STATEFUL-SWAPPING IN THE EMULAB NETWORK TESTBED

by

Prashanth Radhakrishnan

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2008

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Prashanth Radhakrishnan

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

______________________ ______________________________________

Chair: Jay Lepreau

______________________ ______________________________________

John B. Carter

______________________ ______________________________________

John Regehr

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of _______Prashanth Radhakrishnan_______ in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

________________________ ____________________________________

Date

Jay Lepreau
Chair, Supervisory Committee

Approved for the Major Department

____

Martin Berzins
Chair/Dean

Approved for the Graduate Council

____

Robert Brown
Dean of The Graduate School

ABSTRACT

Time-sharing is a useful feature in heavily utilized public network testbeds. Emulab, the popular network research testbed, supports a primitive form of time-sharing by *swapping-out* idle experiments to free up physical resources. However, the swap-out operation destroys experiment runtime state and experimenters have to manually recreate the state upon a subsequent swap-in—a tedious and sometimes impossible proposition.

For a testbed to realize the full benefits of time-sharing, the *experiment swapping* operation should be *stateful*, *transparent* and *fast*. Towards this end, we are extending Emulab with a *stateful-swapping* facility.

We address two key challenges in building this system involving state management and providing application transparency: how to efficiently handle experiment runtime state of several gigabytes so that time-sharing can be fast and how to prevent experiments from perceiving periods of swap-out.

Our approach leverages the advances in Virtual Machine Monitors (VMM) and Storage Virtualization. We address state-related challenges by applying *caching*, *copy-on-write*, *pipelining* and *gray-box* techniques. To prevent the perception of swap-out periods, we virtualize an experiment's notion of time.

This thesis presents the design, an initial implementation and evaluation of the stateful-swapping infrastructure.

For ananda mama

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Jay Lepreau, for letting me work on this project, supporting me for the most part of my masters and, often times, believing in me more than even I did. I would also like to thank my committee, John Carter and John Regehr, for their feedback and patience.

Many thanks to Mike Hibler for his useful suggestions and for reviewing my proposal document. I am very grateful to Anton Burtsev for his collaboration and company during the past two years. Thanks to all the other members of the Flux group for putting up with me.

I would like to thank my parents for their unwavering support and motivation that has kept me from losing it totally. I am eternally grateful to Ananda mama for everything—the void you've left will always remain.

CHAPTER 1

INTRODUCTION

Public network testbeds are heavily used by networking and distributed systems researchers. The demand for the testbed resources usually exceeds capacity. In order to help alleviate this crunch, testbeds resort to techniques that better utilize the physical resources. Some testbeds, such as PlanetLab [31], multiplex resources *spatially* by running several virtual machines on a physical machine. Another approach is to multiplex *temporally* by time-sharing experiments on physical resources. Emulab [50], supports a weak form of time-sharing by "swapping-out" idle experiments.

An Emulab experiment definition—its initial state and network topology—is analogous to a Unix program. A running experiment is analogous to a process, and the persistent store on the file server is analogous to an on-disk filesystem. However, unlike an operating system, a swapped-out experiment loses its runtime state. When the experiment is swapped-in later, the network topology and the disk states are reinitialized. If experiment swapping were analogous to process swapping, then the Emulab scheduler could flexibly preempt low-priority or idle experiments and schedule them later on potentially different physical resources—all this without disrupting the user applications running in the experiments.

Our goal is to extend Emulab with an experiment swapping capability that is *stateful*, *transparent* and *fast*.

This **stateful swapping** facility should preserve both the persistent and volatile runtime state of experiments. Thus, the state that needs to be saved includes node-local memory and disk as well as in-flight network packets.

For stateful swapping to be truly beneficial, it has to be fast—in the order of tens of seconds. From the perspective of the user, this manifests as shorter swapin latencies (i.e., the delay between clicking on the "swapin" button and resuming work on the swapped-in experiment nodes). From the perspective of the Emulab system, faster stateful swapping improves node utilization and enables fine-grained preemptive scheduling policies for experiments. However, the collective runtime state of an experiment, mainly node-local disk plus memory, is typically in the order of tens of gigabytes and naive approaches can lead to several minutes of transfer time. Thus, one of our key challenges is to design efficient state management techniques that can make stateful swapping fast.

Transparency across swapout is another requirement. By transparency, we refer to the guarantee that an application continues to run across swapouts, without behaving erratically. Given that the volatile state is preserved, there are two factors that can affect transparency—external world interactions and perception of current time. External world interactions, especially those that are stateful in nature, will be affected by stateful swapping unless the interaction is specifically designed to work across swapouts. Furthermore, applications would typically see a sudden jump in time, when the experiment is swapped-in after a period of inactivity. We believe that perception of this period of inactivity can have adverse effects on applications. Thus, in order to provide transparency, we need to address the challenges posed by external world interactions and applications' perception of time.

1.1 Focus

In summary, the focus of this thesis is on the two aspects of the stateful swapping system addressed by the following questions:

- How fast can we save and restore experiment runtime state?
- How can we prevent applications and operating systems from perceiving swapouts?

1.2 Approach

Our basic approach is to leverage Virtual Machine Monitors (VMM) for their capability to save and restore node-local volatile state. So, all stateful-swapping experiment nodes are run inside virtual machines (VM).

Towards making stateful-swapping fast, we explore the following techniques:

- *Copy-on-write* for efficient node-local storage.
- *Caching* of golden disk images for faster experiment swapin.
- *Pipelining* data transfer with experiment execution to achieve bounded *experiment swapping* latencies.
- Application of *gray-box* [3] knowledge of filesystem at block layer to achieve better data compression.

In order to make swapout time periods transparent, we implement a mechanism for running experiment nodes in virtual time. We handle the external world interaction problem by leveraging our observation that most Emulab experiments are “closed.” Thus, we only need to support certain well-known external world interactions.

1.3 Transparency vs. Fidelity

While we strive for transparency across swapouts, it is, however, possible that our approaches to efficient stateful-swapping—like the use of COW storage or VMs—may impact application *fidelity*. In order to to differentiate our notions of application transparency and fidelity, consider the following example: some random disk reads taking a fraction of a second longer than the others result in loss of fidelity, but not transparency; whereas the system time jumping ahead by 12 hours, because of an intervening stateful swapout, can result in loss of transparency as well as fidelity.

We have a basic tradeoff between speed and fidelity—for example, COW storage has its own associated costs in terms of read/write penalties and memory consumption; pipelined (or background) data transfer can also affect applications running inside a VM. We believe that the right point in the tradeoff spectrum heavily depends on type of workload run in an experiment. For example, a network intensive experiment which writes to the disk purely for diagnostic purposes, may be able to get the best of both worlds. However, a predominantly disk intensive application may have to choose the fidelity end of the spectrum and sacrifice speed.

1.4 Design Principles

Throughout this work, we have followed three guilding principles. Firsly, we have favored tailoring stable open-source systems over developing systems from scratch. As a result, our design had to operate within the framework of those systems—a reasonable price to pay, we believe, to avoid “re-inventing the wheel.” Secondly, we attempt to keep our implementation as guest OS agnostic as possible. Thus, a single implementation can be reused with multiple operating systems. Finally, we have designed most features described in this thesis to be selectively configurable. Thus, experimenters can create a stateful swapping experiment according to their requirements, which may be based on I/O workloads, usage pattern etc.—the fundamental tradeoff being, faster swapping with more overheads vs. slower swapping with less overheads.

1.5 Contributions

The contributions of this work are mostly software artifacts, which are usable extensions to the systems we built on. The following are the major ones:

- A general-purpose block layer branching storage system for Linux.

- An online free-block detection framework, which is useful in VM-based environments for *filesystem-aware* compression of VM disks and for mitigating the performance impact of copy-on-write in block-based storage systems.
- An infrastructure for Xen virtual machines to operate in a virtual world with certain structured time revealing interactions.

1.6 Scope

The scope of this thesis is largely limited to the client-side of the stateful-swapping infrastructure. That is, we deal with how experiment nodes are managed and how state is transferred to and from experiment nodes. However, we do not discuss the design of the Emulab server that is tailored for stateful-swapping. Specifically, we do not address issues related to server scalability and fault-tolerance. Nevertheless, the client-side infrastructure has been designed with the intention of supporting a scalable server with minimal changes.

Further, the management of stateful-swapping experiments has not yet been automated. Much of the automation effort requires Emulab server modifications—a task best left to Emulab experts.

1.7 Document Roadmap

Chapter 2 provides some background on the systems used in this work, namely, Emulab, Xen VMM and Linux Logical Volume Manager (LVM). Chapter 3 discusses various aspects of our design and concludes with a summary of how stateful-swapping works in the prototype that we have implemented. The implementation issues are outlined in Chapter 4. A preliminary evaluation of the stateful-swapping system is presented in Chapter 5. We discuss related work in Chapter 6 and conclude in Chapter 7.

CHAPTER 2

BACKGROUND

This section presents a background on Emulab, the Xen Virtual Machine Monitor and the Linux Logical Volume Manager (LVM). The stateful-swapping system is targeted for use in the Emulab network testbed. The Xen VMM and LVM are used for managing the node-local volatile and persistent state, respectively.

2.1 Emulab

Emulab is a general purpose testbed facility and "operating system" for networked and distributed system experimentation. It has been actively used by the research and education communities since April 2000. Emulab provides integrated, Web-based access to a wide range of environments including simulated, emulated, and wide-area network resources. However, for the purposes of this thesis, we restrict ourselves to the local area emulation environment that consists of a cluster of commodity PC nodes with configurable network interconnectivity.

An *experiment* is specified by an *ns* script, which describes the experiment's per-node configuration and network topology, besides several other things. When a user swaps in an experiment, the specified network topology is mapped onto physical resources like hosts, network links, delay nodes, etc. Then, Frisbee [21] is used to load experiment nodes with the disk images specified in the ns file.

When an experiment is deemed *idle* by the Emulab system, it is swapped out. This frees up the physical resources allocated to the experiment and destroys all node-local experiment runtime state. However, the experiment configuration persists. The swap-out operation may also be triggered by users.

2.2 Xen Virtual Machine Monitor

Xen [5] is an open source virtual machine monitor for the x86 architecture. Xen consists of the following components.

- A minimal microkernel, called the *hypervisor*, which presents a virtual hardware abstraction that is similar, but not identical, to the underlying hardware—an approach called *paravirtualization*. The Xen hypervisor is also responsible for isolation of VMs and other basic services like VM scheduling, inter-VM communication, etc.
- A privileged management VM (called *domain-0*) for creating and manipulating other VMs. It runs a modified version of Linux. The domain-0 orchestrates VM *save/restore* and *live-migration* [9] across physical machines.
- One or more unprivileged user VMs (called *domain-U*) that run operating systems ported to the Xen hypervisor—i.e., *paravirtualized*.
- Optionally, one or more privileged VMs that host physical device drivers. These VMs authorize and multiplex user VMs' access to hardware devices [24]. Typically, the physical device drivers are hosted in domain-0 itself. We assume this to be the case in the rest of our discussion.

More recently, Xen also supports a *hardware-assisted* virtualization mode, which allows unmodified guest operating systems to be run on Xen. However, that requires processor support for virtualization [45], currently unavailable in Emulab. Furthermore, Xen's support hardware-assisted virtualization mode is still in its early stages of development.

In this work, we have used paravirtualized Xen with user VMs running the Linux OS. But, we have ensured that our design is independent of both the OS running inside user VMs and the mode of virtualization used by Xen. Thus, once machines with virtualization support are available in Emulab, we expect that our implementation could be used on Xen in hardware-assisted mode to statefully swap unmodified proprietary operating systems like Windows.

2.3 Linux Logical Volume Manager

Volume Management creates a layer of abstraction over physical storage. It exports a *volume*, which the filesystem (or any other application) uses like a normal disk partition. Volumes may span several disk partitions and may implement different levels of RAID [30] or even copy-on-write functionality. The latest Logical Volume Manager for Linux is called LVM2 [36], which we henceforth refer to as simply LVM.

LVM has three key entities (Figure 2.1): *physical volume*, *volume group* and *logical volume*. A physical volume is typically a hard disk, or a hard disk partition that has been formatted by LVM. A volume group is made out of one or more physical volumes. Logical volumes are created in a volume group and are allocated space from physical volumes (in the volume group).

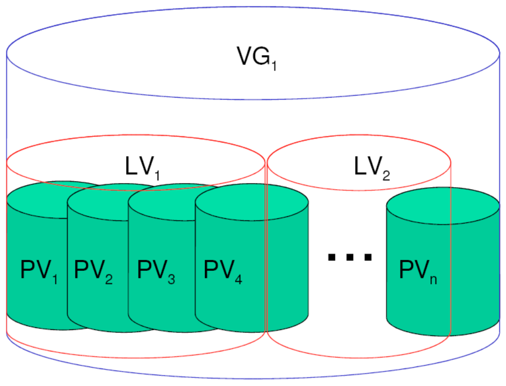

Figure 2.1. Basic LVM Entities (from [20])

A volume group is analogous to the disk, while a logical volume is analogous to the disk partition.

LVM includes a user-level application that manages the ondisk metadata of the LVM entities and communicates with the Device Mapper kernel driver to maintain a kernel-resident block device for each of its logical volumes. The Device Mapper [35] provides a generic framework for volume management. It is used to construct new block devices that map I/O to underlying block devices according to a mapping table. The mapping table specifies how to map ranges of logical volume sectors onto a target device, using one of the supported mapping types. The device mapper currently supports the following mapping types: *linear*, *error*, *striped*, *crypt*, *snapshot-origin*, *snapshot* and *mirror*. Of these, the linear, snapshot, snapshot-origin and mirror mapping types are of interest to us.

A linear volume corresponds to a contiguous region on disk. A snapshot volume corresponds to a point-in-time copy of a linear volume. Unlike the common notion of snapshots as read-only copies, LVM snapshots can be modified independent of the original linear volume. The snapshot-origin is not exported as a volume to user applications. It is a mapping type that is overlaid on linear volumes when they have snapshots attached to them and implements the copy-on-write functionality for writes to the linear volumes. A mirrored volume is made up of two or more linear volumes that replicate the same data. The internals of the snapshot and mirrored volumes are described in Appendix A.

CHAPTER 3

SYSTEM DESIGN

3.1 State Capture

In this chapter, we describe the design choices involved in capturing a stateful-swapping experiment's runtime state.

An experiment's runtime state consists of *persistent* state on the node-local disks, and *volatile* state in per-node processors, memories and the network. We require special means to capture the volatile state.

3.1.1 Node-Local Volatile State

There are different ways to capture an experiment's node-local volatile state, depending on the granularity of the state captured (as a set of processes or the entire system) and the layer at which it is implemented (application-level, OS-level, VMM-level or hardware-based). The following are our requirements for a per-node state capturing system.

- It should capture the state of the entire OS, including all processes. For example, users may experiment with kernel modules.
- It should not require modifications to user applications.
- It should have minimal impact on application performance.
- It should be available for use on commodity hardware and preferrably for the multiple operating systems (such as Linux, FreeBSD, Windows) that Emulab supports.
- It should be hardware agnostic to some extent—for example, independent of devices attached, processor speeds or memory capacities, but possibly specific to a processor architecture (x86). The motivation for this requirement stems from the fact that Emulab has distinct classes of x86-based machines (like pc850, pc3000) and we would like state saved from one class of machines to be restorable on another.

Based on the above requirements, we can eliminate many of the alternatives. Language-based systems are not applicable because they require changes to the user applications or at least require access to application source code. Process checkpointing systems do not capture the OS state.

Implementations that require special hardware are expensive. An OS-level implementation is also not favorable unless it is available on all operating systems. The need for near-native performance rules out simulation-based environments.

Thus, the VMM (Virtual Machine Monitor) appears to be the best layer at which state capture should be done. We use Xen [5], the popular open source hypervisor with a wide developer- and user-base. We leverage Xen's virtual machine (VM) *save* and *restore* capability. The save operation suspends a running VM and dumps its execution context (including virtual processor state and memory) to a file. The restore operation takes a memory dump as input and creates an active VM that resumes execution from the point where it was stopped at save.

We note that the use of VMs also provides the additional benefit of multiplexing multiple machines on a single physical machine. However, considering Xen's high costs in terms of host resources and Emulab's current machine capacities,[1] the degree of multiplexing may not be high—possibly 3-4. We realize that if multiplexing, rather than stateful-swapping, were the end goal, then lightweight virtualization systems, such as Linux *VServer* [17], may be more appropriate.

3.1.2 Node-Local Persistent State

While we do not need special techniques to capture node-local disk state, an important design choice is whether the disk storage for VMs should be local or remote.

Disk states are enormous. By storing disks on the server and having the VMs access their disks over the network, we can eliminate the need for disk image transfers. This leads to shorter stateful-swapping times. However, remote access comes at a cost, in terms of its impact on application fidelity and network and server resources. We believe that these costs of remote access outweigh the benefits. Thus, we use local storage for VMs and employ techniques that minimize the disk state transferred over the network.

3.1.3 Network State

The experiment network state includes the packets that are in-flight when the experiment is swapped-out. In order to ensure application transparency, these packets should not be lost. Thus, we log these packets at the receiver and requeue them in the receiver's receive buffer immediately upon a swap-in. We describe its implementation in section 4.1.2.

[1]Currently, the most powerful test nodes in Emulab have the following configuration: 3.0 GHz processor, 2GB RAM and 2 x 146 GB SCSI disks

3.1.4 Consistency of Distributed State

As part of stateful-swapping, we are checkpointing a distributed system. It is important to ensure that such a checkpoint is consistent. A checkpoint is inconsistent if a node reflects the receive event of a message, whereas the corresponding sender does not reflect the send event. We note that this situation cannot arise in stateful-swapping because, by design, the execution of all nodes in an experiment are suspended before any of them are resumed again. In other words, an experiment's swap-out operation always proceeds to completion before its subsequent swap-in is started. Thus, we do not require the commonly applied technique of tagging messages with Lamport logical clocks [26] for achieving a consistent distributed checkpoint.

3.2 State Retention

During the period between a swap-out and a subsequent swap-in, an experiment's runtime state needs to be stored somewhere. The two possibilities are Emulab repository and experiment physical machines (specifically, the ones that last hosted the experiment).

State retention on experiment machines will result in lesser data transfer over the network and thus, faster stateful-swapping. However, it is in conflict with Emulab's security model, where the users are given root privileges to experiment machines. This allows malicious users to tamper with another experiment's runtime state. One solution to this problem is to restrict user privileges to root on VMs (rather than the physical machines). However, to ensure continued support for traditional Emulab experiments, we will need to restrict these experiments from being hosted on the machines that also host stateful-swapping experiments. This segmentation can lead to reduced node utilization and/or added complexity. Furthermore, such a design will require extensive changes to Emulab access control and security model.

We took the simpler approach of storing the experiment runtime state in the Emulab repository. This effectively retains Emulab's notion of nodes as "containers for swapped-in experiments that are otherwise stateless."

3.3 Disk Delta and Golden Image Caching

We refer to disk changes made between a swap-in and the subsequent swap-out as disk *delta*. We anticipate that the disk delta will be much smaller in comparison to the entire disk. Thus, transferring disk deltas instead of whole-disk images should yield considerable time and bandwidth savings during a stateful swap-out.

3.3.1 Disk Delta Computation

We need to compute the disk delta at the time of a stateful swap-out. The two possible approaches to computing disk delta are identifying the changes *offline* (using content-hashes) or *online* (with dirty bitmaps). The offline approach results in longer swap-out times, whereas the online can potentially incur runtime penalty. In order to keep swap-out times short, we chose to explore the online approach.

3.3.2 Aggregated Delta

Most Emulab experiments use a few popular "golden" disk images. We store VM disk states as *aggregated deltas* that contain all changes made to the golden images. Our intuition is that these aggregated deltas should be considerably smaller than the whole disk image, because experimenters typically do not make pervasive changes to the base image (for example, most of the basic system binaries and applications remain unchanged).

We maintain an aggregated delta for each node in a stateful-swapping experiment. During a swap-in, the per-node aggregated delta is restored on the experiment nodes. During a subsequent swap-out, the disk delta accumulated over the run is saved to the Emulab repository. Before the next swap-in, this swapped-out delta is merged with the aggregated delta. The aggregation happens offline, on the Emulab server.

3.3.3 Caching

We expect the common-case to be the situation where the majority of the stateful-swapping experiments' disk states are contained in a few golden images. By proactively caching these images on experiment nodes, we will only need to transfer the aggregated deltas during a stateful swap-in. This should lead to substantial improvements in stateful swap-in latencies. If the aggregated deltas get as large as the original image, then we may revert to storing entire images.

Note that caching of golden images on experiment nodes does not conflict with Emulab's security model referred to earlier in section 3.2. This is because we do not rely on the presence of the cached image for correctness and we will ensure its validity before use. Ensuring the validity of the cached image also adds to the swap-in latency. Alternatively, the golden images could be preloaded on experiment nodes when experiment nodes are freed. However, this represents a trade-off between node availability and swap-in latency because the preloading activity may keep the nodes unavailable for a longer time. Preloading may make sense when the testbed is not heavily loaded.

In the worst case, where there is no cached image or the cached image is invalid, we will still be able to leverage the multicast capability of Emulab's default image-loader, Frisbee [21], to download the "popular golden image." Multicasting will not be applicable for loading customized disk images.

3.4 Branching

During a stateful swap-in, the intuitive approach to create a VM disk is to overlay the aggregated delta on the golden image. However, this has two important disadvantages. First, it mutates the cached golden image, thus necessitating a disk image reload at every swap-in. Second, when VMs, multiplexed on a physical host, originate from the same golden image—not an uncommon scenario—the overlay approach incurs additional time (for creating a new golden image for each VM) and space (because of duplication of unmodified data in the golden image) costs.

This indicates that we will benefit from a stackable storage system with support for copy-on-write (COW) based *branching*.[2] The COW functionality ensures that there is just one physical copy for unmodified data blocks between the golden image and the aggregated deltas. Moreover, it keeps the original golden image from being mutated. In this approach, we create per-VM branches over the golden image, overlay the aggregated deltas on the branch and use these branches as VM disks.

Furthermore, note that a recursive branching capability will be a straightforward way to compute the disk delta that should be swapped-out. That is, instead of using the branch containing the aggregated delta as the VM disk, we create a second-level branch and use that. As a result, all new writes made during the experiment's run between a swap-in and a swap-out, are saved in the second-level branch. One level of recursion suffices, so the potential performance impact should be minimal.

3.4.1 Design Choices

Implementing a branching storage system poses several design choices in the form of the following questions:

- *Where is the functionality implemented?*
    - Guest OS (inside VM) or VMM (outside VM)
    - Filesystem-level or Block-level

[2]Branch refers to a fork created off a point-in-time copy that can be updated independent of its source. In our case, the different aggregated deltas are branches created off the same golden image.

- *How is data indexed?*
  - By location or content-hash
- *How are writes handled?*
  - Undo-log, Redo-log or Write-Anywhere

3.4.2 Layer

Since we wanted stateful-swapping to be guest OS agnostic, a VMM-level implementation was our unambiguous choice.

Xen VMM virtualizes VM disks at the block layer, transparent to the filesystems inside the VMs. Thus, it is intuitive and more efficient to implement VM disk branching at the VMM block layer rather than the filesystem layer.

A common concern with implementing COW functionality at the block-layer is the need for ensuring consistency of the filesystem above. We note that this problem does not apply to our system because the buffer cache that is consistent with the state of the disk is also saved and restored. However, this precludes observing filesystem state outside of virtual machine runtime context (see section 3.7.1).

3.4.3 Data Indexing

A branching storage system can index disk blocks based on location or content-hash. Indexing by content-hash is more storage efficient because it ensures that a block with a particular content is stored just once. However, we believe that it will have higher performance overheads owing to hash computation and its complete disregard for disk locality. The read/write performance results of Venti [34], a block-level storage system that uses indexing by content-hash, validates our belief. In comparison, indexing by location should incur lesser overheads. Thus, our prototype uses index by location.

3.4.4 Handling Writes

There are three common approaches to handling writes in COW-based storage systems. The undo-log approach (or ”copy-on-write”) mutates the original disk after copying each modified data block’s original contents to the log. The redo-log approach (or ”redirect-on-write”) redirects writes to the log and keeps the original disk intact. In the write-anywhere approach, all block locations, including those of the original disk, are virtualized and writes always go to new locations.

The undo-log based approach does not apply to our case because we do not want to mutate the golden image. Although write-anywhere has the best write performance, it is also the most complex, owing to virtualization of all data blocks and the need for handling fragmentation and garbage collection. We chose the redo-log based approach to avoid the complexity of the write-anywhere implementation.

The resulting VM local storage is illustrated in Figure 3.1.

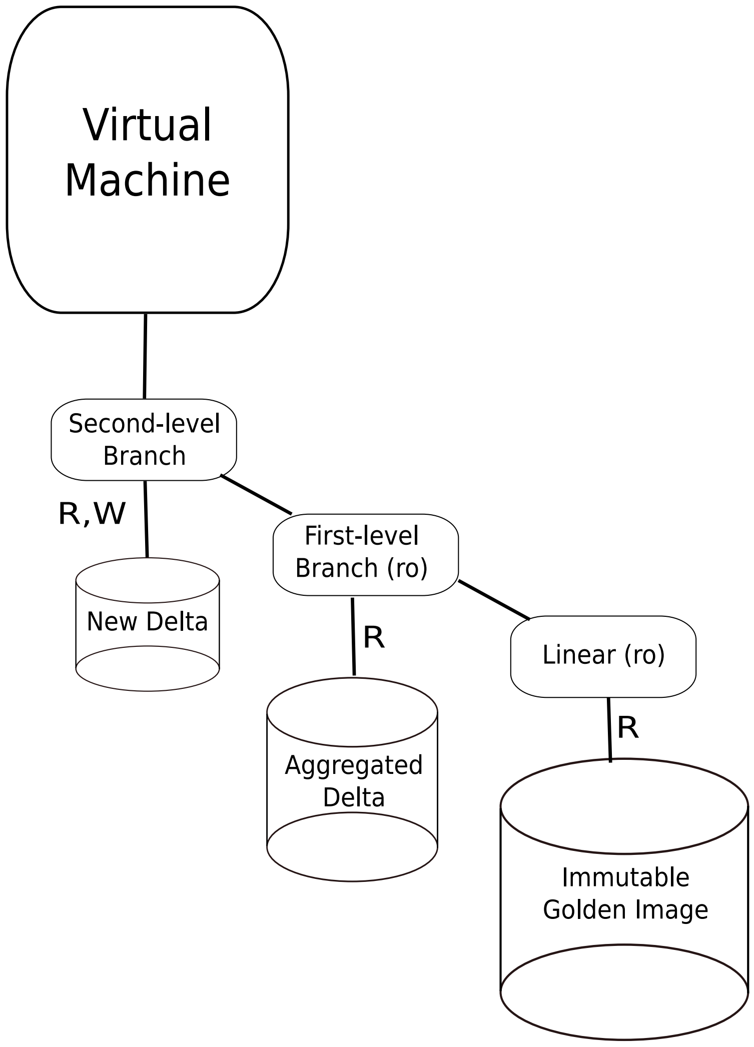

Figure 3.1. Node-local branching storage.

3.5 Node-Local Time

We anticipate that some experiments could be affected by the period of inactivity between a swap-out and a subsequent swap-in (see Appendix B for justification). Even though the runtime state of a node is restored, the "current time" may pose problems. For example, suppose that an application profiles an event with calls to *gettimeofday()* before and after the event. If a stateful swap-out occurred during the measured event, then the measurements will yield absurdly large timing values. The problem here is that, left to themselves, restored VMs see the current time.

Xen VMs rely on the VMM for time. So, we modify the Xen VMM to virtualize the time returned to VMs, to make them believe that time flows continuously, irrespective of swap-outs.

Our prototype has been designed in such a way that users can selectively disable or enable time virtualization at any time. The change comes into play upon the subsequent stateful swap-in. At that time, the VMM will virtualize the VM time—i.e., continue time updates from when the VM suspended—only if time virtualization is enabled. Otherwise, the VM time corresponds to the VMM time.

3.6 External World Interactions

An Emulab experiment node has two kinds of network interfaces: the *experimental network* and the *control network*. The experimental network is specified in the experiment topology and connects nodes within an experiment, whereas the control network is used by the Emulab infrastructure to manage experiments and for communication with machines outside the Emulab network.

An experiment node's interactions on the control network are *external*, because the interacting node on the other end does not participate in stateful-swapping and is outside the time-virtualized world. In contrast, nodes that are reachable over the experimental network will be swapped together and share the same virtualized notion of time.

In the presence of external world interactions, stateful-swapping may affect application transparency. This can happen in two cases. First, when a VM's external world interaction corresponds to a *stateful* network service, the interaction can be disrupted by a swap-out because connection state on the external node cannot be guaranteed to persist during the period of swap-out. Second, if the interactions contain physical timestamps, then a time-virtualized VM could confuse the external world with its notion of time, or the external world could reveal the real time to the VM. Either of these cases may result in unintended behavior. For example, consider the scenario where a time-virtualized VM updates a file on an external world NFS server. Consequently, the VM may put its virtualized timestamp in the *modification time* field of the file. When VMs belonging to experiments in different "time frames" update files on the server, the file modification times will not reflect reality and applications (such as gmake) that rely on them can behave erratically.

3.6.1 The Closed World Assumption

We have observed that most Emulab experiments operate as a *closed world*. That is, most applications that are evaluated on Emulab interact only over the experimental network.

3.6.2 External State Problem

Both experimenters and the Emulab control infrastructure rely on the experimental control network. Experimenters typically use the control network only for experiment setup and monitoring. Potential swap-out interruptions during either of these operations are, at the worst, minor user inconveniences and should not affect the transparency of the application being evaluated. Thus, we assume that interactions with machines outside the Emulab domain can be disrupted by stateful-swapout.

Interactions with the Emulab control infrastructure result from Emulab specific network applications (like event system, idleness monitoring and health monitoring daemons) and well-known network services such as DNS, NFS and NTP. Since our prototype is not integrated with Emulab, the Xen VMs do not run the Emulab specific applications. As part of integrating stateful-swapout in Emulab, these applications may require modifications to handle disruption due to swap-outs. However, we do not address Emulab specific applications in this thesis.

The DNS and NTP services are inherently stateless. The Emulab file server uses NFS version2 [44], which is also stateless by design.[3]

From the foregoing discussion, we conclude that most Emulab experiments do not have any stateful, external-world interaction whose disruption, due to stateful-swapping, may affect application transparency.

3.6.3 Embedded Timestamp Problem

Earlier in the section, we referred to the problems caused by physical timestamps embedded in the external world interactions with a time-virtualized VM. We address these problems by "transducing" those timestamps—i.e., we convert timestamps found in the inbound packets to the VM's *virtual* time frame and those in the outbound packets to the *real* time frame. The

[3]We note that the use of NFS version 4 will likely cause problems with stateful-swapping owing to its use of leases for file locking [6]. Suppose that a VM obtains a lease before being swapped-out. Because of time-virtualization, upon a subsequent swap-in, the VM may continue to use the file assuming it still holds the lock although, in reality, the lease may have expired.

advantage of the transducing approach is that it can be done at the VMM level and does not require changes either to the external world entities or the guest OS in the VM. However, we note that the transducing approach has the following fundamental limitations:

- *Not a general approach.* The semantics of time-revealing interactions have to be well-known. From the network packets, we should be able to figure out the location of timestamps. Further, it is not practically possible to implement transducing for all known protocols containing timestamps. Even so, users may deploy custom proctocols.
- *No encryption.* Even if the protocol is well-known, the timestamps have to be in clear-text. Since the transducing approach interposes on the network packets without the knowledge of the sender or the receiver, it cannot work when timestamps are encrypted—i.e., in the presence of protocols like IPsec or SSL.
- *No future timestamps.* Timestamps in the future cannot be transduced. We need to know the schedule of experiment swappings (i.e., periods of swap-in and swap-out) to be able to transduce time. Naturally, we cannot know the schedule for future swappings. For example, file lease expiration times and HTTP cookie expiration times are timestamps in the future and thus cannot be transduced by packet interposition.

Based on the assumption that Emulab experiments operate in a closed world, the external world interactions now consist of structured interactions with the Emulab control infrastructure. We have identified such interactions where time comes into play, to include the Emulab event system, NTP and NFS. Of these, the Emulab event system is an instance of a custom Emulab application that will need to change to support stateful-swapping. We do not discuss it here. Both NTP and NFS have well-known semantics, transfer data unencrypted and do not include future timestamps.[4] As a result, time transducing can be applied to them.

- *NTP.* A Xen VM would get its time from the management VM on the physical machine, which always runs in current time. It may be sufficient to run the NTP client on the management VM and, thus, we can do away with Emulab NTP interactions on a VM.
- *NFS.* The Use of NFS is much more fundamental. Experimenters use NFS to access their home and *project* directories. Thus, we implement time transducing for NFS, in order to ensure that virtualized timestamps never get written to the NFS server and that VMs in an experiment always see file times corresponding to their "time frame." We present the time transducing logic and the implementation of NFS protocol transducing in section 4.2.2.

[4]We assume that the physical clocks between the Emulab server and the experiment machines are synchronized

In summary, we have ensured that for most Emulab experiments, stateful-swapping, in the presence of external world interactions, will not affect application transparency.

3.7 Optimizations

3.7.1 Free-Block Removal

When a file is deleted or modified (rather than appended), the data blocks occupied by the file are freed. Such blocks can potentially be removed from the disk deltas. As a consequence of operating at the block layer, we lack the knowledge required to identify free filesystem blocks. Disk imaging tools, such as Frisbee, acquire this information by parsing the disk using filesystem-specific plugins. However, this approach is not applicable to our system for the following reason. Those tools operated on a complete and, thus, consistent view of the filesystem state. In contrast, filesystem state on our disks are partial and can be inconsistent. A complete view will require coalescing the disk state with the buffer cache in the memory image. To illustrate the source of inconsistency, consider this scenario: the on-disk free block metadata indicates that block 1024 is free. In reality, block 1024 is allocated on disk. However, the free block metadata has not yet been updated. This is a possible scenario because filesystems typically flush data before metadata.

One possible solution is to parse the memory image, in addition to the disk image, to get a complete view of the filesystem. But this requires a nontrivial amount of filesystem- and OS-specific knowledge. Another solution is to modify the filesystem to tell us the right state online. We dismiss this solution because it is not VM-agnostic. Our solution is a *gray-box technique* [3] that works at the VMM layer and snoops on writes. It uses filesystem-specific plugins to construct the free-block metadata that is consistent (although, not necessarily complete) with respect to the data blocks on the disk. We describe the technique and its implementation in section 4.1.3.

3.7.2 Delta Reordering

Allocation of data on the on-disk partition representing a branch happens on a first write basis—i.e., when a block is written to for the first time since the creation of the branch. Over the course of several swap-outs and swap-ins, a VM's aggregated delta is repeatedly merged with the swap-out disk delta. This aggregated is overlaid on a branch that is used in a read-only fashion.

Owing to the nature of block allocation, it is possible that the access locality of these branches could get highly arbitrary and affect the read performance. As part of the offline delta merging process, we also reorder the disk blocks in the aggregated delta to restore locality.

Suppose that the filesystem reads consecutive blocks 1024 and 1025 that are allocated on a branch. Reordering the delta guarantees that there will not be seeks inbetween the accesses.

3.7.3 Pipelined Swap-in and Swap-out

One potential problem is that as delta size grows, the swapping latencies continue to increase. It is nice to have bounded swapping latencies that are independent of the disk delta size. Towards this end, we explore approaches that pipeline the data transfer involved in stateful-swapping with experiment execution.

In a pipelined swap-in, the VMs are started as soon as the memory images are available. The aggregated disk delta is lazily fetched in the background or on-demand to satisfy read requests for unavailable local blocks. In a pipelined swap-out, the disk delta and memory state is saved to the server, while the VM continues to run. Pipelined swap-out leaves minimal state to be saved after the VM stops.

While the approaches seem symmetric, their applicability depends on our perspective for estimating the latencies. There are two contrasting notions of latencies, depending on whether experiments run in a *batch* mode or in an *interactive* mode. In batch mode, we (specifically, the Emulab system) may be bothered about the experiment switching latencies on a physical node—i.e., the time from when the outgoing experiment stopped to when the incoming experiment started. In an interactive environment, we (specifically, the Emulab user) may be concerned about an experiment's swap-in (or swap-out) latencies—i.e., the time from when the experimenter said "swap in" (or "swap out") to when the experiment started execution (or stopped).

The pipelined swap-in approach will work well in both scheduling modes. However, we postulate that the pipelined swap-out approach may not suit interactive experiments. To illustrate why, suppose an experimenter says "swap my experiment out." She probably does not intend to use the experiment until when she decides to swap it in again. A pipelined swap-out attempts to keep the experiment running while the disk and memory are copied in the background. However, this is fruitless and, even worse, detrimental to resource utilization because stopping the VM and copying out will be faster. Thus, it seems appropriate to use pipelining during swap-out and swap-in for batch mode experiments and only during swap-in for interactive experiments.

During pipelined swapping, some disk I/Os that go over the network to the server may take longer to complete. As a result, this approach can potentially skew the fidelity of applications that are sensitive to disk I/O completion times. Such experiments could disable this feature.

We evaluate the performance impact on applications in Chapter 5.

3.8 Putting It All Together

In this section, we describe the working of the stateful-swapping capability. For brevity, the optimizations discussed in section 3.7 have been excluded. In the following discussion, *pnode* and *vnode* refer to the physical machine and virtual machine, respectively.

A stateful-swapping experiment is specified and customized through a per-pnode configuration file. Most nonessential features (such as time-virtualization, free-block removal, pipelined swapping) can be selectively enabled. It is expected that the Emulab experiment infrastructure will generate this per-pnode configuration file after its resource allocation step that maps vnodes to pnodes. Consistent with the Emulab security model, we permit only vnodes of the same experiment to be multiplexed on a pnode. Furthermore, we assume that each pnode is always preloaded with the default VMM disk image. The life-cycle of a stateful-swapping experiment is detailed below.

1. *Initial Swap-in.* The pnodes first read their configuration file over NFS. If it is an initial swap-in, vnodes are instantiated. Instantiating a vnode entails downloading the vnode base image if it is not already cached, creating a branch forked off the base image, customizing the branch based on the parameters (such as VM IP address, network, hostname, etc.) specified in the pnode configuration file and finally, creating the VMs.
2. *Stateful Swap-out.* During a stateful swap-out, pnodes first synchronize at an *Emulab barrier*[5] to minimize packet losses. Then, the vnodes are suspended. As noted in 3.1, we leverage Xen's save functionality to stop a VM and dump its memory.

   The saved memory images and the disk delta—i.e., the contents of the second-level branches—are compressed and transfered to the server. For faster swap-out, we pipeline compression with data transfer.
3. *Offline Activities.* After the very first swap-out, the aggregated disk delta of each vnode is initialized with the swapped-out disk delta.

   Upon subsequent swap-outs, the swapped-out disk delta of each VM is merged with its aggregated delta to produce the aggregated delta that will be used for the next swap-in. Note that retaining all swap-out deltas and memory images would let the experimenter rollback or branch off any swapped-in state in the past. However, our prototype does not implement this functionality.
4. *Subsequent (Stateful) Swap-in.* For a stateful swap-in, in addition to the vnode base image, pnodes also download the compressed aggregated delta and the memory image of each vnode. For faster swap-in, decompression is pipelined with disk writes. Next, pnodes create branches off the base image and overlay the deltas on these branches. Then, they create second-level

[5]Emulab provides a simple form of a barrier mechanism using a synchronization server that runs on one of the experiment nodes.

branches and resume VMs on these branches. Again, we use Xen's restore functionality to create VMs from saved memory images. Similar to swap-out, the pnodes rendezvous at an Emulab barrier after loading all the VM memory images, but before unpausing the VMs.

Steps (2), (3) and (4) repeat until the experiment is destroyed.

CHAPTER 4

IMPLEMENTATION

4.1 State Management

4.1.1 Branching Storage System

We discussed the motivation for a block-level branching storage system in section 3.4. Such a system basically provides instant, recursive snapshots and branches. Snapshots are frozen point-in-time copies of a storage device while branches are mutable copies derived off snapshots.

Since we did not find any existing open-source Linux implementation of a branching storage system, we decided to build one ourselves. Rather than starting from scratch, we chose to extend LVM snapshots [36]. We explain the details of LVM snapshot implementation in Appendix A. In this section, we discuss our enhancements to LVM and the device mapper that result in a general-purpose branching storage system.

The following are our changes to LVM and device mapper. The first three changes are relevant to the stateful-swapping requirements. The last one, made for a related project [7], improves the usability of LVM snapshots.

4.1.1.1 Immutable Snapshots and Mutable Branches

LVM provides mutable snapshots. That is, both the snapshot and its *origin*[1] can be mutated independently. Thus, an LVM snapshot effectively merges a traditional snapshot and a branch into a single entity. Implementation-wise, LVM snapshots combine the undo and redo log approaches described in section 3.4; it acts as an undo log for changes made to the origin and as a redo log for changes made to the snapshot.

We believe that mutable snapshots do not cleanly support the branching semantics. For example, branching off a mutating origin precludes the ability to try several alternative branches starting from the same source. Moreover, mutable snapshots can make writes to the origin or the snapshots more expensive (see the *Optimizing snapshot COW* bullet below).

[1]The parent or source volume of a snapshot.

Thus, we split the mutable snapshot functionality into its logical components: immutable snapshot and mutable branch. As a result, a snapshot volume is immutable, but its origin is mutable. A branch volume is mutable, however, its origin is immutable.

4.1.1.2 Recursive Branching

We made branches and snapshots recursive in nature, by enabling the stacking of snapshots on linear and branch volumes, and branches on snapshot and other volumes.

I/O performance can degrade with longer branching levels. This is because some I/Os may require lookups on several branch or snapshot devices in the hierarchy before finding the required blocks. However, for stateful-swapout we never go beyond two levels and, as evaluated in Chapter 5, the performance impact is minimal for two levels.

4.1.1.3 Optimizing Branch Writes

We removed unnecessary copy-on-write overheads from the write path of branches. This is important to get near-native performance on writes to branches.

LVM snapshots and branches allocate blocks in units of chunks.[2] When data blocks are written to a branch for the first time, the original contents of the chunks, which will store those data blocks, may need to be copied from the origin to the branch. This copy is required to handle the case where the size of the write does not align with the chunk size.

We optimized for this case by short-circuiting writes that are chunk-aligned. Furthermore, we ensure that the branch always get such writes by choosing a chunk size, such that the file system block size is a multiple of the chunk size.

4.1.1.4 Optimizing Snapshot COW

Write performance of a snapshot origin volume can degrade exponentially with the number of snapshots of the volume. This is because an LVM snapshot is considered independent of other snapshots of the same volume and complete by itself. As a result, a write potentially incurs copy-on-write to each of its snapshots, if the chunk is not already present in them.

We modified the snapshots of a volume to be time-chained and differential—i.e., a snapshot contains only those changes made since the creation of the previous snapshot in the chain. As a result, a volume's write performance will be independent of the number of snapshots it has. However, this change has the potential to make writes to mutable snapshots expensive (a write to

[2] A chunk is the basic unit of copy-on-write in LVM snapshots.

a snapshot may require a copy to its immediately newer neighbor in the chain). Since we have made snapshots to be immutable,[3] this is not a problem.

4.1.2 Packet Logging and Replay

During a swapout, the incoming packets destined to VMs that have just been suspended are logged in the corresponding management VMs. Upon a subsequent swapin, these packets are replayed to the respective user VMs.

- **Logging**

  We leverage the ULOG target of the Linux *iptables* framework for packet logging. The default functionality of the ULOG target is to queue packets to a userland daemon (*ulogd*) for logging on their way to the destination. We modified the ULOG kernel module functionality to begin logging packets only after the Xen *network backend driver*[4] instance corresponding to the user VM that is being suspended has stopped. This ensures that packets that arrive until a VM can accept them, are delivered and never logged.

  Packet logging is enabled from the time when all physical machines in the experiment decide to suspend their VMs, to the time when all the VMs on those physical machines are suspended. We use two barriers, before and after the suspension of VMs, for this purpose. Even after the second barrier, there is a possibility that packets are in the network link (e.g., when a link is shaped with a high delay value). After the second barrier, we keep logging enabled for a duration based on the delay value of the incoming link with the maximum latency.

- **Replay**

  We replay a logged packet by first creating an in-kernel *sk_buff* (Linux kernel packet abstraction) for the packet, and then invoking the *dev_queue_transmit* function with the sk_buff. We have implemented this functionality as a kernel module, to which the packet content is passed using the *proc* filesystem interface.

  The logged packets can be replayed only when the VM they are destined to has resumed execution. Upon resuming, VMs themselves would also transmit packets. Thus, it is possible that new packets arrive before the replay completes. Delivering these new packets before replaying all the logged packets can result in guest OS applications (or protocol stack) seeing

[3]In fact, this was a motivation for making snapshots immutable.

[4]The network backend driver runs in the Xen management VM. It acts as a proxy between the network frontend driver running in the user VM and the physical device driver residing in the management VM.

out-of-order reception of packets. In order to preserve transparency across swapouts, we queue these new packets (at the outgoing edge) until replay completes.

Following a VM restore, Xen drops all queued network requests that are yet to be picked up by the network backend (for transmission) or frontend[5] (for reception) drivers. Furthermore, the Xen frontend-backend driver connection is established only after a user VM has resumed (because the user VM has to participate in the process). As a result, guest OS applications (or network stack) can potentially transmit packets in this short interval between resuming of the VM and functioning of the network frontend driver. These packets are also dropped by Xen. We modified the network frontend driver to preserve the unprocessed requests across VM save/restore and queue the early transmit requests following a resume. These modifications had to be done in the network frontend driver inside the user VM—an exception to our design choice to remain guest OS agnostic (although only to fix something broken in the existing Xen implementation).

4.1.3 Online Free-Block Computation

Section 3.7.1 discussed the need for online free-block computation. In this section, we describe our technique and its implementation.

We build on the insight that filesystems write only allocated blocks to the disk. At the VMM layer, we maintain a shadow copy of the free-block bitmaps that is consistent with the data blocks on the disk. This is done by snooping on all writes.

The snooping algorithm works as follows. When filesystem free-block bitmaps are written, we save the bitmaps. On any other write, if the bitmap corresponding to the written block exists, we reset the corresponding bit. To *warm up* the shadow bitmaps, we preload bitmaps before mounting the filesystem.

This scheme is straightforward to implement for filesystems whose on-disk layouts support fixed size blocks and free-block bitmaps. Some popular filesystems, such as EXT2, EXT3, FFS and ReiserFS, satisfy this criteria. Implementing this scheme for other filesystems, such as FAT32, NTFS and XFS, involves parsing the contents of the metadata blocks to infer the free-blocks.

We have implemented free-block computation for the EXT filesystem because it is arguably the most popular filesystem in Linux (which is the most common OS used in Xen VMs). The

[5]The network frontend driver is a generic virtual device driver that runs in paravirtualized user VMs and talks to the backend driver for transmitting and receiving the user VMs' packets from the real network driver.

components of free-block detection are shown in Figure 4.1.

The free-block calculation is bootstrapped during the first swapin. After setting up the LVM volumes, but before starting the VMs, we initialize the required kernel module and "touch" all the free block bitmaps on the branch device. During a swapout, the VMs are first suspended and then the bitmaps are checkpointed from the kernel module. The checkpointed bitmap files are saved in the Emulab repository and used during the computation of aggregated delta (section 4.1.4). During a swapin, the bitmaps are restored before the VMs are resumed. The checkpointed free

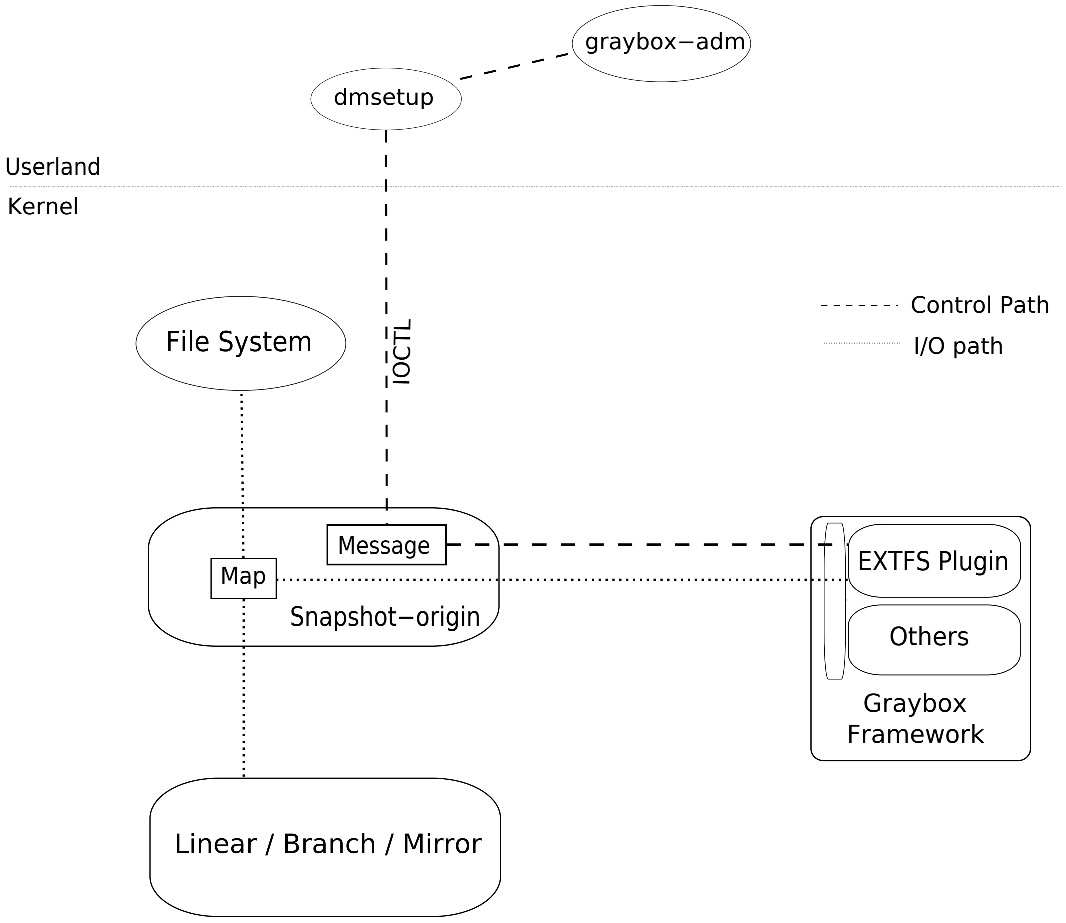

Figure 4.1. Online free block detection: A filesystem-independent kernel framework (*graybox*) exports an interface—functions *ctr*, *dtr*, *checkpoint*, *restore*, *stats* and *process*—implemented by filesystem-specific plugins. A program (*graybox-adm*) for user control communicates with the kernel framework using the device mapper provided interface (*message*). In the absense of snapshot devices, the snapshot-origin layer acts as a simple *I/O passthrough* module overlaid on mutable layers. The snapshot-origin device registers with the graybox infrastructure and in its write path, invokes the *process* function with the in-flight *bio* (Linux block layer I/O abstraction) as argument. The *process* function implements the filesystem-specific snooping algorithm.

block file may also be used in eliminating the free blocks from the swapout delta.

4.1.4 Offline Delta Processing

The offline delta processing happens between a swapout and a subsequent swapin. It involves three steps.

First, we create a new aggregated delta by merging the swapout delta with the current aggregated delta. This is implemented by extending the swapout delta with the chunks that exist in the aggregated delta, but not in the swapout delta. If a chunk exists in both the deltas, then the chunk in the swapout delta contains the most recent copy of the data.

Next, we use the free-block bitmap file from the latest swapout to eliminate free chunks from the aggregated delta. Removal of chunks creates holes in the exception store and thus requires relocation of some existing chunks. For this reason, we do the free-block filtering before the relayout step.

Finally, we relayout the aggregated delta such that the data chunks and mappings are sorted in ascending order of original volume chunk offsets.

4.1.5 Pipelined Swapping

The motivation for pipelined swapping was described in section 3.7.3. In this section we describe its implementation.

We leverage the *mirror* device mapper layer, explained in Appendix A, to implement a simple prototype of pipelined swapping for disk delta. For pipelined swapout of memory, we use Xen's precopy functionality that was designed for live-migration [9].

Figure 4.2 presents the use of the LVM mirror target during pipelined swapping.

During a pipelined swapin, VMs are resumed as soon as the memory image is brought to the experiment nodes. VMs initially access the aggregated delta on the server. We achieve this by layering the first-level branch on the aggregated delta file over NFS. To this branch volume, we then attach a mirror that is backed by local disk storage. This results in mirror resync activity that fetches blocks from the primary mirror on the server to the local disk. During the course of the resync, reads to unsynchronized regions can be expensive because they will be served off the remote copy.

During a swapout, we do the opposite. We add a remote mirror to the recursive branch, which is backed by a file over NFS. The mirror resync activity now copies data to the server. Writes to synchronized regions will be expensive owing to a network transfer in the write path.

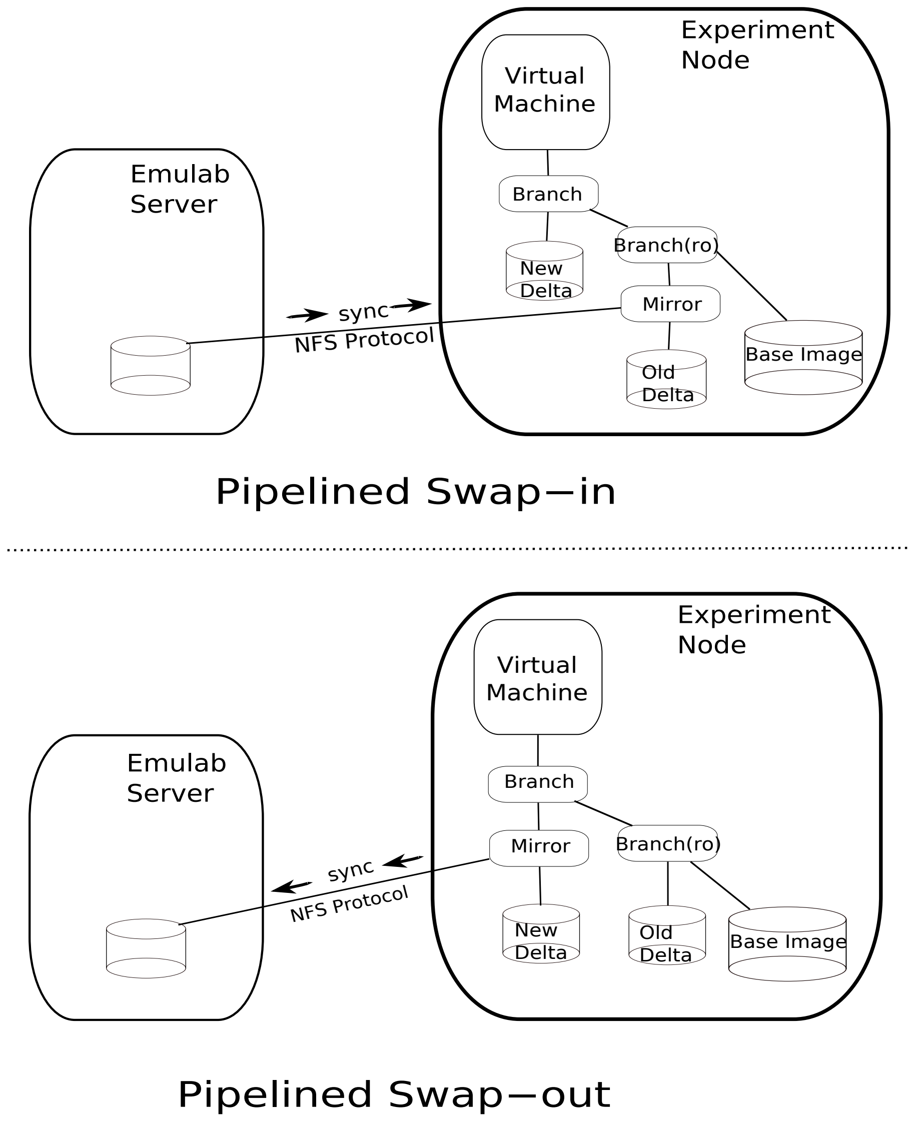

Figure 4.2. Pipelined stateful-swapping: During a pipelined swapin, the contents of the first-level branch are synchronized from the remote mirror to the local mirror. During a swapout, the contents of the second-level branch are synchronized from the local mirror to the remote mirror.

After (or during) the disk mirror resync activity, we initiate Xen's precopy-based *save* operation to a file over NFS. This works by iteratively copying out pages, while the VM continues to execute. At the point when the write-set is deemed small enough, the VM execution is stopped. Then, the last iteration transfers minimal state.

Xen uses the memory precopy technique for ensuring minimal downtime during VM stateful-migration. We added an option to the *xm save* utility to use the precopy mechanism for saving the memory pages to a file. Our modifications to LVM mirrored volumes were two-fold.

4.1.5.1 Rate-Limited Mirror Resync

LVM's mirror resync activity progresses in an aggressive manner. Our initial results showed that this caused more than 100% overhead to the normal I/O load. So, we added a *rate-limit* functionality to the resync activity.

When the normal I/O to the mirrored volume is below a threshold, resync progresses at its usual pace. However, when the I/O activity crosses the threshold, resync is rate-limited. We measure the I/O activity over 500ms periods and use two parameters to tune the rate-limiting functionality. The *minimum I/O threshold* (in blocks/s) parameter is used to set the threshold of normal I/O activity above which the resync activity should be rate-limited. The *rate-limited resync throughput* (in blocks/s) parameter is used to set the maximum throughput of mirror resync when rate-limited by I/O activity.

Furthermore, we have observed that resync activity also hurts a concurrent VM memory image transfer. So, we artificially rate-limit mirror resync during memory image transfer by setting the minimum I/O threshold parameter to 0.

We present the performance results for various values of these parameters in Chapter 5.

4.1.5.2 Preferred Read Mirror

In the current implementation, all reads to a mirrored volume are dispatched to the *primary mirror*.[6] This works fine during the pipelined swapout, when the primary mirror is on the local disk. However, it is inefficient during a swapin, because the primary mirror is remote. We added a *preferred read mirror* parameter to let users choose the mirror that should serve the reads to in-sync regions. In our stateful-swapping prototype, after we create a mirror, we always set this parameter to the mirror on the local disk.

4.2 Time Management

There are two aspects to time management. Virtualizing the local time seen by VMs and transducing time that is embedded in interactions between the time virtualized VMs and the external world. We motivated the need for these in sections 3.5 and 3.6, respectively. In the following sections, we describe the implementation.

[6]The primary mirror is simply the oldest mirror under a mirrored volume.

4.2.1 Xen Time Virtualization

Time-virtualization has been implemented in the Xen hypervisor, so that it can be independent of guest operating systems. However, we have only verified the working of *paravirtualized* Linux under time-virtualization.

4.2.1.1 Time in Xen VMs

A user VM typically relies on the Xen hypervisor for its time. Time information provided to the VM includes the *system time* measured as the number of nanoseconds since boot, an *update stamp*, which is the TSC (hardware Timestamp Counter) value when Xen updated the VM's system time and the *wall clock time* of physical machine boot. This information is stored in a data structure shared between Xen and the VM. VMs compute the current time by reading the hardware TSC and using it to extrapolate the current time based on the recorded system time, update stamp and wall clock time.

Before the VM executes following a restore, Xen updates the time-related fields in the shared data structure with time values from the current physical machine.

4.2.1.2 Xen Modifications

We made the following changes to the Xen hypervisor:

- For the purpose of time-virtualization, we maintain the *real* and *virtual* system times when a VM is resumed. The value of the system time in the shared data structure, immediately upon a restore, corresponds to the virtual system time at resume—this was the system time when the VM was suspended. Subsequent updates to the system time are virtualized based on the real and virtual resume times, and represented by the following equation:

$$v\_system\_time =$$
$$v\_system\_time\_at\_resume + (r\_system\_time - r\_sys\_time\_at\_resume)$$

  v_system_time and $v\_system\_time\_at\_resume$ refer to the current virtual system time and the virtual system time at the most recent resume, respectively. r_system_time and $r_system\_time\_at\_resme$ are the corresponding real system times.

- When a VM is running in time-virtualized mode, we never update the wall clock time in the shared data structure. As a result, its value from the memory image is retained. This wall clock time will be the bootup time of the machine where the current "virtualized time frame" began.

- VMs set one-shot timers[7] through a hypercall to Xen. These timers are set in absolute virtual times. So, we devirtualize them into absolute real times before adding to Xen's callout table.
- We added two hypercalls SET_TIME_VIRT and GET_TIME_VIRT for setting the time-virtualization mode and getting time-virtualization info, respectively. The former is used by the *xm restore* command when invoked with a *-t* option. Stateful-swapping uses this option to turn on time-virtualization when the variable TIME_VIRT is enabled in the experiment configuration file. The GET_TIME_VIRT hypercall gets used in time transducing (section 4.2.2).

4.2.1.3 Limitation

It is not safe for user programs running inside time-virtualized VMs to use TSC for computing time intervals. Relying on TSC values is an unsafe practice anyway, because SMP (symmetric multi-processor) machines may have unsyncronized TSC values.

4.2.2 Time Transducing

In this section, we present the time transducing logic for converting any real timestamp to virtual and vice versa. We also describe how time transducing for NFS has been implemented.

4.2.2.1 Time Transducing Logic

Consider the experiment swapping history in Figure 4.3. The experiment started execution at r_0 and, after several stateful-swapouts, it is swapped-in and running at current time r_C.

By virtualizing an experiment's current time, we remove the gaps owing to its past swapout time periods from the current real time. As a result, if a VM in that experiment does a stat on an NFS file, it may see a timestamp *r* that will not be correctly ordered with respect to events in the experiment's time frame. If *r* were the current real timestamp, it will seem to be a timestamp in the future for the VM. Similarly, when the VM creates an NFS file, it puts a timestamp *v* which corresponds to its current time. However, from the NFS server's time frame, this will be some timestamp in the past. Thus, we need to transduce timestamps *r* and *v* into the corresponding virtual and real timestamps, respectively.

To convert from the real timestamp *r* to an experiment's virtual timestamp, we remove all swapout periods that precede *r*. Similarly, to convert from an experiment's virtual timestamp *v* to the real timestamp, we add all swapout periods that precede *v*.

[7]For example, when a VM has no useful work to do, it sets a one-shot timer asking Xen to it wake up at the next timer-interrupt.

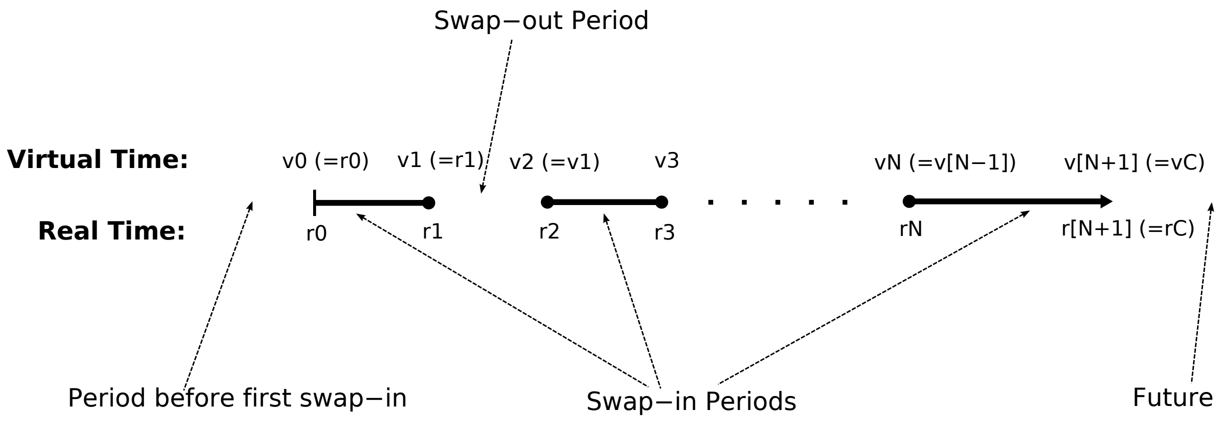

Figure 4.3. Experiment swapping timeline

The conversion from virtual timestamps to real timestamp and vice versa is formalized in Table 4.1. A particular real or virtual timestamp may fall in one of four regions—before the first swapin period, during a swapped-in period, during a swapped-out period or in the future—represented by the rows in the table. Of particular interest is the swapout period. All real timestamps during a swapout period gets mapped to a single virtual timestamp. This results in loss of ordering information between events that occur while an experiment was swapped-out. In contrast, there is no virtual timestamp corresponding to a swapout period.

Here is how time transducing works. Given a real timestamp r, the conditions in the third column are evaluated until one of them is satisfied. When that happens, the expression for $V(r)$ in the corresponding row is evaluated. That gives the virtual timestamp corresponding to r. A similar process is followed for conversion from virtual to real, however, using the last two columns. The value $ERROR$ represents the fact that future timestamps cannot be transduced.[8] The functions $SwapInR$ and $SwapInV$ take a real timestamp and virtual timestamp, respectively, as arguments and return true if the timestamp corresponds to a swapin period. The functions $SwapOutR$ and $SwapOutV$ work similarly.

4.2.2.2 NFS Time Transducing

We first identified the messages containing timestamps in the NFS (version 2) protocol [44]. REQUEST messages, sent from the client to the server, contain virtual timestamps and these messages include SETATTR, CREATE and SYMLINK. RESPONSE messages, sent from the server

[8]In our implementation, we provide some leeway for future timestamps to compensate for minor errors in time synchronization between physical machines. We handle them using the logic for swapped-in period.

to the client, may contain real timestamps and these include GETATTR, SETATTR, LOOKUP, READ, WRITE, CREATE and MKDIR.

Transducing real timestamps in RESPONSE messages works as per Table 4.1. Note that a VM cannot order files that were modified or accessed within a single swapout period. However, it can order these files with respect to modifications done in any other time period. Furthermore, transducing real time to virtual time ensures that the VM will not see bogus timestamps that are in the future.

Transducing virtual timestamps in the REQUEST messages is simpler. This is because the NFS clients always write only their current time in the timestamps sent to the server. We can translate these to real timestamps using the forth column in the “Swapped-in” row of Table 4.1.

4.2.2.3 Implementation Details

We use the GET_TIME_VIRT hypercall to fetch the real and virtual timestamps when a VM is suspended or resumed and append these timestamps to the swapping history. The VM swapping history is saved to the Emulab server during a swapout and restored upon a swapin.

In line with our design decision to remain guest OS agnostic, we have implemented the

Table 4.1. Time Transducing Logic

Period	**Real to Virtual / Outside to Inside**		**Virtual to Real / Inside to Outside**	
	$V(r)$	Conditions $(i = 0, 1, ...N)$	$R(v)$	Conditions $(i = 0, 1, ...N)$
Before Initial Swapin	r	$r \leq r_0$	v	$v \leq v_0$
Swapped-in	$v_i + (r - r_i)$	$r_i < r \leq r_{i+1}$ && $SwapInR(r)$	$r_i + (v - v_i)$	$v_i < v \leq v_{i+1}$ && $SwapInV(v)$
Swapped-out	$v_i = v_{i+1}$	$r_i < r \leq r_{i+1}$ && $SwapOutR(r)$	-	-
Future	$ERROR$	$r > r_C$	$ERROR$	$v > v_C = v_N + (r_C - r_N)$

NFS time transducing functionality in the management VM that runs a customized version of Linux. We have added a new *NFSXDUCE* target to the Linux *iptables* framework. This target parses the NFSv2 information from the in-flight NFS packets and mangles timestamps on specific requests and responses using the time transducing logic described above. The NFSXDUCE target is parameterized by the swapping history timestamps.

CHAPTER 5

EVALUATION

Our evaluation of the stateful swapping prototype will answer the following questions:

- How fast is stateful-swapping?
- How much disk state do stateful-swapping experiments accumulate?
- What are the runtime overheads incurred?
- What is the effect of an interleaving swapout?

5.1 Machine Configuration

Unless specified otherwise, we ran all our experiments on Emulab's *pc3000* experiment nodes, each of which has a 3.0 GHz 64-bit Xeon processor, 2 GB RAM, and two 146 GB, 10000 RPM SCSI disks. We used 3 GB virtual machine disk images running Fedora Core 4 Linux with an ext3 root filesystem; each virtual machine was allocated 256 MB memory. All systems were compiled to use 32-bit addressing.

5.2 Swapping Latencies

We measured the swapping latencies with a single node experiment that adds 275MB of disk data in each swapin session (i.e., between a swapin and a subsequent swapout) for four consecutive swapins. We do this for both compressible (about 5 times) and incompressible data.

5.2.1 Baseline

In Emulab today, a (fresh) swapin takes approximately 5 minutes, whereas a destructive swapout takes about 24 seconds. All latency numbers in the subsequent subsections are marginal latencies over these values.

A strawman approach to stateful swapping is one that transfers the entire disk and memory images at swapin and swapout. For the FC4 image, this approach takes about 4 minutes for swapin and 6 minutes for swapout. These values form the baseline for comparing latencies from our delta-based techniques.

5.2.2 Simple Stateful-Swapping

Simple stateful-swapping uses disk deltas, golden image caching and branching storage. But it does not involve background data transfer in the form of pipelined swapping.

It takes one minute to download the Linux Fedora Core 4 golden image using Frisbee [21] and 30 seconds to verify it by comparing content hashes. These times are independent of the delta size and are incurred once for each physical machine during a stateful-swapin, in the worst-case. As discussed in section 3.3, if the golden image was preloaded by Emulab, then it can be used without verification. In this case, both the download and verification times can be excluded from stateful-swapin latencies.

Figures 5.1 and 5.2 plot the swapin and swapout latencies, respectively, against experiment progress for compressible and noncompressible workloads. The golden image verification or download times are not included in swapin times. As one might expect, the swapin latencies grow with experiment progress. This is because of the increase in aggregated delta size. However, the swapout latency only depends on the data accumulated during a swapin session, which in this case is a constant 275MB. Thus, we see almost horizontal lines for swapout latencies.

Furthermore, transferring compressed data during a swapin leads to shorter latencies if the data is compressible and performs as well as the "no compression" case on incompressible data. On the contrary, during a swapout, attempting a compression of incompressible data performs

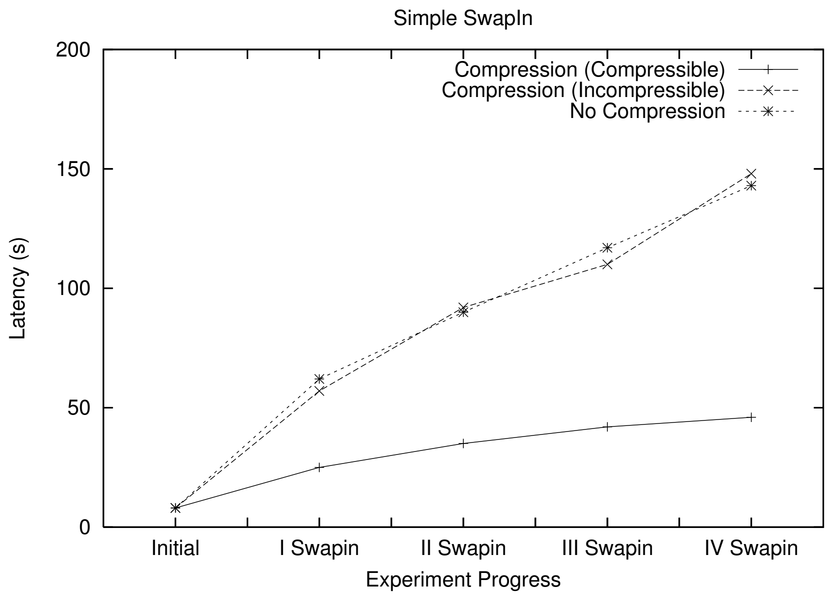

Figure 5.1. Simple Swapin Latencies

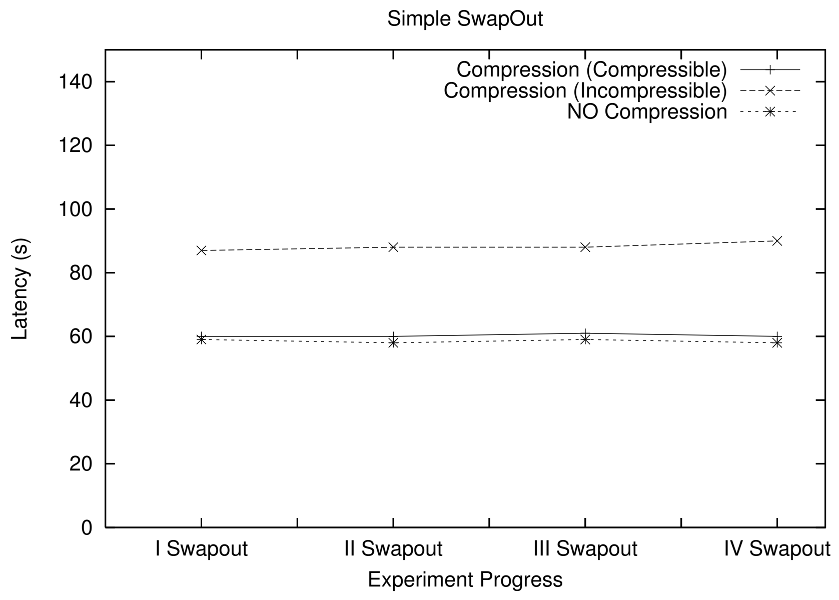

Figure 5.2. Simple Swapout Latencies

much worse than the "no compression." Interestingly, even on compressible data there seems to be no significant gains from using compression. This is because the process of compression (gzip in this case) is generally more time-consuming—about 3 times longer—than decompression. Therefore, we employ compression only during swapin.

5.2.3 Pipelined Stateful-Swapping

Figure 5.3 presents the pipelined swapin latencies for the compressible and noncompressible workloads. The key point to note is that, in contrast to Figure 5.1, the swapin latencies are constant and independent of delta size. Pipelined swapin requires the VM memory image to be downloaded before starting the experiment, and this takes between 17 seconds and 34 seconds depending on the compressibility of the image.

As discussed in section 3.7.3, pipelined swapout helps reduce the interval between experiment suspension and node deallocation, thus shrinking experiment switching times (i.e., the period when no experiment runs on the nodes). We measure the experiment suspension to node release interval as well as the swapout initiation to node release interval. Both these values are plotted in Figure 5.4. The experiment suspension to node release interval takes about 9 seconds on an average—a six-fold improvement compared to the "no compression" case in Figure 5.2.

However, the swapout initiation to node release interval, i.e., the pipelined swapout latency,

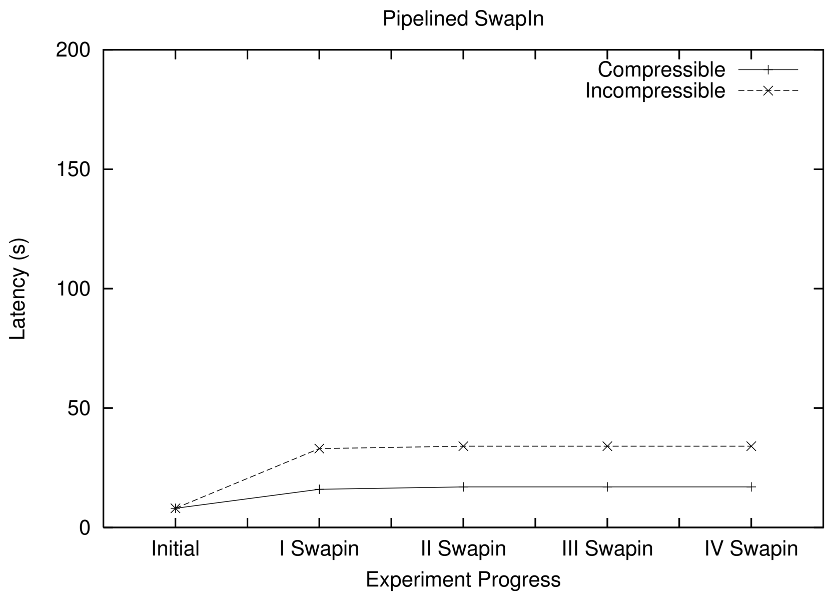

Figure 5.3. Pipelined Swapin Latencies

is 20% longer than the "no compression" case in Figure 5.2. The data transfer during pipelined swapout is slower than the stop and copy approach of Figure 5.2 because the former transfers more data over the wire and is rate-limited by normal IO activity. Furthermore, the swapout latency in the pipelined swapout case also depends on when the data transfer is initiated. This experiment represents the worst case because we initiate data transfer when swapout is triggered. The decision on when to initiate state transfer trades load on the server and possible impact on application fidelity (evaluated later in this section) for shorter swapout latencies.

5.3 State Accumulated

In this section we present the disk delta sizes resulting from common workloads such as system bootup and kernel build.

Booting a paravirtualized linux kernel (derived off the 2.6.18 linux kernel) results in a delta of size 4MB. Building the Linux kernel (version 2.6.18) with the default configuration adds 451MB worth of data to the VM root filesystem, as reported by the *du* utility. The corresponding delta size is 490MB. The metadata overhead is 0.4%. Thus for 451MB filesystem data, the metadata occupies just 1.76MB. The rest (37.24MB) is valid filesystem data.

We have observed that the delta size gets larger with aging. For example, after a series of 3 successive *make*s, the delta size increases to 664MB, despite the fact that the effective used

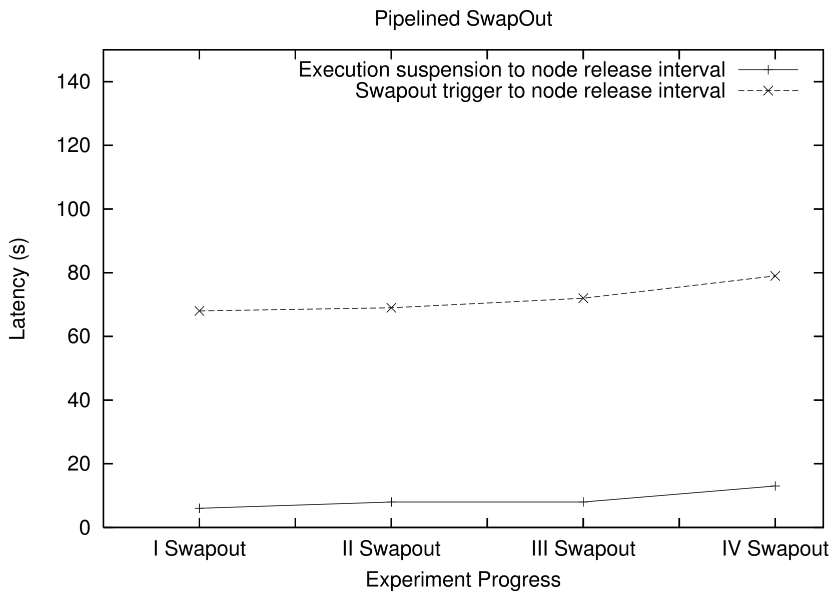

Figure 5.4. Pipelined Swapout Latencies

blocks remain the same (at 490MB). This represents a 35% increase in delta size, just owing to aging.

The free-block elimination optimization, discussed in section 3.7.1, helps this scenario. As Figure 5.5 illustrates, after applying free-block elimination, the delta sizes always correspond to the size of the valid data. This results in as much as 35% space savings due to aging and 94% space savings after a *make clean*.

5.4 Overheads

During normal execution—i.e., in the absence of interleaving swapouts—the overheads incurred by stateful swapping experiments arise from two sources: the VM environment and branching storage.

Xen VMM overheads have been sufficiently studied in earlier work [42, 5]. Our only modification is the time virtualization functionality added to the Xen hypervisor. Since it merely adds an additional arithmatic operation, this should not affect the results from previous measurements. On the other hand, we have made substantial changes to LVM snapshots. Thus, we focus on evaluating the performance of LVM branching storage on a stateful swapout configuration.

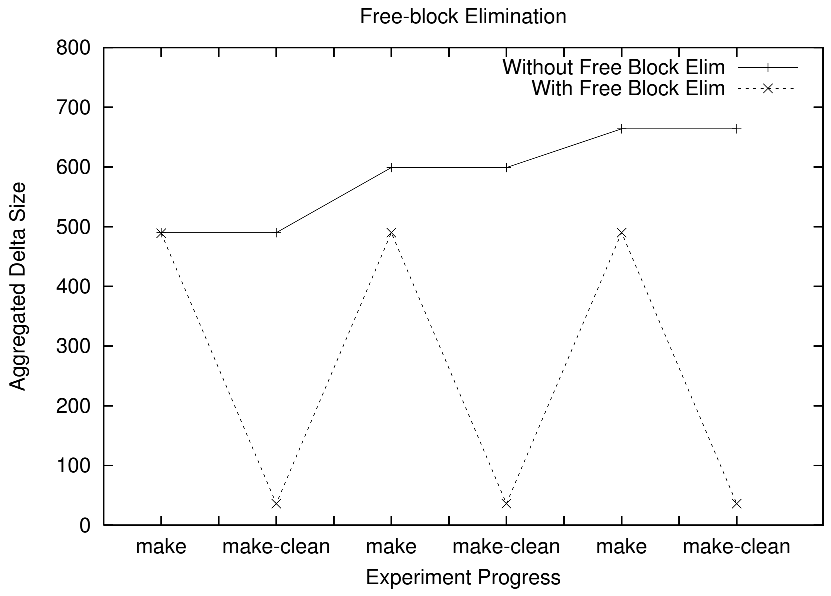

Figure 5.5. Free block elimination on successive *make*s

5.4.1 Branching Storage

We evaluate the performance of LVM branch implementation by comparing it against a simple disk partition and against the original LVM snapshot implementation. We use the Bonnie++ benchmark for this purpose.

Bonnie++ [15] is a popular benchmark that tests filesystem throughput. We configured Bonnie++ to operate on a 512MB file–twice the size of VM memory. The benchmark reads and writes to the 512MB file, one character at a time as well as one block at a time. It also rewrites the file one block at a time.

We ran Bonnie++ on three configurations: a VM running on a two-level branch with our modifications (Branch), a VM running on a two-level branch without our modifications (Branch-Orig) and a VM running over a disk partition (Base).

Aging impacts the IO performance of COW-based systems. For example, initial writes may incur a COW penalty. For stateful swapping purposes, it is important that even initial overheads are minimal, because we frequently (at every swapin) create new COW branch devices. Figures 5.6 and 5.7 show the Bonnie++ results on fresh and aged devices, respectively. By an aged COW device, we mean that all new writes go to existing locations in the log.

On a fresh setup, the sequential block writes on a branch incurs 17% overhead compared to corresponding disk partition writes. This happens because the on-disk metadata regions are

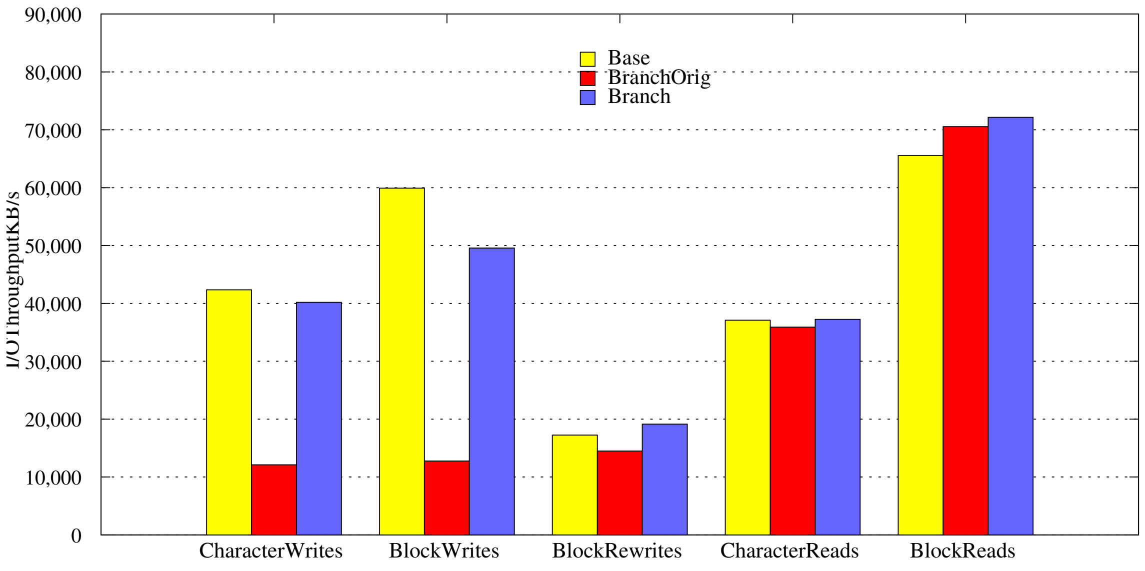

Figure 5.6. Bonnie++ on new storage

distributed over the entire disk and the branch device does additional seeks to update metadata. However, on an aged branch device, the metadata regions are already filled up. Thus, we see that branch performs as well as the disk partition on aged setup. Note also that branch-orig block writes were 74% slower than our modified branch device. Our optimizations eliminated additional copying of blocks if writes are aligned with COW unit size.

In summary, the performance of branch devices is comparable to raw disk partition, except for fresh block writes which incur a 17% overhead.

5.5 Effect of an Intervening Swapout

In this section we study the microscopic effect of an intervening swapout on time, disk and network activity, from the perspective of the guest OS. We specifically focus on these three aspects because our stateful swapping implementation can potentially impact their behavior: we virtualize time across swapouts; we employ COW and background transfer of VM disk data; and, we log/replay inflight network packets.

5.5.1 Time

In order to evaluate the effectiveness of time virtualization we ran a microbenchmark, which continuously does a sleep for 10 milliseconds and measures the difference in time (as reported by *gettimeofday*) across the sleep. The sleep operation is implemented in the Linux kernel using

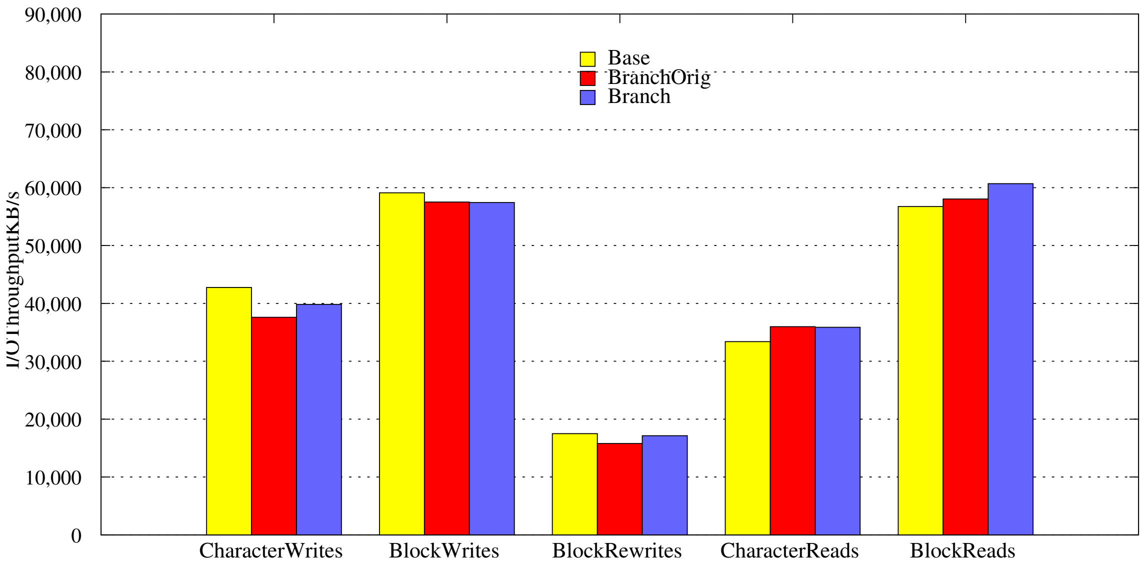

Figure 5.7. Bonnie++ on aged storage

a monotonic clock, which is not affected by our time virtualization. Thus, we expect the sleep operation to take the same time, even across swapouts. However, the time difference will be affected by time virtualization.

Figure 5.8 plots the results from running the benchmark on an idle VM. Note that the reported time difference is actually 20 milliseconds because of the system call overheads. The swapout occured at iteration 119. This results in a 1 millisecond overhead. We believe that the VM participation for tearing down and setting up of device drivers, which happen during VM suspend and resume, respectively, causes this additional overhead across swapouts.

5.5.2 Disk

Our I/O workload iteratively[1] (10 times) copies a 651 MB (more than twice the VM memory) file from the same source to the same destination. In our stateful swapout setup, the source file is in the aggregated delta and the destination file is always written to the second-level delta. We monitor the disk I/O by recording the disk requests (reads + writes) at one second intervals, for the duration of the I/O workload, from inside the VM. The gathered measurements are written to a memory filesystem. When run inside a VM and, in the absence of COW or any intevening swapout, the test took 5 minutes and 52 seconds to complete.

[1]Iteration was required, considering the small VM disk image size and the need for these tests to last for 4-5 minutes

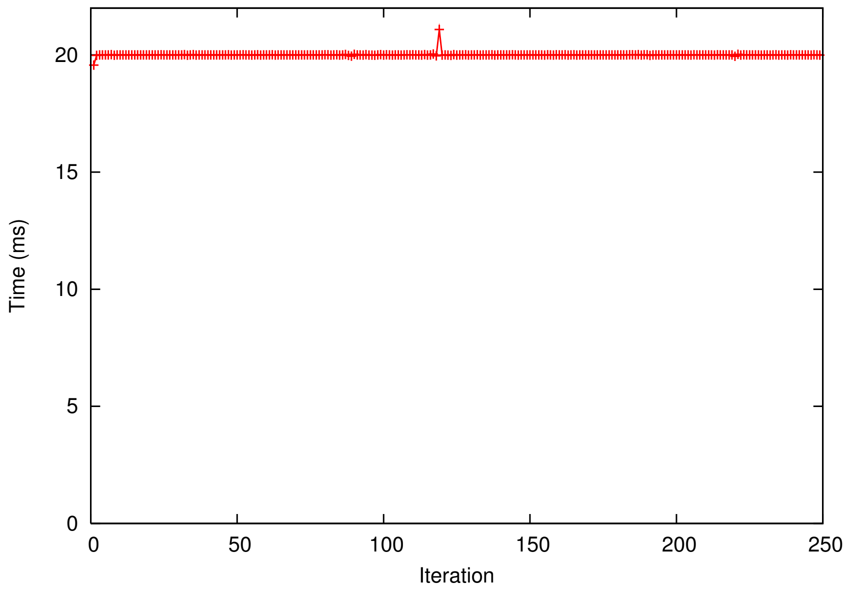

Figure 5.8. Effectiveness of time virtualization – 10 ms time intervals measured across a swapout, which happens at iteration 119.

We report results from four configurations: without any swapout, with a simple intervening swapout, with an intervening swapout that does pipelining-out of data and with an intervening swapout that does pipelining-in of data. In all the tests with a swapout, the swapout was triggered 120 seconds into the run.

Figure 5.9 plots the throughput when there is no intervening swapout. Note that initially, the throughput is lesser. This is owing to the same reason that caused poor Bonnie++ block IO throughtput on fresh branches (section 5.4.1). Furthermore, the test ran 31% faster than the baseline reported above. This is because the source file, which is read from the aggregated delta, and the destination file, which is written to the current delta, were on different physical disks.

5.5.2.1 Effect of a Simple Swap-out

Figure 5.10 presents the IO throughput resulting from a simple (nonpipelined) swapout. There is a perceivable dip in throughput, following the intervening swapin. This can be attributed to the fact that immediately upon a swapin, the writes go to the new branch device, which incurs additional penalty. As a result, there is a 6% increase in total time.

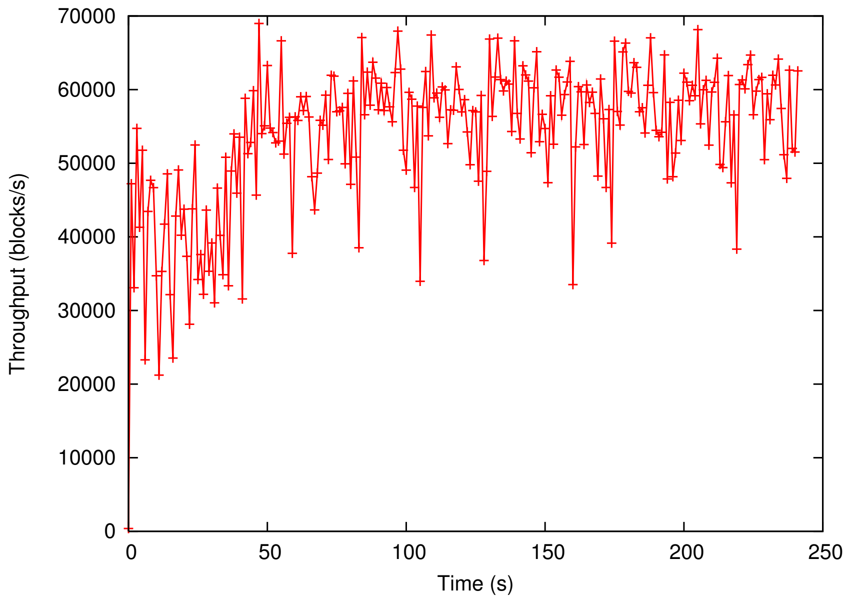

Figure 5.9. I/O throughput, without any interleaving swapout

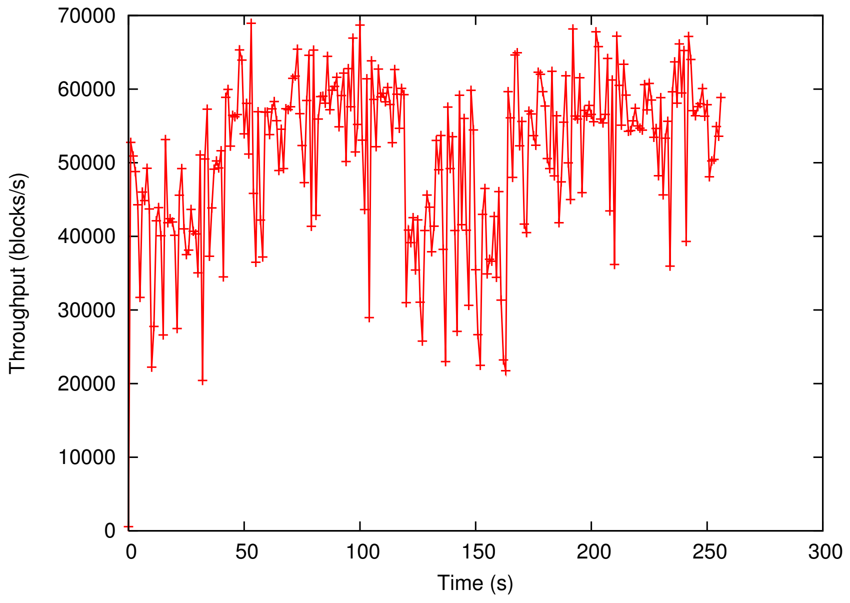

Figure 5.10. I/O throughput across a simple swapout (triggered at 120 seconds)

5.5.2.2 Effect of Pipelined Swap-out

An important factor that affects the IO performance as well as the transfer duration is the rate-limiting of mirror sync activity. Figure 5.11 plots the throughput of copy activity when no rate-limiting is performed. Clearly, this has a huge impact on throughput. Figures 5.12, 5.13 and 5.14 plot the runs when sync activity was rate-limited to 1000, 2000 and 5000 blocks per second, respectively. There is not much impact at 1000, but there are drops in throughput with rate-limiting set to 2000 and 5000. For values below 1000, there was no significant improvements.

Thus, when rate limited, pipelining out data does not cause a significant impact even in the presence of simultaneous write activity.

5.5.2.3 Effect of Pipelined Swap-in

Since the copy workload reads its input file from the aggregated delta, there can be reads that go over the network during the pipeline-in activity.

Figures 5.15 and 5.16 plot the IO throughput of the copy workload when mirror sync activity is not rate limited and when it is limited to 1000[2] blocks per second, respectively. Although rate limiting helps, the rate-limited run incurs a 70% throughput loss for a period of 80 seconds following the swapin. These could be from a combination of remote reads and background writes (sync activity).

In order to study these overheads seperately, we did the following two tests: first, during a swapin, we do not attach a local mirror—all read accesses to the aggregated delta will be remote and there will be no background activity; second, the copy workload was changed to read from an identical file present in the golden image (instead of the aggregated delta)—there will be no remote reads, instead, only background activity. Figures 5.17 and 5.18 present the respective results. From these figures, we can conclude that the effect of background activity is minimal and that remote reads contribute the most to the overheads seen in 5.16.

In summary, pipelined swapin can be potentially expensive, with as much as 70% overheads, but its impact heavily depends on the nature of workloads.

[2]Experimentally, the value of 1000 blocks per second turned out to be optimal, as in the pipelined swapout case.

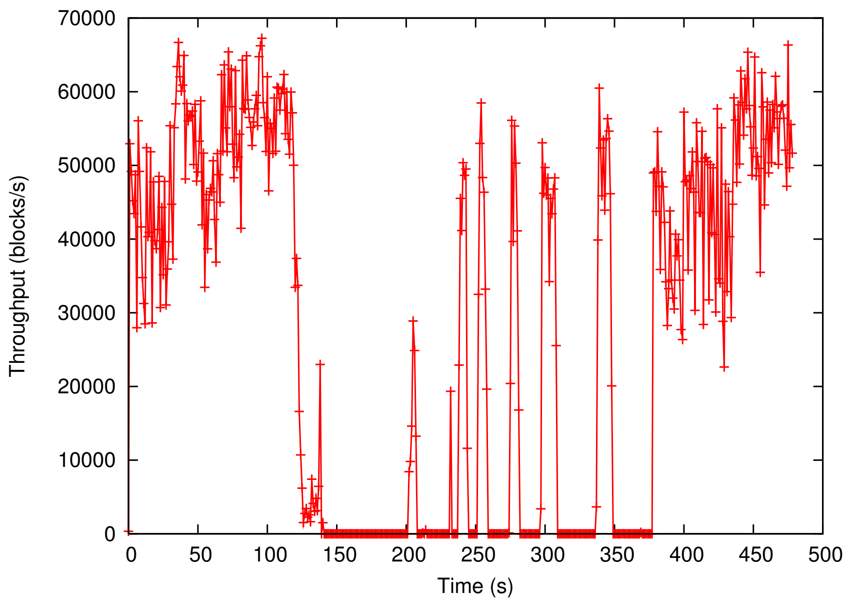

Figure 5.11. I/O throughput across a pipelined swapout, without rate limiting.

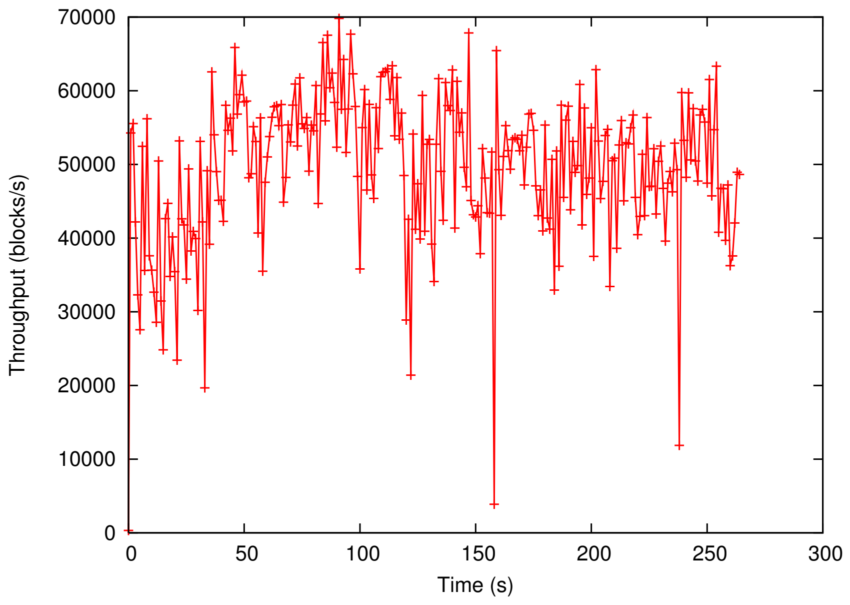

Figure 5.12. I/O throughput across a pipelined swapout, which is rate limited at 1000 blocks per second.

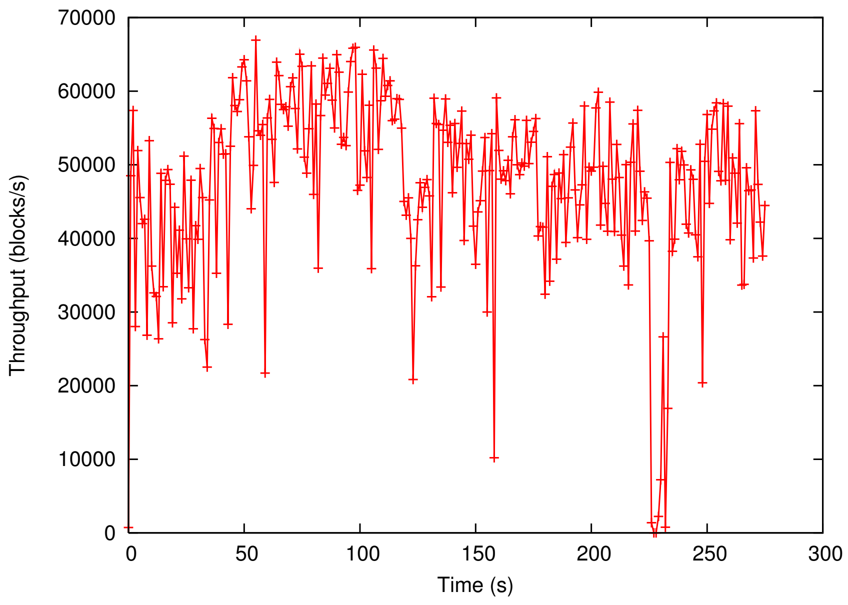

Figure 5.13. I/O throughput across a pipelined swapout, which is rate limited at 2000 blocks per second.

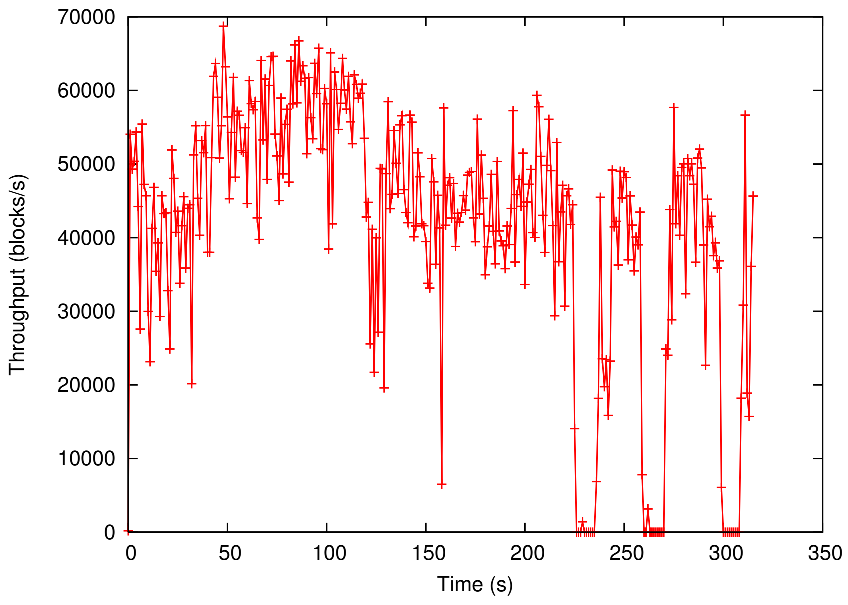

Figure 5.14. I/O throughput across a pipelined swapout, which is rate limited at 5000 blocks per second.

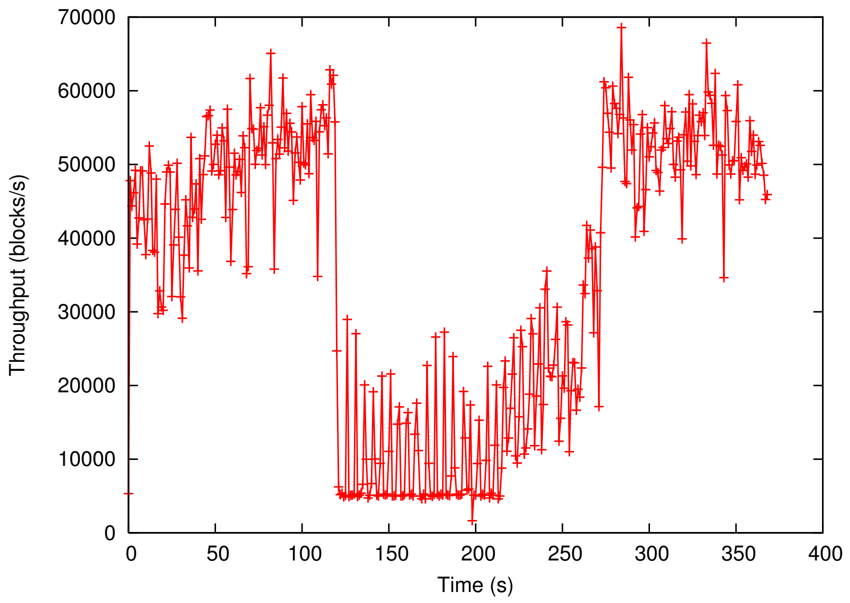

Figure 5.15. I/O throughput across a pipelined swapin, without rate limiting. Reads access aggregated delta.

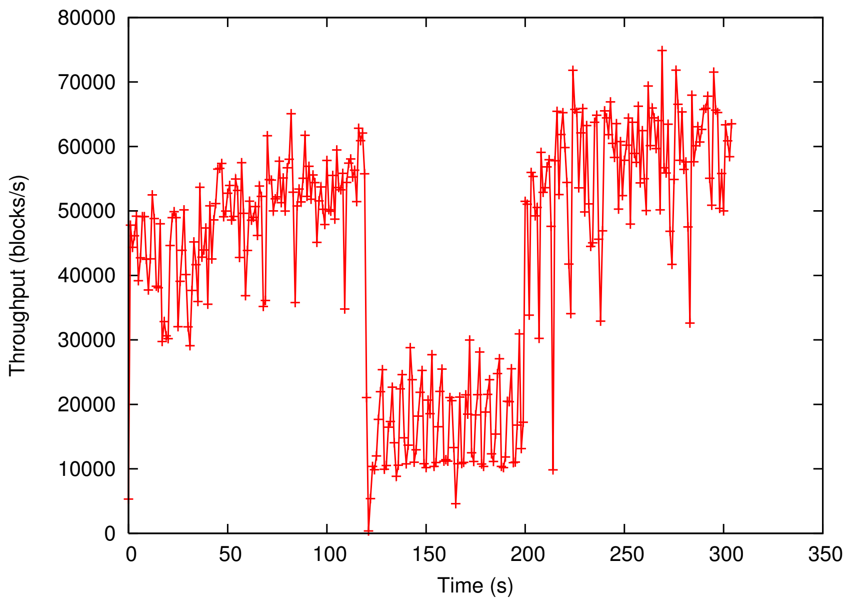

Figure 5.16. I/O throughput across a pipelined swapin, which is rate limited at 1000 blocks per second. Reads access aggregated delta.

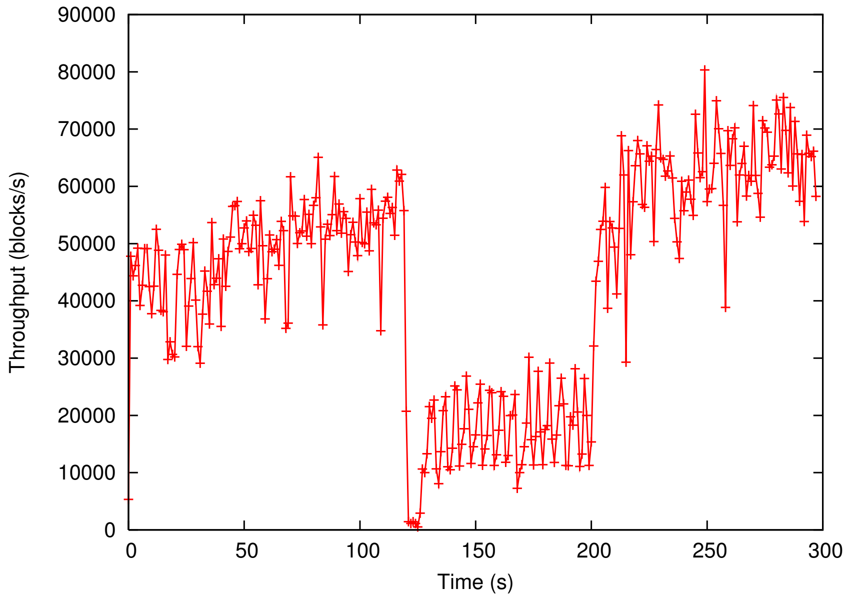

Figure 5.17. I/O throughput across a swapout, where the aggregated delta is remotely accessed following the swapin. Reads access aggregated delta.

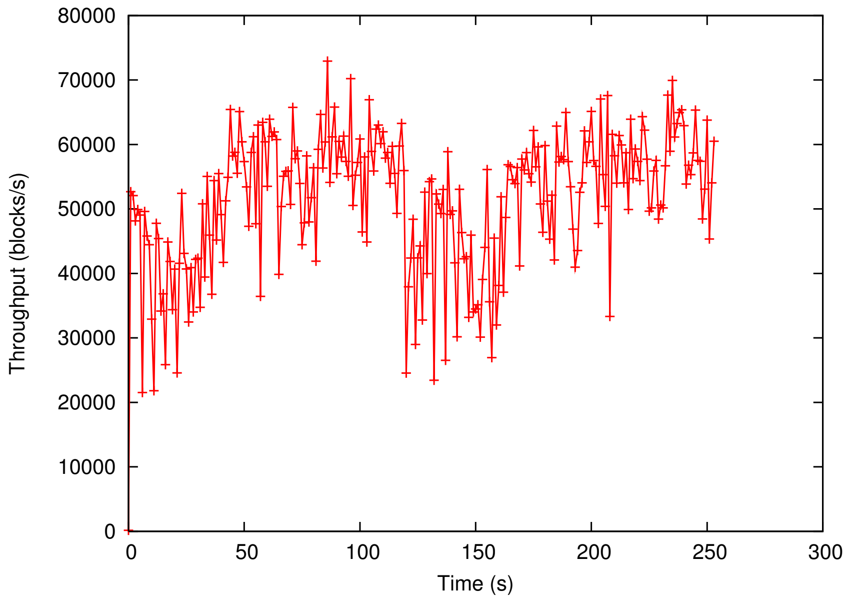

Figure 5.18. I/O throughput across a pipelined swapin, which is rate limited at 1000. Reads access the golden image.

5.5.3 Network

5.5.3.1 SCP Transfer

In this experiment, we do an scp transfer of a 443MB file across VMs on different physical machines. Approximately 10 seconds into transfer, we initiate a stateful-swapout. Since scp is tolerant to packet losses and delays, the scp transfer continues running upon a subsequent swapin and goes to completion.

To measure the effect of a swapout on the packet transfer dynamics, as perceived by user applications, we run a tcpdump session on the receiver VM and log all incoming packets. We plot the tcpdump results in the form of number of packets received per sampling interval of 100 milliseconds over the duration of the transfer. Figures 5.19 and 5.20 show results for two different configurations. In the first configuration, we do not log/replay packets that are in-flight during a swapout, while in the second configuration, such packets are logged and replayed.

As observed from the graphs, logging and replaying the in-flight packets reduces the drop in packet throughput. However, even though we virtualize time, there is still a gap perceived by guest OS applications/network stack. This is because the guest OS cannot send or receive packets immediately before suspension or immediately following resumption. This time is spent by the paravirtualized (frontend) device drivers in the guest OS to break down or establish connection with the corresponding (backend) component in the management VM. The communication between the two driver components uses a publish-subscribe mechanism which is implemented, rather inefficiently, in the userland of management VM.

5.5.3.2 UDP Benchmark

In this experiment, we attempt to measure the effectiveness of our packet log/replay implementation at capturing packets that are in flight and faithfully replaying them. For this purpose, we wrote a simple UDP client-server application, where the client sends a fixed number of packets at a chosen rate, while the server simply reports the number of packets it has received. We vary the packet sending rate to see when packet losses happen across a swapout. Since the minimum time taken by any sleep call in the userland was 10 milliseconds, we had to resort to using busy looping to introduce smaller delays between successive packet sends.[3]

[3]Thus, the interpacket delays are not whole numbers.

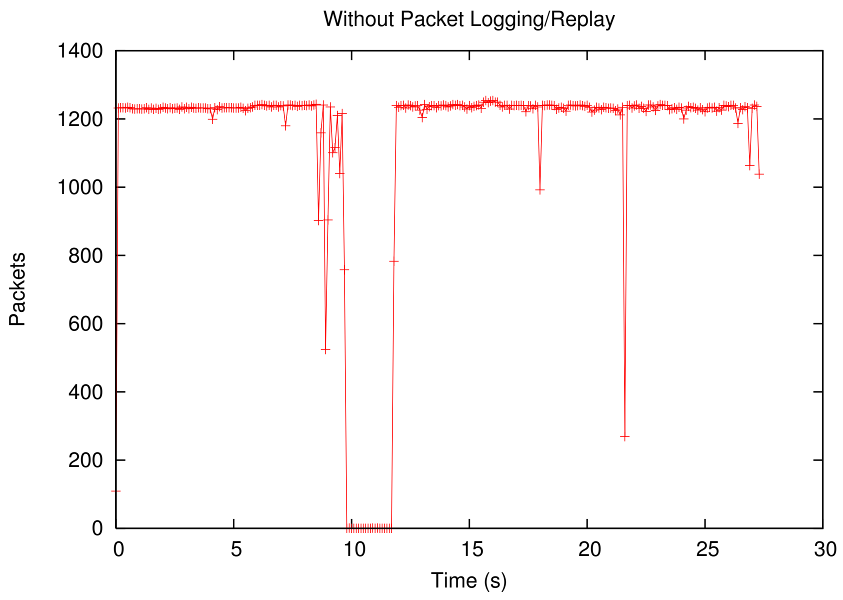

Figure 5.19. SCP with an intervening swapout, with no packet logging/replay.

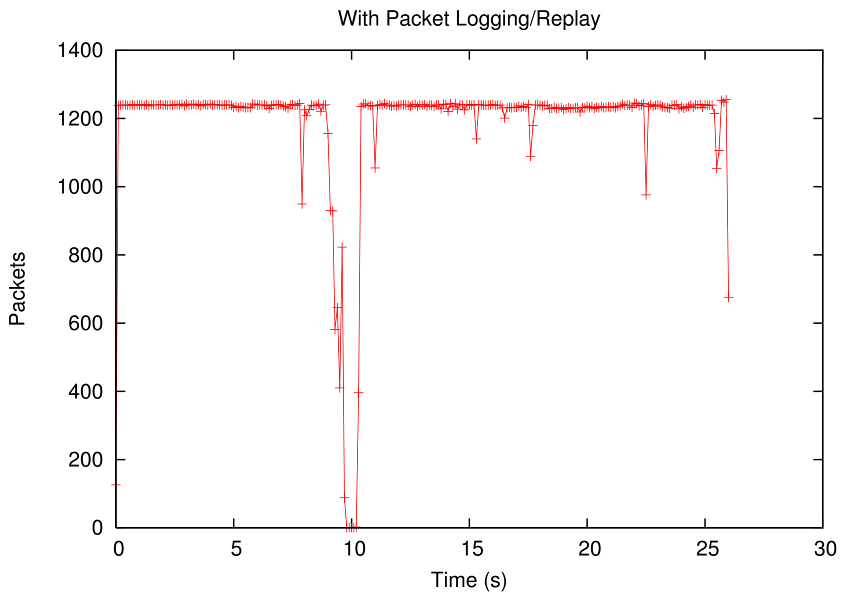

Figure 5.20. SCP with an intervening swapout, with packet logging/replay enabled.

We report the results in Table 5.1. Packet losses typically happen when going from an interpacket delay of 3.3 milliseconds to 4.4 milliseconds. Until all the logged packets have been replayed, we block all the outgoing edges. The packets are lost because the Xen buffer overflows at the outgoing edge of sender.

Table 5.1. Effect of interpacket delay on packet losses.

Inter-packet Delay (ms)	Packet Loss
2.186	Yes
3.280	Yes
4.374	No
10 (sleep)	No

CHAPTER 6

RELATED WORK

6.1 VM-Based Distributed Infrastructure

6.1.1 Collective

Stanford's Collective [8] system uses VMs to deliver managed desktops, called *virtual appliances*, to PC users. These appliances contain OS and other installed applications. PCs include *virtual appliance transceivers*, that cache and run the latest appliances. Mutations to the appliance from a local run are discarded. User data is maintained separately and modifications are periodically backed up to the network repository. Collective fetches data blocks both on demand and proactively.

Collective employs COW technique for delivering appliance updates and for taking periodic backups of user data. Similar to our approach, they cache popular appliances so that application performance does not suffer. In contrast, a cache miss in stateful-swapping causes some delay in swap-in, but does not impact application performance. Unlike Collective, we save and restore the entire VM disk rather than just user data. For a testbed environment, this has the added advantages of preserving user changes to OS or installed applications and also enables a more seamless use of VMs—for example, users are not constrained by where files have to be saved.

The authors of Collective note that workloads optimized for sequential accesses perform poorly on their system. This is because Collective's cache implementation does not consider block sequentiality during allocation. In contrast, VM disk storage in stateful-swapping is more structured. Accesses to sequential data blocks on both the base image and the read-only delta will be sequential on disk—the former, because it is in the form of a complete disk image and the latter, because we reordered the blocks offline (section 4.1.4). There is a possibility that sequential workloads accessing the recursive branch might suffer overheads, because blocks are allocated on a first-write basis. However, we believe that the impact on applications should be less than in Collective.

6.1.2 Virtual Appliances

Inspired from the Collective work, virtual appliances are being used as a new model for software distribution [48]. In this model, vendors provide “pre-built, pre-configured, read-to-run” applications packaged along with an OS in the form of a (possibly generic) VM disk image. To deploy an application, users download the VM disk image and run it on their favorite VMM.

6.1.3 ISR

Internet Suspend/Resume (ISR) [40] employs VMs, local disk cache and portable storage to provide hardware-independent mobility. ISR uses indexing by content hash for local storage and Coda for distributed storage. In contrast, we use COW storage locally and NFS as the distributed filesystem. As noted in section 3.4, we believe that COW suits our LAN environment better than content addressing. Furthermore, the results in the ISR paper [40] indicate that NFS outperforms Coda by 15-50% in LAN environments.

ISR allows dirty state on the client machine to be transferred lazily, after the user leaves. Also, ISR pro-actively prefetches dirty state to client machines in anticipation of the user arriving there. We do not explore either of these approaches because stateful-swapping is targeted for a shared network testbed environment with a different security model (section 3.2).

6.1.4 XenoServer Platform

The XenoServer [25] platform is “a public infrastructure capable of safely hosting untrusted distributed services of uncooperative paying clients.” These services are encapsulated inside VMs.

Similar to our approach, they stack disk delta, called *overlay*, on immutable *template images* that are cached on all client nodes. However, there are two important differences. First, VMs access the overlay over AFS on WAN—contents are brought to the client nodes on demand and cached persistently. Second, they explore an interesting approach to branching functionality at the filesystem-level of VMs. They install a user-level COW NFS server on each client node’s *Management VM* (MVM). This server enables recursive stacking of file hierarchies. During VM deployment, the MVM sets up a VM’s storage area by layering the required overlays and boots the VM from NFS root over the machine-local virtual network.

6.1.5 GVFS

Grid Virtual File System (GVFS) [52] manages data for VM instantiation in Grid computing environments. In GVFS, VMs access their disk over NFS on WAN. However, for improved performance, NFS has been augmented with a local write-back disk cache. In some sense, theirs is an on-demand approach for the entire VM disk. In our case, we use the lazy and eager approaches only for the disk deltas.

6.2 Copy-On-Write Storage

Copy-On-Write (COW) storage has a long history. It has been commonly used to implement versioning and, more recently, with the proliferation of VMs, point-in-time branching.

6.2.1 Block-Level

Block-level implementations of snapshots are common in volume managers [36] [46] and storage arrays [1] [13]. Distributed block-level systems such as Petal [27], Parallax [49] and Olive [2] also provide COW snapshots.

A drawback with block-level snapshot implementation is ensuring data consistency of higher layer applications (such as filesytems or databases) during recovery from snapshots. One approach is to ignore this problem and rely on the higher layer application's recover mechanisms (for example, filesystem *fsck*). Another approach is to interface with such applications to ensure their consistency before taking a snapshot [46]. Although we use block-level branching, the presence of the VM memory image alleviates the consistency problem.

While all these systems allow introspection of disk state at specific points in time, there have been systems, such as Peabody [23], that enable introspection of state at *any* point in time. In industry parlance, this is known as *Continuous Data Protection* [41].

6.2.2 Filesystem-Level

There have been several COW-based versioning filesystems. These systems can be categorized based on the granularity of snapshots (file-level or filesystem-level), implementation method (disk filesystem or syscall), frequency of snapshots and branching capability.

Elephant [39] is a popular system that performed versioning at the granularity of individual files by maintaining an inode log to track versions. It created new versions of files continuously at every *close*. In contrast, NetApp's Snapshots [22], based on Write Anywhere File Layout (WAFL), are versioned at the granularity of the entire filesystem and versioning happens on demand or periodically. WAFL is laid out on disk as a tree of blocks rooted at the root inode. The creation of a snapshot is as simple as duplicating the root inode. WAFL never overwrites any block, except the root inode. It uses a combination of NVRAM and snapshots to ensure consistency of on-disk filesystem at all times. Both WAFL and ZFS [29] provide support for branching.

Ext3cow [32] extends EXT3 filesystem with flexible versioning at the granularity of files as well as the entire filesystem. While all these systems implement versioning at the disk filesystem layer, systems like VersionFS [28] and Wayback [11] work by syscall interposition. These systems can work on any underlying filesystem.

6.2.3 COW Storage for VMs

Given the breadth of past research in versioning systems, we were rather surprised that there was no usable open-source branching storage system for Linux.

Xen team's Parallax [49] was designed for managing VM disk storage in a cluster environment. However, it did not materialize into a usable system either.

As Xen VMM is becoming popular in Linux environments, many have realized the need for better storage software. To our knowledge, there are three Linux-based block-level storage systems that are being developed, concurrent to our efforts.

The Xen team has developed a user-level COW store [51], called *Qcow*, that is based on QEMU. A student at UBC has employed Parallax's radix tree metadata to develop a local branching store called *rsnap* [18]. Like ours, his work also leverages the device-mapper kernel framework. Finally, Zumastor [33], a community project started by Google, intends to develop "a network storage server with enterprise features such as online volume backup, multiple volume snapshots, remote volume replication, integration with Kerberized CITI NFS and Samba."

6.3 Remote Mirroring

Block-level remote mirroring is popularly used for disaster recovery purposes. Updates to a primary copy are synchronously or asynchronously pushed to the remote read-only secondary copy. When disaster strikes primary, secondary takes over. Such systems are both array-based [14] as well as host-based [47].

In contrast, our requirements from remote mirroring was a mechanism to transfer data over the network, with support for transparently redirecting a VM's access to remote blocks. We achieve this by combining the functionalities of LVM mirroring, device-mapper layering and NFS.

6.4 Gray Box Techniques

Our online free-block calculation technique was inspired by earlier work on gray box techniques [3] in storage systems. Specifically, the work on *semantically-smart disk systems* [4] attempted to use filesystem-specific knowledge at disk layer to infer the contents of disk cache, influence filesystem block placement and perform secure deletion. In the same vein, we employed filesystem-specific knowledge about on-disk metadata to infer a consistent view of the free blocks on disk.

6.5 Time Virtualization

There has been recent work by Gupta et al. [19] on slowing down the passage of time from a VM's perspective. The basic goal of that work was to subject VMs to network speeds much higher than what is physically possible. That is, if VM time is slowed down and the real arrival rate of physical events remains the same, then VMs will perceive the physical events to occur much faster than they really do. Gupta et al. used Xen VMMs and took a similar approach as ours for time virtualization.

CHAPTER 7

CONCLUSION

We presented the design and a preliminary implementation of a stateful-swapping facility for the Emulab network testbed. We explored the state and time management issues involved in making experiment swapping *stateful*, *fast*, and *transparent*.

Our work leveraged Virtual Machine Monitors (VMM) for encapsulating node-local state. Towards making stateful-swapping fast, we explored the following techniques:

- *Copy-on-write* for efficient node-local storage
- *Caching* of golden disk images for faster experiment swap-in
- *Pipelining* data transfer with experiment execution to achieve bounded *experiment swapping* latencies
- Application of *gray-box* [3] knowledge of filesystem at block layer to achieve better data compression

Finally, in order to make swap-out time periods transparent, we implemented a mechanism for running experiment nodes in virtual time. We also handled the external world interaction problem by leveraging Emulab's "closed" nature, and by supporting certain well-known external world interactions.

The system at its current state is by no means complete. The following would be some items for further work.

- Automating stateful-swapping in Emulab
- Improving server-side scalability
- Exploring content addressing for optimizing server storage and network transfers
- Providing fidelity guarantees

APPENDIX A

INTERNALS OF LVM SNAPSHOTS AND MIRRORS

A.1 LVM Snapshot Volume

LVM provides mutable snapshots that allow independent mutations to both the original and the snapshot volume. It employs copy-on-write technique to share unmodified disk blocks between an original volume and its snapshot.

The snapshot functionality is implemented by mapping types *snapshot-origin* and *snapshot*. A device with snapshot-origin mapping type is overlaid on the linearly-mapped original device and performs COW—i.e., *copy* to snapshots *on writes* to the original volume, whereas a device with the snapshot mapping type is overlaid on a specially formatted linearly-mapped volume called the *exception store*. Exception store contains the data blocks that were copied from the original volume or written to the snapshot volume. It also stores mappings from block locations on the original volume to block locations on the exception store. The snapshot mapping type exports a view of a complete volume by dynamically mapping data blocks either to the exception store or to the original linear device. Figure A.1 presents an example.

We now describe the I/O path of devices mapped to snapshot and snapshot-origin mapping types.

A.1.1 Snapshot-origin

All reads to the snapshot-origin device are sent to the original device. However, writes effect a copy from the original device to every snapshot whose exception store does not have the blocks being written. After that, the writes are sent to the original volume.

A.1.2 Snapshot

Reads to the snapshot device are sent to the exception store if the blocks being read are already allocated there. Otherwise, they go to the original device. Writes effect a copy from the original device to the exception store, when the blocks being written were not already allocated. After

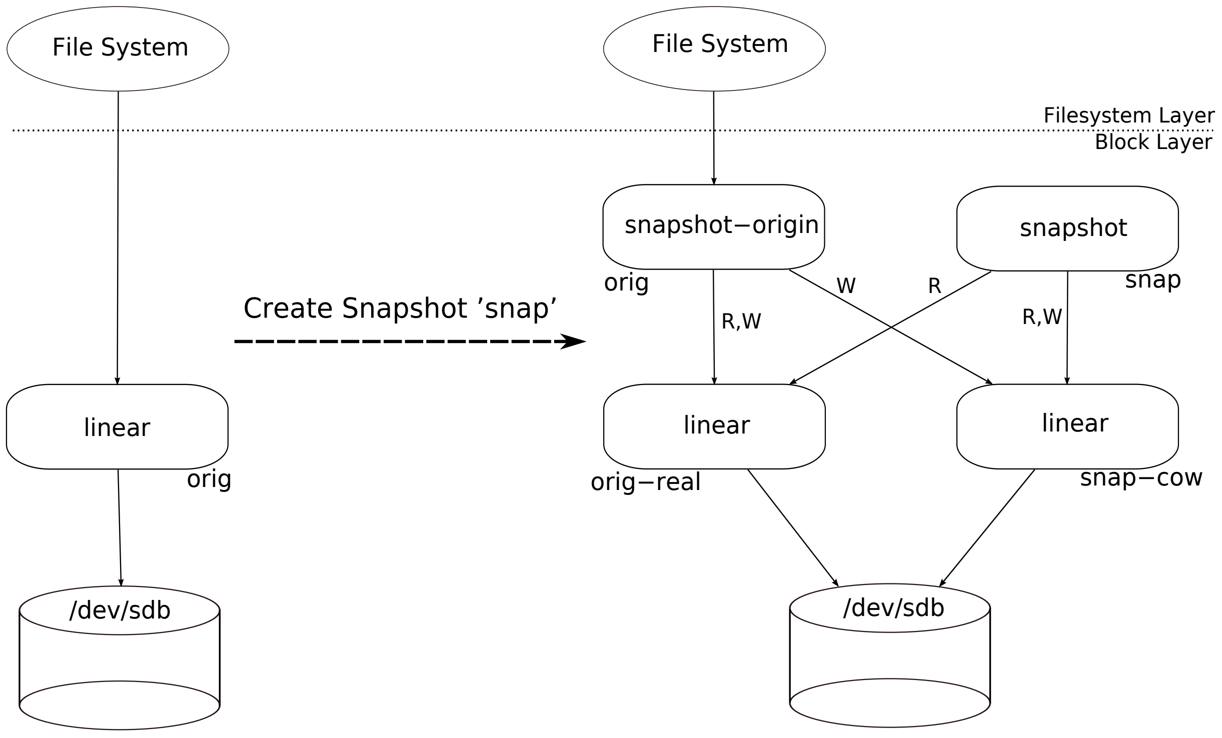

Figure A.1. Snapshot Example: A snapshot *snap* is created off a linearly-mapped volume *orig*. Snapshot creation adds three additional device-mapper devices—*orig-real*, *snap* and *snap-cow*. The device *snap-cow*, with a linear mapping, is the exception store. Device *snap* is mapped to the snapshot mapping type and layered on *snap-cow*. The original *orig* device has been remapped to the *snapshot-origin* mapping type and the new *orig-real* device is mapped to the linear mapping type that *orig* was earlier mapped to. During the creation of *snap*, *orig* is suspended in order to drain in-flight I/O and queue incoming I/O. It is then remapped to the snapshot-origin target and then I/O is resumed. Thus, a snapshot can be created while applications have the original device open.

the copy, the writes are sent to the exception store. The copy before a write is required only if the writes are not aligned with the base unit of COW, called the *chunk*. However, the current implementation of the snapshot mapping type *always* does the copy.

A.2 LVM Mirrored Volume

A RAID1 or mirrored volume consists of a number of replicated linear volumes and a log. These volumes are split into *regions*.[1] The log is in the form of a bitmap that is used to track the state of the regions. Any region can be in one of *sync*, *dirty* or *nosync* states. Regions in sync state

[1]A region is the basic unit of synchronization between the replicated volumes.

have identical contents, while those in dirty state have a pending write in progress. The regions in nosync state have different contents and need to be synchronized with the *primary* mirror.

Writes to a sync region are sent to all the mirrors. The region is set to the dirty state between the time when the writes are issued and the time when they complete. Writes to a nosync region are sent only to the *primary* mirror. All reads are sent to the primary mirror. Reads to sync regions can be load-balanced to improve the read performance. However, the existing implementation of the mirror target does not implement load-balancing.

Concurrent to normal I/O, recovery of nosync regions takes place. Before a region is recovered, I/Os to the region are quiesced. Writes to a region that is being recovered are queued until the recovery completes.

APPENDIX B

IMPACT OF TIME IN STATEFULLY SWAPPED DISTRIBUTED SYSTEMS

In Section 3.5, we motivated the need for time virtualization by claiming that some applications may be affected by the period of inactivity between a swapout and the subsequent swapin. We attempt to substantiate that claim here.

As the first step, we need to understand how systems are typically dependent on time – specifically, wall clock time.[1] Towards this end, we have inspected the use of wall-clock time in seven representative distributed systems that are often evaluated in Emulab. Then, based on this study, we reason how, if at all, these systems could be affected across a swapout.

Our study includes three distributed hash tables (Chord/DHash [12], Pastry/PAST [38] and Bamboo [37]), a peer-to-peer file sharing system (BitTorrent [10]), a distributed filesystem (NFS [6, 44]), a distributed C compiler (distcc [16]) and an Internet overlay (i3 [43]). These are widely used applications and all of them have open source implementations.

B.1 Typical uses of wall clock time

Although some of the studied systems are vastly different, yet, all of them typically end up employing wall-clock time in one of the following four scenarios.

1. *To profile event durations for diagnostic, measurement or other purposes.* All of the studied systems do this. If a swapout occurs during the measured events, the resulting time measurements can be erratic.
2. *To implement an event callback mechanism for scheduling timeouts.* The event expiry time, which is stored as wall clock time, is compared to the current time to decide when an event should fire. This is seen in all the DHTs, BitTorrent and i3. An intervening swapout will cause all events which were supposed to expire during the swapout period to expire immediately

[1]We are bothered about the wall-clock time (which is reported by *gettimeofday()* in C/C++, or *System.currentTimeMillis()* in JAVA), because it will be affected across swapouts. In contrast, the monotonic clock is unaffected.

following a swapin. Thus, with respect to the monotonic clock, the events would expire earlier than scheduled, potentially altering the application behavior.

3. *To store timestamps on persistent objects.* Examples of this include NFS and the DHTs. This does not cause problems with intevening swapouts, unless the stored timestamps are compared with the current time; for example, to implement data expiry in DHTs, such comparisons are made.
4. *To measure packet round trip times (RTT).* For example, Chord/DHash, Bamboo and i3 do this. A swapout can result in the application perceiving very large RTT values, affecting application behavior.

B.2 Impact of swapout

Table B.1 summarizes how the use of wall clock time affects the studied distributed systems, across intervening swapouts. By "impact on functionality," we mean that the system may not work correctly across a swapout (owing to time); whereas, by "impact of fidelity," we mean that the behavior of the system may be altered, but will not be incorrect. We will now discuss the rationale for the conclusions made in Table B.1.

All of the DHT systems store data for an agreed-upon time interval, after which the data is discarded [12]. Clients who want longer storage periods should periodically ask for extension; i.e., refresh. The data expiry timeout is implemented using wall clock time (scenario 2 above), and can thus be shortened by an interleaving swapout. A client (of course, within the same stateful swapping experiment) who refreshes data based on the monotonic clock will think that the DHT system has discarded the data earlier than promised – a violation of DHT's data durability guarantees.

We suspect that the use of leases should not be affected by swapouts as long as the client and server use the same clock source (wall-clock vs. monotonic) – something that cannot be guaranteed if the client is a third-party entity, like in DHTs.

Unfortunately, leases pose problem in the Linux implementation of NFS (found in the Linux kernel source 2.6.18) too. NFSv4 uses leases for locking of files and delegation of file ownership to clients. Client are supposed to renew leases to avoid expiry and they track the lease time (function `nfs4_renew_state()`) using the ticks since boot – the `jiffies` variable – which is transparent to swapouts. However, the server tracks lease intervals (function `nfs4_laundromat()`) using the real time – the `xtime` variable – which will be affected by swapouts.

While we believe that a swapout should not affect the correctness of BitTorrent (and we have

Table B.1. Effect of time on popular distributed systems across intervening swapouts.

	Scenarios Employed	Effect of Intervening Swapout	Use of TSC
BitTorrent [10]	(1), (2)	Potential impact on fidelity	No
NFS (version 4) [6]	(1), (3)	Potential impact on functionality	No
Chord/DHash [12]	(1), (2), (3), (4)	Potential impact on functionality	No
Bamboo [37]	”	”	Yes
Pastry/PAST [38]	”	”	No
distcc [16]	(1)	Impacts only the reported statistics	No
i3 [43]	(1), (2), (4)	Potential impact on fidelity	No

also confirmed this experimentally), it can skew the performance fidelity, owing to its use of scenario 2 above. For example, BitTorrent uses timeouts for periodically choking [10] peers, for throttling download/upload rates, etc. Owing to a swapout, these timeouts can expire prematurely and cause peers to alter their behaviour – for example, choosing a different peer to download from.

The fidelity of i3 can also be affected by swapouts. i3 clients calculate the proximity of servers, in order to pick the closest server(s) [43]. It does so by sending periodic ping packets and measuring the RTT – scenario 4 above. So, an i3 client's choice of server(s) can be affected by swapouts, although this is arguably a rare scenario.

Distcc is the only exception which seems to use wall clock time exclusively for diagnostic purposes. As a result, its behavior should not be affected by swapouts.

B.3 Conclusion

Based on this study, we have reason to believe that time virtualization can be important for distributed applications run in Emulab.

REFERENCES

[1] 3PAR. 3PAR Virtual Copy. Retrieved 01-April-2008 from http://www.3par.com/documents/3PAR-vc-br-06.0.pdf.

[2] AGUILERA, M. K., SPENCE, S., AND VEITCH, A. Olive: distributed point-in-time branching storage for real systems. In *NSDI'06: Proceedings of the 3rd Conference on 3rd Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 27–27.

[3] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and control in gray-box systems. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 43–56.

[4] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BAIRAVASUNDARAM, L. N., DENEHY, T. E., POPOVICI, F. I., PRABHAKARAN, V., AND SIVATHANU, M. Semantically-smart disk systems: past, present, and future. *SIGMETRICS Perform. Eval. Rev. 33*, 4 (2006), 29–35.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), pp. 164–177.

[6] BRIAN PAWLOWSKI AND SPENCER SHEPLER AND CARL BEAME AND BRENT CALLAGHAN AND MICHAEL EISLER AND DAVID NOVECK AND DAVID ROBINSON AND ROBERT THURLOW. The NFS Version 4 Protocol. *Proceedings of the 2nd International System Administration and Networking Conference (SANE2000)* (2000), 94.

[7] BURTSEV, A., RADHAKRISHNAN, P., HIBLER, M., AND LEPREAU, J. Time-travel in closed distributed systems. Poster at the Third Symposium on Networked Systems Design and Implementation (NSDI '06), May 2006.

[8] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The collective: a cache-based system management architecture. In *NSDI'05: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 19–19.

[9] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 20–20.

[10] COHEN, B. Incentives build robustness in bittorrent. Tech. rep., bittorrent.org, 2003.

[11] CORNELL, B., DINDA, P. A., AND BUSTAMANTE, F. E. Wayback: a user-level versioning file system for linux. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), USENIX Association, pp. 27–27.

[12] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (October 2001).

[13] EMC CORPORATION. EMC SnapView. Retrieved 01-April-2008 from http://www.emc.com/pdf/products/clariion/SnapView2_DS.pdf.

[14] EMC CORPORATION. EMC SRDF. Retrieved 01-April-2008 from http://www.emc.com/products/product_pdfs/ds/srdf_ds_l523-7.pdf.

[15] GNU/LINUX. Bonnie++. Retrieved 01-April-2008 from http://sourceforge.net/projects/bonnie/.

[16] GNU/LINUX. distcc. Retrieved 01-April-2008 from http://distcc.samba.org/.

[17] GNU/LINUX. VServer. Retrieved 01-April-2008 from www.linux-vserver.org.

[18] GUPTA, A. Rsnap: Recursive writable snapshots for logical volumes. Master's thesis, 2006.

[19] GUPTA, D., YOCUM, K., MCNETT, M., SNOEREN, A. C., VAHDAT, A., AND VOELKER, G. M. To infinity and beyond: time-warped network emulation. In *NSDI'06: Proceedings of the 3rd Conference on Symposium of Networked Systems Design & Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 7–7.

[20] HEINZ MAUELSHAGEN, REDHAT. LVM2 Talk. Retrieved 01-April-2008 from http://people.redhat.com/h̃einzm/talks.

[21] HIBLER, M., STOLLER, L., LEPREAU, J., RICCI, R., AND BARB, C. Fast, scalable disk imaging with frisbee. In *Proc. of the 2003 USENIX Annual Technical Conf.* (San Antonio, TX, June 2003), USENIX Association, pp. 283–296.

[22] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 19–19.

[23] III, C. B. M., AND GRUNWALD, D. Peabody: The time travelling disk. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 241.

[24] K. FRASER AND S. HAND AND R. NEUGEBAUER AND I. PRATT AND A. WARFIELD AND M. WILLIAMSON. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Workshop on Operating System and Architectural Support for On-Demand IT Infrastructure* (2004).

[25] KOTSOVINOS, E., MORETON, T., PRATT, I., ROSS, R., FRASER, K., HAND, S., AND HARRIS, T. Global-scale service deployment in the xenoserver platform. In *Proceedings of the First Workshop on Real, Large Distributed Systems* (2004).

[26] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[27] LEE, E. K., AND THEKKATH, C. A. Petal: distributed virtual disks. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ACM Press, pp. 84–92.

[28] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2004), USENIX Association, pp. 115–128.

[29] OPENSOLARIS. ZFS at OpenSolaris.org. Retrieved 01-April-2008 from http://www.opensolaris.org/os/community/zfs.

[30] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), ACM Press, pp. 109–116.

[31] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I* (Princeton, New Jersey, October 2002).

[32] PETERSON, Z., AND BURNS, R. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage 1*, 2 (2005), 190–212.

[33] PHILLIPS, D. Zumastor Linux Storage Server. In *Ottawa Linux Symposium* (June 2007).

[34] QUINLAN, S., AND DORWARD, S. Awarded best paper! - venti: A new approach to archival data storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), USENIX Association, p. 7.

[35] REDHAT. Device-Mapper Resource Page. Retrieved 01-April-2008 from http://sources.redhat.com/dm/.

[36] REDHAT. LVM2 Resource Page. Retrieved 01-April-2008 from http://sourceware.org/lvm2.

[37] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a dht. In *Proc. of the 2004 USENIX Annual Technical Conf.* (2004), USENIX Association.

[38] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (October 2001).

[39] SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., AND VEITCH, A. C. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems* (1999), pp. 2–7.

[40] SATYANARAYANAN, M., KOZUCH, M. A., HELFRICH, C. J., AND O'HALLARON, D. R. Towards seamless mobility on pervasive hardware.

[41] SNIA. CDP Buyer's Guide. Retrieved 01-April-2008 from http://www.snia-dmf.org/library/DPI_BuyersGuide_WEB_20070412.pdf.

[42] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *EuroSys '07: Proceedings of the 2007 Conference on EuroSys* (New York, NY, USA, 2007), ACM Press, pp. 275–287.

[43] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet indirection infrastructure. In *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2002), ACM Press, pp. 73–86.

[44] SUN MICROSYSTEMS, INC. NFS: Network File System Protocol Specification. Retrieved 01-April-2008 from http://www.ietf.org/rfc/rfc1094.txt.

[45] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C., ANDERSON, A. V., BENNETT, S. M., KAGI, A., LEUNG, F. H., AND SMITH, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[46] VERITAS SOFTWARE. VERITAS FlashSnap. Retrieved 01-April-2008 from http://eval.veritas.com/downloads/news/ESG-FlashSnap.pdf.

[47] VERITAS SOFTWARE. VERITAS Volume Replicator. Retrieved 01-April-2008 from http://www.bigvent.pl/files/vvr_white_paper.pdf.

[48] VMWARE, INC. Virtual Appliances - Overview. Retrieved 01-April-2008 from http://www.vmware.com/appliances/learn/overview.html.

[49] WARFIELD, A., ROSS, R., FRASER, K., LIMPACH, C., AND HAND, S. Parallax: managing storage for a million machines. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 4–4.

[50] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[51] XENSOURCE. Xen Blktap Qcow Disks. Retrieved 01-April-2008 from http://lxr.xensource.com/lxr/source/tools/blktap/README.

[52] ZHAO, M., ZHANG, J., AND FIGUEIREDO, R. Distributed file system support for virtual machines in grid computing. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 202–211.