Remus: High Availability via Asynchronous Virtual Machine Replication

Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield[*]

Department of Computer Science
The University of British Columbia
{brendan, geoffrey, dmeyer, feeley, norm, andy}@cs.ubc.ca

Abstract:

Allowing applications to survive hardware failure is an 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, while completely preserving host state such as active network connections. Our approach encapsulates protected software in a virtual machine, asynchronously propagates changed 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.

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 that are capable of maintaining and switching to backups in the face of failure. Commercial high availability systems that aim to protect modern servers generally use specialized hardware, customized software, or both (e.g [12]). In each case, the ability to transparently survive failure is complex and expensive enough to prohibit deployment on common servers.

This paper describes *Remus*, a software system that provides OS- and application-agnostic high availability on commodity hardware. Our approach capitalizes on the ability of virtualization to migrate running VMs between physical hosts [6], and extends the technique to replicate snapshots of an entire running OS instance at *very* high frequencies — as often as every 25ms — between a pair of physical machines. Using this technique, our system discretizes the execution of a VM into a series of replicated snapshots. External output, specifically transmitted network packets, is not released until the system state that produced it has been replicated.

Virtualization makes it possible to create a copy of a running machine, but it does not guarantee that the process will be efficient. Propagating state synchronously at every change is impractical: it effectively reduces the throughput of memory to that of the network device performing replication. 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*. System state is not made externally visible until the checkpoint is committed — we achieve high-speed replicated performance by effectively running the system tens of milliseconds in the past.

The contribution of this paper is a practical one. Whole-system replication is a well-known approach to providing high availability. However, it usually has been considered to be significantly more expensive than application-specific checkpointing techniques that only replicate relevant data [15]. Our approach may be used to bring HA “to the masses” as a platform service for virtual machines. In spite of the hardware and software constraints under which it operates, this system provides protection equal to or better than 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.

1.1 Goals

Remus aims to make mission-critical availability accessible to mid- and low-end systems. By simplifying provisioning and allowing multiple servers to be consolidated on a smaller number of physical hosts, virtualization has made these systems more popular than ever. However, the benefits of consolidation come with a hidden cost in the form of increased exposure to hardware failure. Remus

addresses this by commodifying high availability as a service offered by the virtualization platform itself, providing administrators of individual VMs with a tool to mitigate the risks associated with virtualization.

Remus's design is based on the following high-level goals:

Generality. It can be prohibitively expensive to customize a single application to support high availability, let alone the diverse range of software upon which an organization may rely. To address this issue, high availability should be provided as a low-level service, with common mechanisms that apply regardless of the application being protected or the hardware on which it runs.

Transparency. The reality in many environments is that OS and application source may not even be available to modify. To support the broadest possible range of applications with the smallest possible barrier to entry, high availability should not require that OS or application code be modified to support facilities such as failure detection or state recovery.

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.

These are lofty goals, entailing a degree of protection well beyond that provided by common HA systems, which are based on asynchronous storage mirroring followed by application-specific recovery code. Moreover, the desire to implement this level of availability *without* modifying the code within a VM necessitates a very coarse-grained approach to the problem. A final and pervasive goal of the system is that it realize these goals while providing deployable levels of performance even in the face of SMP hardware that is common on today's server hardware.

1.2 Approach

Remus runs paired servers in an active-passive configuration. 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. This allows the primary server to remain productive, while synchronization with the replicated server is performed asynchronously. The basic stages of operation in Remus are given in Figure 1.

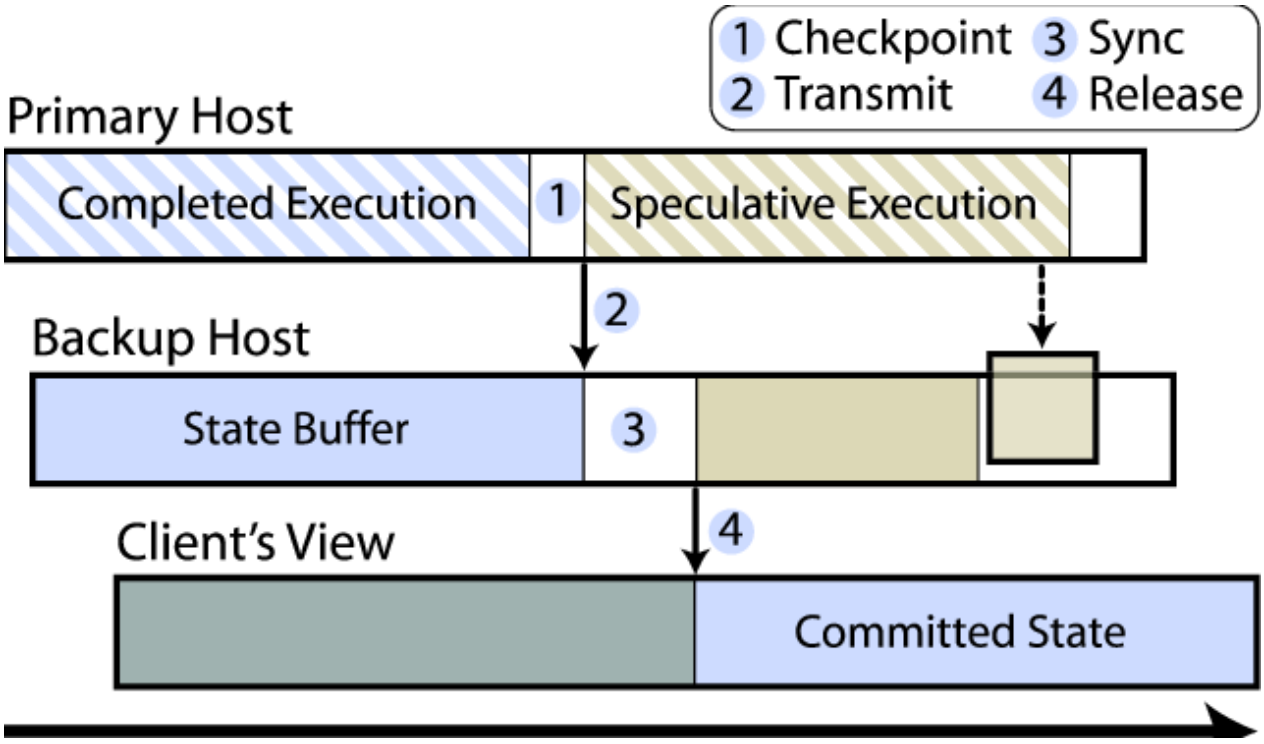

Figure 1: Speculative execution and asynchronous replication in Remus.

VM-based whole-system replication. Hypervisors have been used to build HA systems before [4]. In that work, virtualization is used to run a pair of systems in lock-step, and additional support has been added to ensure that VMs on a pair of physical hosts follow a deterministic path of execution: external events are carefully injected into both the primary and fallback VMs so that they result in identical states. Enforcing such deterministic execution suffers from two fundamental problems. First, it is highly architecture-specific, requiring that the system have a comprehensive understanding of the instruction set being executed and the sources of external events. Second, it results in an unacceptable overhead when applied in multi-processor systems, where shared-memory communication between processors must be accurately tracked and propagated [8].

Speculative execution. Replication may be achieved either by copying the state of a system or by replaying input deterministically. We believe the latter to be impractical for real-time operation, especially in a multi-processor environment. Therefore, Remus does not attempt to make computation deterministic — 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. However, the state of the replica needs to be synchronized with 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 output[1] 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.

Asynchronous replication. Buffering output at the primary server allows replication to be performed *asynchronously*. The primary host can resume execution at the moment its machine state has been captured, without waiting for acknowledgment from the remote end. Overlapping normal execution with the replication process yields substantial performance benefits. This enables efficient operation even when checkpointing at intervals on the order of tens of milliseconds.

2 Design and Implementation

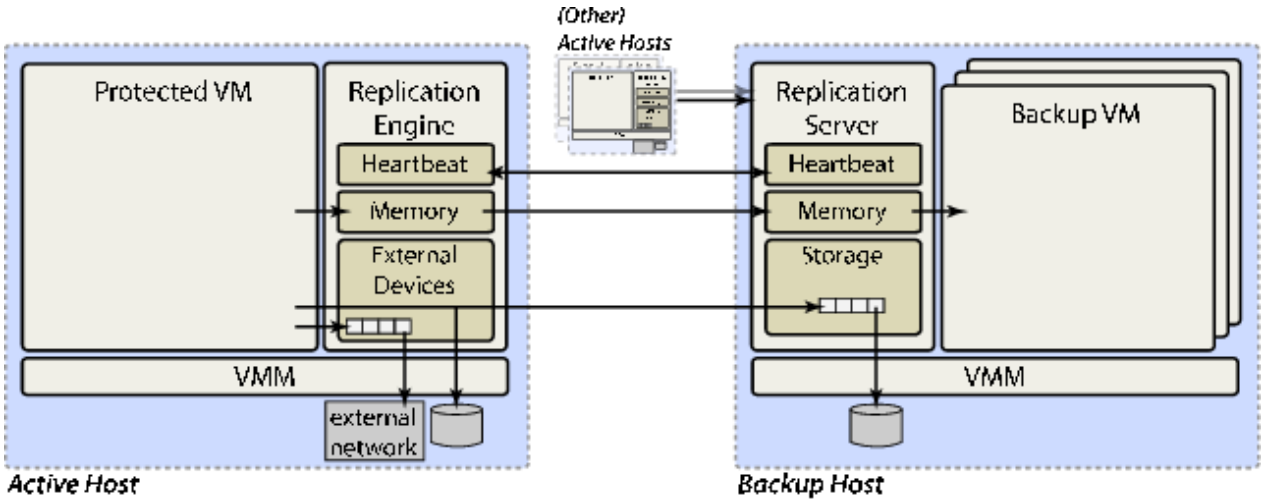

Figure 2: Remus: High-Level Architecture

Figure 2 shows a high-level view of our system. We begin by encapsulating the machine to be protected within a VM. 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* VM to a *backup* physical host. On the backup, the VM 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 happens very

frequently – we include benchmark results at frequencies up to forty times per second, representing a whole-machine checkpoint including 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. To maintain disk replication, 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 buffered disk writes are applied 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 VM. This consumes a relatively small amount of the backup host's resources, allowing it to concurrently protect VMs running on multiple physical hosts in an N-to-1-style configuration. Such a configuration gives administrators a high degree of freedom to balance the degree of redundancy against resource costs.

2.1 Failure model

Remus 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 output will be made externally visible until the associated system state has been committed to the replica.

Our goal is 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. It uses a pair of commodity host machines, connected over redundant gigabit Ethernet connections, and survives the failure of any one of these components. By incorporating block devices into its state replication protocol, it avoids requiring expensive, shared network-attached storage for disk images.

We do not aim to recover from software errors or non-fail-stop conditions. As observed in [*5*], this class of approach 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 HA products, which provide protection for virtual machines today [*31*, *30*]. 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 VM 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 VM running and network connections intact. Exposed state is not lost and disks are not corrupted.

2.2 Pipelined Checkpoints

Checkpointing a running virtual machine many times per second places extreme demands on the host system. Remus addresses this by aggressively pipelining the checkpoint operation. We use an epoch-based system in which execution of the active VM is bounded by brief pauses in execution in which changed state is atomically captured, and external output is released when that state has been propagated to the backup. Referring back to Figure 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 [6], but as described later in this section it has been dramatically optimized for high-frequency checkpoints. With state changes preserved in a buffer, the VM 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.

2.3 Memory and CPU

Checkpointing is implemented above Xen's existing machinery for performing live migration [6]. Live migration is a technique by which a virtual machine is relocated to another physical host with only slight interruption in service. To accomplish this, memory is copied to the new location while the VM continues to run at the old location. During migration, writes to memory are intercepted, and dirtied pages are copied to the new location in rounds. After a specified number of intervals, 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 100ms. Total migration time is a function of the amount of memory in use by the guest, and its *writable working set* [6], which is the set of pages changed repeatedly during

guest execution.

Xen provides the ability to track guest writes to memory using a mechanism called *shadow page tables*. When this mode of operation is enabled, the VMM maintains a private ("shadow") version of the guest's page tables and exposes these to the hardware MMU. Page protection is used to trap guest access to its internal version of page tables, allowing the hypervisor to track updates, which are propagated to the shadow versions as appropriate.

For live migration, this technique is extended to transparently (to the guest) mark all VM memory as read only. The hypervisor is then able to trap all writes that a VM makes to memory and maintain a map of pages that have been dirtied since the previous round. Each round, the migration process atomically reads and resets this 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 VM and enters a final "stop-and-copy" round, where any remaining pages are transmitted and execution resumes on the destination host.

Remus implements checkpointing as repeated executions of the final stage of live migration: each epoch, the guest is paused while changed memory and CPU state is 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.

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 overhead. This is not the case when capturing frequent VM 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.

Remus optimizes checkpoint signaling in two ways: First, it reduces the number of inter-process requests required to suspend and resume the guest domain. Second, it entirely removes xenstore from the suspend/resume process. In the original code, when the migration process desired to suspend a VM it sent a message to xend, the VM management daemon. Xend in turn wrote a message to xenstore, which alerted the guest by an event channel (a virtual interrupt) that it

should suspend execution. The guest's final act before suspending was to make a hypercall[2] which descheduled the domain and caused Xen to send a notification to xenstore, which then sent an interrupt to xend, which finally returned control to the migration process. This convoluted process could take a nearly arbitrary amount of time — typical measured latency was in the range of 30 to 40ms, but we saw delays as long as 500ms 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 VM suspension. In concert, these two notification mechanisms reduce the time required to suspend a VM to about one hundred microseconds – an improvement of two orders of magnitude over the previous implementation.

In addition to these signaling changes, we have increased the efficiency of the memory copying process. First, we quickly filter out clean pages from the memory scan, because at high checkpoint frequencies most memory is unchanged between rounds. Second, we map the guest domain's entire physical memory into the replication process when it begins, rather than mapping and unmapping dirty pages at every epoch — we found that mapping foreign pages took approximately the same time as copying them.

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 VM after 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 previously they were always destroyed after their state had been replicated). A similar operation is necessary in order to re-arm watches in xenstore.

Asynchronous transmission. 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 was reduced by approximately 10% at 20 checkpoints per second.

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 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 work was deferred until the domain was restored on the other host. These modifications are available in Xen as of version 3.1.0.

These changes are not strictly required for correctness, but they do improve the performance of the checkpoint considerably, 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 VMs.

2.4 Network buffering

Most networks cannot be counted on for reliable data delivery. Therefore, networked applications must either accept packet loss, duplication and reordering, or 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, since their loss will appear as a transient network failure and will not affect the correctness of the protected state. 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 has been lost.

Figure 3 depicts the mechanism by which we prevent the release of speculative network state. Inbound traffic is delivered to the protected host immediately, but outbound packets generated since the previous checkpoint are queued until the current state has been checkpointed and that checkpoint has been acknowledged by the backup site. We have implemented this buffer as a linux queuing discipline applied to the guest domain's network device in *domain 0*, which responds to two RTnetlink messages. Before the guest is allowed to resume execution after a checkpoint, the network buffer receives a `CHECKPOINT` message, which causes it to insert a barrier into the outbound queue preventing any subsequent packets from being released until a corresponding release message is received. When a guest checkpoint has been acknowledged by the backup, the buffer receives a `RELEASE` message, at which point it begins dequeueing traffic up to the barrier.

There are two minor wrinkles in this implementation. The first is that in linux, queueing disciplines only operate on *outgoing* traffic. Under Xen, guest network interfaces consist of a frontend device in the guest, and a corresponding backend device in *domain 0*. Outbound traffic from the guest appears as *inbound* traffic on the backend device in domain 0. Therefore in order to queue the traffic, we convert the inbound traffic to outbound by routing it through a special device

called an *intermediate queueing device* [16]. This module is designed to work at the IP layer via iptables [27], but it was not difficult to extend it to work at the bridging layer we use to provide VM network access in our implementation.

The second wrinkle is due to the implementation of the Xen virtual network device. For performance, the memory used by outbound networking traffic is not copied between guest domains and domain 0, but shared. However, only a small number of pages may 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. Unfortunately, the network output buffer can result in messages being in flight for a significant amount of time, which results in the guest network device blocking after a very small amount of traffic has been sent. Therefore when queueing messages, we first copy them into local memory and then release the local mappings to shared data.

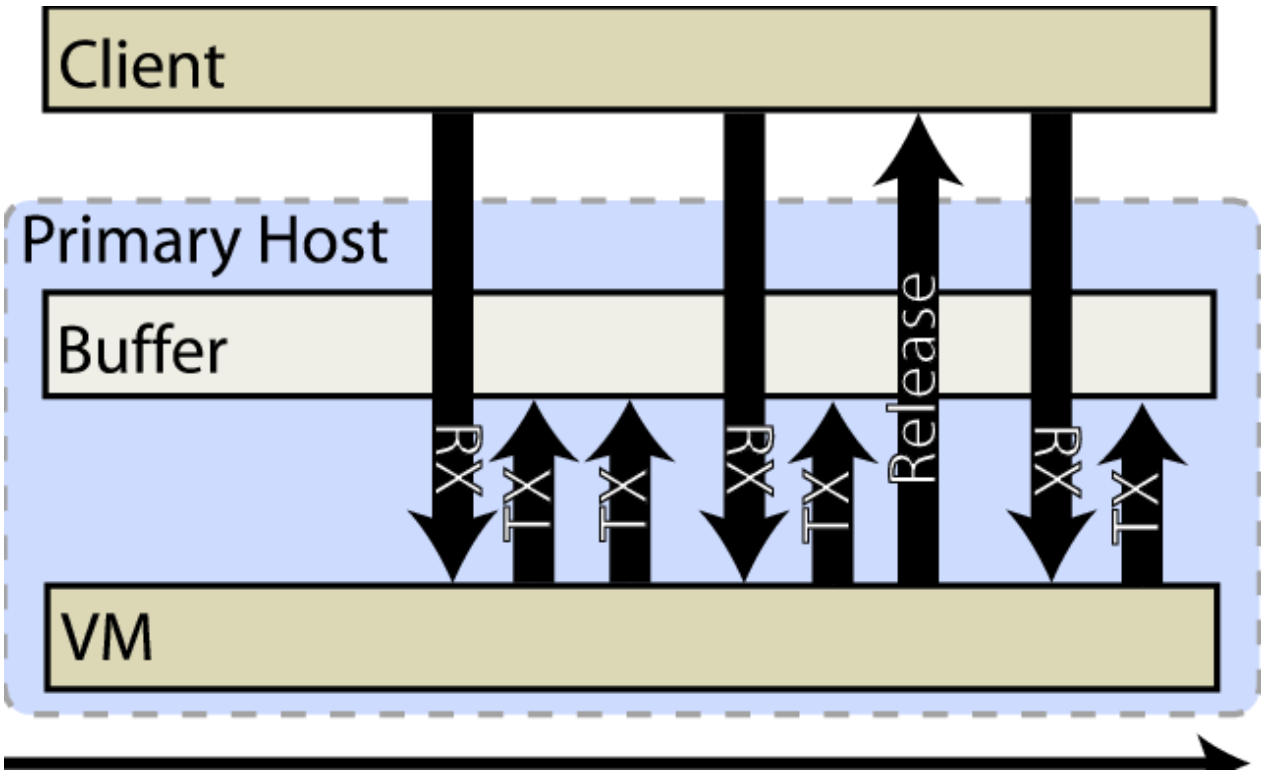

Figure 3: Network buffering in Remus.

2.5 Disk buffering

Disks present a rather different challenge than network interfaces, largely because they are expected to provide much stronger reliability guarantees. In particular, when a write has been acknowledged by a disk, an application (or file system) expects to be able to recover that data even in the event of a power failure immediately following the acknowledgment. While Remus is designed to

recover from a single host failure, it must preserve crash consistency even if *both* hosts fail. Moreover, the goal of providing a general-purpose system precludes the use of expensive mirrored storage hardware designed for HA applications. Therefore Remus maintains a complete mirror of the active VM'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 4 gives a high-level overview of the disk replication mechanism

As with the memory replication subsystem described in Section 2.3, writes to disk from the active VM are treated as write-through: they are immediately applied to the primary disk image, and asynchronously mirrored to an in-memory buffer on the backup. This approach provides two direct benefits: First, it ensures that the active disk image remains crash consistent at all times; in the case of both hosts failing, the active disk will reflect the crashed state of the externally visible VM at the time of failure (the externally visible VM resides on the primary host if the primary host has not failed or if the backup also fails before it has been activated, otherwise it resides on the backup). Second, writing directly to disk accurately accounts for the latency and throughput characteristics of the physical device. This obvious-seeming property is of considerable value: accurately characterizing disk responsiveness is a subtle problem, as we ourselves experienced in an earlier version of the disk buffer which held write requests in memory on the primary VM until checkpoint commit. Such an approach either buffers writes, under-representing the time required to commit data to disk and allowing the speculating VM to race ahead in execution, or conservatively over-estimates write latencies resulting in a loss of performance. Modeling disk access time is notoriously challenging [*28*], but our implementation avoids the problem by preserving direct feedback from the disk to its client VM.

At the time that the backup acknowledges that a checkpoint has been received, disk updates reside completely in memory. No on-disk state may be changed until the entire checkpoint has been received, as this would prevent the backup from rolling back to the most recent complete checkpoint. Once the checkpoint is acknowledged, the disk request buffer may be applied to disk. In the event of a failure, 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 VM: if the backup fails after activation but before data is completely flushed to disk, its on-disk state might not be crash consistent.

Only one of the two disk mirrors managed by Remus is actually valid at any given time. This point is critical in recovering from multi-host crashes. This property is achieved by the use of an activation record on the backup disk, which is written

after the most recent disk buffer has been completely flushed to disk and *before* the backup VM begins execution. In recovering from multiple host failures, this record may be used to identify the valid, crash consistent version of the disk.

The disk buffer is implemented as a Xen block tap module [32]. The block tap is a device which allows a process in the privileged domain to efficiently interpose itself between the frontend disk device presented to a guest VM and the backend device which actually services requests. The buffer module logs disk write requests from the protected VM and mirrors them to a corresponding module on the backup, which executes the checkpoint protocol described above and then removes itself from the disk request path before the backup begins execution in the case of failure at the primary.

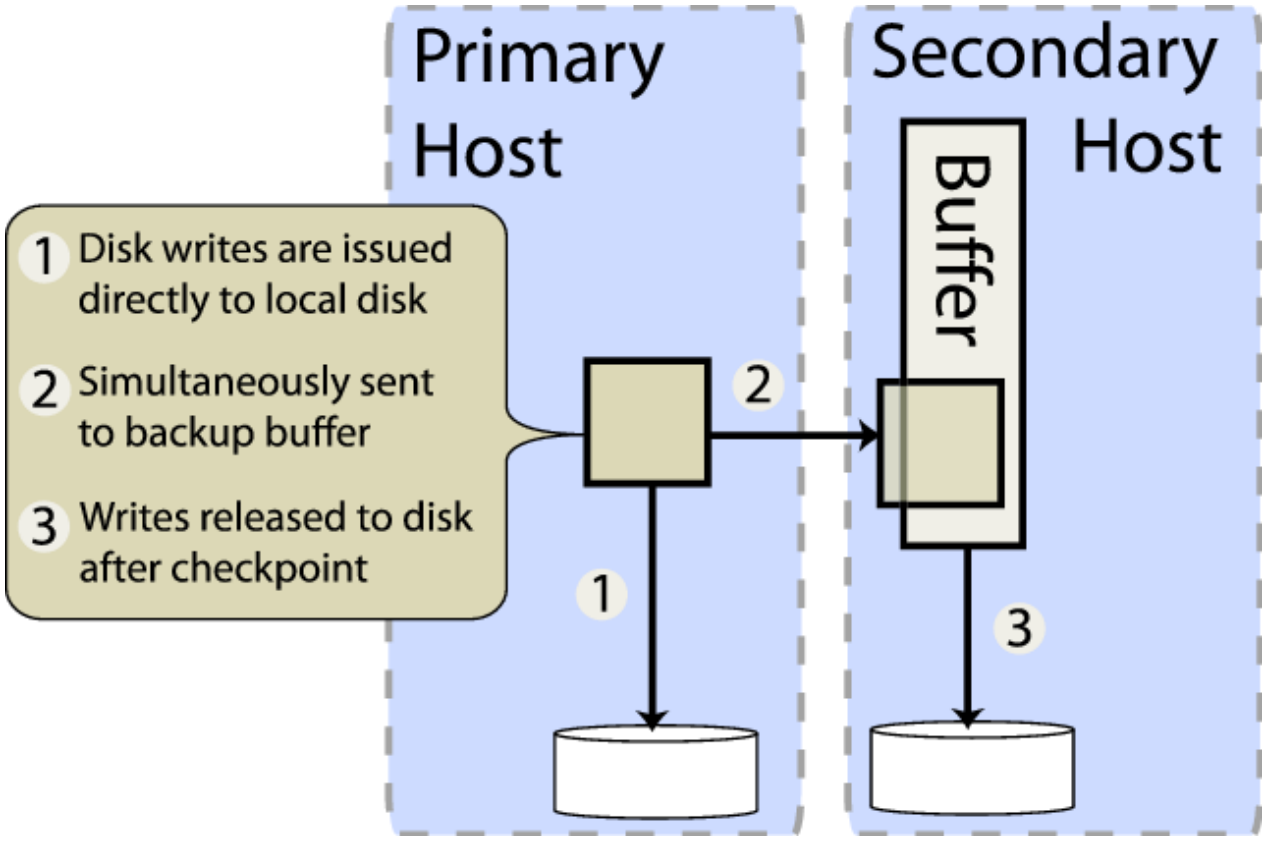

Figure 4: Disk write buffering in Remus.

2.6 Detecting Failure

Remus's focus is on demonstrating that it is possible to provide advanced high availability in a general and transparent way using commodity hardware and without modifying the protected applications. We currently use a simple failure detector that is directly integrated in the checkpointing stream: a timeout of the backup responding to commit requests will result in the primary assuming that

the backup has crashed and disabling protection. Similarly, a timeout of new checkpoints being transmitted 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. We feel that in this case, we have designed Remus to the edge of what is possible with commodity hardware.

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. In this section, we characterize the overheads resulting from our approach for a variety of different workloads, in order two answer two questions: (1) Is this system practically deployable? (2) What kinds of workloads are most amenable to our approach?

Before measuring the performance impact, we must establish that the system functions correctly. We accomplish this by injecting network failures at each phase of the replication protocol, while putting substantial disk, network and CPU load on the protected system. We find that the backup takes over for the lost primary within approximately one second in every case, preserving all externally visible state, including active network connections.

We then evaluate the overhead of the system on application performance across very different workloads. We find that a general-purpose task such as kernel compilation incurs approximately a 50% performance penalty 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 output-commit delay on the network interface.

Based on this analysis, we conclude that although Remus is efficient at state replication, it does introduce significant network delay, 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 (although there are a number of optimizations which have the potential to noticeably reduce network delay, some of which we discuss in more detail following the benchmark results). We feel that we have been conservative

in our evaluation, using benchmark-driven workloads which are significantly more intensive than would be expected in a typical virtualized system; the consolidation opportunities such an environment presents are particularly attractive because system load is variable.

3.1 Test environment

Unless otherwise stated, all tests were run on IBM eServer x306 servers, consisting of one 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 was Xen 3.1.2, modified as described in Section 2.3, and the operating system for all virtual machines was linux 2.6.18 as distributed in Xen 3.1.2, with the modifications described in Section 2.3. The protected VM was allocated 512 MB of total RAM. To minimize scheduling effects from the VMM, domain 0's VCPU was pinned to the first hyperthread. One physical network interface was bridged to the guest virtual interface and used for application traffic, one was used for administrative access, and the last was used for replication (we did not bond interfaces for replication, but this is immaterial to the tests we performed). Virtual disks were provided by disk images on the SATA drive, exported to the guest using the tapdisk AIO driver.

3.2 Correctness verification

As discussed in Section 2.2, Remus's replication protocol operates in four distinct phases: (1) checkpoint changed state and increment the epoch of network and disk request streams, (2) replicate system state, (3) when the complete memory checkpoint and corresponding set of disk requests has been received, send a checkpoint acknowledgement from the backup, and (4) upon receipt of the acknowledgement, release outbound network packets queued during the previous epoch. To verify that our system functions as intended, we tested deliberately induced network failure at each stage. For each test, the protected system executed a kernel compilation process in order to generate disk, memory and CPU load. To verify the network buffer, we simultaneously executed a graphics-intensive X11 client (`glxgears`) attached to an external X11 server. Remus was configured to take checkpoints every 25 milliseconds throughout. Each individual test was repeated twice.

At every failure point, the backup successfully took over execution of the protected system, with only minor network delay (about one second) noticeable while the backup detected the failure and activated the replicated system. The glxgears client continued to run after a brief pause, and the kernel compilation task continued to successful completion. We then gracefully shut down the VM and executed a forced file system check on the backup disk image, which reported no inconsistencies.

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 run 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, and Postmark focuses on disk performance.

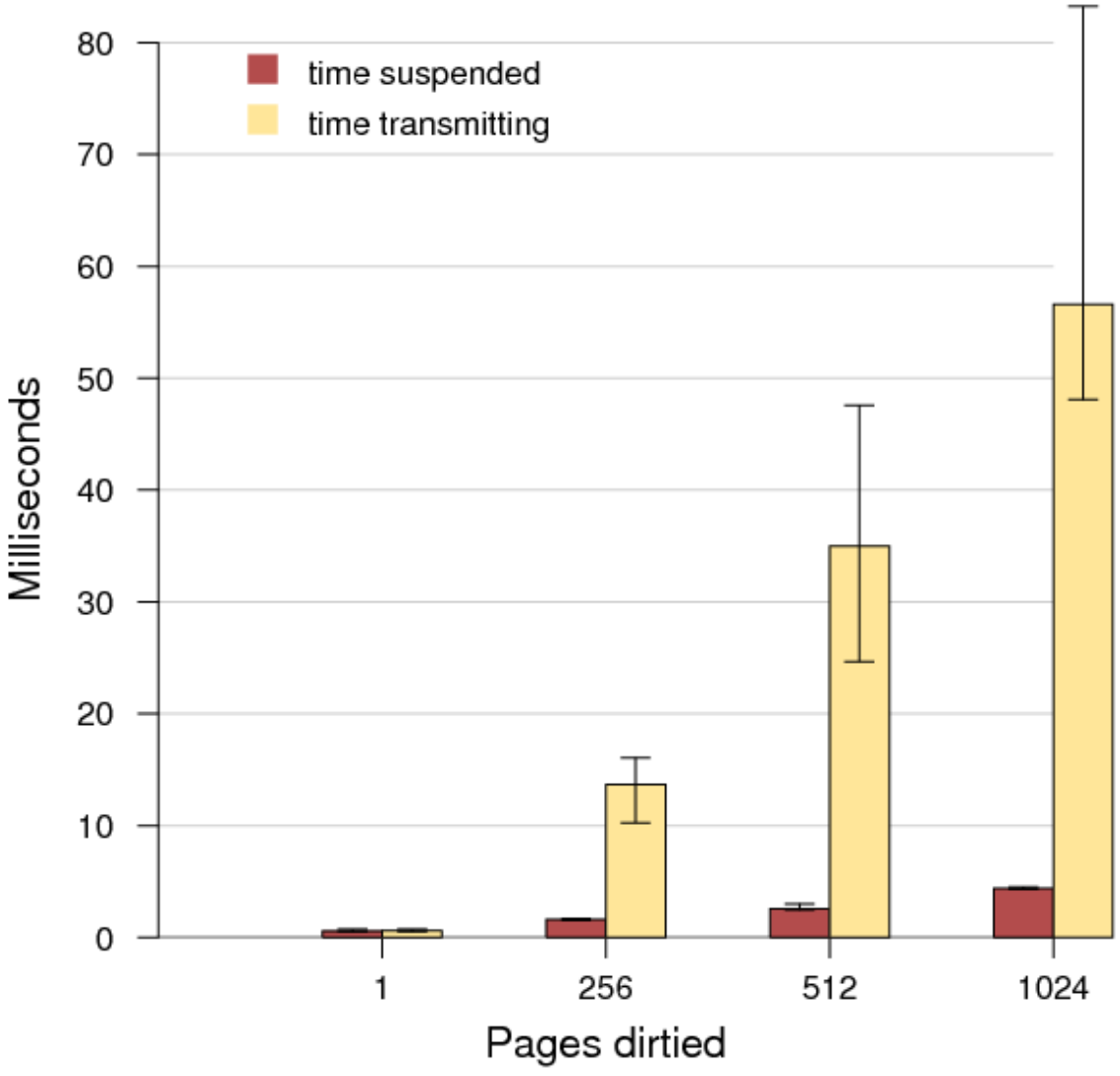

Figure 5: Checkpoint time relative to pages dirtied.

To better understand the following measurements, we performed a microbenchmark measuring the time spent copying guest state (while the guest was suspended) and the time spent sending that data to the backup relative to the number of pages changed since the previous checkpoint. We wrote an application to repeatedly change the first byte of a set number of pages and measured times over 1000 iterations. Figure 5 presents the average, minimum and maximum recorded times spent in the checkpoint and replication stages, within a 95% confidence interval. It shows that the bottleneck for checkpoint frequency is replication time.

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. This is a balanced workload that tests CPU, memory and disk performance.

Figure 6 shows protection overhead when configured to checkpoint at rates of 10, 20, 30 and 40 times per second, compared to a baseline compilation in an unprotected virtual machine. Total measured overhead at each of these frequencies was 31%, 52%, 80% and 103%, respectively. Overhead scales linearly with checkpoint frequency within the rates we tested. We believe that the overhead measured in this set of tests is reasonable for a general-purpose system, even at a checkpoint frequency of 40 times per second.

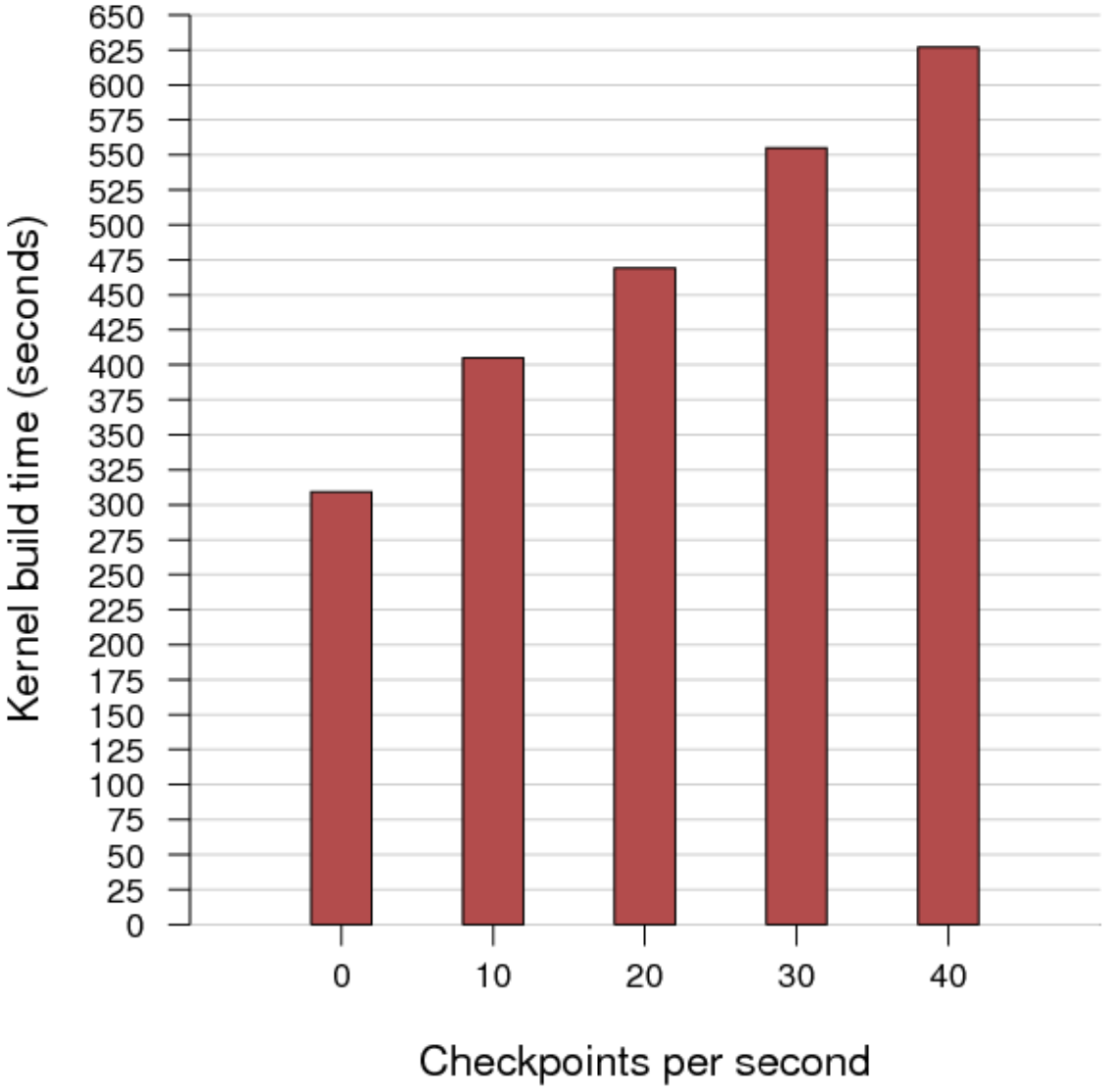

Figure 6: Kernel build time by checkpoint frequency.

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. 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 7 shows SPECweb performance at various checkpoint frequencies relative to an unprotected server. These scores are primarily a function of the delay 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 checkpointed state has been acknowledged, the effective network delay can be higher than the configured checkpoint interval. Remus does ensure that the VM is suspended at the start of every epoch, but it cannot currently ensure that the total amount of state to be replicated per epoch does not exceed the bandwidth available during the configured epoch length. Because the *effective* checkpoint frequency is lower than the configured rate, 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, or average latencies of 100ms, 61ms, 49ms and 43ms respectively.

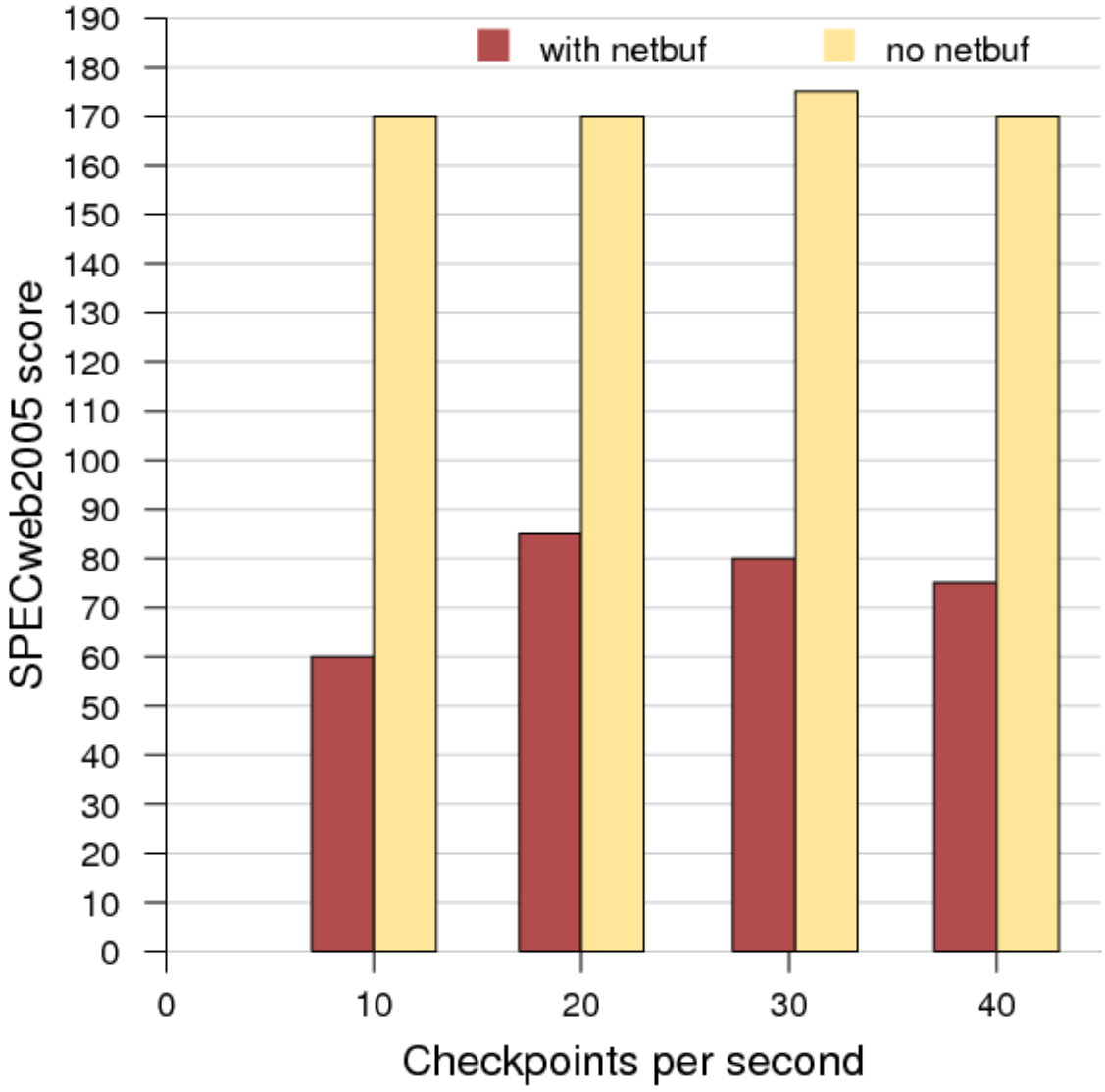

Figure 7: SPECweb scores by checkpoint frequency (native score: 305)

SPECweb is a RAM-hungry workload which is also very sensitive to network latency. This makes it a poor fit for our current implementation, which trades network delay for memory throughput. Figure 8 demonstrates the dramatic effect delay between the client VM and the web server has on SPECweb. We used the Linux *netem* [19] queueing discipline to add varying degrees of delay to the outbound link from the web server (virtualized but not running under Remus). For comparison, Figure 7 also shows 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

falling short of the configured rate). Deadline scheduling and page compression, discussed in Section 3.4 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.

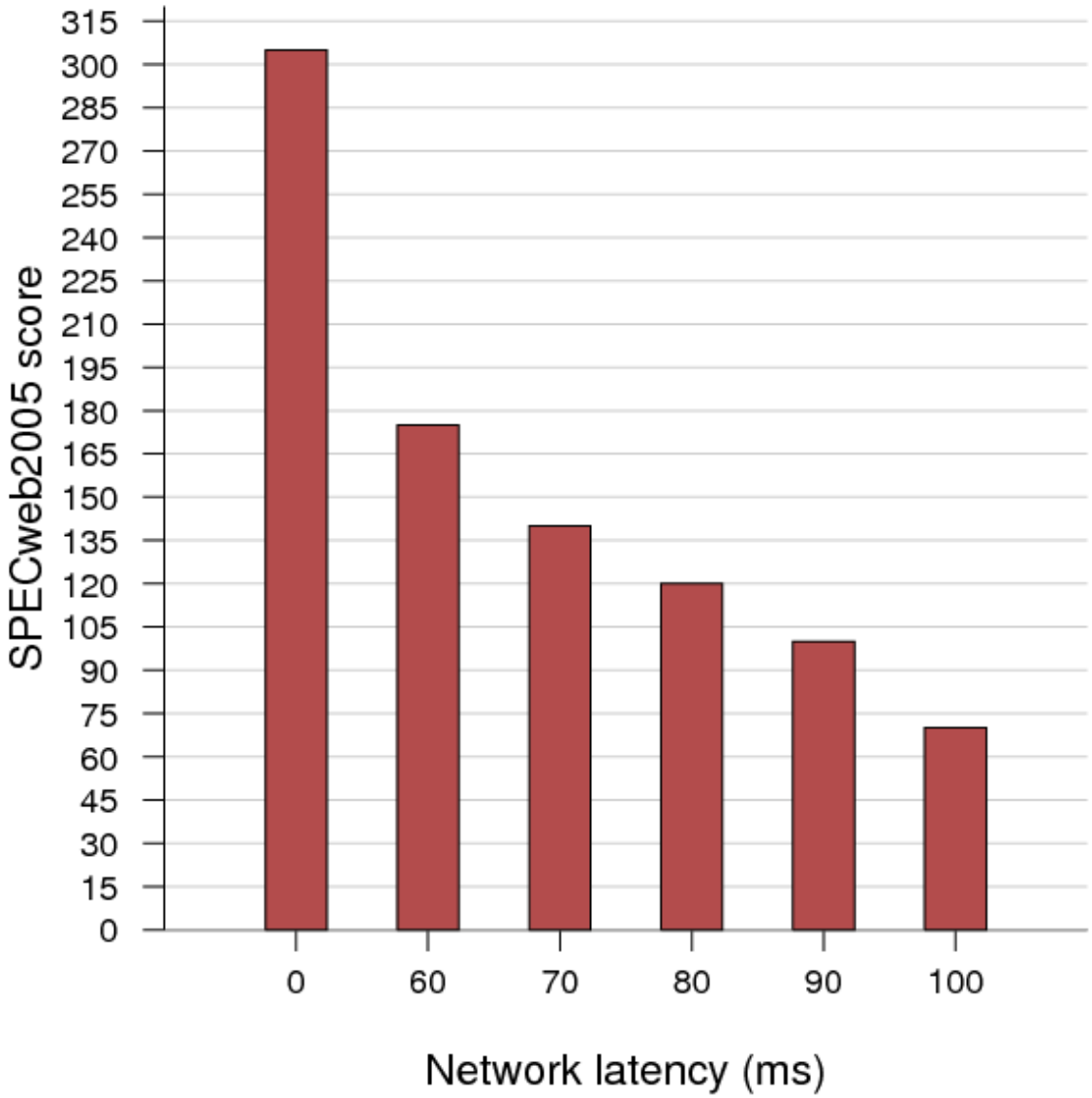

Figure 8: The effect of network delay on SPECweb performance.

Postmark. The previous sections characterize 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 throughput and disk response time. 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 9 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.

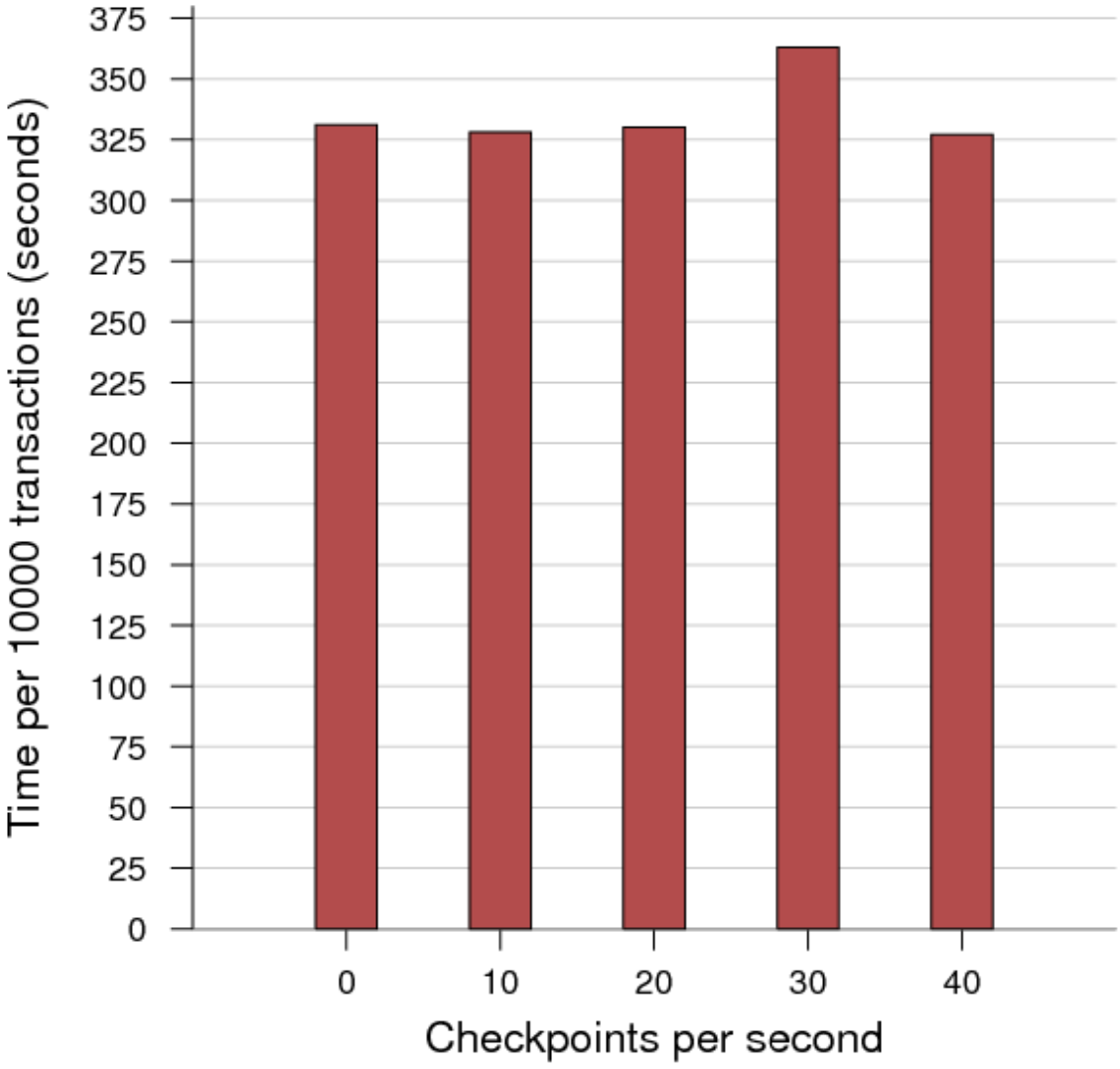

Figure 9: The effect of disk replication of Postmark performance.

3.4 Potential optimizations

Although we believe the performance overheads shown earlier in this section are reasonable for what they provide, we are eager to reduce them further, particularly for latency-sensitive workloads. In addition to more careful tuning of the existing code, we believe the following techniques have the potential to greatly increase performance.

Deadline scheduling. The amount of time required to perform a checkpoint is currently variable, depending on the amount of memory to be copied. Although Remus ensures that the protected VM is suspended at the start of each epoch, it currently makes no attempt to control the amount of state which may change between epochs. To provide stricter scheduling guarantees, the rate at which the guest operates could be deliberately slowed [*10*] between checkpoints, depending on the number of pages dirtied. Applications which prioritize latency over throughput, such as those modeled by the SPECweb benchmark discussed in Section 3.3, may enable this throttling for improved performance. To perform such an operation, the shadow page table handler could be extended to invoke a callback when the number of dirty pages exceeds some high water mark, or it may be configured to pause the virtual machine directly.

Page compression. It has been observed that disk writes typically only alter 5–20% of a data block [*35*]. If a similar property holds for RAM, we may exploit it in order to reduce the amount of state requiring replication, 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 prototyped a basic 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 significant 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.

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 observe a typical compression ratio of 10:1 on the replication stream. Figure 10 shows the bandwidth consumed in MBps for a 60-second period of the kernel compilation benchmark described in Section 3.3. The cache size was 8192 pages and the average cache hit rate was 99%.

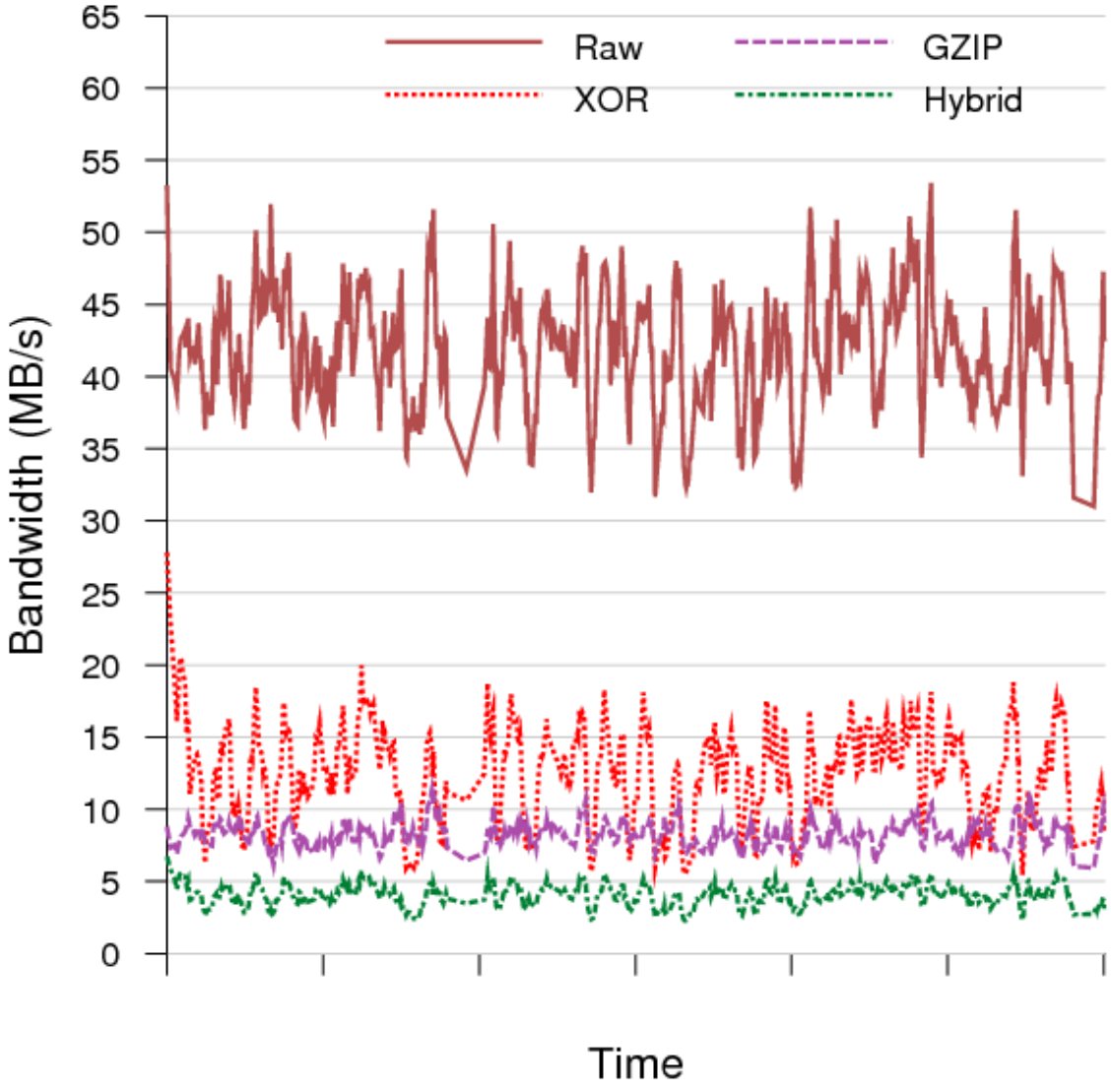

Figure 10: Comparison of bandwidth requirements using various compression schemes.

Compressing the replication stream can consume additional memory and CPU resources on the replicating host, but lightweight schemes such as the XOR compression technique should pay for themselves through the reduction in bandwidth required for replication and consequent reduction in network buffering delay.

Copy-on-write checkpoints. The current implementation pauses the domain at each checkpoint for an amount of time linear in 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 supplying the Xen shadow paging system with a userspace-mapped buffer 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 reading them directly from the guest. When it had finished replicating pages, their space in the buffer could be marked for reuse by the Xen COW module. If the buffer were to become full, the guest could simply be paused, resulting in a graceful degradation of service from COW to stop-and-copy operation.

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. Replication at the virtualization layer 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 than process checkpointing in the operating system, in application libraries or in applications themselves, because it must replicate the entire system instead of individual processes. It can also be less efficient, because it may replicate unnecessary state. 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.

Virtual machine migration. As described earlier, Remus is built on top of the Xen support for live migration [*6*], 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*].

Like 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 the VMM.

Another significant drawback of deterministic replay as 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 access shared memory. There have been some attempts to address this problem. For example, *Flight Data Recorder* [34] 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 [8]. 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.

Virtual machine logging and replay. Virtual machine logging has been used for purposes other than high availability. For example, in *ReVirt* [9], 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* [13] that, like ReVirt, replays the system for forensics only.

Operating system replication. There are many operating systems, such as *Accent* [25], *Amoeba* [18], *MOSIX* [1] and *Sprite* [23], which support process migration, mainly for load balancing. 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 is 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* [22] attempts to introduce a virtualization layer within the linux kernel. This approach must be rebuilt for every operating system, and carefully maintained across versions.

Library approaches. Some application libraries provide support for process migration and checkpointing. This support is commonly for parallel application frameworks such as CoCheck [29]. Typically process migration is used for load balancing and checkpointing is used to recover an entire distributed application in the event of failure.

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 [14] provides a limited form of copy-on-write snapshots of a block store. *Parallax* [33] significantly improves on this design by providing limitless lightweight copy-on-write snapshots at the block level. The *Andrew File System* [11] 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* [26] is a software abstraction over a block device which transparently replicates it to another server.

Speculative execution. Using speculative execution to isolate I/O processing from computation has been explored by other systems. In particular, SpecNFS [20] and Rethink the Sync [21] use speculation in a manner similar to us in order to make I/O processing asynchronous. 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.

5 Future work

This section briefly discusses a number of directions that we intend to explore in order to improve and extend Remus. As we have demonstrated in the previous section, the overhead imposed by our high availability service is not unreasonable. However, the implementation described in this paper is quite young. Several potential areas of optimization remain to be explored. Upon completion of the targeted optimizations discussed in Section 3.4, we intend to investigate more general extensions such as those described below.

Introspection optimizations. Remus 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 on the backup. 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 memory-copying process 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.

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 make it possible for 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 [24], which used an early version of the checkpointing infrastructure provided by Remus.

Disaster recovery. Remus is a product of the SecondSite [7] project, whose aim was to provide geographically diverse mirrors of running systems in order 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, network delay will be an even larger concern. Additionally, network reconfigurations will be required to redirect Internet traffic accordingly.

Log-structured datacenters. We are extending Remus to capture and preserve the complete execution history of protected VMs, rather than just the most recent checkpoint. By mapping guest memory into Parallax [17], our virtual block store designed for high-frequency snapshots, we hope to be able to efficiently store large amounts of both persistent and transient state at very fine granularity. This 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 and so forth).

6 Conclusion

Remus is a novel system for retrofitting high availability onto existing software running on commodity hardware. The system uses virtualization to encapsulate a protected VM, and performs frequent whole-system checkpoints to asynchronously replicate the state of a single speculatively executing virtual machine.

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: HA may simply be “switched on” for specific virtual machines. As with any HA 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 HA facilities, and it does so without requiring special-purpose hardware.

Remus represents a previously unexplored point in the design space of HA for modern servers. The system allows protection to be simply and dynamically provided to running VMs 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.

Acknowledgments

The authors would like to thank their paper shepherd, Arun Venkataramani, and the anonymous reviewers for their insightful and encouraging feedback. We are also indebted to Anoop Karollil for his aid in the evaluation process. This work is supported by grants from Intel Research and the National Science and Engineering Research Council of Canada.

References

[1]

BARAK, A., AND WHEELER, R. Mosix: an integrated multiprocessor unix. 41–53.

[2]

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* (New York, NY, USA, 2003), ACM Press, pp. 164–177.

[3]

BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), ACM Press, pp. 169–179.

[4]

BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault-tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (December 1995), pp. 1–11.

[5]

CHANDRA, S., AND CHEN, P. M. 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)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 91.

[6]

CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association.

[7]

CULLY, B., AND WARFIELD, A. 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.

[8]

DUNLAP, G. *Execution Replay for Intrusion Analysis*. PhD thesis, University of Michigan, 2006.

[9]

DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)* (2002).

[10]

GUPTA, D., YOCUM, K., MCNETT, M., SNOEREN, A. C., VAHDAT, A., AND VOELKER, G. M. To infinity and beyond: time warped network emulation. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (2005).

[11]

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (1988), 51–81.

[12]

HP. NonStop Computing. http://h20223.www2.hp.com/non-stopcomputing/cache/76385-0-0-0-121.aspx.

[13]

KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 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.

[14]

Lvm2. http://sources.redhat.com/lvm2/.

[15]

MARQUES, D., BRONEVETSKY, G., FERNANDES, R., PINGALI, K., AND STODGHILL, P. Optimizing checkpoint sizes in the c3 system. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)* (April 2005).

[16]

MCHARDY, P. Linux imq. http://www.linuximq.net/.

[17]

MEYER, D., AGGARWAL, G., CULLY, B., LEFEBVRE, G., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Parallax: Virtual disks for virtual machines. In *EuroSys '08: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), ACM.

[18]

MULLENDER, S. J., VAN ROSSUM, G., TANENBAUM, A. S., VAN RENESSE, R., AND VAN STAVEREN, H. Amoeba: A distributed operating system for the 1990s. *Computer 23*, 5 (1990), 44–53.

[19]

netem. http://linux-net.osdl.org/index.php/Netem.

[20]

NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM Press, pp. 191–205.

[21]

NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. 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.

[22]

OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 361–376.

[23]

OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The sprite network operating system. *Computer 21*, 2 (1988), 23–36.

[24]

PENG, G. Distributed checkpointing. Master's thesis, University of British Columbia, 2007.

[25]

RASHID, R. F., AND ROBERTSON, G. G. Accent: A communication oriented network operating system kernel. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles* (New York, NY, USA, 1981), ACM Press, pp. 64–75.

[26]

REISNER, P., AND ELLENBERG, L. Drbd v8 – replicated storage with shared disk semantics. In *Proceedings of the 12th International Linux System Technology Conference* (October 2005).

[27]

RUSSELL, R. Netfilter. http://www.netfilter.org/.

[28]

SCHINDLER, J., AND GANGER, G. Automated disk drive characterization. Tech. Rep. CMU SCS Technical Report CMU-CS-99-176, Carnegie Mellon University, December 1999.

[29]

STELLNER, G. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)* (Honolulu, Hawaii, 1996).

[30]

SYMANTEC CORPORATION. Veritas Cluster Server for VMware ESX. http://eval.symantec.com /mktginfo/products/Datasheets /High_Availability/vcs22vmware_datasheet.pdf, 2006.

[31]

VMWARE, INC. Vmware high availability (ha). http://www.vmware.com/products/vi/vc/ha.html, 2007.

[32]

WARFIELD, A. *Virtual Devices for Virtual Machines*. PhD thesis, University of Cambridge, 2006.

[33]

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.

[34]

XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture* (New York, NY, USA, 2003), ACM Press, pp. 122–135.

[35]

YANG, Q., XIAO, W., AND REN, J. 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 (Washington, DC, USA, 2006), IEEE Computer Society, pp. 289–301.

---

[*]
also of Citrix Systems, Inc.

[1]
Remus buffers network and disk output. Other devices, such as the console or the serial port, are presumed to be used for local administration and therefore would not require buffering. However, nothing prevents these devices from being buffered as well.

[2]
Paravirtual 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.

This document was translated from L^AT_EX by H^EV^EA.