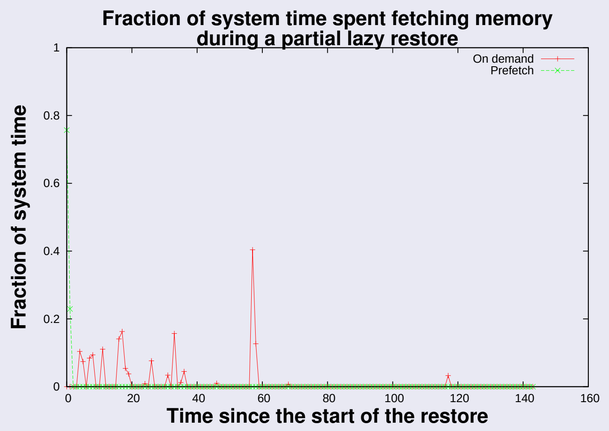


Figure: Partial lazy restore of RHEL using A-bit scanning. The restore process eagerly fetches only the working set, then starts RHEL. The prefetched pages significantly reduce the fraction of system time spent fetching memory after the system starts.

Partial Lazy Restore with Memory Tracing

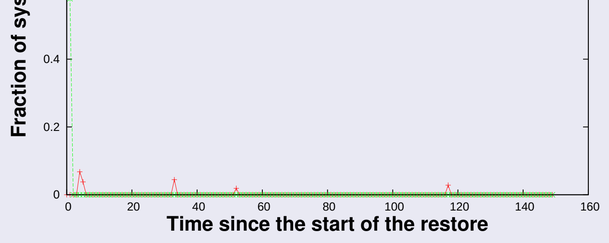


Figure: Partial lazy restore of RHEL using memory tracing. This method is more accurate than A-bit scanning because the system reverts back to the beginning of the tracing period, so memory accesses during the restore are likely to be identical to the traced accesses.

Performance Comparison

	Prefetched	Skipped	Lazy
Eager	262,144	N/A	N/A
Lazy	0	N/A	1,675
Access	11,181	11,181	75
Trace	11,295	11,293	11

Table: For each method, we compare the number of pages that were prefetched, the number of page accesses that were skipped because of prefetching and the number of pages that were lazily restored. Both A-bit scanning and memory tracing do well at predicting memory accesses and reducing the amount of memory that needs to be restored lazily.