Optimizing VM Checkpointing for Restore Performance in VMware ESXi

Irene Zhang[*] University of Washington
Tyler Denniston[*] MIT CSAIL
Yury Baskakov VMware
Alex Garthwaite[*] CloudPhysics

Abstract

Cloud providers are increasingly looking to use virtual machine checkpointing for new applications beyond fault tolerance. Existing checkpointing systems designed for fault tolerance only optimize for saving checkpointed state, so they cannot support these new applications, which require better restore performance. Improving restore performance requires a predictive technique to reduce the number of disk accesses to bring in the VM's memory on restore. However, complex VM workloads can diverge at any time due to external inputs, background processes, and timing variation, so predicting which pages the VM will access on restore to reduce faults to disk is impossible. Instead, we focus on predicting which pages the VM will access *together* on restore to improve the efficiency of disk accesses.

To reduce the number of faults to disk on restore, we group memory pages likely to be accessed together into *locality blocks*. On each fault, we can load a block of pages that are likely to be accessed with the faulting page, eliminating future faults and increasing disk efficiency. We implement support for locality blocks, along with several other optimizations, in a new checkpointing system for VMware ESXi Server called Halite. Our experiments show that Halite reduces restore overhead by up to 94% for a range of workloads.

1 Overview

The ability to checkpoint and restore the state of a running virtual machine has been crucial for fault tolerance of virtualized workloads. Recently, cloud providers have been exploring new applications for VM checkpointing. For example, they want to use checkpointing to save and power off idle VMs to conserve energy. Restoring a checkpointed "template" VM could be used to clone new VMs on demand, which would enable fast, dynamic allocation of VMs for stateless workloads.

Unlike traditional fault tolerance applications, these new applications depend on efficient *restore* of checkpointed VMs. For example, using checkpointing for dynamic allocation of VMs depends on the ability to quickly start up a VM on demand. Checkpointing systems designed to support fault tolerance only restore on failures, so they optimize for *checkpoint save* performance instead. As a result, previous work rarely addresses restore beyond basic support, so existing systems would offer poor performance for these new applications.

Virtual machine checkpointing takes a snapshot of the state of a VM at a single point in time. The hypervisor writes any temporary VM state, like VM memory, to persistent storage and then reads it back into memory when restoring the checkpoint. Since memory images can be large, VMware ESXi uses a technique called *lazy restore* that loads the memory image from disk while the VM runs. While the VM's memory is partially on disk, any access to on-disk pages causes a fault that requires a disk synchronous access before the VM's execution can resume. Pauses in execution for faults to disk can quickly degrade the usability of the VM.

Improving lazy restore performance requires a predictive technique that reduces the number of faults to pages on disk. However, it is impossible to predict which pages the VM will access on restore; the VM's execution might diverge at any time due to timing differences or external inputs, particularly with complex workloads that have many background tasks and user applications. Previous work [24] based on predicting which pages the VM would access on restore could not cope with divergence, leading to poor performance for complex workloads like Windows desktop applications.

Rather than reducing the number of faults to disk by predicting the pages that the VM will access on restore, we instead predict the pages that the VM will access *together* on restore. On each fault to disk during lazy restore, we prefetch a few pages that are likely to be accessed with the faulting page, rather than prefetching before the VM's execution begins. This technique is more resilient to divergence since the prefetching decision is based directly on pages that have been accessed by the VM after the restore. There is a smaller penalty for incorrect predictions because only a few pages are prefetched at a time.

To allow for efficient prefetching on restore, we sort pages likely to be accessed together into *locality blocks* in the VM's checkpointed memory image. On restore,

[*]Work done while all authors were at VMware.

we load an entire locality block on each fault to disk. Since the other pages in the locality block are likely to be accessed with the faulting page, we eliminate faults to disk for those pages. We implement this technique in a new VM checkpointing system for VMware ESXi Server called *Halite*.

Halite uses two techniques to predict the access locality of memory pages. The first uses the VM's memory accesses during lazy save. The VM continues running past the checkpoint while its memory is written to disk, so the VM's execution during lazy restore is actually a re-execution of the VM's execution during lazy save. While the exact execution of the VM will vary on restore due to divergence, there is less change to the access locality. Pages accessed together during lazy save are likely to be accessed together again on restore.

Since the VM may not access all of its pages during checkpointing, we must use a second technique for predicting access locality. For the unaccessed pages, Halite uses locality in the guest operating system's *virtual address space* to predict access locality. Pages that are mapped together in the virtual address space are likely to be accessed together, so locality in the virtual address space is another good predictor of access locality.

We designed and implemented Halite as an improved VM memory checkpointing system for VMware ESXi 5.1 [22] and included other optimizations to the current system. Halite performs fine-grained compression of the checkpointed memory file, so that compression can be done in parallel on checkpoint save and only a small amount of decompression is required for each fault to disk on checkpoint restore. Compression increases the effectiveness of locality blocks because more pages can fit into each block. Unlike ESXi, Halite makes extensive use of threads to parallelize work during checkpoint save and restore, including threads for compression and I/O. Halite dynamically throttles background work during lazy save and restore to avoid disk contention.

The next section reviews some new applications for VM checkpointing that VMware has explored. Section 3 gives background on the current virtual machine memory checkpointing system in VMware ESX 5.1. Section 4 describes Halite's new memory file layout with locality blocks. Section 5 describes the algorithms that we use for predicting access locality. Section 6 details the other optimizations in Halite. Section 7 gives implementation details including the algorithm for saving and restoring VM memory in Halite. Section 8 presents our experimental results. Section 9 gives an overview of related work, including our previous work, and Section 10 concludes.

2 Checkpointing Workloads

The primary motivation for Halite is to improve the checkpoint restore performance of ESXi, enabling a variety of new and emerging use cases. In contrast to fault tolerance scenarios, where restore is uncommon and happens only on failure, these new use cases depend on efficient checkpoint restore.

2.1 Dynamic VM Provisioning

One of the advantages to cloud computing is the ability to allocate the appropriate amount of computing resources for any workload. This allocation does not have to be static; as a workload requires more or less resources, the number of allocated VMs can be increased or decreased. However, most cloud infrastructures are not able to quickly bring more VMs online. On Amazon EC2, it can take up to 10 minutes to bring up a VM [1]. Due to this delay, users must keep a buffer of unused VMs to handle spikes in requests. Running a number of idle VMs is both a waste of resources and still may not be sufficient to protect against severe spikes in usage.

Halite enables fast checkpoint restore from a template VM image, similar to VM fork supported by Snowflock [10], Kaleidoscope [2] or FlurryDB [14]. This feature allows users to better scale their resource allocation with usage. Using a checkpointed VM image with a running Apache server, a VM could be online and handling user requests in a few seconds. Using a checkpointed VM also offers advantages over quickly booting a VM; the applications in the VM benefit from a warm cache and several applications can be running in the VM without the overhead of application start up times, which can sometimes be long. Alternatively, for some workloads, Halite gives users the ability to allocate a single stateless VM for each incoming connection. Customers have requested this feature because it is an easy solution to ensure security between users.

2.2 Energy Conservation

Virtualization reduces energy usage with server consolidation, but conserving energy consumed by idle VMs is still a serious concern in cloud deployments. Some systems have explored turning off servers [3] or suspending idle VMs [5] to conserve power, but all of these systems struggle with restarting servers or VMs. They use predictive techniques to restart VMs in advance. When these techniques incorrectly predict usage patterns, either energy is wasted powering on VMs that are not needed, or users are forced to wait for the needed VM to restart.

Halite makes it much easier to turn off idle VMs without suffering from poor performance when the VM is needed again. Using Halite, the user can checkpoint VMs and power them off. Complex predictive models are not required with Halite because suspended VMs can be quickly restarted on demand.

2.3 Virtual Desktop Infrastructure

Large companies have started to move toward converting desktop PCs into VMs running in a datacenter. These virtual desktops are easier to maintain and reduce the amount of hardware needed. However, since there are more users sharing hardware, users can see performance degradation when there are spikes in usage. In particular, VMware has observed a "boot storm" problem, where all users arrive at work in the morning and attempt to boot their VMs close in time, leading to severe disk contention. VM checkpointing can be used to mitigate this problem. If users checkpoint their desktop VM before going home (or an energy conservation system checkpoints it for them), then they can simply restore their VM in the morning. Restoring a checkpointed VM requires reading much less from disk than a full boot, easing contention on the disk. In addition, Halite efficiently restores the VM in much less time than booting a VM, further reducing the disk usage and wait time for the user.

3 ESXi Checkpointing

In order to give some background and motivation to our work, we describe the state of the art in virtual machine checkpointing implemented in the current release of VMware ESXi. We primarily discuss the mechanism for checkpointing VM memory and not other VM state.

3.1 ESXi Save and Restore Algorithm

ESXi has used lazy checkpointing since VMware ESXi 4.0. Lazy checkpointing allows the VM to run while its memory is saved or restored, reducing the amount of downtime. ESXi implements a generic copy-on-write scheme similar to the one described here [20] and similar to Xen's implementation [4]. VMware's implementation depends on ESXi's memory tracing mechanism to track write accesses, which is also used for Halite.

On checkpoint save, ESXi pauses the VM's execution and saves its CPU and device state. ESXi installs memory traces on all of the VM's pages and resumes the VM. When the VM writes to an unsaved page, it triggers the trace on that page. ESXi saves the page and removes the memory trace before allowing the write to proceed.

While the VM runs, the hypervisor concurrently writes out memory pages using a background thread. This thread ensures that the checkpointing process finishes in a reasonable amount of time. The background thread walks the VM's physical address space, saving any pages that have not been already written. It removes the trace on any page that it saves to avoid triggering the trace later. The checkpoint save is complete when the background thread has walked the entire address space.

ESXi supports lazy checkpoint restore using the swap subsystem by treating a restoring VM like a VM with all of its memory swapped. This implementation was chosen for its simplicity and ease of deployment. To restore a VM, ESXi sets up the checkpointed memory file as the swap file and then restarts the VM at the checkpoint. On each access by the VM, a single memory page is swapped in from the memory file. Concurrently, a background thread touches swapped out pages to ensure that the restore finishes in a reasonable period of time.

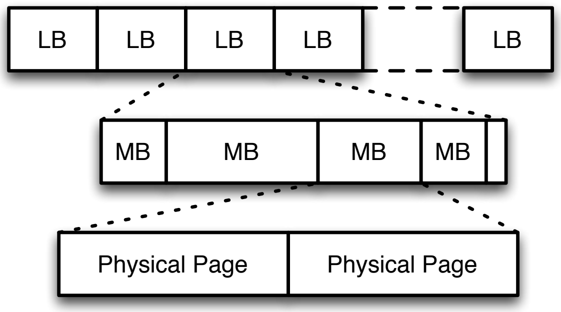

Figure 1: Block layout of checkpointed memory file. Memory blocks (MB) consist of several pages in physical address order. Locality blocks (LB) contain a variable number of compressed memory blocks. Memory blocks are grouped into locality blocks based on access locality.

3.2 Memory File Organization

ESXi saves the VM's memory in physical address order from physical address 0 to the VM's memory size. This file layout is simple and requires no metadata, but is not optimal for either checkpoint save or restore. On every write to an unsaved page, ESXi must save the page before allowing the VM to continue executing. Since these writes are to random memory pages, the disk accesses are random as well. ESXi avoids having to write to disk on each write access by buffering, but buffered pages still cannot be written out sequentially because of the file layout. These writes can degrade the VM's performance if the rate of writes to memory is high.

This organization is even worse for checkpoint restore. On every access to an unrestored memory page, the VM must pause while waiting for the page to be read from disk. Since physical memory is inherently random access, every access to an unrestored memory page requires the disk to seek and read a single page, leading to poor disk performance. Because ESXi treats the memory as swapped, the hypervisor only reads one 4K page from disk on each access, further degrading disk performance.

4 Halite Memory File Organization

This section describes Halite's memory file layout. Halite uses a significantly different memory file organization from ESXi, with locality blocks and fine-grained compression. Figure 1 shows the layout of the Halite memory file. Locality blocks are crucial for efficient prefetching on each fault to disk during restore.

4.1 Memory Blocks

For simplicity of implementation and to reduce the size of meta-data, Halite divides the VM's physical memory into fixed-size, aligned blocks of a few pages each called *memory blocks*. Halite uses memory blocks as the smallest unit of processing (i.e., compression, buffering, etc.), so memory blocks must be small enough that there is still some access locality in the physical address space.

We found using a memory block size of two pages was a good trade-off in our implementation; it halves the amount of meta-data, but remains small enough for workloads with poor access locality in the physical address space like Windows applications. Larger memory blocks sizes can be used for Linux because of its use of a buddy allocator[1], but performed poorly for Windows.

4.2 Locality Blocks

Halite groups a number of memory blocks into each locality block, based on access locality. Halite loads an entire locality block on each fault to disk, reducing disk accesses and increasing disk efficiency. For simplicity, locality blocks are fixed-size. Due to the fixed size of locality blocks, they contain a variable number of compressed memory blocks and some empty space. Locality blocks also reduce the size of meta-data needed for compression because only the byte offset within the locality block is needed. Larger locality blocks increase the efficiency of the disk, but also increase the latency of each fault to disk on restore. In our implementation, we used a locality block size of 64KB, which we found to be a good trade-off between efficiency and latency.

5 Access Locality Prediction

Halite uses two techniques to predict access locality for grouping memory pages into locality blocks. The first technique traces the execution of the VM past the checkpoint during the lazy save. The first technique only works for pages that are accessed during the lazy save, so we combine it with a second technique that uses guest virtual address locality to predict access locality. Both of these techniques are standard in CPU prefetching [19], although not at the 4KB page level.

5.1 Lazy Save Memory Accesses

In our previous work [24], we observed that the checkpoint restore period is a re-execution of the checkpoint save period since the VM restores back to the point in time at the beginning of the lazy save period. Unlike working set restore, Halite uses the VM's memory accesses during lazy save to predict access locality, rather than access ordering. If the VM accessed page X, followed by page Y during lazy save, then it is highly likely that if the VM accesses X or Y on restore, it will also access the other, even with divergence due to timing or different external inputs.

Halite groups the pages that were accessed together during lazy save into locality blocks. The first N pages accessed by the VM are stored in one locality block, the next N in another, where N is the size of a locality block. The number of pages in a locality block can vary due to compression. Halite does this sorting during the save process by writing pages out to locality blocks as they are accessed. Simply filling locality blocks in access order allows Halite to fill locality blocks without post-processing, to easily fill a locality block at a time, and to write locality blocks out sequentially to disk as they are filled.

For VMs with more than one virtual CPU (vCPU), Halite separates pages into locality blocks based on the vCPU. Since each vCPU is running a separate thread of execution, we believe that an access to page X on one vCPU, followed by an access to page Y on another vCPU is not a good predictor of access locality since differences in timing can easily cause divergence. Sorting based on vCPU simply requires Halite to fill one locality block per vCPU at a time.

5.2 Guest Virtual Address Space

Divergence from the VM's execution during lazy save on restore is unavoidable; there will be pages that weren't accessed during lazy save that are accessed on restore. For pages that are not accessed during lazy save, we use guest virtual address space locality to predict access locality. This technique assumes that if page X and Y are adjacent in the virtual address space, then an access to page X or Y is a good predictor that the VM will also access the other page. Previous work [16] has shown that the virtual address space is a better predictor of access locality than the physical address space.

Halite sorts pages not accessed by the VM during the checkpoint save into locality blocks based on virtual address. The first N mapped pages in a guest virtual address are stored in one locality block, the next N in another. Again, N may vary due to compression. Halite collects page table roots as the VM runs. The background thread in Halite walks the guest virtual address space using the guest page tables. As the background thread scans, it fills locality blocks in the order it encounters pages and writes them out sequentially to disk. We only save a single copy of each memory page. Memory pages that are mapped in more than one guest address are saved the first time we encounter them in a page table.

[1]Linux's buddy allocator increases access locality in the physical address space by mapping contiguous physical addresses to virtual addresses whenever possible.

6 Halite Checkpointing Optimizations

This section introduces other improvements made in Halite to ESXi's checkpointing system. These optimizations include dynamic background thread throttling, compression, zero page optimizations, and threading. Some of them take advantage of Halite's more sophisticated memory file organization, while some of them are just general improvements to the ESXi checkpointing infrastructure.

Dynamic Background Thread Throttling The background thread in ESXi is designed to ensure that the checkpointing process finishes, even if the VM does not access all of its memory pages. When the VM is rapidly touching pages, disk access to the checkpointing file becomes a bottleneck and the background thread begins to contend with the VM. However, we observed that if the VM is accessing pages rapidly, there is no reason for the background thread to run since the checkpointing process is clearly still making progress. Therefore, we only run the background thread in Halite if the checkpointing process is not progressing, which keeps the background thread from contending with the VM. We do this throttling for both checkpoint save and restore.

Compression Compression reduces the size of the checkpointing image, which reduces not only the size on disk of the image, but also the amount of data that needs to be moved to and from the disk for the checkpoint. Reducing the disk space required can be important, especially if the VM has a large memory size, but reducing the amount of I/O is even more important because the disk is a bottleneck during checkpointing. Compression also allows more memory to be prefetched on each page fault with the same amount of I/O. In Halite, each memory block in a locality block is separately compressed. We chose to compress memory blocks instead of whole locality blocks to allow more parallelization and to reduce the amount of decompression required on each page fault. We found that using smaller blocks for compression has minimal impact on the compression ratio.

Zero Pages ESXi scans guest memory for pages that are completely zero and does copy-on-write sharing of those pages. For these pages, there is no reason to read or write the page for checkpointing, so Halite tracks these pages and does not include them in the memory image. Halite also does this for pages that the VM has never touched, and therefore, are not backed in the hypervisor.

Threading Halite introduces several threads to allow more parallel processing of memory pages. On checkpoint save, these threads reduce the amount of time that the VM has to be paused on each page fault. Halite only needs to pause the VM long enough to copy the memory page to a buffer; threads perform the compression and writing the memory out to the checkpointing file. On restore, the faulting page must be read in from disk and decompressed synchronously, so threads cannot improve the performance. However, Halite decompresses and restores the other memory blocks in the locality block using threads in parallel. This minimizes the work for each prefetched page on a page fault and eliminates the work required if the VM later accesses one of the prefetched pages.

7 Implementation

We implemented Halite using VMware ESXi 5.1. Halite replaces ESXi's existing checkpointing mechanisms because they are not designed to asynchronously process memory and restore memory that is not organized in physical address order. In addition, ESXi does not save memory to a separate file by default; memory is normally stored in one file with other checkpointed state. Halite required a separate file because there is no way to anticipate the size of the region required for checkpointed memory due to compression. ESXi does not support compression of the memory image. Halite only replaces the VM memory checkpointing system, so ESXi still handles saving any other VM state.

7.1 Halite Save and Restore Algorithm

Like ESXi, Halite uses lazy checkpointing, but Halite does copy-on-access, rather than copy-on-write checkpointing to capture access locality. However, Halite buffers pages on checkpoint save and writes to disk sequentially, rather than randomly, so the overhead is small.

On both checkpoint save and restore, Halite tries to do as much work asynchronously as it can. During the lazy save period, when a trace triggers, Halite simply copies the memory block to a buffer and removes all of the traces on the pages in that block. Later the memory block is compressed by a thread and copied to a locality block. When the locality block fills up, another thread writes it out to disk. The thread also updates the mapping to record which locality block contains each memory block.

When the VM faults on an unrestored page, Halite consults the map to find which locality block it needs to fetch. The locality block might already be in memory if it was prefetched by a previous fault. If not, Halite reads the locality block, and decompresses and restores just the memory block of the faulting page. The other pages in the locality block will be decompressed by threads later.

The background thread in Halite works similarly to ESXi, except it is throttled as described in Section 6.

8 Evaluation

Our evaluation answers several questions about the performance of Halite:

- How does Halite compare to ESXi 5.1 for some representative workloads?

- How do locality blocks compare to other block organizations?
- How does compression impact the performance of Halite for workloads with differing amounts of compressibility?
- How much do locality blocks contribute to the performance benefits offered by Halite?
- How does restoring a checkpointed VM using Halite compare to a cold boot of the VM?

We evaluated Halite using a synthetic workload and three application benchmarks. pgbench [18] is a standard database benchmark for PostgreSQL. Worldbench is a Windows desktop application workload. We also designed a simulated Apache [6] web server benchmark. Our workloads represent a range of complexity from the simple synthetic workload to the complex Worldbench benchmark. We ran our experiments on a server with a 2.3GHz 8-core AMD Opteron processor and 24GB of RAM. All of the VMs and checkpoints are stored on a 15,000 RPM Seagate 1TB drive running VMFS-5.

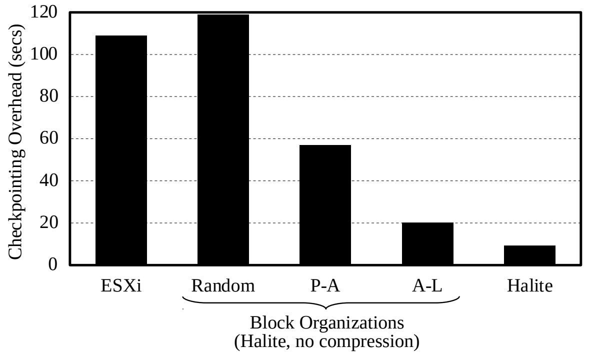

Figure 2: Synthetic workload performance for the current ESXi implementation, Halite with different memory file organizations and Halite with memory file optimizations. The middle bars give Halite performance for an uncompressed memory file using (from left to right) random blocks, physical address blocks (P-A) and access locality blocks (A-L). Performance is given as the increase in runtime caused by the checkpoint restore (lower is better).

8.1 Microbenchmark

First, to evaluate the performance benefit of Halite and its various optimizations in a controlled environment, we created a synthetic workload generator. The benefit of the workload generator is having control over every aspect of the workload. VM workloads tend to be very complex, making it difficult to isolate the source of performance differences.

The workload sequentially accesses memory using one thread per vCPU, with each thread accessing a separate region of memory. We ran our workload on Red Hat 6 Enterprise Server in a VM with 4 vCPUs and 2GB of RAM. The workload has a working set size of 256MB. It allocates 1GB of memory for the test and fills that memory with data that is 50% compressible. In order to increase physical memory fragmentation, the workload allocates memory in a 16 page stride. Workload performance is measured by the time needed to access 100,000 pages. We checkpointed the VM in the middle of the workload test run, then restored the checkpoint and recorded the time to complete the test. The VM restores back to the start of the checkpointing, so the result does not include any overhead from creating the checkpoint. We separately discuss the cost of checkpoint save in Section 8.3.

8.1.1 Checkpoint Restore Overhead

We tested the overhead imposed by checkpoint restore on the microbenchmark for several different test configurations. We tested the current implementation of VM checkpointing in ESXi against Halite with all of its optimizations. We also measured the impact of locality blocks compared to other block organization schemes. We used a version of Halite without any memory file optimizations, such as compression and the zero page optimization, but with different block organizations. We do not use compression in this test, so each file block holds 8 memory blocks or 16 pages. Thus, on each fault to disk, 15 other pages in the block are “prefetched” from disk. We tested three organizations: memory pages grouped into blocks randomly (random blocks), memory pages grouped by physical address (physical address blocks) and locality blocks.

Random blocks should be the lower bound worst-case performance; on each fault to disk, 15 random pages are loaded along with the faulting page. Physical address blocks use the same memory file organization as the current ESXi implementation, but group 16 pages into a block to read for each fault to disk instead of a single 4KB page. This organization simulates the performance of ESXi if we had added Halite’s other optimizations, but kept the memory file organization. The test using locality blocks simulates Halite, but isolates the effects of locality blocks from Halite’s other memory file organizations, including compression and the zero page optimization.

Figure 2 gives the performance overhead of checkpoint restore for each of our test configurations. Each test result is the average of 10 test runs. The baseline runtime of the workload generator is 54 seconds on average. The restore overhead is given as the increase in runtime of the synthetic workload due to the VM being restored in the middle of the run. We use the same snapshot for all test configurations by reformatting the same memory file, so the performance before the checkpoint is identical and the difference in runtime is only due to the restore process.

Comparing ESXi and Halite, Halite reduces the restore overhead by 100 seconds or more than 10x. The three

Table 1: Efficiency of each type of block organization (no compression) given by number of blocks faulted in from disk and percentage of other pages in the blocks later accessed.

	Disk Accesses	VM Access %
Random	22,975	19%
P-A	14,501	36%
A-L	6,908	83%

bars in the middle of the graph show the impact of varying just the block organization. It is important to isolate the performance impact of different block organizations to understand the benefit offered by locality blocks.

As expected, random blocks perform the worst. Random blocks perform worse than even ESXi, although reading blocks of pages from disk should increase disk efficiency. This result shows that reading blocks from disk only improves performance if the other pages in the block are useful for eliminating future faults to disk. Otherwise, using bigger blocks only increases the latency of each fault to disk without reducing the overall number of faults. Physical address blocks halve the overhead compared to the random organization because the other pages in a block are more likely to be accessed, eliminating some faults to disk. This improvement is due to access locality in the physical address space.

Locality blocks perform the best, improving performance by 6x over random blocks and almost 3x over physical address blocks. We see further improvement because locality blocks have better access locality than physical blocks. More of the other pages in the block are likely to be accessed after the faulting page, eliminating more faults to disk. These results show how crucial block organization is for restore performance.

8.1.2 Memory File Organization

We can see how different block organizations impact performance by looking at the checkpointing statistics collected by Halite. Table 1 gives the total number of faults to disk for each block organization and the percentage of prefetched pages the VM accessed after each fault. Prefetched pages are the pages other than the faulting page in a block of pages brought in from disk. Accesses to prefetched pages eliminate faults to disk, improving disk efficiency.

It is clear that locality blocks lead to more efficient disk access than the other block organizations. There are fewer faults to disk and more pages in each faulted block are eventually accessed, eliminating more faults and increasing disk efficiency.

We collected hypervisor statistics on the percentage of

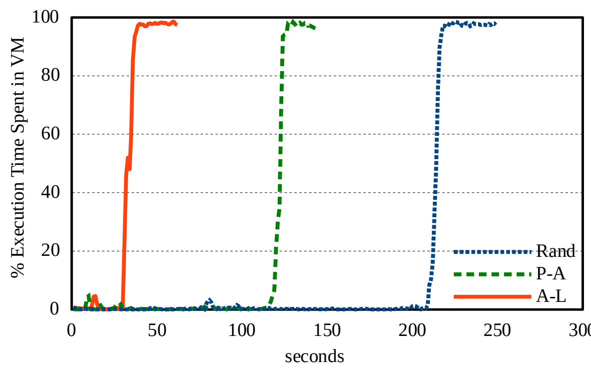

Figure 3: VM performance during lazy restore given as % of time spent executing in the VM (higher is better). Results are given for three block organizations: random, physical address blocks (P-A) and access locality blocks (A-L)

time spent running guest code to show the impact of faults to disk. Faults to disk pause the VM's execution, leading to less time spent running guest code and a reduction in performance for the guest. For the best performance, we want to return the guest to running almost 100% of the time as soon as possible.

Figure 3 shows the impact on guest execution during restore for the different block organizations. Locality blocks improve performance by reducing the period of time where the VM does not run much due to faults to disk. With locality blocks, the VM sees a large number of page faults for the first 30 seconds. That time increases to 120 seconds with a physical address blocks, and to 210 seconds with a random blocks. The total time to restore the VM also decreases from locality blocks to random blocks.

During the restore, the VM sees a large number of faults until the working set is entirely faulted in. The period where the VM sees performance degradation is not directly related to the test overhead given in Figure 2 because the VM is making some progress during that time. Locality blocks pack the VM's working set into fewer blocks and brings the working set in with fewer faults to disk.

For some workloads, we could prefetch these page before starting the VM as we previously proposed [24]. However, the working set cannot be determined before the VM restores for some workload, and in fact, changes while the VM runs. For those workloads, Halite provides better performance because locality blocks group pages that are likely to be accessed together into blocks, so Halite will still be able to efficiently fault in the working set, whereas working set restore would hurt performance by prefetching pages that are not needed before the VM starts. We saw this reduction in performance for working set restore with the Worldbench workload presented

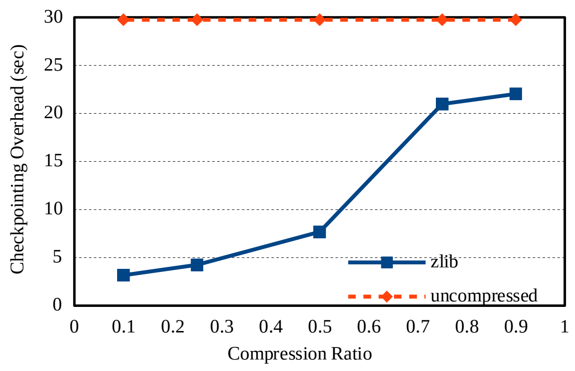

Figure 4: Restore overhead for Halite using compressed and uncompressed memory files for workloads with varying compression ratios. Smaller compression ratio means that the workload is more compressible. Performance is given as the increase in the synthetic workload runtime (lower is better).

in Section 8.2.1, whereas Halite provides a significant performance improvement.

8.1.3 Compression

The performance impact of including compression in Halite depends heavily on the compressibility of the workload. Thus, we compare Halite performance with a compressed and uncompressed memory file for workloads with different compression ratios. We varied the compression ratio of the memory allocated by the workload generator by packing some percentage of every page with already compressed data. These tests were performed using zlib [13]. Halite also supports LZRW [23]; however, we found that zlib performed similarly to or better than LZRW in most cases.

Figure 4 shows the performance results for workloads that compress to 10% of their original size to workloads that compress to 90% of their original size. The performance improvement due to compression varies from a 89% improvement to a 26% improvement depending on the compressibility of the VM's memory.

8.2 Application Benchmarks

In addition to our synthetic benchmark, we also evaluated the performance of Halite for a number of representative application benchmarks. These workloads vary in complexity, and therefore, the amount of divergence they exhibit. Worldbench is a simulated desktop application workload running in Windows XP. It is the most divergent; there are timing variances in the inputs from the workload generator, and Windows XP has many background processes that run at various times. In comparison, pgbench running on PostgreSQL in a server Linux install is more deterministic. The benchmark is deterministic and there are few background processes. Finally, our Apache server benchmark is designed to have divergence in the random selection of pages that we request from it, but it runs on top of Linux and has a small working set.

The workloads also vary in their compressibility, which affects the performance benefit of Halite, as shown in the previous section. The Worldbench checkpointed memory file only compresses to 67% of its original size due to a large number of media files in memory. The pgbench checkpointed memory file compresses to 10% of its original size due to pgbench filling the database with patterns.

8.2.1 Worldbench

Worldbench is a simulated desktop workload with typical desktop applications like word processing, web browsing and video editing. Worldbench closely simulates the expected workload of a VDI deployment described in Section 2.3. We ran Worldbench in Windows XP in a VM with 1GB of memory and 2 vCPUs. We used the multitasking test from the test suite that simulates a browser workload and media encoding. Worldbench reports the amount of time taken to run the test suite once. We checkpointed the VM 10 minutes into the test run, so the first 10 minutes are identical across runs. Each test is the average of 10 test runs.

We evaluated the performance of Worldbench on ESXi and three Halite configurations. The current ESXi implementation of checkpointing uses a memory file that is organized by physical address and reads one 4KB page on each fault to disk. The first Halite configuration is Halite (P-A), which gives the performance of Halite using physical blocks. Halite (P-A) uses Halite's checkpointing optimizations like threading and background thread throttling, but none of the memory file optimizations like locality blocks, compression and zero page optimization. This configuration simulates the performance of ESXi with the Halite optimizations that would be easy to add to ESXi, like increasing the block size faulted in from disk and throttling the background thread, but not changing the memory file layout.

The next Halite configuration is Halite (A-L), which gives the performance of Halite with locality blocks, which is the key contribution in Halite. This configuration isolates the performance impact of locality blocks, from other memory file optimizations like compression and zero page optimization. The last configuration is Halite with all optimizations including compression and zero page optimization. This configuration gives the total performance benefit of Halite over ESXi.

The baseline performance of Worldbench without checkpointing is 816 seconds. Figure 5 gives the Worldbench results as the average increase in the runtime due to checkpointing. Again, there is no performance impact from saving in these results because the checkpoint is

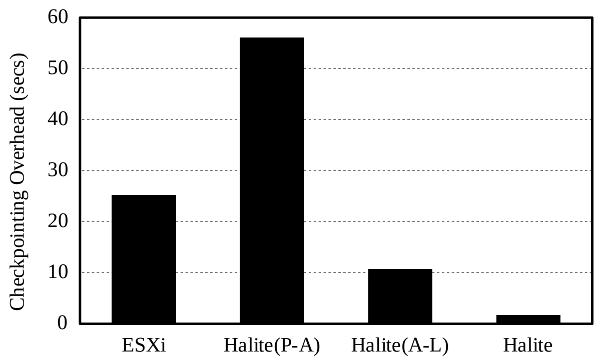

Figure 5: Checkpointing overhead for Worldbench given as number of seconds increase in runtime over baseline (lower is better).

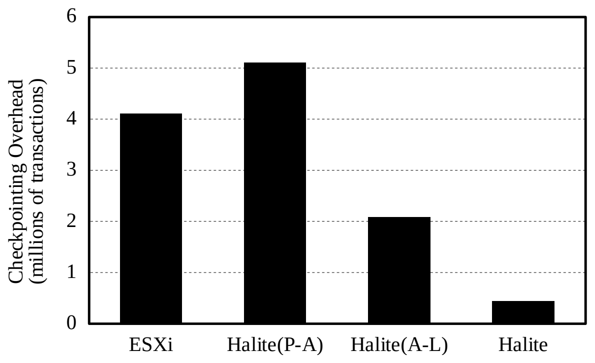

Figure 6: Checkpointing overhead for pgbench given as a reduction in transactions completed in 5 minutes over baseline (lower is better).

before the beginning of the save process, so the increase in runtime is only due to the lazy restore process. All configurations use the same memory file, re-organized into the appropriate block organization, so there is only one checkpoint across all of the configuration tests.

ESXi increases the runtime of Worldbench by more than 25 seconds due to faults to disk during the lazy restore period. Halite (P-A) adds a number of optimizations to ESXi, including physical blocks, which should increase disk efficiency. However, Worldbench has poor access locality in the physical address space, so physical blocks actually reduce performance like random blocks did in our microbenchmark test. Halite (P-A) actually increases the runtime overhead by almost 2x, up to 56 seconds on average.

Like Halite (P-A), Halite (A-L) also uses blocks, but locality blocks instead of physical blocks. Halite (A-L) reduces the runtime overhead by almost 6x compared to Halite (P-A), showing the importance of block organization based on access locality. Halite, which includes compression, further improves performance, reducing the checkpointing overhead to an average of 1.6 seconds. Compared to the current implementation of ESXi, Halite reduces restore overhead by 94%. This performance is a significant improvement over our previous work [24], which did not cope with divergence well and actually reduced performance for Worldbench.

8.2.2 pgbench

pgbench is a benchmarking tool, based on TPC-B, for the PostgreSQL database used to test the performance of a database installation. We used VMware's vFabric PostgreSQL [21] based on PostgreSQL 9.0. We ran pgbench and PostgreSQL in a Red Hat 6 Enterprise server in a VM with 2GB of memory and 4 vCPUs. pgbench measures database performance by recording the total number of transactions completed within a timed run.

We used a pgbench run of 5 minutes with 16 clients running on 4 threads. We ran pgbench with only select queries because we found the performance to be more consistent and it avoided disk contention. The default 85/15 read/write mix showed similar performance improvements, but with a larger range of performance over test runs. We checkpointed pgbench at the beginning of the run and collected the test results after restoring. vFabric PostgreSQL is designed to be an in-memory database, so we sized our database to 1.8GB.

We used ESXi with the same Halite configurations as the Worldbench experiments. pgbench shows the difference between the different configurations for a workload that has less divergence. The baseline performance of pgbench with no checkpoint taken is 6.9 million transactions. Figure 6 shows checkpointing overhead for pgbench as the reduction in number of transactions completed in 5 minutes. For example, pgbench completes 2.8 million transactions after restoring from a checkpoint on ESXi, a reduction of 4.1 million over the baseline. We chose to plot the overhead metric because it stays constant regardless of the length of the test.

For pgbench, Halite (P-A) only increases checkpointing overhead by 20% compared to ESXi. This increase is smaller than Worldbench because Linux workloads have better physical address locality due to Linux's buddy allocator. Still, Halite (A-L) reduces performance overhead by more than 2x for both ESXi and Halite (P-A). Halite with compression and the zero page optimization further reduces the overhead by 75%. Compared to ESXi, Halite reduces performance overhead by 89%.

8.2.3 Apache Webserver

To evaluate the performance of Halite for dynamic VM allocation that we described in Section 2.1, we created an experiment to simulate an Apache server application running in a VM. The test uses an Apache server with an

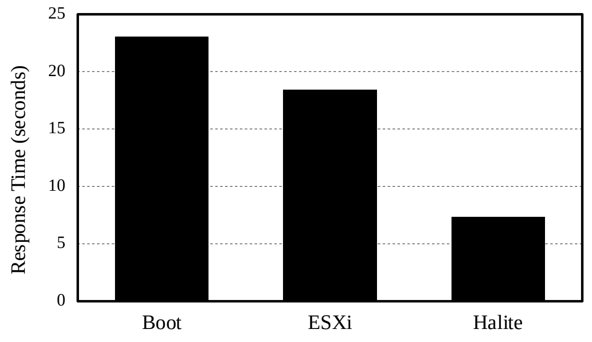

Figure 7: Apache Response Times. Time until first response (lower is better) from Apache server starting from boot of the VM, an ESXi checkpoint and a Halite checkpoint.

HTML dump of Spanish Wikipedia[2]. The test client requests pages randomly with a Gaussian distribution from a set of 10,000 pages from the dump. Before checkpointing, the client makes 10,000 requests to warm the cache. We ran Apache in Ubuntu 10.04 Server in a VM with 2 vCPUs and 2GB of memory.

We tested the performance of our server application with a few scenarios. First, we measured the response latency of the server when booting the VM and starting the web server on the first HTTP request. This scenario simulates dynamic allocation of VMs without using checkpointing. Next, we measured the response latency when restoring a checkpoint of a VM with a running Apache server using the current ESXi implementation. This scenario reflects the performance of using ESXi checkpointing for dynamic VM allocation. Finally, we tested the latency using Halite to restore the server VM.

Figure 7 gives the response times for each setup. Dynamically allocating a new VM for each connection requires 23 seconds on average to respond to the first HTTP request. Restoring a checkpointed VM with a running Apache server using the current ESXi checkpointing implementation gives response times of 18.4 seconds on average. Using Halite reduces response time to 7.3 seconds. Halite reduces the response time of a dynamically allocated web server VM by a factor of three compared to a cold boot of the VM and a factor of 2.5 compared to a restore using the current ESXi implementation, making it much more feasible to dynamically allocate VMs.

We also measured the response times for subsequent HTTP requests to the web server. Once the connection to the server has been established, our test client issues 10,000 random page requests, also with a Gaussian distribution. These measurements further show the benefits of Halite as well as showing the benefit of using a checkpoint,

[2]`http://dumps.wikimedia.org/eswiki/`

Table 2: Average response time and maximum response time for the first 10,000 requests, excluding the first request.

	Avg. Response Time	**Max Response Time**
Boot	18 ms	25.168 s
ESXi	13 ms	9.333 s
Halite	3 ms	3.010 s

Table 3: Checkpoint save overhead for ESXi and Halite given as increase in runtime or reduction in transactions over baseline.

Workload	**ESXi**	**Halite**
Synthetic workload (sec)	1	4
pgbench (millions of trans.)	.31	.76
Worldbench (seconds)	11	5

rather than a newly booted VM, for dynamic allocation. Table 2 shows the average and maximum response time for HTTP requests issued to the web server.

Since the VM's cache is on disk at the beginning of all of these scenarios, the performance of the web server depends entirely on how efficiently the VM's cache can be filled from disk. Using a checkpointed VM reduces both average and maximum response times. The web server running in the VM already has a warmed cache, so the VM's working set just has to be restored, the cache does not have to be refilled. Restoring the checkpoint using ESXi reduces the response times, however, Halite performs the best because it is able to most efficiently restore the VM's working set from disk. Halite reduces the average response time of the web server by a factor of 6 and the maximum response time by a factor of more than 8 over booting the VM and starting the web server for each connection.

8.3 Checkpoint Save

Since most of the work for checkpoint save is done asynchronously in ESXi and Halite, the difference in performance between the two is minimal. Halite does copy-on-access checkpointing, which increases the checkpointing overhead, but writes out to disk sequentially, which reduces the overhead.

Table 3 gives performance results for our synthetic workload, pgbench and Worldbench. Performance was measured for each workload after a checkpoint was taken in the middle of the run. For the synthetic workload and pgbench, the additional read traces only reduced performance by 7-8%. These two workloads are both read-only workloads, so the performance impact is higher than for a more balanced read-write workload. Halite performs better than ESXi for Worldbench due to the more write-heavy workload.

9 Related Work

Most previous checkpointing systems focused on checkpoint save performance for supporting fault tolerance, so there is a limited body of work on improving checkpoint restore performance. For systems that do not support lazy restore, the memory file organization is not important, so it is frequently not addressed. We also describe some techniques used by process and file system checkpointing systems to optimize checkpointing.

9.1 Virtual Machine Checkpointing

There is not a large body of work on virtual machine checkpointing and almost all of it focuses on checkpoint save performance. Many [15, 20] do not address the restore algorithm at all.

Commercial hypervisors all include support for checkpointing and restoring VMs, however not all support lazy checkpoint save or restore due to the complex implementation. For systems that do not support lazy checkpoint restore, the disk layout of checkpointed memory does not matter, although performance could be improved using compression. Xen supports lazy checkpointing [4], but it is not clear whether it supports lazy restore or what organization is used for checkpointed memory.

Our previous work [24] addressed the issue of lazy restore performance by prefetching the working set of the VM's memory before restarting the VM. However, we found that, while it was effective for simple workloads like MPlayer running on basic Linux, it offered little benefit for more complex workloads, like Windows desktop applications. These complex workloads have more divergence, and since working set restore depends on predicting which memory pages the VM will access on restore, it cannot cope with divergence. In contrast, Halite focuses on predicting which pages the VM will access *together* on restore, making it more effective for a wider range of workloads, including a 94% reduction in restore overhead for Windows workloads.

9.2 Other Virtualization

One related area of work is VM migration. Post-copy migration suffers from the same performance challenges as lazy restore due to the network latency while the VM is paused waiting for the page to be copied from the source. However, the organization of VM memory is not a factor because the VM's memory is not on disk on the source, so it can be accessed in any order with no performance penalty.

Hines et al. [7] implemented a background page walking thread that adaptively picks the order in which it walks depending on the last access. For each access, the page walker will try to push some of the other pages around that access in the physical address space. However, previous work [16] found that the guest virtual address space is a more reliable predictor of access locality and we found in our experiments that locality in the physical address space can be poor.

VM fork is another solution for dynamic allocation of virtual machines that requires restoring memory while the VM runs. Snowflock [10] depends on there being a small difference between forked VMs that the memory can be sent from the parent to the child with little performance degradation for the child VM. Kaleidoscope [2] groups pages based on what the page is used for as another way to predict access locality. Our approach is more general because it does not require paravirtualization to categorize pages.

9.3 Process Checkpointing

Previous work in optimizing checkpointing for individual or distributed processes has focused primarily on checkpoint save, but not checkpoint restore. Plank et al. [17] implemented the process checkpointing system *Ickp* using copy-on-write checkpointing as well as compression as an optimization. Li et al. [11] compare performance characteristics of four algorithms for checkpoint/restart of parallel programs. The work of Liao et al. [12], called Concurrent CKPT, aims to improve on the CLL algorithm by avoiding page table manipulation. However, all of this work focuses solely on checkpoint save performance, and does not discuss checkpoint restore.

9.4 Fast OS and Application boot

There has been some work on organizing operating systems and application files to improve boot times for both. Windows uses a mechanism called SuperFetch [8] that orders files on disk in the order that they are accessed during boot. SuperFetch uses an adaptive algorithm that tracks past boot processes to predict the order of accesses. Like Halite, SuperFetch addresses the performance of restoring some set of data by reordering the data on disk in a more optimal way. Unlike Halite, SuperFetch attempts to predict the order of all accesses, not just the locality, so performance suffers when there is divergence. However, divergence may be less of a concern for booting the OS.

Joo et al. [9] implemented a system that predicts and prefetches application data on application startup to optimize for interleaving application execution and I/O. Like other predictive techniques, theirs suffers from reduced performance on divergence, although divergence is less of a problem for applications. They do not address disk layout at all because their system is designed for SSDs. However, they could improve performance on SSDs by reorganizing the data for prefetching into fewer blocks.

10 Conclusion

We presented a new checkpointing system, Halite, for VMware ESXi that reduces restore overhead for a range of workloads. Halite predicts which pages the VM will access together on restore and groups these pages into *locality blocks*. We showed that locality blocks offer significant performance benefits over other block organizations and copes well with divergence in complex VM workloads like Windows desktop applications. In particular, locality blocks outperform physical address blocks by 10x for Windows. Combining locality blocks with Halite's other optimizations, Halite reduces the overhead of checkpoint restore in VMware ESXi to 1.6 seconds for a Windows desktop workload, a reduction of 94%. This significant improvement in restore performance allows Halite to efficiently support new applications for VM checkpointing.

11 Acknowledgements

Many thanks to Karen Zee, Ron Mann and Dan Ports for their discussions on this work. Thanks to all of the members of the monitor group for their support throughout the project. Thanks to Steve Gribble and Pete Hornyack for their insightful evaluation of the evaluation. Thanks to the entire University of Washington systems lab and to our anonymous reviewers for their paper comments. Special thanks to our manager, Joyce Spencer, for her continued support and encouragement throughout this work.

References

[1] Amazon. Amazon EC2 FAQ. `aws.amazon.com/ec2/faqs/`.

[2] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. In *Proceedings of the 6th European Conference on Computer Systems*, EuroSys '11, pages 273–286, New York, NY, USA, April 2011. ACM.

[3] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating System Principles*, SOSP '01, pages 103–116, New York, NY, USA, October 2001. ACM.

[4] Patrick Colp, Chris Matthews, Bill Aiello, and Andrew Warfield. VM Snapshots, February 2009. `http://www.xen.org/files/xensummit_oracle09/VMSnapshots.pdf`.

[5] Tathagata Das, Pradeep Padala, Venkata N. Padmanabhan, Ramachandran Ramjee, and Kang G. Shin. Litegreen: saving energy in networked desktops using virtualization. In *Proceedings of the USENIX Annual Technical Conference*, USENIX'10, pages 3–3, Berkeley, CA, USA, June 2010. USENIX Association.

[6] Apache Software Foundation. Apache http server project, 2012. `http://httpd.apache.org/`.

[7] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 5th Conference on Virtual Execution Environments*, VEE '09, pages 51–60, Washington, DC, USA, March 2009. ACM.

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

[9] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. Fast: quick application launch on solid-state drives. In *Proceedings of the 9th Conference on File and Storage Technologies*, FAST '11, Berkeley, CA, USA, February 2011. USENIX Association.

[10] Horacio Andrs Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th European Conference on Computer Systems*, EuroSys '09, pages 1–12, Nuremberg, Germany, April 2009. ACM.

[11] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Parallel & Distributed Systems*, pages 874–879, August 1994.

[12] Jianwei Liao and Yutaka Ishikawa. A new concurrent checkpoint mechanism for real-time and interactive processes. In *Proceedings of the 34th Computer Software and Applications Conference*, COMPSAC '10, pages 47–52, Washington, DC, USA, 2010. IEEE Computer Society.

[13] Jean loup Gailly and Mark Adler. zlib. `zlib.net`.

[14] Michael J. Mior and Eyal de Lara. Flurrydb: a dynamically scalable relational database with virtual machine cloning. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 1–9, New York, NY, USA, 2011. ACM.

[15] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th Conference on Virtual Execution Environments*, VEE '11, pages 75–86, New York, NY, USA, March 2011. ACM.

[16] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: coalesced large-reach TLBs. In *Proceedings of the Conference on Microprogramming and Microarchitecture*, MICRO '12. IEEE, December 2012.

[17] James S. Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, June 1994.

[18] PostgreSQL. pgbench. `http://www.postgresql.org/docs/devel/static/pgbench.html`.

[19] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[20] Michael H. Sun and Douglas M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Georgia Institute of Technology, 2010.

[21] VMware. VMware vfabric postgres. `http://www.vmware.com/products/application-platform/vfabric-postgres/overview.html`.

[22] VMware. VMware vSphere Hypervisor. `www.vmware.com/products/vsphere-hypervisor/overview.html`.

[23] Ross N. Williams. `http://www.ross.net/compression/introduction.html`.

[24] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th Conference on Virtual Execution Environments*, VEE '11, pages 87–98, New York, NY, USA, March 2011. ACM.