HydraVM: Low-Cost, Transparent High Availability for Virtual Machines

Kai-Yuan Hou*, Mustafa Uysal†, Arif Merchant‡, Kang G. Shin* and Sharad Singhal§
*Department of Eletrical Engineering and Computer Science
University of Michigan, Ann Arbor, Mighican
{karenhou,kgshin}@eecs.umich.edu
†VMware Inc.
muysal@vmware.com
‡Google Inc.
arif.merchant@acm.org
§HP Labs
sharad.singhal@hp.com

***Abstract*—Existing approaches to providing high availability (HA) for virtualized environments require a backup VM for every primary running VM. These approaches are expensive in memory because the backup VM requires the same amount of memory as the primary, even though it is normally passive. In this paper, we propose a storage-based, memory-efficient HA solution for VMs, called *HydraVM*, that eliminates the passive memory reservations for backups. HydraVM maintains a complete, recent image of each protected VM in shared storage using an incremental checkpointing technique. Upon failure of a primary VM, a backup can be promptly restored on any server with available memory. Our evaluation results have shown that HydraVM provides protection for VMs at a low overhead, and can restore a failed VM within 1.6 seconds without excessive use of memory resource in a virtualized environment.**

***Keywords*-Virtualization; High Availability; VM Checkpointing; VM Restore; Shared Storage;**

Submission Category: Regular paper

Word Count: 9345 words

Declaration: The material has been cleared through all authors' affiliations.

I. Introduction

Virtualization is widely used in contemporary IT infrastructures for it enables flexible partition and dynamic allocation of physical computing resources. In a virtualized environment, multiple virtual machines (VMs) are consolidated on a physical server to reduce deployment and management costs. However, when a physical server failure occurs, for example, due to unexpected power loss or hardware faults, all the VMs running on that server suffer service outages. Providing high availability (HA) for VMs to survive failures of their physical hosts is therefore a crucial task.

Automatically restarting the VMs affected by a physical host failure in other healthy hosts (e.g., [1]) is usually not enough. Users of the VMs notice disruption of services, as they may need to wait more than 10 seconds for a failed VM to reboot and become available for use again. The applications that were running in the VM before its failure may not resume execution from where they left off when the VM restarts from its disk image. Human intervention is usually needed as applications are *stateless* if not specially designed and instrumented.

A *stateful* approach is required to reduce application downtime and lost work from physical host failures. Traditional state-machine replication [2] and primary-backup replication [3] approaches add complexities to building highly available applications. Virtualization assists to reduce this complexity by providing HA support at the infrastructure level for any application running in a VM. The current state-of-the-art HA solution for VMs [4] is to provide, for each primary VM, a backup VM in a different physical host. The backup VM is normally passive (not operating). It acts as a receptacle of the primary VM's state changes, and can quickly take over execution from the primary's most recent state when the primary fails. The cost of this approach is high, because the backup VM reserves as much memory as the primary, and this memory space cannot be used by other VMs even though the backup is inactive until a fail-over is required. The reservation of backup memory degrades resource efficiency in a virtualized environment, offsetting some of the benefits gained through consolidation.

In this paper, we propose a *storage-based, memory-efficient* approach, called *HydraVM*, to providing high availability support for virtualized environments. Our primary design objective is to protect VMs from failures of their physical hosts *without* any idle backup memory reservations. Instead of creating backup VMs to keep track of the primary VM state, we maintain a fail-over image that contains a complete, recent memory checkpoint of the primary VM in a networked, shared storage, which is commonly available in a virtualized environment. In the event of primary failure, a spare host is provisioned, and we restore the failed VM based on its fail-over image and a consistent disk state, and activate the VM promptly to take over execution.

The storage-based HydraVM approach has several benefits. It uses inexpensive shared storage for maintaining VM fail-over images in place of expensive DRAM, reducing the hardware costs for providing HA support. It frees up the memory reserved by idle backup VMs for better usage, improving resource efficiency in a virtualized environment. Spare memory can be replenished to existing VMs for enhancing performances, or used to host additional VMs and upgrade system throughputs. Since HydraVM maintains the fail-over image in a shared storage instead of a dedicated backup machine, in case of a failure, the affected VM may be recovered on any physical host that has access to the storage. This allocation flexibility allows fail-over to any host that currently has available memory capacity, which is critical given the highly variable utilization of hosts in a virtualized environment.

The remainder of this paper is organized as follows. The next section provides an overview of HydraVM, and describes the rationale of its design. Section III and IV details the protection and recovery mechanisms of HydraVM against failures. Our experimental results are presented and analyzed in Section V. Section VI summarizes related work. We discuss future directions and conclude the paper in Section VII.

II. Design Rationale and Overview of HydraVM

A. Motivation and Objectives

The design of HydraVM is motivated by two key observations. First, existing approaches [4], [5] for providing high availability to VMs maintain a backup for each (primary) VM to quickly take over when a failure occurs. The backup VM consumes as much main memory as the primary, even though it stays passive for most of time. This idle reservation effectively *doubles* the memory requirement of each VM without providing any additional productivity in the normal course of operation.

Our second observation is that it is difficult to either take advantages of or adjust the idle reservation of backup memory. Current platform virtualization technologies, such as Xen and VMware ESX, either do not support or do not prefer machine-wide paging to "schedule" physical page frames across hosted VMs, because a meta-level page replacement policy can introduce performance anomalies due to unintended interactions with the memory management systems in the guest VMs [6]. As a result, once a memory area is reserved (although not actively used) by a backup VM, it cannot be utilized by other running VMs. The backup VM could possibly be "paged out" when remaining idle [7]. However, it needs to be swapped in very frequently to synchronize with the primary execution state, creating a non-trivial overhead. The popular memory ballooning technique [6] used for run-time management of VM memory allocations is not helpful for shrinking backup memory reservation either, because the backup VM is not operational and hence cannot exercise its balloon driver.

Our primary design goal is therefore to *eliminate* the idle reservation of backup memory, and our approach is to "maintain" backup VMs in a stable storage instead. HydraVM maintains in a networked, shared storage a fail-over image for each protected (primary) VM, based on which a backup VM can be quickly restored and activated to take over when the primary fails. The shared storage for storing VM fail-over images is fail-independent to the physical servers hosting VMs. Such a shared storage accessible to every VM server host is usually provided in a virtualized environment to facilitate management of VMs via, for example, a storage area network (SAN) or cluster filesystem, and built with storage-level redundancy mechanisms which can be leveraged to guarantee reliability for the fail-over images.

Besides eliminating unnecessary memory reservations, HydraVM must also satisfy several other properties to be practically useful. It should provide protection for VMs against failures of physical servers at low overheads to be deployable in real-world systems. In the event of a host failure, fail-over of the affected VMs must occur quickly, and the amount of completed work lost due to the failure should be reasonably small.

B. System Overview

HydraVM has two operating modes, *protection* and *recovery*. Figure 1 illustrates the operation of HydraVM.

A primary VM runs in a host that may fail. During the normal execution of the primary, HydraVM operates in the protection mode. Protection for the primary VM begins when HydraVM takes an initial full checkpoint of the VM, which contains the complete VM execution state. The full checkpoint is taken only once at the beginning. The complete VM memory state captured is stored as the VM fail-over image in the VM checkpoint store, while a VM disk state consistent with the memory state in the image being kept in the VM disk server, which hosts the virtual disks of the primary VM. VM checkpoint store and VM disk server can be one or separate storage servers in the shared storage system of a virtualized environment.

As the primary VM operates, HydraVM keeps track of the execution state of the primary by taking VM checkpoints periodically, so that in case there is a failure, the primary can be transparently recovered from a recent state. HydraVM takes checkpoints of the primary VM *incrementally* (Section **??**). It doesn't have to send the full VM memory image everytime the VM state is captured, but only changes to the primary state since the last checkpoint taken, to be consolidated in the fail-over image in the VM checkpoint store. HydraVM also implements a *copy-on-write* (CoW) technique (Section III-B) to minimize the performance disruptions to the primary VM from continuous checkpointing.

Once a failure of the primary host is detected, HydraVM switches to the recovery mode and reacts to the failure. A restoration host with sufficient available memory is selected

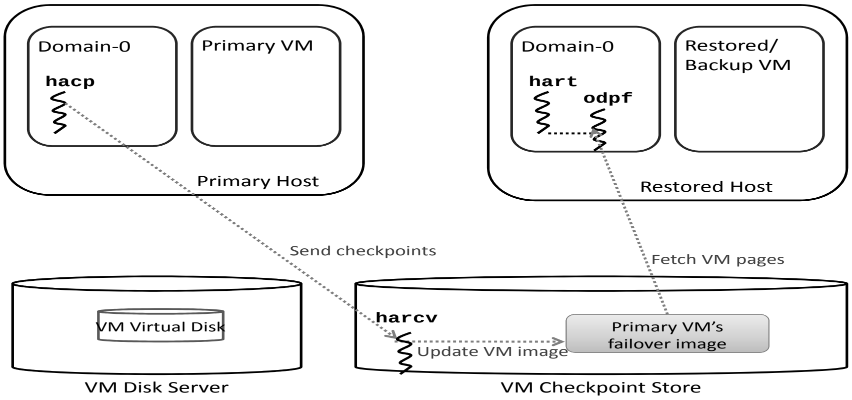


Figure 1. HydraVM overview.

and provisioned, either from the stand-by nodes, or from the surviving nodes, in which the failed primary VM is restored based on the fail-over image in the VM checkpoint store and a consistent persistent state in the VM disk server. To perform fail-over quickly, HydraVM implements *slim VM restore* (Section IV-A) to load the minimum information needed from the fail-over image, instantiate the backup VM, and immediately activate the VM to take over execution from the most recent state recorded before the primary failure. The VM memory contents not loaded during fail-over are provided to the VM *on-demand* when the VM accesses them (Section IV-B).

HydraVM maintains fail-over images on stable storage, eliminating the memory reservations made by idle backup VMs. In order to limit the amount of completed work lost upon failure, frequent VM checkpoints need to be taken, which requires the time to store each incremental checkpoint to be short. We propose to use a solid state device (SSD), such as a flash drive, to hold the VM fail-over images. SSDs are cheaper than DRAM, and faster than hard disks. They provide much better performance than mechanical drives, for both sequential and random I/Os [8]. By holding fail-over images on a SSD, the average time required to consolidate incremental checkpoints can be significantly reduced, and as a result, checkpoints of a protected VM can be taken more frequently, achieving greater protection of the VM.

While we propose deployment of a SSD-enabled checkpoint store to further improve system performance, HydraVM is not confined to systems that have SSDs available. In Section V, we evaluate HydraVM on both a disk and a SSD-based checkpoint store and demonstrate the feasibility of our approach. Although a SSD-based system enables checkpoints of a primary VM to be taken more frequently, we point out that not all applications require a very high checkpointing frequency. Users of the applications that need to be checkpointed extremely frequently (and yet, do not have a checkpointing mechanism built-in) may be willing to accept the cost of existing approaches that maintain a designated backup VM in memory [4], [9], [5], [10]. HydraVM complements existing HA approaches by providing a memory-efficient solution to providing VM protection against host failures that is practically useful and deployable in real-world virtualized environments.

We built a prototype HydraVM system on the Xen hypervisor [11]. HydraVM implements its VM protection and recovery mechanisms for high availability as Xen management commands (`hacp` and `hart`) to be invoked via the `xm` user interface. These commands can be used by HA provisioning agents to initiate the protection for a primary VM and the restoration of a backup VM in case the primary fails. In the next two sections, we provide details of the design and implementation of the HydraVM protection and recovery mechanisms.

III. HydraVM Protection

HydraVM tracks *changes* to the VM state by taking incremental checkpoints. A checkpointing daemon, called `hacp`, runs in the privileged management VM (Domain 0 in Xen's terminology) in the primary host, as shown in Figure 1. It takes incremental checkpoints of the primary VM periodically and sends the checkpoint data over the network to the VM checkpoint store. A receiver daemon, `harcv`, runs in the checkpoint store and receives checkpoints from the primary periodically. It merges all of the VM state changes included in each checkpoint and maintains a correct and consistent fail-over image for the primary VM.

A. *Incremental Checkpointing to Storage*

HydraVM implements incremental VM checkpointing similar to that proposed in Remus [4] and Kemari [9]. Unlike these approaches, HydraVM stores the checkpoints taken in a shared storage rather than in server memory. The `hacp` checkpointing daemon takes an initial full snapshot of the primary VM and stores the VM image in the VM checkpoint store as a regular file. This image file has approximately the same size as the amount of memory configured for the primary VM. It stores all memory pages of the primary VM sequentially in their order, along with the virtual CPU state, and contains information describing the configuration of the primary VM, for example, its resource usage and virtual device state.

As the primary VM executes, HydraVM tracks the changes of its memory and virtual CPU state by a series of incremental checkpoints. To create an incremental checkpoint, the `hacp` checkpointing daemon temporarily pauses the primary VM, while it identifies the changed state and copies them to a buffer. We leverage the *shadow page tables* support [12] provided by the Xen hypervisor to identify the state that has changed since the last checkpoint taken. Shadow page tables are turned on for the primary in *log-dirty* mode. In this mode, the hypervisor maintains a private (shadow) copy of the guest page tables in the primary and uses page protection (marking all VM memory read-only) to track writes to the primary's memory. After the memory pages that are dirtied since the last checkpoint are identified, they are copied to a buffer along with the changed CPU state, and the primary VM is un-paused and can resume execution.

While the primary VM continues its execution, the `hacp` daemon transmits the buffer containing the changed state to the `harcv` receiver daemon running in the VM checkpoint store. The `harcv` daemon receives each incremental checkpoint in its entirety, and can either store the checkpoint as an individual patch file to be merged with the VM fail-over image during the activation of a bakcup VM, or update the VM image to contain the most recent page contents included in the incoming checkpoint. HydraVM uses the latter approach, since merging a series of checkpoint patch files involves reading them from disk and committing their contents to various locations in the fail-over image, and is a time-consuming process. If the merging is performed when restoration of a backup VM is required, the fail-over time would be unacceptably long. In HydraVM, the `hacrv` daemon updates the fail-over image with the set of dirtied memory pages and changed CPU states contained in a checkpoint as it is received, and commits all changes to the storage media to guarantee durability. It then sends an acknowledgement back to the `hacp` daemon confirming the completion of this checkpoint. Note that the set of state changes captured in one incremental checkpoint is applied to the fail-over image in its entirety; changes to the image are not made until the entire incremental checkpoint has arrived at the checkpoint store to ensure that a correct and consistent image of the primary VM is available for fail-over in the shared storage at all times.

B. *Copy-on-Write Checkpointing*

The primary VM is paused for taking each incremental checkpoint to ensure that a consistent set of memory and CPU state changes are captured together in one checkpoint. While the primary VM is paused, the `hacp` daemon first identifies the set of memory pages dirtied since the last checkpoint, gains access to these dirtied VM pages, and then copies the contents of the dirty pages to a separate buffer. The dirty pages need to be copied in order to isolate their contents from the primary VM while the `hacp` daemon transmits them to the shared storage in parallel with primary execution. Otherwise, if the primary VM modifies a dirty page before `hacp` is able to send the page out, by the time the page is transmitted, its contents have changed from what was captured in the checkpoint, and this may corrupt the consistency of the checkpoint.

Pausing the primary VM for every incremental checkpoint interrupts its normal execution, and hence, it is critical for the duration of these interruptions to be minimized. It takes a non-trivial amount of time for `hacp` to gain access to all dirtied VM pages in an incremental checkpoint. Multiple hypercalls are needed to ask the hypervisor to map batches of dirty pages belonging to the primary VM for the daemon. We enhance the implementation by requesting access to the entire primary memory space once at the beginning of protection of the VM, so that `hacp` need not map regions of the primary's memory repeatedly everytime when the primary is paused.

Copying the contents of the dirty pages also significantly increases the VM pause time. In fact, not every dirty page in a checkpoint needs to be duplicated. If a dirty page is not modified by the resumed primary execution, its content remains consistent with the checkpoint captured and the `hacp` daemon can transmit the page directly without making a copy. When the primary VM touches a dirty page, it does not create any problem if the primary's modification happens after `hacp` sent the page to the shared storage. The content of a dirty page needs to be duplicated only if the page has not been transmitted to the checkpoint store and is about to be modified by the primary. HydraVM implements Copy-on-Write (CoW) checkpoints to copy these pages only.

CoW checkpointing is implemented by maintaining a bitmap which indicates the memory pages to be duplicated before modifying them. While the primary VM is paused, we mark the bits in the CoW bitmap for all dirty pages included in the checkpoint captured, and resume primary execution without copying the dirty page content. While the primary VM is running, if a page marked in the CoW bitmap is detected to be modified, we duplicate its content to a separate buffer before allowing the modification. The hacp daemon is notified, so that it may send the copied page, not the modified page being used by the primary VM, to the checkpoint store. For all other dirty pages, hacp transmits the VM pages directly. CoW checkpointing reduces the VM pause time by not copying page content while the VM is paused. It also reduces page duplication costs, as pages are copied only for a subset of the dirty pages included in each incremental checkpoint.

HydraVM focuses on tracking VM memory state in a storage-based fail-over image that can be used to recover the protected VM in case a failure occurs. For correct recovery, a VM disk state consistent with the fail-over memory image is needed. To meet this requirement, our system currently hosts VM virtual disks in LVM [13] logical volumes and uses the the snapshot capability of LVM to capture the VM disk state at the time of an incremental checkpoint. LVM snapshots are instant and incur a minimum overhead since it uses a CoW technique for disk blocks to record only changes made to a logical volume after a snapshot is created.

IV. HydraVM Recovery

When a primary failure is detected, a restoration host with sufficient memory is provisioned. A VM restore agent, called hart as shown in Figure 1, is invoked to quickly bring up the failed VM in the restoration host based on its fail-over image in the checkpoint store. The restored VM starts operation immediately after being created, while its memory space is only partially populated during the fail-over. As the VM executes, it sends requests for the missing memory contents to a memory page fetching daemon, called odpf, which then provides the pages needed from the fail-over image.

A. Slim VM Restore

To quickly bring up a backup VM is especially challenging for HydraVM, because in our approach, VM fail-over image is not kept in server memory, but on a networked, stable storage. As a result, we must quickly load the fail-over image into memory in the restoration host and activate the backup VM. However, it is impractical to delay the execution of the VM until the complete VM image is loaded, as this can take an unacceptable length of time—20 to 40 seconds in our experiments with 1-2GB VM images. We perform a "slim" VM restore that loads only the minimum amount of information from the VM image required to instantiate the backup VM, and then resumes the execution of the VM immediately, without waiting for its memory contents to be fully populated.

In the event of a fail-over, the hart agent first loads the configuration information from the fail-over image in the VM checkpoint store. The configuration of the VM describes its allocated resources and virtual devices, and is used by the VM restore agent to create a VM container with sufficient memory to host the backup VM, establish a communication channel between the backup VM and the hypervisor, and properly connect virtual devices to the backup VM. A few memory pages that are shared between the VM and the hypervisor are loaded subsequently. These pages contain important information about the VM, for example, running virtual CPU, wall-clock time, and some architecture-specific information, and are critical to starting VM execution.

The memory space of the backup VM is then initialized with all page table pages of the VM. Page tables in the primary VM before its failure contain references to the physical memory frame in the primary host. When checkpointing a page table page in the primary during protection, all such references are translated to *pseudo-physical frame numbers* [14], which are host-independent indices to the contiguous memory space as seen by the VM. When the hart agent loads a page table page from the VM image, it walks through the page table entries, assigns physical memory frames in the restoration host for the VM pages, and updates the references to point to the allocated memory in the restoration host. Finally, the virtual CPU state is loaded, and the backup VM is switched on for execution.

B. Fetching VM Pages On-demand

Immediately after the VM resumes execution, its memory space is only partially populated. Only a minimum number of VM pages (those loaded during slim VM restore) are in-place in the VM memory space and ready for use; no data pages of the VM are loaded from the fail-over image.

As the backup VM executes and accesses its memory pages, valid contents must be supplied for the execution to proceed. In HydraVM, memory references of the backup VM are intercepted by the hypervisor. If a memory page accessed by the restored VM is not yet present in the VM's memory space, a page fetch request is sent to the odpf fetching daemon, which loads the contents of the requested memory page from the VM image in the VM checkpoint store into proper location in the restored VM's memory space; hence, execution of the backup VM can proceed.

Memory references made by the restored VM can be intercepted in different ways. One approach is to mark all the resident pages in the backup VM as *not-present* in their page table entries while loading page tables during the slim VM restore. As a result, a page fault exception is forced and trapped into the hypervisor whenever the backup accesses a page. However, this approach requires significant changes to the guest OS kernel. Instead, we leverage Xen's support of shadow page tables to detect memory accesses from the backup VM. Since the "shadow" copy of the guest page tables maintained by the hypervisor is initially empty and to be filled in as guest pages are accessed, faults on shadow page tables can be used as indicators of VM memory references.

Before finishing the slim VM restore, the hart restore agent enables shadow page tables for the restored VM, and the VM is un-paused to resume execution. At this time, the odpf fetching daemon is ready to service page fetch requests from the VM. When a page that is not yet present in the VM's memory space is accessed, the VM is paused and a page fetch request is issued to the fetching daemon, specifying the index of the requested page. Once the fetching daemon finishes loading the page contents, the backup VM is un-paused and continues executing. By fetching VM pages on demand, no unnecessary pages are brought in for the backup VM, reducing the number of I/Os needed. Although backup VM's execution is inevitably interrupted by page fetches, the frequency of such interruptions is significantly reduced once the working set of the VM is brought in.

V. Evaluation

We built a prototype of HydraVM on the Xen 3.3.2 hypervisor. In this section, we evaluate the effectiveness and efficiency of our system. Our evaluation focused on the following two key questions:

- What type of VM protection does HydraVM provide without any idle memory reservations, and what are the associated operational overheads?
- When a machine failure is detected, how quickly does HydraVM bring a failed VM back alive, and how is the operation of the restored VM affected by the fail-over performed?

Next, we describe the test environment and the workloads used in our experiments. We then present the experimental results and our analysis.

A. Testbed, Methodology, and Workloads

All of our experiments were run on a set of HP Proliant BL465c blades, each equipped with two dual-core AMD Opteron 2.2 GHz processors, 4 GB RAM, two Gigabit Ethernet cards, and two 146 GB SAS 10K rpm hard disks. We set up our testbed as illustrated in Figure 1, one blade for each host. All four blades in our testbed are in the same LAN.

The VM under test runs in the primary host while being protected by HydraVM, and uses one dedicated network interface in the host (eth0). The virtual disks of the protected (primary) VM are hosted in the VM disk server under LVM, and mounted in the primary host via NFS. The hacp checkpointing daemon runs in Domain 0, takes periodic checkpoints of the primary VM at different checkpointing frequencies, and sends the checkpoint content to the VM checkpoint store via the other network interface in the primary host (eth1).

In the event of the primary host failure, the failed primary VM is brought up in the restoration host based on the fail-over image in the VM checkpoint store, and a consistent version of its disk state in the VM disk server. In our experiments, we forcefully stop the primary VM to emulate the occurrence of a primary failure. The restored VM continues operation in the restoration host with its memory contents being provided on-demand by the odpf memory page fetching daemon. The restored VM and page fetching daemon uses separate network interfaces in the restoration host.

We installed an Intel 80 GB X25M Mainstream SATA II MLC SSD in the VM checkpoint store to understand the behavior of the HydraVM system using a disk and a SSD-based checkpoint store. Note that in all experiments VM virtual disks were hosted in the VM disk server on a hard drive. Unless otherwise stated, the VM under test was configured with 512 MB of memory, one virtual CPU, and one virtual NIC. The VM is pinned to use one physical processor in its hosting machine, while Domain 0, in which HydraVM executes its protection and recovery tools, is pinned to use the other one.

We ran two workloads, respectively, in the VM under test for our evaluation. We compiled a linux kernel source tree (version 2.6.22.7) using the default configuration and the *bzImage* target. Kernel compilation exercises the virtual memory system, disk, and processor of the VM. We also ran a video transcoding workload using ffmpeg [15], which is an open-source project that provides a set of libraries and programs for streaming and converting multimedia contents. In our experiments, we used ffmpeg to convert a series of MPEG2 videos to AVI format. The total amount of video data being converted was 2.3 GB.

B. Storage-based VM Protection

HydraVM stores the fail-over image and periodic VM checkpoints in stable storage. We first evaluate the VM protection provided by HydraVM based on two types of storage devices, a hard drive and a SSD. We ran the two workloads in the primary VM, respectively. Checkpoints of the primary VM were taken periodically throughout the execution of the workloads, and we varied the time between successive checkpoints.

Table I summarizes the average size of the incremental checkpoints taken during the execution of the workloads, as well as the average time required to apply all dirty pages contained in a checkpoint to the fail-over image and commit the changes to the storage media under different checkpointing frequencies. When configured to take checkpoints every 10 seconds, HydraVM pauses the primary VM and starts a new checkpoint 10 seconds after the VM was paused when the previous checkpoint started. Video transcoding touches twice as many memory pages compared to kernel compilation in the 10-second checkpointing intervals, but it does not require a proportionally longer time to commit the checkpoints. This is because video transcoding touches memory mostly sequentially, resulting in series of sequential writes when updating the fail-over image, which can be

Kernel Compilation						
Configured Checkpointing Frequencies	Every 10 sec		Every 5 sec		Every 2 sec	
Checkpoint Storage	HD	SSD	HD	SSD	HD	SSD
Checkpoint Size (MB)	46.1	46.3	40.8	40.5	40.2	36.7
Checkpoint Commit Time (sec)	5.0	3.2	4.6	2.9	4.6	2.7
Actual Checkpoint Frequencies (sec)	10.0	10.0	5.2	5.0	5.0	3.0

Video Transcoding						
Configured Checkpointing Frequencies	Every 10 sec		Every 5 sec		Every 2 sec	
Checkpoint Storage	HD	SSD	HD	SSD	HD	SSD
Checkpoint Size (MB)	92.4	91.9	58.4	51.7	40.5	26.6
Checkpoint Commit Time (sec)	6.1	4.3	4.6	2.9	3.5	1.7
Actual Checkpoint Frequencies (sec)	10.0	10.0	5.8	5.0	3.9	2.2

Table I
THE AVERAGE SIZES OF INDIVIDUAL INCREMENTAL CHECKPOINTS AND AVERAGE TIME REQUIRED TO COMMIT EACH CHECKPOINT TO THE VM CHECKPOINT STORAGE BASED ON A HARD DISK (HD) OR A FLASH DEVICE (SSD).

performed more efficiently. For both workloads, it takes a shorter amount of time to commit checkpoints to a fail-over image hosted on a SSD than a hard drive, and each checkpoint finishes committing within the configured checkpointing interval.

A checkpoint can contain a larger amount of data than can be completely written to storage before the configured checkpointing interval ends. In this case, the next checkpoint is not started until the previous one has committed to storage fully, in order to ensure the consistency of the VM fail-over image at all time. This delay of successive checkpoints creates discrepancies between the *configured* and *actual* checkpointing frequencies. For example, when HydraVM is configured to take checkpoints every 5 seconds and uses a disk-based checkpoint store, some of the checkpoints were not able to finish committing within the 5-second interval, and the time elapsed between consecutive checkpoints are measured to be longer, as indicated by the actual checkpointing frequency in Table I. The configured every 5 seconds frequency was achieved on a SSD-based checkpoint store for both workloads, as I/Os are more efficient on a SSD. When configured to take checkpoints every 2 seconds, successive checkpoints are delayed even on a SSD-based system. However, the delays are shorter when using a SSD, and hence the smaller checkpoints generated during the shorter actual checkpointing intervals compared to the case of using a hard drive.

C. Overheads of VM Protection

When being protected, the primary VM is paused periodically for each incremental checkpoint to be taken, resulting in constant disruptions to VM operation. The checkpoint data are transferred to the checkpoint store when the primary resumes running, interfering with primary's normal operation. We therefore evaluate the costs of VM protection using VM pause time and workload overhead as two important metrics.

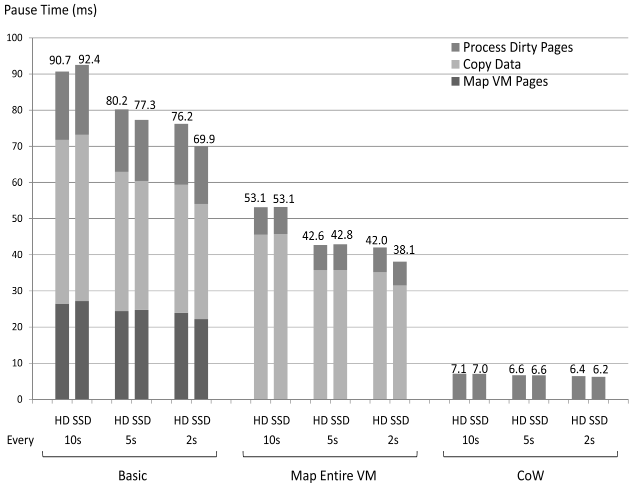
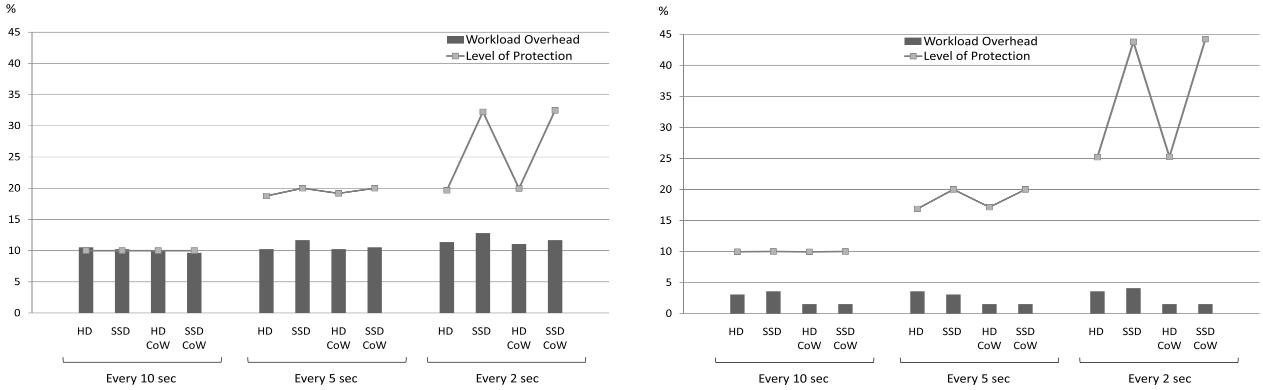


Figure 2. The average VM pause times for the kernel compilation workload resulted from different checkpointing techniques under different configured checkpointing frequencies.

We first present a detailed evaluation of VM pause time using the kernel compilation workload as an example. The total height of each bar in Figure 2 represents the average time during which the primary VM is paused for an incremental checkpoint to be taken (also indicated by the numbers above each bar.) We distinguish two major components of the VM pause time, the time for the `hacp` checkpointing daemon to gain access to the set of dirtied VM pages that composes the checkpoint (”Map VM Pages”), and the time to copy the contents of those pages to a buffer to be transferred after the VM resumes execution (”Copy Data”).

We compare the basic incremental checkpointing technique and the two enhancements of requesting access to the entire primary memory when the VM protection start and CoW checkpoints (described in Section III-B.) Mapping the entire primary VM at the beginning reduces the average VM



(a) Kernel Compilation

(b) Video Transcoding

Figure 3. The protection overheads on workload execution and the corresponding levels of protection the workload gains using the basic and CoW checkpointing techniques under different configured checkpointing frequencies.

pause time by 41% to 46% for the evaluated checkpointing frequencies, and together with CoW checkpointing, achieves a 92% of reduction on VM pause time.

For the first two checkpointing techniques, the VM pause times are largely in proportional to the checkpointing intervals; the larger an interval is, the more memory pages are dirtied during the interval and the longer it takes to map and copy those dirty pages, and hence the longer the VM pause time. When configured to take checkpoints every 2 seconds, the pause times observed on a disk and SSD-based system are different because the times actually elapsed between consecutive checkpoints are longer when using a hard drive, as discussed in Section **??**, resulting in more memory pages getting dirtied, and hence the longer pause time. When using the CoW technique, the VM pause times are flat across the checkpointing frequencies evaluated, since we only mark a bit per dirtied VM page in the CoW bitmap while the VM is paused, without actually mapping and copying any pages.

We evaluate the workload overheads by comparing the workload execution time measured in a periodically checkpointed VM with the baseline execution time in a VM that is not protected/checkpointed. Figure 3(a) and 3(b) show the results for kernel compilation and video transcoding, respectively. Our experiments compare the basic and CoW (included the enhancement of mapping the entire VM memory) techniques on a disk and a SSD-based system. The results show that it does not incur an undue overhead on workloads even when HydraVM is configured to take checkpoints as frequently as every 2 seconds. In all evaluated cases, kernel compilation runs less than 13% slower, and it takes no more than 4% longer time for video transcoding to finish.

When checkpoints are taken every 10 seconds, workloads incur similar overheads on a disk and a SSD-based system. When using CoW checkpointing, overheads for kernel compilation are lowered, slightly for kernel compilation, and more for video transcoding. CoW checkpointing duplicates only 48% of the checkpointed pages the basic technique copies throughout the execution of kernel compilation, and no more than 17% of the pages for video transcoding.

To further understand the workload overheads in different cases, let us consider the corresponding *level of protection* the workload gains. The primary VM may receive greater protection by being checkpointed more frequently, resulting in less loss of completed work in the event of a fail-over. In the baseline case when no checkpoints are taken, the primary gains 0% of protection. Let effective checkpointing per second be 100% of protection the primary may receive. The level of protection can be computed by dividing the total number of checkpoints actually taken by the workload execution time. For example, when configured to take checkpoints every 2 seconds for kernel compilation on a hard drive-based system, a total of 75 checkpoints were taken throughout the 392 seconds of workload execution. The level of protection for this experiment case is 75/392 = 19%.

Looking at the level of protection the VM actually received helps us understand the seemingly counter-intuitive situation when a SSD-based system incurs higher workload overheads than a hard drive. Since checkpoint committing is faster on a SSD, more checkpoints are taken throughout the workload execution, and hence the higher degree of protection achieved. Consequently, the primary VM is paused more frequently, and larger total amount of checkpoint data are transferred while the workload is running, resulting in a larger overhead.

In summary, experimental results show that protection overheads depend upon the workload characteristics, and are largely in proportional the actual level of protection the primary VM receives. In all our experiments, the `hacp` daemon uses less than 7% of CPU.

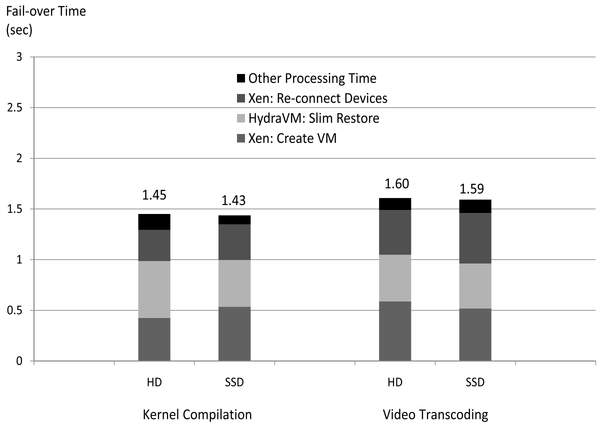


Figure 4. The average fail-over time required to bring up a failed primary VM in which a kernel compilation and video transcoding workload has completed 50% of its execution.

D. Restoration of a Failed VM

HydraVM quickly restores a backup VM from the fail-over image upon detection of a primary failure performing a slim VM restore. This section evaluates the effectiveness of this mechanism. We run the two workloads in the primary VM, respectively. The primary was checkpointed periodically. After completing about 50% of each workload (executing 200 seconds for kernel compilation and 110 seconds for video transcoding) and finishing the last incremental checkpoint in this time period, we forcefully stop the primary VM, and launch the slim VM restore to bring up a backup VM in the restoration host.

Both workloads resumed execution correctly in the restored VM from the most recent checkpointed state in the VM checkpoint store after a brief pause, during which fail-over is performed. Figure 4 shows our measurements of the fail-over time required. The numbers shown are average results over three experimental runs. The results demonstrate that HydraVM is capable of restoring a failed VM very promptly within 1.6 seconds of time, which is usually acceptable to human users.

We observe that restoring a VM from a SSD-hosted fail-over image does not show significant performance improvements. Therefore, we further break the fail-over time into three major components, and find that our implementation of slim VM restore is efficient and only composes about one-third of fail-over time. Over 60% of time are spent by the recycled Xen code to create VM container and await for virtual devices reconnecting to the restored VM.

Page table pages form the majority of VM data being loaded from storage during slim VM restore. In our experiments, an average of 1033 and 1135 page table pages (about 4 MB of data out of the 512 MB VM memory space) are loaded during restore. They are loaded more quickly on SSD by an average of 0.04 seconds for the two workloads. Due to the small amount of data being loaded from storage and the small percentage of influence storage devices have on total fail-over time, a hard drive and a SSD-based shared storage performs almost equally well on failed VM restoration.

Kernel Compilation		
Checkpoint Storage	HD	SSD
Memory Contents Fetched (MB)	128	
Workload Overhead (%)	8	8

Video Transcoding		
Checkpoint Storage	HD	SSD
Memory Contents Fetched (MB)	454	
Workload Overhead (%)	11	11

Table II
THE OVERHEAD INCURRED WHILE FINISHING THE LAST 50% OF EXECUTION FOR KERNEL COMPILATION AND VIDEO TRANSCODING BECAUSE OF FETCHING VM PAGES ON-DEMAND AND THE AMOUNT OF MEMORY DATA FETCHED.

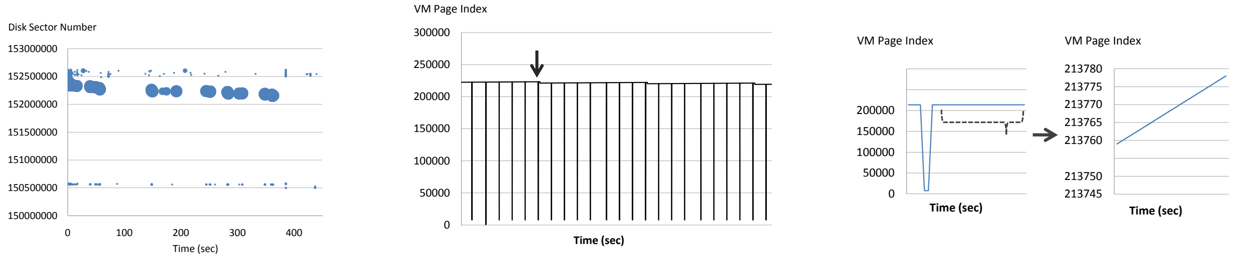
E. Operation of a Restored VM

Immediately after restored, the missing memory pages are fetched from the fail-over image on the shared storage when needed. To understand the impacts of fetching memory contents on-demand, we compare the time required to finish executing the last 50% of the workloads described in the previous subsection with the remainder execution time measured in a VM the memory space of which is fully populated.

The workload overheads are summarized in Table II. The results show that fetching memory pages on-demand as the VM executes does not introduce a prohibitive overhead, and therefore, this mechanism is practically deployable to enable slim VM restore, switching on a restored backup VM immediately after loading only critical information from the fail-over image, minimizing the fail-over time required.

Our experimental results do not show performance benefits when fetching from a fail-over image hosted on a SSD-based shared storage. This is surprising, since we expected that on-demand fetching of memory pages would generate many small random reads from the fail-over image, which are much faster on SSDs. To further understand the VM page fetching behavior, we conducted a separate set of experiments. We execute the entire kernel compilation workload in a VM brought up by slim restore. As the workload executes, we record the indexes (pseudo-physical frame numbers) of the memory pages requested by the VM. In addition, we record using the *blktrace* [16] tool the I/O activities that actually happen on the storage device in the shared storage to service the VM page fetching requests from the fail-over image.

Figure 5(b) shows the pattern of VM page requests This page fetching pattern repeats throughout the execution of



(a) The I/O activities servicing the page fetch requests.

(b) The page fetching pattern that repeats for the entire benchmark.

(c) A zoomed-in view of the page fetching pattern.

Figure 5. The pattern of the page fetches requested by the backup VM recorded for the kernel compilation benchmark and a zoomed-in view of the pattern.

kernel compilation. (The complete pattern not shown for clarity.) As can be observed in the zoomed-in view of this pattern, illustrated in Figure 5(c), page accesses made by the VM show substantial spatial locality. Most of the requests fetch the VM page next to the one fetched in the previous request, which only requires reading an adjacent block in the VM fail-over image on the shared storage.

Figure 5(a) shows the I/O activities on the block device servicing the recorded page fetch requests for the entire execution of the kernel compilation benchmark. The disk access pattern also shows substantial spatial locality. Although over 30,000 4K memory pages were requested by the backup VM during the compilation process (about 128 MB of memory contents loaded, as reported in Table II), only about 3,000 I/O requests were sent to the block device in the shared storage, as reported by blktrace. Each bubble in Figure 5(a) represents an I/O request recorded, and the size of the bubble represents the size of the request. We found that instead of a series of 4K block reads, many of the I/O activities fetched 512 disk sectors (256 KB of data, 64 4K blocks) in one request, as shown by the large bubbles in the figure. This observation leads to our finding that optimizations built in the filesystem (e.g. file prefetching) are effective in improving hard drive performance to support the demand page fetching mechanism in HydraVM, bridging the performance gap between mechanical disks and flash storage devices.

F. A Comparative Perspective

We conclude our evaluation by comparing HydraVM with alternative HA solutions for VMs.

Popular hypervisors usually provide the functionalities to create complete VM checkpoints as regular files on persistent storage, and start a VM from its image file. These can be used as the simplest techniques to provide high availability. For example, on a Xen platform, “VM save” takes a full checkpoint of a VM. When a failure occurs, “VM restore” can be used to load the checkpoint from disk and creates a backup VM to take over. The first set of bars in Figure 6 shows that it can take as long as 35 seconds to take *each* full checkpoint of a 1G VM and send it to a shared storage, and up to 20 seconds to bring such a VM back to operation from the checkpointed image using the VM save and restore approach.

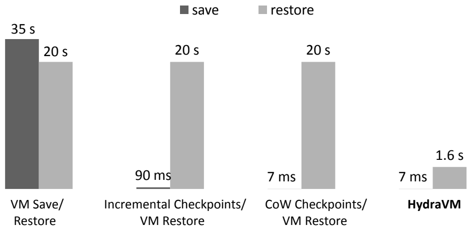


Figure 6. A comparative perspective on the overall contribution of HydraVM.

Although VM save/restore does not reserve memory for the backup until a fail-over is required, the long time delays makes it practically unuseful. Instead of saving complete VM state, incremental checkpointing captures only the changed state from the last checkpoint. The “save” bar of the second group shows that the time during which the VM is interrupted because of checkpointing is greatly reduced to milli-second scale. The amount of data that needs to be transferred for each checkpoint is reduced from the entire VM (1 GB) to several MB of dirty pages, depending on the workloads running in the VM. HydraVM maintains the complete VM image in a shared storage, while other incremental checkpointing approaches, such as Remus and

Kemari, reserve in another host as much memory as the primary to store the VM image.

Interruption of the primary VM is reduced further using CoW checkpointing, as shown in the third group of the figure. However, a 20-second fail-over time, resulted from loading the entire VM image for restore, is prohibitively expensive. The last set of the bars in the figure shows that by integrating slim VM restore and fetching memory pages on-demand as the VM executes, the fail-over time is significantly reduced. The results in this group differentiate HydraVM from other similar approaches: without reserving additional memory, HydraVM take continuous checkpoints to protect a VM with minimal interruption of the VM, and restores a VM within 1.6 seconds in the event of a failure, achieving the goals of low-overhead checkpointing and responsive fail-over.

VI. Related Work

To the best of our knowledge, this is the first attempt to provide high availability in virtualized environments that takes into account both recovering a VM from its recent execution state and reducing the computing resources committed to the protection of the VM. Our approach focused on elimination of the passive reservation of main memory for the backup VMs set up for high availability.

Hypervisor vendors offer high-availability solutions that rely on restarting the failed VM in another node [1]. Unlike HydraVM, these solutions simply reboot the failed VM from its disk image. Although they do not require additional resources until a failure occurs, they do not recover the application state, and take significantly longer than HydraVM to perform a fail-over.

Various other hypervisor-based approaches have been proposed to keep track of VM execution state and perform a stateful fail-over, minimizing the work lost in the event of a failure. Bressoud and Schneider [5] proposed lock-step execution of a backup VM. In their approach, a hypervisor intercepts the instructions and interrupts executed on the primary VM, and sends them for a lock-step replay on the backup VM. VMware Fault-Tolerance [10] has recently been offered as a commercial product based on a similar idea of letting two VMs execute an identical sequence of instructions so that one may take over the other if it fails. These approaches require the participating VMs to have identical memory configurations to execute alongside, and therefore, are not resource-efficient.

Remus [4] and Kemari [9] continuously take incremental checkpoints of a protected VM to record its execution state. Remus checkpoints the primary VM at fixed, sub-second intervals, while Kemari takes a checkpoint whenever the primary VM is about to interact with external devices, such as disk and network interfaces. These approaches maintain a backup VM in RAM and keep the backup VM synchronized with its primary VM by continuously updating the backup VM image kept in memory with the incremental checkpoints taken at the primary. Throughout the protected execution of the primary, additional memory is reserved for its backup VM, even though the backup is normally passive (not operating). Our approach adapts a periodic, incremental checkpointing technique that is most similar to Remus'. However, we trade off main memory with shared storage to store the backup VM image, resulting in better cost- and resource-efficiency.

The concept of copy-on-write checkpoints was also discussed in Remus, and we are aware of multiple implementations of this well-known systems technique. Colp *et al.* developed VM Snapshots [17] which provides a set of API's for fast snapshoting of a VM. Different from our approach, their implementation uses FUSE (Filesystem in Userspace) [18] driver support, and involves modifications to the guest kernel. Sun *et al.* [19] also implemented copy-on-write for lightweight VM checkpointing. However, their system currently focuses on taking complete snapshots of a VM, instead of incremental checkpoints, which are required in the context of VM high availability.

Our approach to rapid VM fail-over using on-demand fetching of memory pages draws on the work on post-copy-based VM migration [20] and SnowFlock [21]. Post-copy VM migration is an alternative approach to live VM migration [12]. Without copying the entire VM image over to the target node, it resumes VM execution at the target right after shipping the processor state from the source VM, and as the VM runs in the target node, memory pages are fetched/pushed from the source VM to populate the memory space of the target VM. SnowFlock is a system that implements "VM fork" similar to "process fork". SnowFlock creates VM replicas (child VMs) rapidly by shipping a condensed (parent) VM image, called a *VM descriptor*, to the target hosts and instantiating multiple child VMs based on the descriptor. As the child VMs execute, they fetch memory contents from the parent VM on-demand. Our approach applies a similar concept to address a completely different problem in the context of high VM availability; we load minimum VM state from the fail-over VM image in the shared storage into server memory to quickly restore a backup VM, minimizing the fail-over time required, and, similarly, supply memory pages for the backup VM on-demand.

Our approach is very effective in reducing the excessive cost of computing resources committed to providing high VM availability. While other memory-saving techniques such as compression [22], differencing [23], and page caching can be used to reduce the memory pressure caused by the creation of backup VMs, none of these techniques will completely eliminate it.

VII. Conclusions

In this paper, we proposed HydraVM, a storage-based, memory-efficient way of achieving high availability in virtualized environments. Unlike current VM fail-over approaches, which require twice the memory the VM uses, our approach requires minimal extra memory. HydraVM stores VM fail-over images in a shared storage, and can promptly restore a failed VM on any host that has access to the shared storage, which allows any host with available capacity to be used as the backup.

We adapt a continuous, incremental checkpointing technique to track the state of the protected VMs and keep their fail-over images in the shared storage recent. We implement copy-on-write checkpoints to further reduce the checkpointing overhead. Our experimental evaluation demonstrates the effectiveness of the proposed techniques. In the event of a fail-over, our slim VM restore mechanism brings a backup VM back to operation within 1.6 seconds, by loading only the critical VM state data from the VM image kept in the shared storage. Execution of the backup VM is resumed promptly, and proceeds by fetching memory contents from the VM image as needed. Experimental results also show that our methods do not introduce excessive overhead on the protected primary VM during normal execution, or on the backup VM after restored. Using HydraVM, a VM can be restored quickly enough to meet the requirements of most users and applications at a minimal cost in resources.

References

[1] "VMware Inc. VMware high availabiltiy (HA)," http://www.vmware.com/products/vi/vc/ha.html, 2007.

[2] F. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[3] N. Budhiraja, K. Marzullo, F.B.Schneider, and S. Toueg, "The primary-backup approach," 1993.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchisonson, and A. Warfield, "Remus: High-availability via asynchronous virtual machine replication," in *Proceedings of the 5th conference on Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008, pp. 161–174.

[5] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 80–107, 1996.

[6] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002, pp. 181–194.

[7] A. Burtsev, M. Hibler, and J. Lepreau, "Aggressive server consolidation through pageable virtual machines," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (Poster Session)*, 2008.

[8] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *USENIX Annual Technical Conference*, Jun. 2010.

[9] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual machine synchronization for fault tolerance," in *Proceedings of the USENIX Annual Technical Conference 2008 (Poster Session)*, Jun. 2008.

[10] "VMware Inc. VMware fault tolerance (FT)," http://www.vmware.com/products/fault-tolerance/, 2008.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machine," in *Proceedings of the 3rd conference on Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[13] "LVM2 resource page." http://sourceware.org/lvm2/.

[14] D. Chisnall, *The Definite Guide to the Xen Hypervisor*. Prentice Hall Press, 2007.

[15] "FFmpeg," http://www.ffmpeg.org/, Retrieved September 2010.

[16] A. D. Brunelle, "blktrace user guide," http://www.cse.unsw.edu.au/ aaronc/iosched/doc/blktrace.html, 2007, retrieved September 2010.

[17] P. Colp, C. Matthews, B. Aiello, and A. Warfield, "Vm snapshots," in *Xen Summit*, Feb. 2009.

[18] "FUSE filesystem in userspace," http://fuse.sourceforge.net/, Retrieved September 2010.

[19] M. H. S. D. M. Blough, "Fast, lightweight virtual machine checkpointing," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-10-05, 2010.

[20] M. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, Mar. 2009.

[21] H. A. Lagar-Cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the ACMSIGOPS/EuroSys European Conference on Computer Sysmtems 2009*, Apr. 2009.

[22] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, May 2005.

[23] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008, pp. 309–322.