

Generalized High Availability via Virtual Machine Replication

by

Brendan Cully

B.Sc., New York University, 2001

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

October, 2007

© Brendan Cully 2007

Abstract

Allowing applications to survive hardware failure is a expensive undertaking, which generally involves re-engineering software to include complicated recovery logic as well as deploying special-purpose hardware; this represents a severe barrier to improving the dependability of large or legacy applications. We describe the construction of a general and transparent high-availability service that allows existing, *unmodified* software to be protected from the failure of the physical machine on which it runs. *Remus* provides an extremely high degree of fault tolerance, to the point that a running system can transparently continue execution on an alternate physical host in the face of failure with only seconds of downtime, completely preserving host state such as active network connections. We describe our approach, which encapsulates protected software in a virtual machine, asynchronously propagates VM state to a backup host at frequencies as high as forty times a second, and uses speculative execution to concurrently run the active VM slightly ahead of the replicated system state.

Contents

Abstract	ii
Contents	iii
List of Figures	v
Acknowledgements	vi
1 Introduction	1
1.1 Goals	3
1.2 Approach	4
1.2.1 Virtual machine replication	5
1.2.2 Speculative execution	6
1.2.3 Asynchronous replication	7
2 Design	8
2.1 Failure model	11
2.2 Pipelined checkpoints	12
2.3 Master control program	13
2.4 Memory and CPU	13
2.4.1 Migration enhancements	14
2.4.2 Checkpoint support	18
2.4.3 Asynchronous transmission	19
2.4.4 Guest modifications	19
2.5 Network buffering	20

2.6	Disk buffering	22
2.7	Detecting failure	26
3	Evaluation	27
3.1	Test environment	28
3.2	Failover performance	29
3.3	Benchmarks	29
3.3.1	Kernel compilation	29
3.3.2	SPECweb2005	31
3.3.3	Postmark	33
3.4	Potential optimizations	36
3.4.1	Deadline scheduling	36
3.4.2	Page compression	37
3.4.3	Copy-on-write checkpoints	41
4	Related Work	42
4.1	Virtual machine migration	42
4.2	Virtual machine logging and replay	43
4.3	Operating system replication	44
4.4	Library approaches	44
4.5	Replicated storage	44
4.6	Speculative execution	45
5	Future Work	46
6	Conclusion	49
	Bibliography	51

List of Figures

1.1	Speculative execution and asynchronous replication.	5
2.1	High-level architecture.	10
2.2	Normal suspend path.	17
2.3	Accelerated suspend protocol.	18
2.4	Network buffering.	21
2.5	Disk write buffering.	23
3.1	Kernel build times at various checkpoint frequencies.	30
3.2	SPECweb performance at various checkpoint frequencies	32
3.3	The effect of network latency on SPECweb performance.	34
3.4	The effect of disk replication on Postmark performance.	35
3.5	SPECweb2005 checkpoint latency versus pages dirtied per epoch.	38
3.6	Checkpoint latency versus dirty rate	39
3.7	Effectiveness of page compression.	40

Acknowledgements

I am deeply indebted to my supervisors, Mike Feeley and Andrew Warfield. Their wisdom, humour and encouragement were an inspiration, as was the seemingly boundless generosity with which they offered their time. I would also like to extend my sincere thanks to Norm Hutchinson for his enthusiasm and support, and for being my second reader. Thanks as well to the distributed systems group, and in particular Geoffrey Lefebvre, Dutch Meyer and Kan Cai.

Chapter 1

Introduction

Highly available systems are the purview of the very rich and the very scared. However, the desire for reliability is pervasive, even among system designers with modest resources.

Unfortunately, high availability is hard — it requires that systems be constructed with redundant components and that they be able to both detect component failure and seamlessly activate its backup. Commercial high availability systems that aim to protect modern servers generally use specialized hardware, customized software, or both (e.g., [13]). In each case, the technology required in order to be able to transparently survive failure is complex and expensive enough that it prohibits deployment on common servers. To better support the increasing use of network-based applications and systems, it is desirable to deliver the level of availability traditionally associated only with higher-end installations to those based on commodity hardware and software.

This thesis describes *Remus*, an entirely software-based system that provides high availability on commodity hardware without operating system or application-specific knowledge. Its approach capitalizes on the demonstrated ability of virtualization to migrate active, running virtual machines between physical hosts [7], extending that technique to replicate snapshots of an entire running system at *very* high frequencies — as often as once every 25ms — between a pair of physical machines. Using this technique, our system discretizes the execution of a virtual machine into a series of replicated step-wise snapshots. Externally-visible events, in particular transmitted outbound network packets, are not released until the system state responsible for their generation has been

replicated.

The contribution of this thesis is a practical one. Whole-system replication is a well-known approach to providing high availability. However, it is commonly considered to be significantly more expensive than checkpointing techniques that use application-specific knowledge in order to replicate only relevant data [16]. Unfortunately, such approaches require careful coding for every protected application, making them unsuitable for rapid development or for the use of off-the-shelf applications. Attempts have been made to implement somewhat more general protocols in library code, but application developers must be careful to observe the constraints of the framework they choose. When they don't, replication may silently fail or worse, the replica may become inconsistent. Operating system-level techniques such as process migration solve some of these problems, but they are only possible when the operating system source code is available. They are difficult to implement and must be rigorously analyzed for correctness with each revision of the underlying operating system. For these reasons, we have revisited machine-level replication. In this thesis, we demonstrate a system constructed on commodity hardware which uses whole-system replication to transparently survive host failures without requiring modifications to application or operating system code. This approach may be used to bring high availability “to the masses” as a service offered to virtual machines by the virtualization platform.

Virtualization makes it *possible* to create a copy of a running machine, but it does not guarantee that the process will be *efficient*. Synchronously propagating state at every change is impractical: it effectively reduces the throughput of local state to that of the network device performing replication. Therefore, rather than running two hosts in lock-step [4] we allow a single host to execute *speculatively* and then checkpoint and replicate its state *asynchronously*. Externally visible state is not made visible until a checkpoint has been committed on a backup host — we achieve high-speed replicated performance by effectively running the system tens of milliseconds in the past.

In spite of the hardware and software constraints under which we have delib-

erately designed this system, it is able to provide protection equal to or better than significantly more expensive commercial offerings. Many existing systems only actively mirror persistent storage, requiring applications to perform recovery from crash-consistent persistent state. In contrast, Remus ensures that regardless of the moment at which the primary fails, no externally-visible state is ever lost.

The remainder of the thesis presents the architecture of our system and discusses the challenges that were involved in capturing high-frequency checkpoints of running VMs on commodity hardware. The resulting system is completely functional and allows high availability to be offered as a service by the virtualization platform, retrofitting dependability onto existing software.

1.1 Goals

When designing Remus, we first considered the characteristics of an idealized high availability solution targeting mid- to low-end systems. We believe that such a system must focus on the following guiding principles:

Generality - Performing customization of hardware, operating systems, libraries and applications is costly and difficult. For resource-constrained organizations, it can be prohibitively expensive. Therefore, any modification required to common off-the-shelf and open-source software and hardware should be minimized, or if possible, eliminated entirely.

Transparency - The natural granularity at which to provide redundancy is at the level of whole-machine state. Operating at finer granularity is (from a purely functional perspective) both unnecessary and complicated, because it forces designers to determine what aspects of whole-system state (e.g., file descriptors) must additionally be restored in the event of an error, and to both capture and recreate this state wherever it may reside in the host. This requires intimate cooperation between the provider of redundancy and the operator of the system to be protected. It would be preferable to be able to separate these two entities into separate administrative domains, making high availability as

simple to add to an existing server as, for example, a new hard drive.

Reasonable performance - The system should not consume excessive resources, and should run on commodity equipment. While it is unrealistic to expect a system to provide high availability without some overhead, the additional resources required to provide performance equivalent to unprotected operation should be no more than some small factor of those of the protected system. In short, the system should be deployable for real workloads.

Multiprocessor support - Concurrent multi-processors are now the norm for data centers. Therefore approaches in which overheads increase in proportion to the degree of multiprocessing should be considered detrimental to the success of any practical high availability solution.

Seamless failure recovery - No externally visible state should ever be lost in the case of single-host failure. Furthermore, failure recovery should proceed rapidly enough that it appears as nothing more than temporary packet loss from the perspective of external users. Established TCP connections should not be lost or reset.

1.2 Approach

Remus is able to achieve the above goals by replicating a host in a primary-backup fashion. We employ three major techniques in order to overcome the difficulties traditionally associated with this approach. First, we base our system on a virtualized infrastructure to facilitate whole-system replication. Second, we increase system performance through speculative execution, which decouples external output from synchronization points. Finally, asynchronous replication enables the primary server to remain productive, even as synchronization with the replicated server is occurring. The basic stages of operation in Remus are given in Figure 1.1.

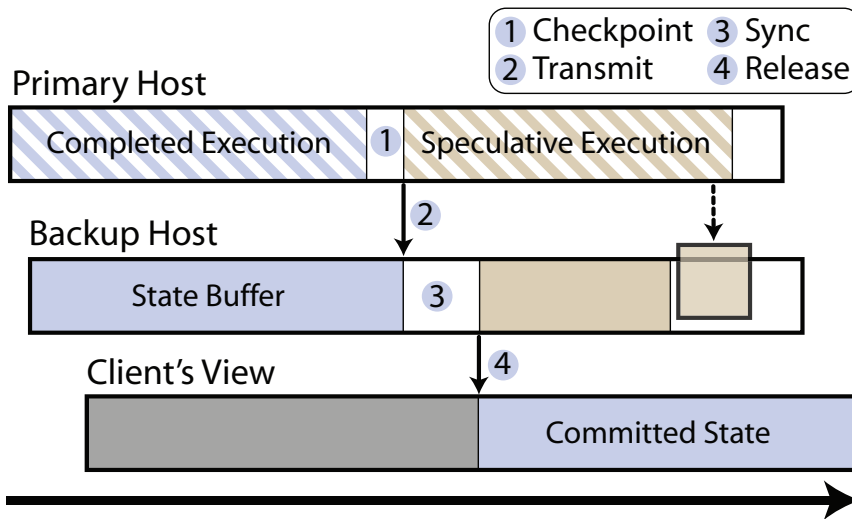


Figure 1.1: Speculative execution and asynchronous replication.

1.2.1 Virtual machine replication

Hypervisors have been considered as a mechanism for building high availability systems in the past [4]. In this previous work, virtualization has been used to run a pair of systems in lock-step, ensuring that virtual machines on a pair of physical hosts follow an identical, deterministic path of execution. Machine state changes very rapidly; attempting to synchronously propagate every change to local memory and CPU state between hosts would be wildly impractical. To reduce the required bandwidth between the primary and the backup server to manageable levels, some systems choose to replicate only the input events themselves, and then to replay them on the backup host. This design leverages the natural bandwidth and latency symmetry between input and output channels.

Unfortunately, it is not easy to log events with enough precision that they will produce repeatable output when replayed – it is much easier to virtualize a system than to make it execute deterministically. Interrupt jitter may change the order in which processes are scheduled. Multiple processors may access shared memory in different order due to clock skew. It is a difficult task (and one which must be performed again at almost every hardware revision) even

to log the exact point in an execution stream at which a given external input is delivered; it is harder still to reproduce that exact moment during replay. There have been some impressive attempts to do this [10, 14], but they do not replay in real time. Very few have attempted the considerably harder problem of replaying an SMP system [9], and none that we know of have achieved it without overheads that often make multiprocessor performance *worse* than uniprocessor.

Systems based upon event replay also require that the backup virtual machine be continuously active: all computation performed by the primary host must be executed again on the backup in order to bring it to the same state. Replicating the state itself avoids this requirement, allowing the backup to operate as a passive state recorder which consumes few resources. This permits the construction of n-to-1 redundant systems, allowing the system designer a high degree of freedom to balance resource expenditure against failure protection.

1.2.2 Speculative execution

Due to the difficulties involved in performing deterministic machine replay discussed in the previous section, particularly in the case of multi-processor systems, Remus has been designed to tolerate non-deterministic computation. There is a very real possibility that the output produced by a system after a given checkpoint will be different if the system is rolled back to that checkpoint and its input is replayed. Therefore, Remus requires that the backup be synchronized with the primary at all times, *from the perspective of an external viewer*. This means that the state of the replica must be synchronized with that of the primary only when the output of the primary has become externally visible. Instead of letting the normal output stream dictate when synchronization must occur, we can buffer the output until a more convenient time, performing computation *speculatively* ahead of synchronization points. This allows a favorable trade-off to be made between output latency and runtime overhead, the degree of which may be controlled by the administrator.

Remus is particularly concerned with network and disk output. Other de-

vices, such as the console or the serial port, are presumed to be used for local administration and therefore would not require buffering. However, nothing intrinsically prevents these devices from being buffered as well.

1.2.3 Asynchronous replication

Buffering output at the primary server allows replication to be performed *asynchronously*. The primary host can resume execution at the instant that a consistent checkpoint of its machine state has been captured, without waiting for acknowledgment from the remote end. Any output associated with the checkpoint is simply queued until it has committed. Overlapping normal execution with the replication process yields substantial performance benefits, and permits reasonably efficient operation even when checkpointing at intervals on the order of tens of milliseconds.

Chapter 2

Design

Figure 2.1 shows a high-level view of our system. We begin by encapsulating the machine to be protected within a virtual machine. Our implementation is based on the Xen virtual machine monitor [2], and extends Xen’s support for live migration to provide fine grained checkpoints. An initial subset of our checkpointing support has been accepted into the upstream Xen source.

Remus achieves high availability by propagating frequent checkpoints of an *active* virtual machine to a *backup* physical host. On the backup, the virtual machine image is resident in memory and may begin execution immediately if failure of the active system is detected. Because the backup is only periodically consistent with the primary, all network output must be buffered until state is synchronized on the backup. When a complete, consistent image of the host has been received, this buffer is released to external clients. The checkpoint, buffer, and release cycle can be performed very frequently. We have tested it against heavy workloads at frequencies as high as forty times a second, representing a whole-machine checkpoint of network and on-disk state every 25 milliseconds.

Unlike transmitted network traffic, disk state is not externally visible. It must, however, be propagated to the remote host as part of a complete and consistent snapshot. In order to achieve replication of block data, all writes to the primary disk are transmitted asynchronously to the backup, where they are buffered in RAM until the corresponding memory checkpoint has arrived. At that point, the complete checkpoint is acknowledged to the primary, which then releases outbound network traffic, and the disk writes are applied from the buffer to the backup disk.

It is worth emphasizing that the virtual machine does not actually execute on the backup host until a failure occurs. It simply acts as a receptacle for checkpoints of the active virtual machine. As a consequence, only a small amount of resources are used on the backup host, allowing it to concurrently protect virtual machines running on multiple physical hosts in an n-to-1-style redundant configuration. This provides administrators with the flexibility to balance redundancy and resource costs when provisioning high availability.

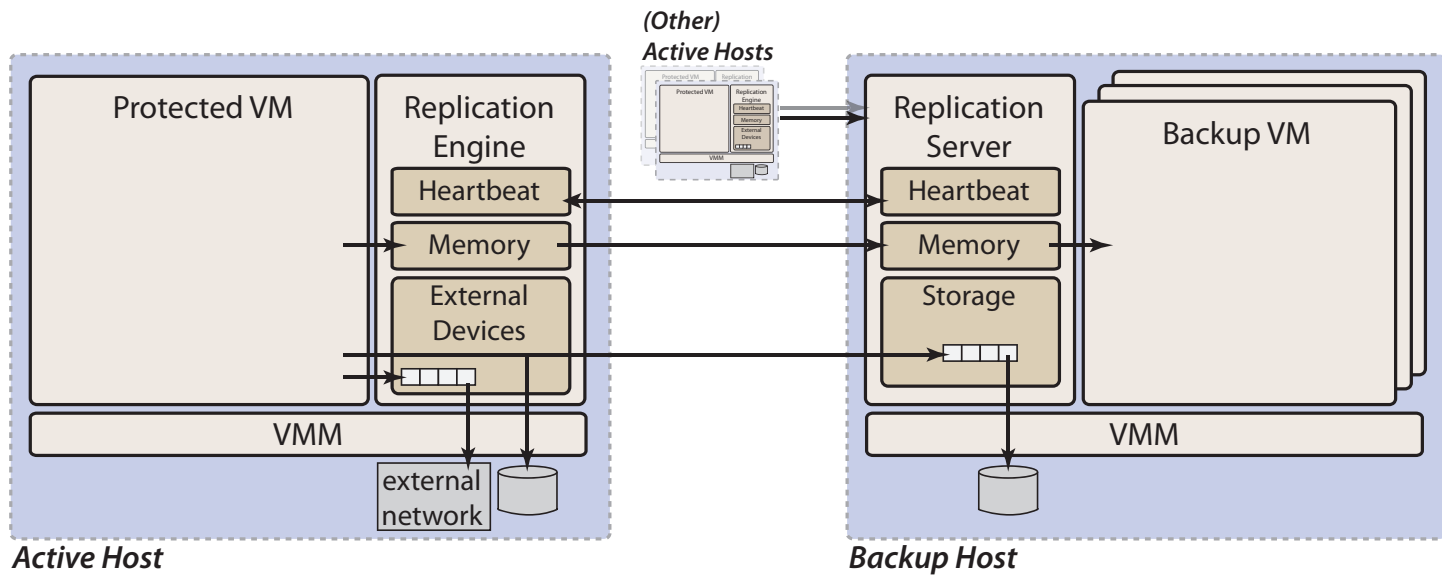


Figure 2.1: High-level architecture.

2.1 Failure model

This system provides the following properties:

1. The fail-stop failure of any single host is tolerable.
2. Should both the primary and backup hosts fail concurrently, the protected system's data will be left in a crash-consistent state.
3. No state generated in speculation will be made externally visible until it has been acknowledged as committed to the backup.

Remus aims to provide completely transparent recovery from fail-stop failures of a single physical host. The compelling aspect of this system is that high-availability may be easily retrofitted onto existing software running on commodity hardware. The system uses a pair of commodity host machines, connected over redundant gigabit Ethernet connections, and survives the failure of any one of these components. It does not require the use of expensive, shared, network-attached storage for disk images, because it includes block device requests in the state replication stream.

We do not aim to recover from software errors or non-fail-stop conditions. As observed in [5], the methodology we have chosen provides complete system state capture and replication, and as such will propagate application errors to the backup. This is a necessary consequence of providing both transparency and generality.

Our failure model is identical to that of commercial high availability products, which provide protection for virtual machines today [31, 32]. However, the degree of protection offered by these products is substantially less than that provided by Remus: existing commercial products respond to the failure of a physical host by simply rebooting the virtual machine on another host from its crash-consistent disk state. Our approach survives failure on time frames similar to those of live migration, and leaves the virtual machine running and network connections intact. Exposed transient state is not lost and file system consistency is preserved.

2.2 Pipelined checkpoints

Checkpointing a running virtual machine at rates of up to forty times a second places extreme demands on the checkpoint system. Remus addresses this by aggressively pipelining the checkpoint operation. We use an epoch-based system in which execution of the active virtual machine is bounded by brief pauses in execution during which dirty pages are written to a local buffer. Concurrently with the execution of the virtual machine, this buffer is replicated to the backup host, which acknowledges a checkpoint when the buffer has been received in its entirety. Finally, the backup may apply new memory and CPU state to the local virtual machine image, and write out any disk blocks associated with committed checkpoints.

Referring back to Figure 1.1, this procedure can be divided into four stages: (1) Once per epoch, pause the running VM and copy any changed state into a buffer. This process is effectively the stop-and-copy stage of live migration [7], but as described later in this section it has been dramatically optimized for high-frequency checkpoints. With state changes preserved in a buffer, the virtual machine is unpaused and speculative execution resumes. (2) Buffered state is transmitted and stored in memory on the backup host. (3) Once the complete set of state has been received, the checkpoint is acknowledged to the primary. Finally, (4) buffered network output is released.

The result of this approach is that execution is effectively discretized at checkpoint boundaries; the acknowledgment of a completed checkpoint by the backup triggers the release of network traffic that has been buffered and represents an atomic transition into the new epoch.

If failure of the primary node is detected, execution resumes on the backup from the point of the most recently committed checkpoint. Likewise, if the backup fails, then the primary node disables replication and reverts to unprotected execution. The following sections describe the replication control process, and then describe in more detail the nuances of managing the memory, network and disk subsystems.

2.3 Master control program

Virtual machine failure protection is initiated by a command-line tool called *protect*, which first synchronizes initial disk state between the primary and the backup, and then starts the primary virtual machine, engaging the network and disk buffering subsystems described in the following sections. After opening a control channel to the disk buffer, it spawns a customized version of the live migration process, and then enters a loop in which it requests checkpoints at an interval requested by the operator. When the migration process receives a checkpoint request, it suspends the guest domain and then notifies *protect*, which then issues a checkpoint message to the disk buffer as described in Section 2.6. When the disk buffer has completed the checkpoint it notifies the control process, which then sends a message to the network buffer to release packets queued from the now-completed epoch. This cycle is repeated until the domain terminates or a failure event is detected.

2.4 Memory and CPU

Checkpointing is implemented above Xen's existing machinery for performing live migration [7]. Live migration is a technique by which a virtual machine is relocated to another physical host with only slight interruption in service. To do this, memory is copied to the new location while the virtual machine continues to run at the original location. During migration, writes to memory are intercepted, and dirtied pages are copied to the new location in rounds. After a specified number of rounds, or when no forward progress is being made (because the virtual machine is writing to memory at least as fast as the migration process can copy it out), the guest is suspended and the remaining dirty memory is copied out along with the current CPU state. At this point the image on the new location is activated. Total downtime depends on the amount of memory remaining to be copied when the guest is suspended, but is typically under 100 milliseconds. Total migration time is a function of the amount of memory in

use by the guest and the size of its *writable working set* [7].

During live migration, writes to virtual memory are tracked by the hypervisor, using a mechanism called *shadow page tables*. When this mode is activated, the virtual machine monitor maintains a private (“shadow”) version of the guest’s page tables and exposes these to the hardware memory management unit. Hardware page protection is used to trap guest access to its internal version of page tables. This allows the hypervisor to track page table updates, which it propagates to the shadowed versions as appropriate before returning control to the guest.

To provide migration, this technique is extended to transparently (to the guest) mark all virtual machine memory as read only. The hypervisor is then able to trap all writes that a virtual machine makes to memory in order to maintain a map of pages that have been dirtied since the previous round. Each round, the migration process atomically reads and resets this dirty map, and the iterative migration process involves chasing dirty pages until progress can no longer be made. As mentioned above, the live migration process eventually suspends execution of the virtual machine and enters a final “stop-and-copy” round, during which any remaining pages are transmitted before execution is resumed on the destination host.

Remus implements checkpointing as repeated executions of the final stage of live migration: at the end of each epoch, the guest is paused and updated memory and CPU state are copied to a buffer. The guest then resumes execution on the current host, rather than on the destination. Several modifications to the migration process are required in order to provide sufficient performance and to ensure that a consistent image is always available at the remote location. These are described below.

2.4.1 Migration enhancements

In live migration, guest memory is iteratively copied over a number of rounds and may consume minutes of execution time; the brief service interruption

caused by the singular stop-and-copy phase is not a significant component of total migration overhead. This is not the case when capturing frequent virtual machine checkpoints: *every* checkpoint is just the final stop-and-copy phase of migration, and so this represents a critical point of optimization in reducing checkpoint overheads. An examination of Xen’s checkpoint code revealed that the majority of the time spent while the guest is in the suspended state is lost to scheduling, largely due to inefficiencies in the implementation of the xenstore daemon that provides administrative communication between guest virtual machines and domain 0.

For the final round of memory copying, the guest must be suspended to create a consistent image. Paravirtual guests perform several steps in addition to pausing themselves before the final round¹, including disconnecting devices and the connection to the Xenstore registry, and preparing a suspend record for use during restore, including information about the location of pages shared with Xen and the state of the guest’s VCPUs. Thus suspending the guest is a cooperative process in which the migration process requests that the guest suspend itself, then awaits notification that it has done so. This procedure, depicted in Figure 2.2, is surprisingly complicated.

1. The migration process sends a message to xend via stdout, asking it to suspend the guest.
2. Xend writes an entry into the “control/shutdown” node in the guest’s portion of xenstore, on which the guest has registered a watch.
3. Xenstore fires the watch in the guest via an event channel.
4. The guest reads the contents of the watch and then calls its suspend routine.

¹Paravirtual Xen guests contain code specifically for suspend requests that are responsible for cleaning up Xen-related state such as shared memory mappings used by virtual devices. In the case of hardware-virtualized (e.g., Windows) VMs, this state is completely encapsulated by Xen’s device model, and these in-guest changes are unnecessary.

5. At the end of the suspend function, the guest issues a suspend hypercall, which will not return control to the guest until dom0 issues a hypercall requesting it.
6. After the monitor has paused the domain, it sends a notification to xenstore that a domain has changed state.
7. Xenstore iterates through each active domain until it finds one that has changed state. It then writes this information into a xenstore node on which xend has registered a watch, and fires the watch.
8. Xend reads the watch, discovers that the domain has been suspended, and notifies the migration process via stdout. At this point it may also perform other suspend-related activities, such as invoking external scripts to checkpoint the state of attached disks or other devices.

This process requires cooperation from several independent processes. Some of these processes (particularly xenstore, but also xend) can introduce large amounts of delay, and the signalling between them can cause significant jitter due to scheduling artifacts in domain 0. Although these delays are not a serious problem for low-frequency, heavyweight operations like domain save, they form a significant portion of the downtime during live migration, and dwarf most of the other costs of frequent checkpointing. This convoluted process can take a nearly arbitrary amount of time – we frequently measured latencies in the range of 30 to 40 milliseconds, but saw delays as long as 500 milliseconds in some cases.

Remus's optimized suspend code streamlines this process by creating an event channel in the guest specifically for receiving suspend requests, which the migration process can invoke directly. Additionally, a new hypercall is provided to allow processes to register an event channel for callbacks notifying them of the completion of virtual machine suspension. The streamlined suspend process is depicted in Figure 2.3. In concert, these two notification mechanisms reduce the time required to suspend a virtual machine to about one hundred microseconds

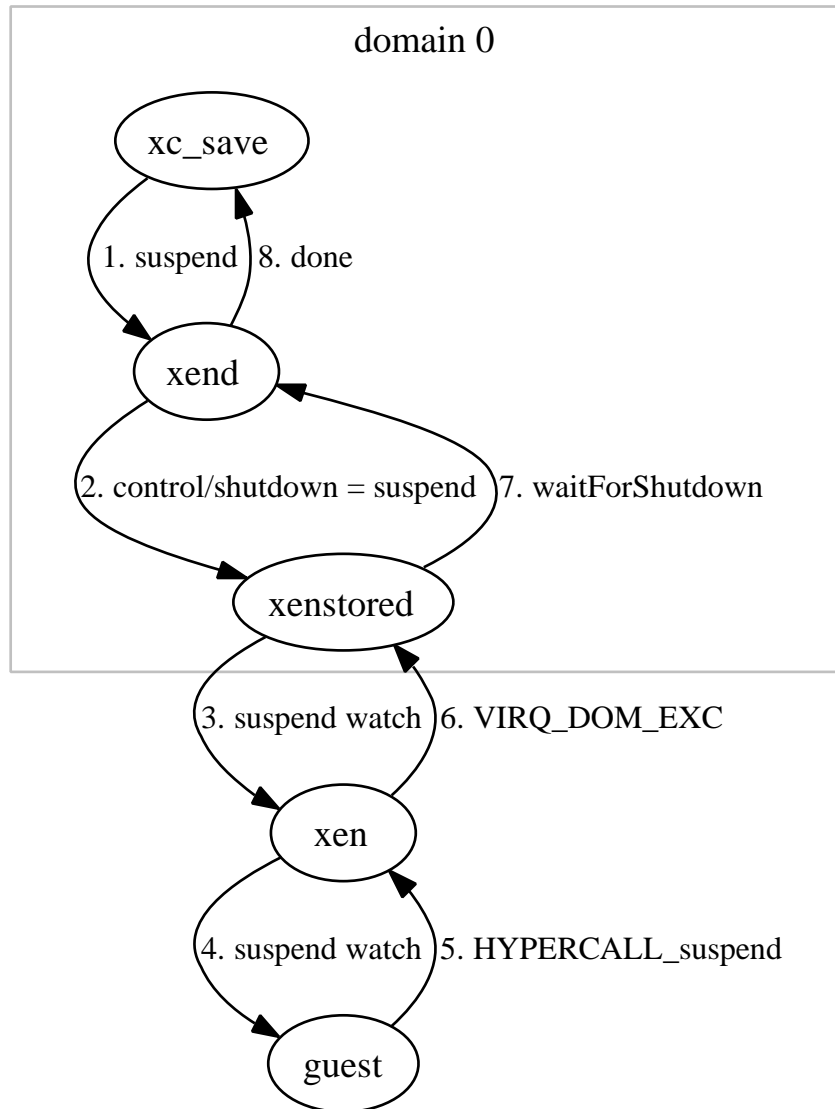


Figure 2.2: Normal suspend path.

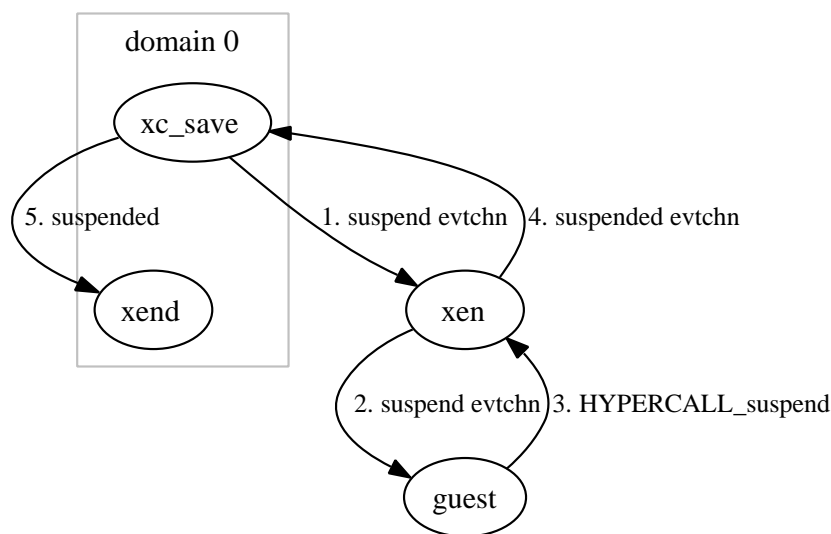


Figure 2.3: Accelerated suspend protocol.

– an improvement of two orders of magnitude over the previous implementation.

2.4.2 Checkpoint support

Providing checkpoint support in Xen required two primary changes to the existing suspend-to-disk and live migration code. First, support was added for resuming execution of a domain after it had been suspended; Xen previously did not allow “live checkpoints” and instead destroyed the virtual machine in the process of writing its state out. Second, the suspend program was converted from a one-shot procedure into a daemon process. This allows checkpoint rounds after the first to copy only newly-dirty memory.

Supporting resumption requires two basic changes. The first is a new hypercall to mark the domain as schedulable again (Xen removes suspended domains from scheduling consideration, because the only action performed on suspended domains previously was to destroy them after their state had been replicated). A similar operation is necessary in order to re-arm watches in xenstore.

2.4.3 Asynchronous transmission

One of the guiding principles of our design is that checkpointing should be as asynchronous as possible. At the frequency with which we perform checkpoints, any pause in the operation of the guest, however slight, will have an impact on its throughput². Therefore, in order to allow the guest to resume operation as quickly as possible, the migration process was modified to copy touched pages to a staging buffer rather than delivering them directly to the network while the domain is paused. This results in a significant throughput increase: the time required for the kernel build benchmark discussed in Section 3.3.1 was reduced by approximately 10% at 20 checkpoints per second.

While buffering improves performance by reducing the amount of time that a machine must be paused, it also highlights a limitation of our current implementation. Although the guest can continue to run while the buffer is being propagated, subsequent checkpoints cannot begin until the current checkpoint has completed. A larger buffer will naturally take longer to transmit; an excessively large buffer may delay the next checkpoint beyond the requested checkpoint frequency. The optimal size of the buffer is approximately $BW/(CF + T)$, where BW is the bandwidth of the replication channel, CF is the checkpoint frequency, and T is the time required to copy out state to the buffer. For example, when checkpointing every 40ms over a 1 Gbps link, and given a conservative copy-out time of 10ms, the checkpoint process should use a buffer no larger than 50Mb or 6MB.

2.4.4 Guest modifications

As discussed above, paravirtual guests in Xen contain a suspend handler that cleans up device state upon receipt of a suspend request. In addition to the notification optimizations described earlier in this section, the suspend request

²This is, in fact, the *only* part of the replication process which is truly speed-critical. Because the remaining stages of replication are performed asynchronously, latencies of a few milliseconds are tolerable. For this reason our current implementation performs reasonably well in spite of the fact that it is written in Python.

handler has also been modified to reduce the amount of work done prior to suspension. In the original code, suspension entailed disconnecting all devices and unplugging all but one CPU. This is unnecessary work when the domain is simply being checkpointed, and so it was removed. These modifications were applied to Xen for use in the case that a migration attempt or a save operation fails, and they have been available since version 3.1.0.

These changes are not strictly required for correctness, but they do improve the performance of the checkpoint noticeably, and involve very local modifications to the guest kernel. Total changes were under 100 lines of code in the paravirtual suspend handler. As mentioned earlier, these modifications are not necessary in the case of non-paravirtualized virtual machines.

2.5 Network buffering

Most networks cannot be counted on for reliable data delivery. Therefore, networked applications either accept packet loss, duplication and reordering, or they use a high-level protocol such as TCP which provides stronger service guarantees. This fact simplifies the network buffering problem considerably: transmitted packets do not require replication: their loss will appear as a transient network failure and not affect the correctness of the protected system. However, it is crucial that packets queued for transmission be held until the checkpointed state of the epoch in which they were generated is committed to the backup; if the primary fails, these generated packets reflect speculative state that is discarded, and so they must not become visible in this circumstance.

Network buffering is implemented as a linux queueing discipline, which performs buffering on the network interface representing the virtual machine in domain 0. Its operation is depicted in Figure 2.4. All outbound packets are queued until an external commit message arrives via the linux netlink protocol. When the commit message is received, the tail of the queue is marked, and packets are released to the network. When the previously-marked tail packet has been delivered, the device returns to buffering mode. This protocol allows the

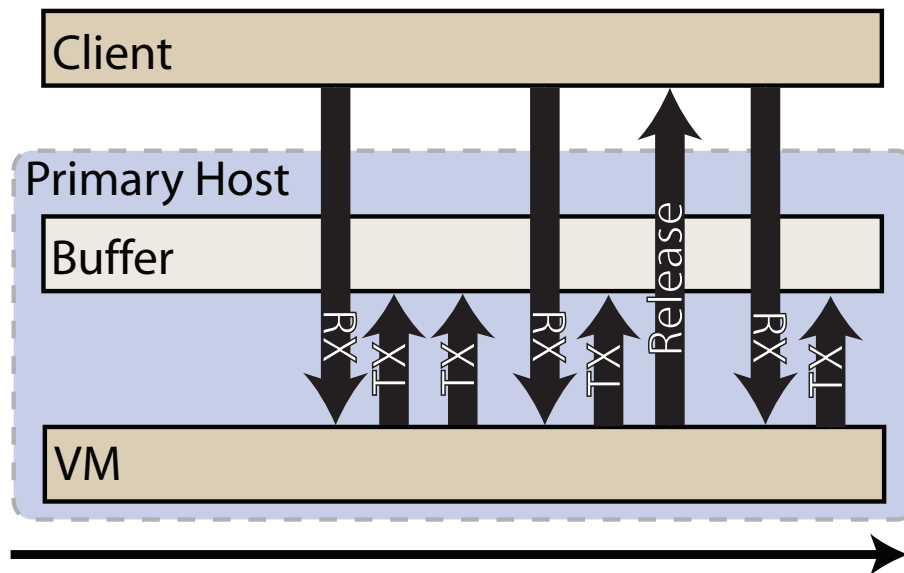


Figure 2.4: Network buffering.

flush to proceed asynchronously, with the guest permitted to use the interface while the commit proceeds.

There are two minor wrinkles in this implementation. The first is that in linux, queuing disciplines only operate on *outgoing* traffic. Under Xen, the network interface consists of two separate devices, one in the guest and one in domain 0. Outbound traffic from the guest appears as *inbound* traffic on the bridge device in domain 0. Therefore in order to queue the traffic, we must convert the inbound traffic to outbound by routing it through a special device called an *intermediate queuing device* [17]. This module is designed to work at the IP layer via iptables [28], but it was not difficult to extend it to work at the bridging layer used to provide virtual machine network access in our implementation.

The second wrinkle is due to Xen's implementation of network packet delivery between virtual machines. For performance, the memory used by outbound networking traffic is not copied between guest domains and domain 0, but rather shared through grant table mappings managed by Xen. The hypervisor only

permits a small number of pages to be shared at any one time. If messages are in transit between a guest and domain 0 for only a brief time, this limitation is not noticeable. However, the network output buffer can result in messages being in flight for a significant amount of time, with the result that the guest network device becomes blocked after a small amount of traffic has been sent. Therefore when messages are queued, the driver first copies them into local memory and then releases the local mappings to shared data.

2.6 Disk buffering

Disks present a rather different challenge than do networks in providing consistent state replication. While Remus is designed to recover from a *single* host failure, it must preserve the consistency semantics that the guest expects of persistent storage even if *both* hosts should fail. Moreover, the goal of providing a general-purpose system precluded the use of expensive mirrored storage hardware designed for high-availability applications. Instead, Remus maintains a complete mirror of the active virtual machine's disks on the backup host. Prior to engaging the protection system, the current state of the disk on the primary is mirrored to the backup host. Once protection has been engaged, writes to persistent storage are tracked and checkpointed similarly to updates to memory. Figure 2.5 gives a high-level overview of the disk replication mechanism

Much like memory updates, writes to disk from the speculative virtual machine are treated as write-through: they are immediately applied to the local disk image, and concurrently mirrored to an in-memory buffer on the backup. This approach provides two direct benefits: First, it ensures that the local disk image remains crash-consistent at all times; in the case of that both hosts fail, the local disk will reflect the crashed state of the active virtual machine at the time of failure. Second, writing directly to disk accurately represents the latency and throughput required to talk to the physical device. This obvious-seeming property is of considerable value; Remus first began with an in-memory buffer which encountered some difficulty accurately characterizing disk responsiveness.

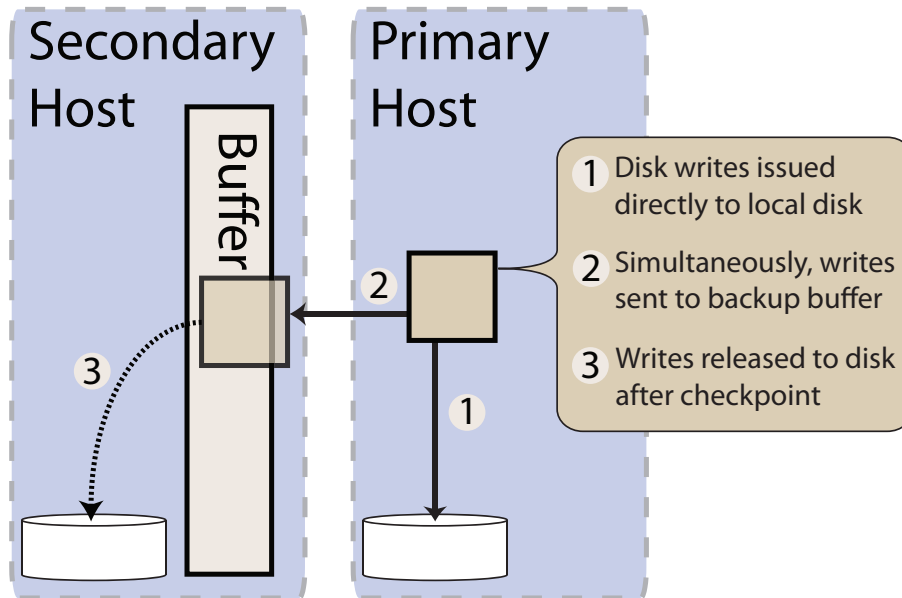


Figure 2.5: Disk write buffering.

Earlier implementations either buffered writes, under-representing the time required to commit data to disk and allowing the speculating virtual machine to race ahead of real persistent storage, or conservatively overestimated write latencies which resulted in a loss of performance. Modeling disk response time is notoriously challenging [29]; Remus avoids this by using the disk itself as a model.

At the time the backup acknowledges a checkpoint as committed, disk updates reside completely in memory. The backup must not overwrite the on-disk state produced by any speculative execution, because this would prevent it from recovering the state of the disk at the time of the last committed checkpoint. Once a checkpoint commit has been received, the disk request buffer may be applied to disk. In the event of a failure event at the primary host, Remus will wait until all buffered writes have been applied before resuming execution. Although the backup could begin execution immediately using the request buffer as an overlay on the physical disk, this would violate the disk semantics presented to

the protected virtual machine: A crash of the backup after such a resumption and before data had completed writing out to disk could potentially result in inconsistent on-disk state.

Only one of the two disk mirrors managed by Remus is valid at any given time. In the case that both the primary and the backup hosts fail, this point becomes critical. This property is provided through the use of a persistent on-disk assertion, which indicates at all times which physical disk is active. After flushing buffered writes to disk and *before* resuming execution of the protected VM, Remus writes a timestamped activation record asserting that the local copy of the disk is current and consistent. When recovering from the failure of both the primary and the backup, this assertion may be used to identify which disk contains the valid, crash-consistent image.

The disk buffer is implemented as a client of the *block tap* device [33, 34]. The Xen block tap device allows a userspace process to interpose on the block request/response stream between a virtual machine and its underlying device. Remus uses a custom tap module which interposes on disk write requests from the protected virtual machine and asynchronously mirrors them to the backup as they are written to the local disk. To provide checkpoint synchronization, this device opens two control FIFOs at startup, allowing the protection service to insert checkpoint records into the disk stream at the appropriate time, and to receive acknowledgment from the backup before releasing queued outbound network traffic. On the primary host, checkpoint requests are simply passed along to the replica, and acknowledgments are likewise passed from the replica to the notification FIFO.

On the replica, the tapdisk module maintains state in two separate RAM disks. One is the *speculation buffer*, which holds write requests associated with speculative execution on the primary host. The other is the *write-out buffer*, which contains the state of the most recent checkpoint as it writes itself to disk. Whenever it receives a checkpoint message, any state currently in the speculation buffer is moved into the write-out buffer, superseding any data at

the same sector addresses which may already be in the write-out buffer³, and a commit receipt is immediately reported to the primary: If the primary fails, it is sufficient to have the contents of the disk associated with the most recent checkpoint in RAM, as long as the replica is not activated until that state has been applied to disk. In the case of failure, the speculation buffer is discarded and execution at the backup is permitted once the write-out buffer has been entirely drained. The contents of this buffer are applied to disk according to the following algorithm:

1. Sort outstanding writes by sector address.
2. Merge contiguous sectors into single requests.
3. Submit as many requests as the underlying driver will accept.
4. Wait for a completion callback from the underlying driver. When the completion callback is received, repeat step 1.
5. When the final request has completed, the disk state is consistent.

The sectors are sorted for two reasons: first, it is very likely to improve physical locality for the underlying disk, whose performance is generally dominated by the number of seeks it must perform; second, this allows the checkpoint process to aggressively merge requests. Although the number of outstanding requests is bounded by the tapdisk interface, the size of each request is not. Merged requests provide a substantial performance benefit.

It should be noted that the write ordering produced by this algorithm does not necessarily match that of the guest. It is possible that an interrupted write-out process will leave the disk in a non-crash-consistent state. The disk validity assertion described above protects the system as a whole from relying on such a disk.

³In normal operation this will never happen — the protection service waits for the previous checkpoint commit to be acknowledged before issuing a new one. But it would be possible to receive a new checkpoint before the last one had committed, and the driver is designed with this in mind.

2.7 Detecting failure

Remus's focus is on demonstrating that advanced high availability may be provided in a general and transparent way using commodity hardware and without modifying the applications that it protects. We currently use a simple failure detector that is directly integrated in the checkpointing stream: a timeout waiting for the backup to respond to commit requests will result in the primary assuming that the backup has crashed and disabling protection. Similarly, a timeout during transmission of checkpoint data from the primary will result in the backup assuming that the primary has crashed and resuming execution from the most recent checkpoint.

The system is configured to use a pair of bonded network interfaces, and the two physical hosts are connected using a pair of Ethernet crossover cables (or independent switches) on the protection NICs. Should both of these network paths fail, Remus does not currently provide mechanism to fence execution. Traditional techniques for resolving partitioning (i.e. quorum protocols) are notoriously difficult to apply in two host configurations. It is the responsibility of the system administrator to ensure that external network connections share fate with the replication channel.

Chapter 3

Evaluation

Remus has been designed with the primary objective of making high availability sufficiently generic and transparent that it may be deployed on today's commodity hardware. We believe that the system we have implemented meets these goals admirably. But to be practically useful, the overhead it entails must not be prohibitive. In the following section, we evaluate our implementation against three very different workloads in order to reveal its performance characteristics across a wide range of possible operating environments.

Before evaluating the overhead imposed by our high availability service, we first measure the behaviour of the system when a single host fails. In case of failure at the primary, we find that the backup becomes fully active within approximately 1.2 seconds, without any loss of visible state – established TCP connections continue without interruption.

We then evaluate the overhead of the system on application performance. We do this by measuring the performance of a protected system compared to an unprotected system for a few common workloads. In order to better understand these results we present a set of microbenchmarks which pinpoint the mechanisms most affecting performance. We find that a general-purpose task such as kernel compilation performs at somewhat better than half native speed when checkpointed 20 times per second, while network-dependent workloads as represented by SPECweb perform at somewhat more than one quarter native speed. The additional overhead in this case is largely due to added network latency.

Based on this analysis, we conclude that although Remus is efficient at state

replication, it does introduce significant network latency, particularly for applications that exhibit poor locality in memory writes. Thus, applications that are very sensitive to network latency may not be well-suited to this type of high availability service. However, it is worth mentioning that our implementation currently lacks a number of optimizations which have the potential to significantly reduce network latency, some of which are discussed in more detail following the benchmark results. It should also be pointed out that SPECweb latency is especially high due to its sustained memory burn rate; bursty applications will generally experience less latency. More general purpose workloads, as represented by the kernel compilation benchmark, can be protected with an overhead of roughly 100%.

3.1 Test environment

Unless otherwise stated, all tests were run on IBM eServer x306 systems, consisting of a 3.2 GHz Pentium 4 processor with hyperthreading enabled, 1 GB of RAM, 3 Intel e1000 GbE network interfaces, and an 80 GB SATA hard drive. The hypervisor is Xen 3.1.0 modified as described in Section 2.4, and the operating system for all virtual machines was linux 2.6.18 as distributed in Xen 3.1.0, with the modifications described in Section 2.4.4. Each domain is provided one VCPU. Domain 0 is pinned to the first physical hyperthread, and the guest domain to the second. The first network interface connects the primary and secondary physical hosts via a private switch. It is used for the replication protocol. The second interface is publicly routed, and is used for administrative access. The third interface is used for application data. Virtual machine networking is provide in bridged mode on the third interface. Virtual disks are provided from disk image files on local SATA drives, exported to the guests using the tapdisk AIO driver.

Although we did not have SMP equipment available for testing multi-CPU checkpointing, the only additional overhead this should entail is a small additional operation during guest suspension in order to quiesce other CPUs. This

overhead is trivial relative to that required to stop and copy the complete set of changed state per round, and therefore highly unlikely to have a visible effect on checkpointing performance.

3.2 Failover performance

To evaluate failover performance, we logged into the primary host via SSH and began a kernel compilation. We simultaneously started a “ping” process to measure round-trip time from an external host to the protected server every 200ms. We then disconnected power from the primary host at arbitrary times during the build, and observed that (1) the SSH session to the primary host remained unbroken after power failure and recovery, and (2) the kernel build continued to successful completion. Using the ping test, we recorded the elapsed time during which the server was unavailable to be approximately 1.2 seconds.

3.3 Benchmarks

In the following section, we evaluate the performance of our system using a variety of macrobenchmarks which are meant to be representative of a range of real-world workload mixtures. The primary workloads we examine are a kernel compilation test, the SPECweb2005 benchmark, and the Postmark disk benchmark. Kernel compilation is a balanced workload which stresses the virtual memory system, the disk and the CPU. SPECweb primarily exercises networking performance and memory throughput. Postmark focuses exclusively on disk performance, which is not a significant component of the overhead of the other workloads.

3.3.1 Kernel compilation

The kernel compile test measures the wall-clock time required to build linux kernel version 2.6.18 using the default configuration and the *bzImage* target. Compilation uses GCC version 4.1.2, and make version 3.81. The measured

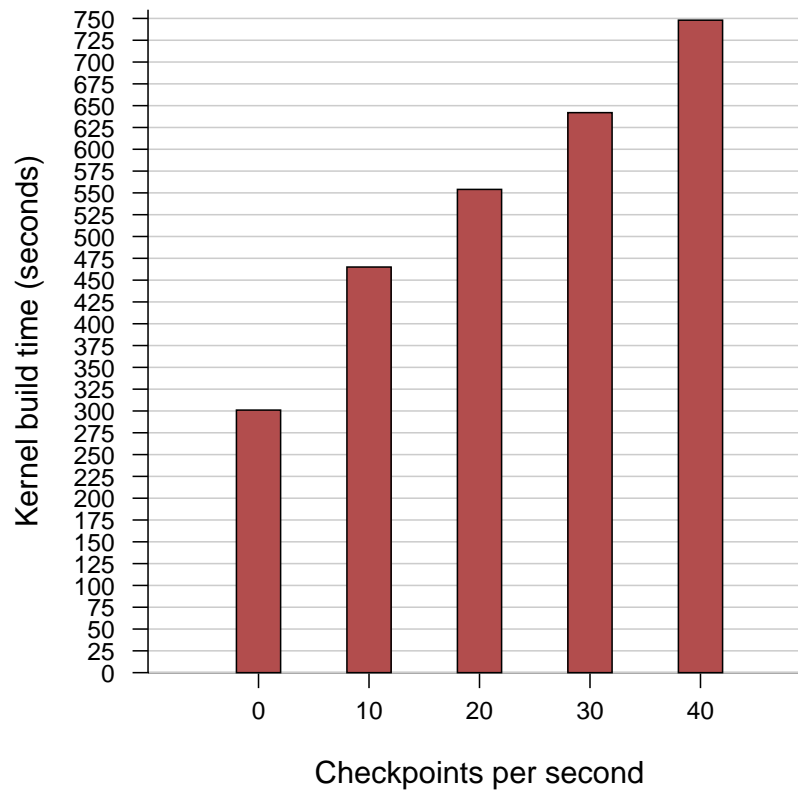


Figure 3.1: Kernel build times at various checkpoint frequencies.

times are preceded by an untimed build followed by a make clean in order to reduce cold cache effects.

This is a balanced workload that tests CPU, memory and disk performance. Figure 3.1 shows protection overhead when configured to checkpoint at rates of 10, 20, 30 and 40 times per second. Overhead scales linearly with checkpoint frequency. At a frequency of 20 times per second, total measured overhead is 84%. We believe this represents a realistic overhead for general-purpose systems.

3.3.2 SPECweb2005

The SPECweb benchmark is composed of at least three separate systems: a web server, an application server, and one or more web client simulators. We configure these as three VMs on distinct physical machines. The application server and the client are configured with 640 MB out of 1024 MB total available RAM. The web server and backup are provisioned with 2048 MB of RAM, of which 1024 is allocated to the web server VM, which is the system under test. Although this leaves domain 0 with more than 900 MB of RAM, the replication engine only requires 20-30 MB. The SPECweb scores we mention in this section are the highest results we achieved with the SPECweb e-commerce test maintaining 95% “good” and 99% “tolerable” times.

Figure 3.2 shows SPECweb performance at various checkpoint frequencies relative to an unprotected server. These scores are primarily a function of the latency imposed by the network buffer between the server and the client. Although they are configured for a range of frequencies, SPECweb touches memory rapidly enough that the time required to propagate the memory dirtied between checkpoints sometimes exceeds 100ms, regardless of checkpoint frequency. Because the network buffer cannot be released until the checkpoint state is propagated, the effective network latency is significantly higher than the configured checkpoint frequency. The size of the checkpoint buffer constrains to some degree how far ahead of replication the domain may run, but it is only a coarse-grained throttle. Because the effective checkpoint frequency is lower than the configured frequency, and network latency dominates the SPECweb score, performance is relatively flat across the range of configured frequencies. At configured rates of 10, 20, 30 and 40 checkpoints per second, the average checkpoint rates achieved were 9.98, 16.38, 20.25 and 23.34 respectively, for average latencies of 100ms, 61ms, 49ms and 43ms respectively. To confirm that SPECweb performance was due to network latency, we performed the tests again with network buffering disabled. In this case the overhead, primarily due to the stop-and-copy phase of checkpointing, was 44%.

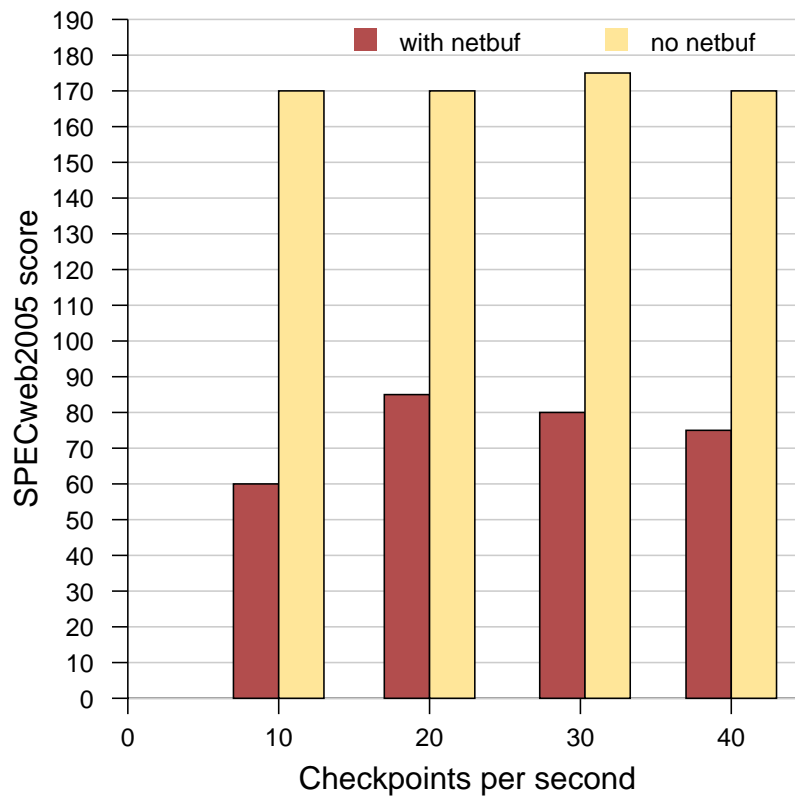


Figure 3.2: SPECweb performance at various checkpoint frequencies, with and without network output buffering. Native score: 305.

SPECweb is a RAM-hungry workload which is also very dependent on network latency. This makes it a poor fit for our current implementation, which trades network latency for memory throughput. Figure 3.3 demonstrates the dramatic effect of latency between the client VM and the web server. We used the linux *netem* [19] queueing discipline to add varying degrees of delay to the outbound link from the web server (virtualized with 1024 MB RAM, but not running under our protection service). For comparison, we also show protection overhead when network buffering is disabled, to better isolate network latency from other forms of checkpoint overhead (again, the flat profile is due to the effective checkpoint rate being lower than the configured rate). Deadline scheduling, discussed in Section 3.4.1, and page compression, discussed in Section 3.4.2 are two possible techniques for reducing checkpoint latency and transmission time. Either or both would reduce checkpoint latency, and therefore be likely to increase SPECweb performance considerably.

3.3.3 Postmark

The previous sections have characterized network and memory performance under protection, but the benchmarks used put only moderate load on the disk subsystem. In order to better understand the effects of the disk buffering mechanism, we ran the *Postmark* disk benchmark (version 1.51). This benchmark is sensitive to both disk response time and throughput. To isolate the cost of disk replication, we did not engage memory or network protection during these tests. Configuration was identical to an unprotected system, with the exception that the virtual disk was provided by the tapdisk replication module. Figure 3.4 shows the total time required to perform 10000 postmark transactions with no disk replication, and with a replicated disk committing at frequencies of 10, 20, 30 and 40 times per second. The results indicate that replication has no significant impact on disk performance.

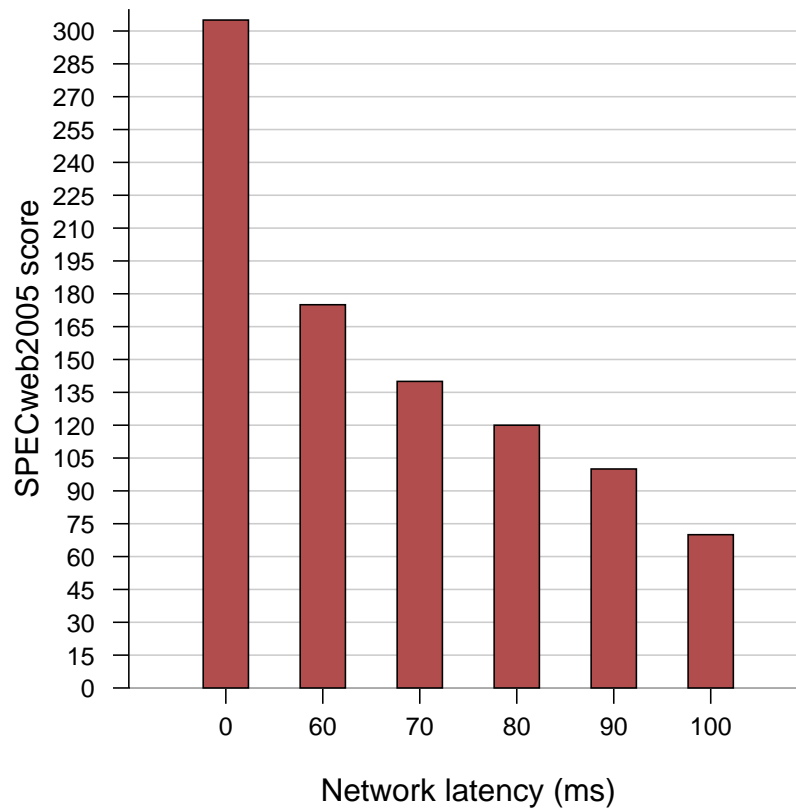


Figure 3.3: The effect of network latency on SPECweb performance.

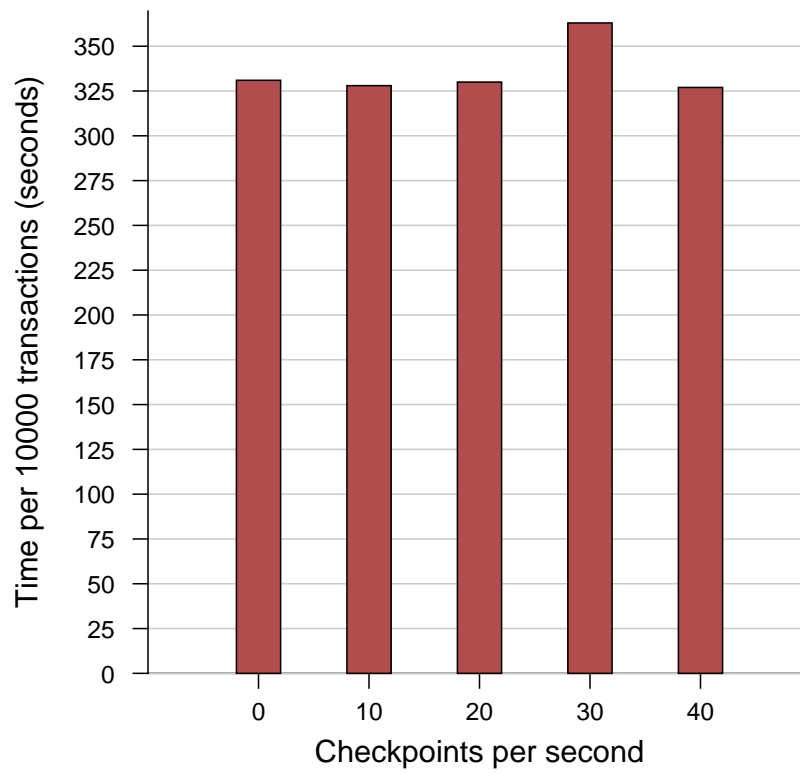


Figure 3.4: The effect of disk replication on Postmark performance.

3.4 Potential optimizations

Although we believe that the performance overhead as measured earlier in this chapter is reasonable for the level of protection that it provides, we are eager to reduce it further, particularly for latency-sensitive workloads. In addition to simply streamlining the existing code (which is largely unoptimized python in our current implementation), we believe the following techniques have the potential to greatly increase performance.

3.4.1 Deadline scheduling

Remus allows checkpoint requests at any frequency greater than the scheduler quantum of the domain 0 operating system (typically about 10ms), but the time required to perform a checkpoint and propagate it to the backup may exceed the interval between checkpoints, depending on the amount of state which must be copied. Figure 3.5 illustrates the strong correlation between the number of pages dirtied per epoch and the amount of time required to propagate the checkpoint, based on samples from a SPECweb benchmark in which the requested checkpoint frequency was 20 checkpoints per second. Although it is theoretically possible to take new checkpoints before the last checkpoint has completely propagated, this would not reduce effective latency: network output release must wait until propagation. Because memory bandwidth is far higher than disk bandwidth, epochs in which memory is heavily written to may delay subsequent checkpoints beyond their scheduled time. Figure 3.6 illustrates the effective checkpoint rate during kernel compilation compared to the requested rate. At higher frequencies, the protected system has less time in which to touch memory per epoch, but there is also less time to propagate it. At lower frequencies, the time available for both activities is increased, and furthermore multiple writes to the same memory location are coalesced during checkpointing. Theoretically, checkpoint frequency can be guaranteed at the interval of M/B where M is the total amount of physical memory available to the guest and B is the bandwidth of the replication link. This rate is far too low to support

realistic high availability deployment, but may be of use in other applications, such as periodic checkpoints for restartable computations.

It would be desirable to provide stricter scheduling guarantees. One possible solution would be to adjust the rate at which the guest operates [11], deliberately slowing it down between checkpoints depending on the number of pages it touches. Applications which prioritize latency over throughput, such as those modeled by the SPECweb benchmark discussed in Section 3.3.2, could enable such throttling for improved performance. In order to perform this type of operation, the shadow page handler may be extended to pause the guest when the number of pages currently dirty exceeds a high water mark. It would also be possible for Xen to invoke a callback by an event channel when the number of dirty pages exceeds some high water mark, allowing a user-level process to apply more sophisticated scheduling adjustments which may be configured through Xen's extensible domain scheduler.

3.4.2 Page compression

Writes to memory rarely change every byte on a page. It has been observed that disk writes typically only alter 5–20% of a data block [37]. This fact can be exploited to reduce the amount of data transmitted by sending only the delta from a previous transmission of the same page.

To evaluate the potential benefits of compressing the replication stream, we have implemented a proof-of-concept compression engine. Before transmitting a page, this system checks for its presence in an address-indexed LRU cache of previously transmitted pages. On a cache hit, the page is XORed with the previous version and the differences are run-length encoded. This provides effective compression when page writes do not change the majority of the page. Although this is true for much of the data stream, there remains a significant fraction of pages that have been modified to the point where XOR compression is not effective. In these cases, a general-purpose algorithm such as that used by gzip may achieve a higher degree of compression.

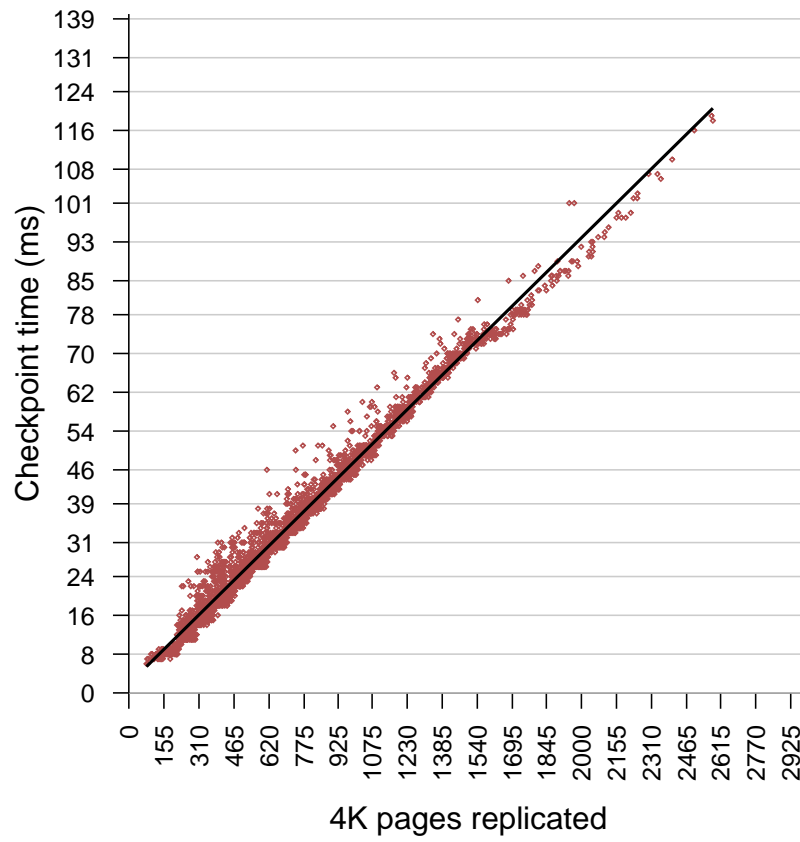


Figure 3.5: SPECweb2005 checkpoint latency versus pages dirtied per epoch.

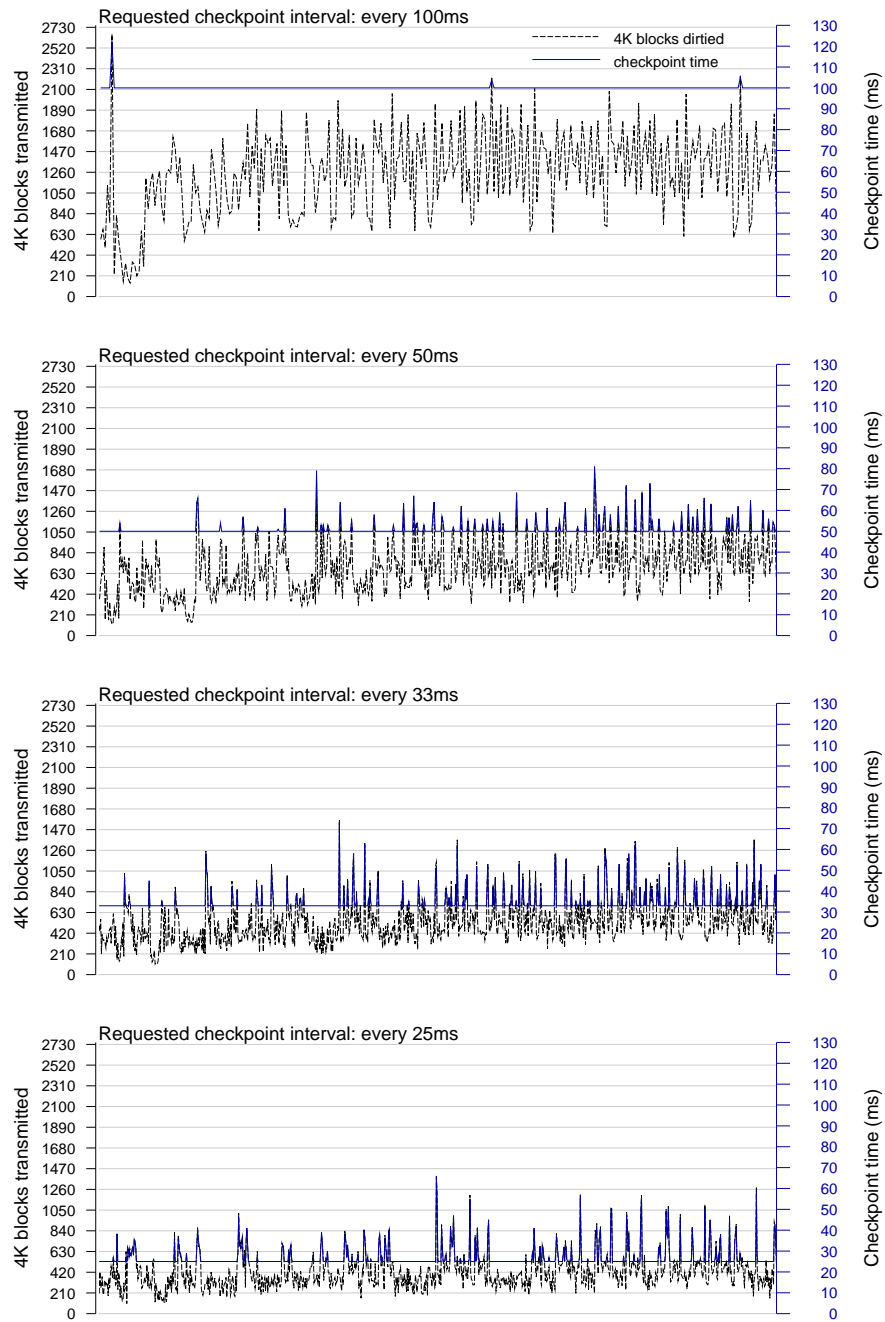


Figure 3.6: Checkpoint latency relative to the number of pages dirtied per epoch during kernel compilation.

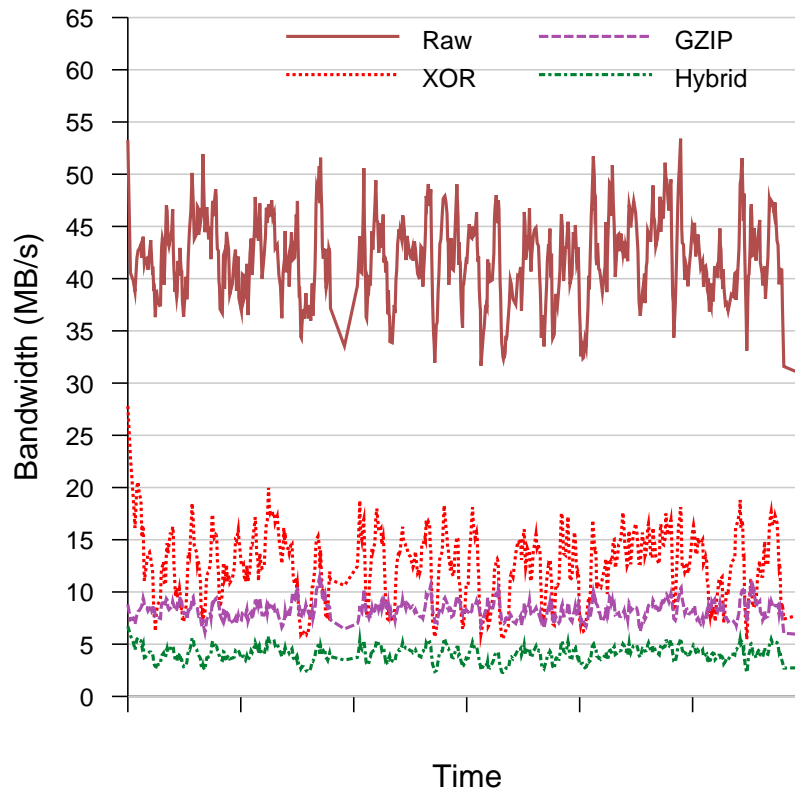


Figure 3.7: Comparison of the effectiveness of various page compression mechanisms.

We found that by using a hybrid approach, in which each page is preferentially XOR-compressed, but falls back to gzip compression if the XOR compression ratio falls below 5:1 or the previous page is not present in the cache, we could obtain a typical compression ratio of 10:1 on the replication stream. Figure 3.7 shows the bandwidth in megabytes per second for 60 seconds of a kernel build test on a protected domain with 1GB of RAM. The cache size was 8192 pages and the average cache hit rate was 99%.

Replication stream compression will consume additional memory and CPU

resources on the replicating host, but lightweight schemes such as the XOR compression technique mentioned above, which may be trivially vectorized to execute efficiently on modern processors, should easily pay for themselves through a reduction in bandwidth required for replication, and its consequent improvement in network buffering latency.

3.4.3 Copy-on-write checkpoints

In order to capture a consistent checkpoint, Remus in its current implementation must pause the domain for an amount of time linear to the number of pages which have been dirtied since the last checkpoint. This overhead could be mitigated by marking dirty pages as copy-on-write and resuming the domain immediately. This would reduce the time during which the domain must be paused to a fixed small cost proportional to the total amount of RAM available to the guest. We intend to implement copy-on-write by providing the Xen shadow paging code with a buffer of domain 0 memory into which it could copy touched pages before restoring read-write access. The replication process could then extract any pages marked as copied from the COW buffer instead of directly mapping guest domain memory. When it has finished replicating pages, their space in the buffer can be marked for reuse by the Xen COW module. If the buffer becomes full, the guest may simply be paused, resulting in a graceful degradation of service from COW to stop-and-copy operation.

Chapter 4

Related Work

State replication may be performed at several levels, each of which balances efficiency and generality differently. At the lowest level, hardware-based replication is potentially the most robust solution. Hardware, however, is much more expensive to develop than software and thus hardware replication is at a significant economic disadvantage. Virtualization-layer replication has many of the advantages of the hardware approach, but comes at lower cost because it is implemented in software. Like hardware, however, the virtualization layer has no semantic understanding of the operating-system and application state it replicates. As a result it can be less flexible — replicating the entire system instead of individual processes — and less efficient — replicating some state unnecessarily — than process checkpointing in the operating system, in application libraries or in applications themselves. The challenge for these higher-level approaches, however, is that interdependencies among state elements that comprise a checkpoint are insidiously difficult to identify and untangle from the rest of the system and thus these checkpointing mechanisms are significantly more complex than checkpointing in the virtualization layer.

4.1 Virtual machine migration

As described earlier, Remus is built on top of the Xen support for live migration [7], extended significantly to support frequent, remote checkpointing. Bradford et al. extended Xen's live migration support in another direction: migrating persistent state along with the migrating guest so that it can be restarted on a remote node that does not share network storage with the originating system[3].

Similar to Remus, other projects have used virtual machines to provide high availability. The closest to our work is Bressoud and Schneider's [4]. They use the virtual machine monitor to forward the input events seen by a primary system to a backup system where they are deterministically replayed to replicate the primary's state. Deterministic replay requires much stricter constraints on the target architecture than simple virtualization and it requires an architecture-specific implementation in VMM.

Another significant drawback of deterministic replay exemplified by Bressoud and Schneider's work is that it does not easily extend to multi-core CPUs. The problem is that it is necessary, but difficult, to determine the order in which cores accesses the shared memory. There have been attempts to address this problem. For example, *Flight Data Recorder* [36] is a hardware module that sniffs cache coherency traffic in order to record the order in which multiple processors access shared memory. Similarly, Dunlap introduces a software approach in which the CREW protocol (concurrent read, exclusive write) is imposed on shared memory via page protection [9]. While these approaches do make SMP deterministic replay possible, it is not clear if they make it feasible due to their high overhead, which increases at least linearly with the degree of concurrency. Our work sidesteps this problem entirely because it does not require deterministic replay.

4.2 Virtual machine logging and replay

Virtual machine logging has been used for purposes other than high availability. For example, in *ReVirt* [10], virtualization is used to provide a secure layer for logging state changes in the target system in order to provide better forensic evidence for intrusion detection systems. The replayed system is a read-only copy of the original system, which is not meant to be run except in order to recreate the events involved in a system compromise. Logging has also been used to build a *time-travelling debugger* [14] that, Like ReVirt, replays the system for forensics only.

4.3 Operating system replication

There are many research operating systems that support process migration, primarily for load balancing. Some examples are *Accent* [26], *Amoeba* [18], *MOSIX* [1], *Sprite* [24] and *V* [6]. The main challenge with using process migration for failure recovery is that migrated processes typically leave residual dependencies to the system from which they were migrated. Eliminating these dependencies is necessary to tolerate the failure of the primary host, but the solution has proved to be elusive due to the complexity of the system and the structure of these dependencies.

Some attempts have been made to replicate applications at the operating system level. *Zap* [23] attempts to introduce a virtualization layer within the linux kernel. This approach must be rebuilt for every operating system, and carefully maintained across versions.

4.4 Library approaches

Some application libraries provide support for process migration and checkpointing. This support is common for parallel applications for example *CoCheck* [30]. Typically process migration is used for load balancing and checkpointing is used to recover an entire distributed application in the event of failure.

4.5 Replicated storage

There has also been a large amount of work on checkpointable storage for disaster recovery as well as forensics. The Linux Logical Volume Manager [15] provides a limited form of copy-on-write snapshots of a block store. *Parallax* [35] significantly improves on this design by providing limitless lightweight copy-on-write snapshots at the block level. The *Andrew File System* [12] allows one snapshot at a time to exist for a given volume. Other approaches include *RSnapshot*, which runs on top of a file system to create snapshots via a series of

hardlinks, and a wide variety of backup software. *DRBD* [27] is a software abstraction over a block device which transparently replicates it to another server. Because these approaches replicate only persistent state, they cannot provide true high availability. It would be straightforward to replace Remus' native disk replication subsystem with other block-level approaches. Checkpointable file systems could also be used, with some loss of generality.

4.6 Speculative execution

Using speculative execution to isolate I/O processing from computation has been explored by other systems. In particular, SpecNFS [21] and Rethink the Sync [22] use speculation in a manner similar to us in order to gain asynchrony of I/O processing. Remus is different from these systems in that the semantics of block I/O from the guest remain entirely unchanged: they are applied immediately to the local physical disk. Instead, our system buffers generated network traffic to isolate the externally visible effects of speculative execution until the associated state has been completely replicated.

Chapter 5

Future Work

This chapter briefly discusses a number of directions that we intend to explore in order to improve performance and extend the service which Remus provides in its current implementation. As we have shown in the previous chapter, the overhead imposed by the high availability service is not unreasonable. However, the implementation described in this thesis is quite young. Consequently, several potential areas of optimization remain. Upon completion of the targeted optimizations discussed in Section 3.4, we intend to investigate more general optimizations and extensions, some of which are described below.

Introspection optimizations. This implementation currently propagates more state than is strictly necessary. For example, buffer cache pages do not need to be replicated, since they can simply be read in from persistent storage again. To leverage this, the virtual disk device could log the addresses of buffers provided to it for disk reads, along with the associated disk addresses. The process responsible for copying memory could then skip over these pages if they had not been modified after the completion of the disk read. The remote end would be responsible for reissuing the reads from its copy of the disk in order to fill in the missing pages. For disk-heavy workloads, this should result in a substantial reduction in state propagation time. More advanced uses of introspection might infer information similar to that captured by Valgrind [20] in order to determine whether dirty memory is actually in use by guest applications (it may have been returned to a free page pool, for example) before propagating it to the backup.

Output dependency analysis. For some classes of applications, the latency induced by network output buffering represents a significant loss of performance. Applications that perform their own checkpointing can reduce this overhead because they have access to dependency information between the state that they must checkpoint and the output that they produce, whereas Remus must assume that every outbound network packet is dependent on any uncheckpointed state. It may be interesting to explore ways of exporting this dependency information to Remus in order to increase performance, in a cooperative manner reminiscent of the techniques Xen uses to paravirtualize guest operating systems.

Hardware virtualization support. Due to the lack of equipment supporting hardware virtualization in our laboratory at the time of development, we have only implemented support for paravirtualized guest virtual machines. But we have examined the code required to support fully virtualized environments, and the outlook is quite promising. In fact, it may be somewhat simpler than the paravirtual implementation due to the better encapsulation provided by virtualization-aware hardware.

Cluster replication. It would be useful to extend the system to protect multiple interconnected hosts. While each host can be protected independently, coordinated protection would allow internal network communication to proceed without buffering. This has the potential to dramatically improve the throughput of distributed applications, including the three-tiered web application configuration prevalent in managed hosting environments. Support for cluster replication could be provided by a distributed checkpointing protocol such as that which is described in our colleague Gang Peng's master's thesis [25], which used an early version of the checkpointing infrastructure provided by Remus in order to support restartable parallel computations.

Disaster recovery. Remus was born from the SecondSite [8] project, whose aim was to provide geographically diverse mirrors of running systems in order to survive physical disaster. We are in the process of planning a multi-site deployment of Remus in order to experiment with this sort of configuration. In a

long distance deployment of Remus, network delay will be an even larger concern. Additionally, network reconfigurations will be required to redirect Internet traffic accordingly.

Log-structured datacenters. We are interested in extending Remus to capture and preserve the complete execution history of protected VMs rather than just the most recent checkpoint. This class of very detailed execution data should be very useful in building advanced debugging and forensics tools. It may also provide a convenient mechanism for recovering from state corruption whether introduced by operator error or by malicious agents (viruses, worms, and so forth).

Chapter 6

Conclusion

This thesis has presented Remus, a novel system by which high availability may be retrofitted onto existing software running on commodity hardware, without requiring significant changes to either. By encapsulating an existing system with an virtual machine, it can capture whole-machine state which represents a checkpoint of execution at an instant in time. These checkpoints are propagated asynchronously to a backup host while the protected virtual machine continues to run speculatively. Although network output must be buffered between checkpoints in order to ensure consistency between primary and backup hosts, Remus performs checkpoints at such high frequency (we have tested it at up to 40 times per second) that the effective latency induced by buffering need not be higher than that experienced between hosts on the internet.

Providing high availability is a challenging task and one that has traditionally required considerable cost and engineering effort. Remus commodifies high availability by presenting it as a service at the virtualization platform layer: high availability may simply be “switched on” for specific virtual machines. As with any high availability system, protection does not come without a cost: The network buffering required to ensure consistent replication imposes a performance overhead on applications that require very low latency. Administrators must also deploy additional hardware, which may be used in N-to-1 configurations with a single backup protecting a number of active hosts. In exchange for this overhead, Remus completely eliminates the task of modifying individual applications in order to provide high availability facilities, and it does so without requiring special-purpose hardware.

Remus represents a previously unexplored point in the design space of high availability for modern servers. The system allows protection to be simply and dynamically provided to running virtual machines at the push of a button. We feel that this model is particularly attractive for hosting providers, who desire to offer differentiated services to customers.

While high availability is an interesting challenge in its own right, we believe that the high-frequency checkpointing mechanism we have engineered in support of Remus will have many other interesting applications, ranging from forensics and error recovery tools based on replayable history to software engineering applications such as concurrency-aware time-travelling debuggers.

Bibliography

- [1] Amnon Barak and Richard Wheeler. Mosix: an integrated multiprocessor unix. pages 41–53, 1999.
- [2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
- [3] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179, New York, NY, USA, 2007. ACM Press.
- [4] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 1–11, December 1995.
- [5] Subhachandra Chandra and Peter M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 91, Washington, DC, USA, 2002. IEEE Computer Society.
- [6] David Cheriton. The v distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

-
- [7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, Berkeley, CA, USA, 2005. USENIX Association.
- [8] Brendan Cully and Andrew Warfield. Secondsite: disaster protection for the common server. In *HOTDEP'06: Proceedings of the 2nd conference on Hot Topics in System Dependability*, Berkeley, CA, USA, 2006. USENIX Association.
- [9] George Dunlap. *Execution Replay for Intrusion Analysis*. PhD thesis, University of Michigan, 2006.
- [10] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)*, 2002.
- [11] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time warped network emulation. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.
- [12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.
- [13] HP. NonStop Computing. <http://h20223.www2.hp.com/nonstop-computing/cache/76385-0-0-0-121.aspx>.
- [14] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX*

-
- Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.
- [15] Lvm2. <http://sources.redhat.com/lvm2/>.
- [16] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghill. Optimizing checkpoint sizes in the c3 system. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, April 2005.
- [17] Patrick McHardy. Linux imq. <http://www.linuximq.net/>.
- [18] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.
- [19] netem. <http://linux-net.osdl.org/index.php/Netem>.
- [20] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM Press.
- [21] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 191–205, New York, NY, USA, 2005. ACM Press.
- [22] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006. USENIX Association.
- [23] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

-
- [24] John K. Ousterhout, Andrew R. Cherenon, Frederick Douglass, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.
- [25] Gang Peng. Distributed checkpointing. Master’s thesis, University of British Columbia, 2007.
- [26] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *SOSP ’81: Proceedings of the eighth ACM symposium on Operating systems principles*, pages 64–75, New York, NY, USA, 1981. ACM Press.
- [27] Philipp Reisner and Lars Ellenberg. Drbd v8 – replicated storage with shared disk semantics. In *Proceedings of the 12th International Linux System Technology Conference*, October 2005.
- [28] Rusty Russell. Netfilter. <http://www.netfilter.org/>.
- [29] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU SCS Technical Report CMU-CS-99-176, Carnegie Mellon University, December 1999.
- [30] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS ’96)*, Honolulu, Hawaii, 1996.
- [31] Symantec Corporation. Veritas Cluster Server for VMware ESX. http://eval.symantec.com/mktginfo/products/Datasheets/High_Availability/vcs22vmware_datasheet.pdf, 2006.
- [32] VMware, Inc. VMware high availability (ha). <http://www.vmware.com/products/vi/vc/ha.html>, 2007.
- [33] Andrew Warfield. *Virtual Devices for Virtual Machines*. PhD thesis, University of Cambridge, 2006.

-
- [34] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the development of soft devices. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.
- [35] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. 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.
- [36] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 122–135, New York, NY, USA, 2003. ACM Press.
- [37] Qing Yang, Weijun Xiao, and Jin Ren. Trap-array: A disk array architecture providing timely recovery to any point-in-time. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 289–301, Washington, DC, USA, 2006. IEEE Computer Society.