Transparent Checkpoints of Closed Distributed Systems in Emulab

Anton Burtsev Prashanth Radhakrishnan [†] Mike Hibler Jay Lepreau

University of Utah, School of Computing
{aburtsev, mike, lepreau}@cs.utah.edu

[†] NetApp
shanth@netapp.com

Abstract

Emulab is a testbed for networked and distributed systems experimentation. Two guiding principles of its design are realism and control of experimentation. There is an inherent tension between these goals, however, and in some aspects of the testbed's design, Emulab's implementers favored realism over control. Thus, Emulab provides wide-ranging control over an experiment's environment and initial conditions, but relatively little control over its *execution*—in particular, the ability to suspend, preempt, or replay the experiment.

We have extended Emulab with a new means of control over experiment execution: the ability to cleanly checkpoint the execution of the set of nodes and networks that comprise an experiment. Conventional checkpoint mechanisms can easily degrade the fidelity of experiment results as a consequence of checkpoint downtimes, overheads of background state saving, and unintended distributed checkpoint synchronization effects. In this paper we demonstrate a checkpointing technique that is *transparent* with respect to the execution of the system under test, almost completely concealing the underlying checkpoint activity.

Building on our checkpoint mechanism, we have implemented two powerful facilities for experiment execution control: the ability to *preemptively swap-out* experiments without losing their run-time state, and the ability to *time-travel* through the run of a system.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—checkpoint/restart; D.4.7 [*Operating Systems*]: Organization and Design—distributed systems

General Terms Design, Performance

Keywords distributed checkpointing, transparent checkpointing, Emulab, network testbed

[†] Work performed while at the University of Utah.

1. Introduction

Emulab is a public testbed for networked and distributed systems experimentation [White 2002]. It manages a pool of physical machines and a large switching infrastructure, providing a flexible way to remotely and automatically configure almost any experimental network within a matter of minutes. It is designed and built in support of two main principles: realism and control of experimentation.

Realism means that the experiment environments created by Emulab are like production environments to a fine level of detail. This is key to ensuring that the results of experiments performed in Emulab are meaningful for predicting how systems will behave when deployed in the real world. Experiments in Emulab are carried out on real hardware (typically PCs) running production operating systems, with nodes connected by real networks. An entire physical infrastructure is allocated for the experiment upon creation—dedicated physical nodes connected by physical, switched Ethernet links. Users are given full freedom to install and configure any component of the software stack inside an experiment.

Control over experiments is essential for effective research and repeatability of experiment results. Most experiments in Emulab never cross the boundaries of their experimental networks, communicating with the external world only through a set of well-defined channels. The "closed" nature of Emulab-based experiments—controlled external dependencies and side-effects—offers a unique opportunity to reproduce, modify, and re-execute experiments in a directed way.

Emulab today provides a great deal of control over an experiment's environment and its initial conditions. However, Emulab permits relatively little control over experiment *execution*. Experiments can be modified and re-launched in a controlled way, but it is not currently possible to suspend, preempt, rollback, or replay them. This is because is it is difficult to provide such controls without sacrificing the realism of the execution environments that Emulab provides, and which Emulab's users demand.

In this paper, we report our experience in providing new controls over experiment execution that do not interfere with realism. We have extended Emulab with two new features: the ability to preemptively suspend experiments without losing their run-time state, and the ability to time-travel through (i.e., rollback and replay) the run of a system. Both rely on the ability to *checkpoint the execution of a computer network*. Our goal for checkpointing is to be *transparent* with respect to the execution of the system under test: transparency here means that a run of the system with checkpointing is the same as it would be without checkpointing *as observed from within the system*.[1] All measurable parameters of the system (e.g., CPU allocation, elapsed time, disk throughput, network delays, and bandwidth) should remain identical to the non-checkpointed execution.

Although checkpointing is a well-studied technique, to our knowledge checkpoint transparency as defined here has never been addressed as a separate problem. In the local (single-machine) case, transparency of a checkpoint typically means having the smallest possible downtime in order to save the system state: a downtime that can be optimized to be on the order of several milliseconds [Cully 2008]. The price for achieving this small downtime is having background work done concurrently with the running system to pre-copy or copy-on-write the necessary state [Osman 2002, Srinivasan 2004, Cully 2008]. This background activity can affect the realism of execution.

In the distributed case, the checkpointer must ensure consistency of a checkpoint across the network [Chandy 1985]. Intuitively, consistency means that the system is saved in a state that can only occur during a failure-free, correct execution [Elnozahy 2002]. To provide this logical correctness, existing solutions may drop, suspend, or replay external communication until the checkpoint commits. This also perturbs realism of execution by introducing packet delays, timeouts, traffic bursts, and replay buffer overflows.

Our approach to retaining realism is to conceal checkpoint activity from the system under test. We employ a straightforward principle: if we can "instantly" freeze the execution and time of an experiment, the system will not be affected by the checkpoint. Checkpoint transparency requires that time and execution of a system under test are stopped atomically from the point of view of that system.

To ensure atomicity of the checkpoint in the local case, we implement a *temporal firewall*: a minimal layer of control inside a system's kernel, designed to isolate time and execution of the checkpointing code from the rest of the system. We virtualize time and atomically stop execution of all code running inside the temporal firewall, making the checkpoint transparent to that part of the kernel code and to all user code. Only the code that participates in the checkpoint runs outside of the firewall, and only that code will see "non-transparent" behavior.

To provide atomicity of a checkpoint across the network (i.e., over an entire Emulab experiment), we perform a coordinated checkpoint. First, to reduce the effects of checkpoint skew between machines, we synchronize the clocks of all the machines that are involved in an experiment. This helps to avoid packets that may be sent or received in the short time that some machines are suspended and others are not. Second, to avoid replay of packets that are in-flight due to network delay, we checkpoint the in-flight network state using Emulab's "delay nodes."

The primary contribution of our work is the implementation of a transparent checkpointing mechanism for a distributed system. We detail the challenges that must be overcome to implement a checkpointer that is *transparent*—i.e., that does not interfere with observable behaviors—to a distributed system under test. We present a novel mechanism, the *temporal firewall*, that can be used as a general way to virtualize time and execution inside the Linux kernel. We show how our emphasis on transparency changes the classical implementation of a distributed checkpoint. Finally, we evaluate our work by implementing two powerful control mechanisms on top of the transparent checkpoint: *stateful swap-out* and *time travel* for experiments in Emulab.

2. Emulab Background

Emulab is a widely used, time- and space-shared testbed facility and "operating system" for networked and distributed system experimentation [White 2002].

To use the Emulab testbed, a user creates an *experiment* that defines the static and dynamic configuration of a network. The static part describes the devices in the network, the links between them, and the configuration of these elements. This includes the operating systems that are loaded onto the devices and the bandwidth, latency, and loss characteristics of network links. Although Emulab can manage many types of devices, in this paper we are concerned with experiments that use PC-class devices connected by wired network links. The dynamic portion of an experiment describes events that are scheduled to occur within an experiment, e.g., program executions.

Once an experiment is created, it can be *swapped in*. Emulab maps the network description within the experiment onto physical resources of the testbed [Ricci 2003]. Emulab allocates nodes and network links from its pool of resources and configures them. Nodes are loaded with the user's choice of software [Hibler 2003] and then booted. Network connections are built by creating VLANs within Emulab's switching infrastructure. Special conditions on network links are established by interposing *delay nodes* on the paths between nodes [Ricci 2007]. A delay node shapes the traffic on the link according to the experiment's specification, but is otherwise transparent to the experimental network.

Once a user's experiment is swapped in, the user can login to the allocated nodes and perform work. In addition to the nodes and networks allocated to his or her experiment, an experimenter can use additional services that are provided by Emulab. First, users can access their resources through a

[1] This definition is different from "checkpoint transparency," which is traditionally defined with respect to implementation: i.e., a checkpoint mechanism is transparent if it requires no modification to the checkpointed system.

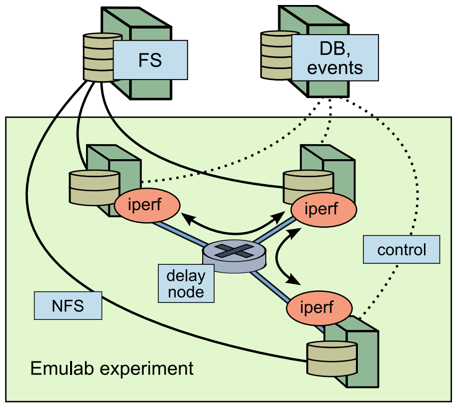

Figure 1. A three-node Emulab experiment running the "iperf" benchmark. Emulab emulates network links using switch configurations and special *delay nodes*. Emulab servers provide control interfaces to experiment nodes, an event system for controlling experiment execution, NFS file storage, and other services.

dedicated control network that provides access to all Emulab machines. Second, to bootstrap, monitor, and control the experiment, users rely on network services that are provided by Emulab: DNS, NTP, NFS-mounted persistent storage, and a distributed event system supporting synchronized activities across nodes. Figure 1 illustrates a simple three-node Emulab experiment and highlights core Emulab services.

Most users perform their work within the *closed world* provided by Emulab: i.e., systems under test interact only within the experimental network, and perhaps with the network services that Emulab provides. In a typical Emulab experiment, an experimenter uses NFS-mounted filesystems to store experiment applications, access experiment scripts, and store experiment results. To orchestrate an experiment execution, users rely on Emulab's event system, which implements an experiment-wide notification system.

When a user is finished working, he or she tells Emulab to *swap-out* the experiment. (A swap-out may also occur if Emulab believes that the experiment is idle.) This releases the hardware resources that were allocated to the experiment, *without* preserving the dynamic state of those resources. The experiment itself stays within Emulab's database, however, so that it may be swapped-in again in the future. If and when that happens, the experiment starts again from its initial state, i.e., with freshly configured resources.

3. Challenges of Transparent Checkpointing

The checkpointing of applications has long been a useful technique. Originally aimed at providing crash recovery, it is now also used for debugging and system analysis [King 2005, Zeller 2002, Tucek 2007, Dunlap 2002, Yang 2006]. This shift in the purpose of checkpoint systems changes the focus for the properties they must provide: applications are expected to behave similarly across checkpointed and non-checkpointed runs, not just run with acceptable overhead.

This section explains the challenges we faced in concealing checkpointing from systems under test, given the constraints imposed by our need to preserve system-level realism for Emulab-based experiments as described above.

3.1 Encapsulation and atomicity

Checkpointing relies on the ability to encapsulate the state of a running system. Encapsulation ensures that a checkpointed system has no external dependencies and thus can be saved and restarted later. Typically, encapsulation is achieved through the use of a virtualization layer, often implemented by a virtual machine. Virtualization is used to create a clear boundary separating system dependencies, that is, a boundary between the hosted "guest system" and the physical hardware managed by the virtual machine "hypervisor."

Traditionally, virtual machines are thought of as sealed containers that fully virtualize all resources, and which therefore can be stopped and resumed at any time. Transparency of a local checkpoint is taken for granted. In practice, however, maintaining full encapsulation at all times can significantly degrade the performance of a virtual system. To improve performance, a virtual environment may choose to violate full isolation and allow sharing of some state between a guest system and the hypervisor. This creates a challenge for checkpoint transparency, because such a virtual machine cannot be stopped atomically: some state of the guest system must be encapsulated at the time of the checkpoint.

For example, to avoid an extra layer of memory virtualization, the paravirtualized interface of Xen [Barham 2003] exposes the physical memory layout to the guest operating system. To modify its page tables, a paravirtualized guest kernel executes code that uses real physical addresses. Thus, it cannot be checkpointed and then resumed on a physical machine with a different memory layout.

Even more work must be done to ensure the encapsulation of virtual devices. These devices serve the dual purposes of protecting and multiplexing the real hardware among guest systems. Like their physical counterparts, virtual devices maintain state: they go through the process of initialization and often establish user sessions as well. Furthermore, to improve performance, virtual devices handle guest requests asynchronously and hence can store a significant number of "in-flight" requests. To achieve encapsulation, during a checkpoint the virtual machine has to shutdown its devices. When resumed, the devices have to be reconnected.

Generally, there are two ways to encapsulate the state shared between a virtual machine and hypervisor. In the first, the virtual machine can be checkpointed externally by the hypervisor, requiring no work on the part of the virtual machine to encapsulate its run-time state. When resumed, the shared state is recreated by the hypervisor. The advantage of this approach is that it is possible to ensure the atomicity of a checkpoint with respect to the execution inside the virtual machine. Unfortunately, it requires the full virtualization of physical memory and rather complex reasoning about virtual devices (e.g., the state they share, the protocol they follow). For example, if it is impossible to describe the shared state declaratively, the restore protocol must replay communication between the virtual machine and hypervisor to recompute it [Swift 2004, Lagar-Cavilla 2007].

In the second approach, shared state is serialized and encapsulated by the virtual machine itself. An advantage of this approach is that a virtual machine can reuse existing shutdown/boot protocols to tear down and re-establish connections to the shared state. As a result, implementation of a checkpoint requires only minimal changes to a virtual machine and hypervisor. Unfortunately, the checkpoint loses its atomicity, because the guest system must participate in its own checkpoint. This violates transparency even if checkpointing is concealed by virtualized time.

Our solution lies between these two approaches. We rely on a virtual machine to encapsulate its state: however, we implement an isolation layer within the guest kernel that hides checkpointing from most of the guest kernel and from all user-level applications.

3.2 Checkpoint synchronization

The asynchronous nature of any distributed system makes it impossible to trigger checkpoints on all nodes simultaneously. An immediate result of unsynchronized checkpoints is that a running system becomes partially suspended during a checkpoint, i.e., some nodes stop before others. This creates two major anomalies observed by the system, packet delays and in-flight packets. Packet delays are caused by a sender being suspended before the receiver. In the worst case, such delays result in packet timeouts and retransmissions. In-flight packets are caused by the receiver being suspended before the sender. (Packets “in flight” due to the bandwidth-delay product of a link are different and are discussed in the next section.)

Once introduced to the system, in-flight packets cannot be removed transparently. In-flight packets are strictly artifacts of unsynchronized checkpointing; their presence and subsequent replay will necessarily affect experiment realism. It is impossible to drop or replay in-flight packets without breaking the fidelity of experimentation.

To avoid losing in-flight packets, a checkpoint algorithm typically logs and replays these packets immediately after completing the checkpoint. Assuming a zero-delay network, new packets may begin to arrive at a node immediately after it is resumed. To avoid out-of-order delivery, these new packets must be queued behind the in-flight packets logged during the checkpoint [Elnozahy 2002].This results in longer delays for the new packets and possible throughput degradation. An obvious way to mitigate these delays is to drain the queue more rapidly by replaying in-flight packets faster. However, this breaks realism by creating an artificial traffic burst.

Another complication of increasing replay speed is that under a high steady-state load, network receive buffers in the system under test will be filled to near capacity, a natural consequence of flow-controlled protocols such as TCP. Therefore, any attempt to inject more packets during replay may result in unintended packet loss if these buffers overfill.

Finally, a checkpointed system will experience the effects of poor synchronization not only when nodes are being suspended for checkpointing, but also when they are being resumed. At restart, some nodes will be restarted before others, leading to problems with packet delays and in-flight packets.

Thus, a primary design principle of our checkpointing solution is to minimize the number of in-flight packets. Unless the downtime of a local checkpoint can be reduced to the order of packet inter-arrival time, it is impossible to avoid packet logging and replay for a traditional, non-coordinated checkpoint [Chandy 1985]. Our approach is to synchronize clocks across the network and implement a coordinated, distributed checkpoint ensuring that all nodes are suspended for checkpoint near-simultaneously.

3.3 Bandwidth-delay product

Irrespective of checkpoint synchronization, a large number of packets may be in flight at the moment of checkpoint due to network delays. The number is limited by the bandwidth-delay product of each link. In contrast to in-flight packets caused by imprecise checkpoint synchronization, it is generally possible to replay bandwidth-delay packets without violating the fidelity of network experimentation. In practice, however, two issues make transparent replay a challenging task: the need for per-path replay, and the implementation of an accurate delay emulator.

The transparency of replay with respect to packet delays requires an implementation of a per-path replay. The problem stems from the potential asymmetry of bandwidth-delay products on different paths. After a checkpoint, the replay log will be dominated by packets from paths with high bandwidth-delay products. When resumed, packets from faster, low-delay paths will experience anomalous delays as they wait for the replay of large bandwidth-delay paths.

Replaying packets from different paths separately, one can ensure in-order delivery for each path, but avoid blocking across different paths. An implementation of this approach, however, requires the knowledge of paths between nodes, which may not be available.

An alternative to logging per path is to log per flow. Flows are convenient as they can be easily identified at the receiver—packets from the same source-destination pair belong to the same flow. A limitation of this approach is that it assumes the ability to identify the source and destination addresses for any protocol.

An additional complication for implementing an accurate replay mechanism is that, in environments such as Emulab, link characteristics are implemented through emulation [Rizzo 1997]. To retain the realism of delay emulation during replay, the rather complex functionality of a delay emulator must be implemented on every node participating in the replay of in-flight packets.

Our solution follows two design choices. First, we do not assume the availability of per-path or per-stream information for Emulab experiments. Emulab is intended to support experimentation with any protocols above Layer 2. Second, instead of implementing an accurate per-node replay mechanism, we rely on Emulab's existing traffic-shaping architecture to capture all in-flight packets, using delay nodes.

4. Checkpoint Implementation

As stated earlier, we use virtualization to provide encapsulation for our checkpoint implementation. Since a key element of Emulab's design is to provide flexibility by allowing configuration for all levels of the software, we rejected process-level containers in favor of low-level, full-system virtualization. To further meet the Emulab goal of realism, we chose a paravirtualization strategy. Our checkpoint mechanism is based on the Xen virtual machine monitor [Barham 2003], a simple, extensible, and mature open-source hypervisor.

We extended Xen's live-migration mechanisms to support live checkpoint, i.e., saving the memory and device state of a running system.[2] The following sections describe how we ensure the transparency of a checkpoint in both the local and distributed cases. We implement our prototype using a paravirtualized Linux kernel supported by Xen.

4.1 Atomicity

The transparency of a checkpoint requires that the time and execution of a system be stopped atomically. Xen's lack of full resource virtualization, as well as the complexity of the state shared between a virtual machine and the hypervisor, forced us to abandon the idea of implementing atomicity entirely by means of the hypervisor, i.e., externally with respect to a virtual machine.

Instead, we maintain atomicity internally, with the aid of the guest kernel. Inside the guest kernel, we create a small control layer that we call a *temporal firewall*. We suspend execution and time inside the firewall. The code needed to reach an encapsulated state runs outside of the firewall. This design allows us to avoid massive changes to the hypervisor and rely on the existing checkpoint code in the guest kernel. At the same time, it ensures checkpoint atomicity for the bulk of the guest system, which is inside the firewall.

The code needed to perform a checkpoint is a mixture of synchronous and asynchronous activities, spread across different parts of the Linux kernel. To isolate this code from that running inside the temporal firewall, we modified the mechanism controlling execution flow inside the Linux kernel (Figure 2).

[2] Similar functionality was concurrently implemented and committed to the mainstream Xen by the Remus team [Cully 2008].

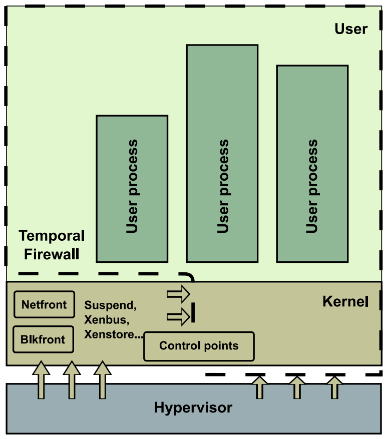

Figure 2. Temporal firewall

To preserve the functionality of the checkpoint mechanism, we identified which activities must be executed outside of the firewall. These consist of the "suspend" thread, virtual device drivers, and the event-driven XenBus handlers used for checkpoint coordination with the virtual machine monitor. No user-level activity is needed, and only six kernel threads need to run during a checkpoint. Although we suspend all device drivers for a checkpoint, block device drivers need their IRQ handlers to run outside of the firewall in order to drain in-flight requests before shutting down connections to shared state. XenBus event channels and watch handlers must also run to provide cross-domain communication.

Although no user-level code is currently necessary during the Xen checkpoint, we have implemented mechanisms to control both user- and kernel-level execution. The Linux kernel provides primitives to control user-level code at the granularity of threads. Since the user-level code executes solely inside user-level threads, by selectively stopping these threads we can control all user-level activity.

Control over the kernel code is less straightforward. There are four main types of activity inside the Linux kernel: kernel threads; interrupt signal handlers (IRQs, interrupts, and exceptions); deferrable functions (soft IRQs, tasklets, and workqueues); and timer jobs. Execution is transferred between these activities in either a synchronous or asynchronous way. However, we only needed to modify a small amount of code to enforce the firewall, i.e., to allow execution of code outside of the firewall.

First, we modified the `schedule` function, which computes the next thread to run, to selectively stop threads inside the kernel. By controlling `schedule`, we can prevent execution of user-level threads and stop the set of kernel threads that process kernel workqueues. At the same time, we do not break scheduling entirely. The threads needed for checkpointing continue to run and share the CPU. Although many scheduling algorithms need some notion of progressing time to account for CPU consumption and to implement preemptive scheduling, the Linux kernel does not. This allows us to run during a checkpoint without any notion of progressing time, with kernel threads behaving cooperatively, yielding the CPU while waiting for external events.

Second, we modified kernel interrupt and soft-IRQ dispatch handlers to selectively suspend interrupt activity. However, this mechanism is not needed in practice, since we suspend all activity that would normally generate interrupts. By suspending all user-level and kernel threads, we stop the origins of most system activity. Furthermore, there are no deferred functions required for checkpointing and virtual device drivers are suspended, so there are no unwanted IRQs coming from outside the guest system. The only IRQs we receive during a checkpoint are from the XenBus virtual device, which is needed for checkpointing and is left running outside of the firewall. Finally, we do not need to firewall exception handlers in general: none should occur in a properly functioning Linux kernel. The only exception that might occur is a page fault, which we run outside the firewall.

We did not need to modify the return path for interrupts and system calls. Although the return path has the ability to pass execution to a soft-IRQ handler or the scheduler, we already control execution of these elements, so there is no need for additional mechanisms.

Lastly, as a consequence of virtualizing time for a guest system (discussed below), we effectively stop the timer interrupt handler during a checkpoint. Thus, timer jobs inside the system will not be scheduled since time does not progress.

4.2 Time virtualization

Traditionally, to keep track of time, an operating system relies on counting periodic timer interrupts. Due to the concurrent execution of multiple virtual machines and contention for the CPU, this approach is impossible in a virtualized environment. First, if the guest system is not running at the moment of interrupt delivery, the interrupt can be delayed or lost. Second, a guest system may not run continuously between interrupts. Finally, the correct operation of the guest kernel and user-level applications may rely on the presence of a high-precision time source.

To address these issues, Xen exposes wall-clock time, system time since boot, and run-time state statistics to the guest system through shared memory regions that Xen updates periodically. Additionally, the guest can account time by requesting periodic timer interrupts, and by accessing the hardware time-stamp counter register (TSC). To obtain the most recent time values, the guest system interpolates time values with nanosecond precision by reading a hardware time-stamp register, which is used to compute the time passed since the last memory update.

To conceal the passage of time associated with a checkpoint, we virtualize all of these time sources and prevent the real time from leaking inside the temporal firewall. First, we prevent the hypervisor from updating the time values during a checkpoint. Second, since both time values are interpolated by accessing the time-stamp register, we restrict the guest system's access to it during a checkpoint. By suspending the system time, we entirely stop time accounting inside the Linux guest. In particular, we prevent updating of the `xtime` and `jiffies` variables, and stop processing of the POSIX timers.

We must also virtualize the hypervisor's run-time state statistics. These statistics reflect the amount of time a guest system spends in one of four states: running, runnable but not scheduled due to CPU contention, blocked by some other system activity, or idle. In the face of contention for the physical CPU, a guest system relies on run-time state statistics to improve its scheduling decisions. To virtualize the run-time state statistics, we modify the hypervisor to suspend accounting of state changes during a checkpoint.

Finally, we virtualize periodic, polling, and single-shot timers by suspending them for the duration of a checkpoint. By stopping the periodic timer, we suspend the delivery of timer interrupts to the guest kernel.

4.3 Checkpoint synchronization

As discussed in Section 3, the transparency of a distributed checkpoint requires atomicity across the network. Essentially we need to stop all of the nodes in a distributed system at the same time.

Ideally, the checkpoint system should be able to trigger a checkpoint immediately in response to any system event (e.g., arrival of a network packet, or execution of a break or watch point). In practice, it is impossible to ensure the atomicity of such an approach because of the lack of checkpoint synchronization across the system. A checkpoint notification mechanism that triggers a checkpoint on every node will experience variable delays due to network transmission, stack processing, and virtual-machine scheduling.

Alternatively, one can rely on clock synchronization algorithms to schedule the execution of a global checkpoint across a network. Potentially, this approach delivers much higher synchronization accuracy, since it is only limited by clock synchronization error. The disadvantage is that checkpoint events must be scheduled in advance on all nodes.

We support both techniques in our implementation using a common mechanism. Recall that Emulab has a high-speed, low-latency control network that provides access to all nodes (Section 2). On top of this we have implemented a fast publish-subscribe checkpoint notification bus. All nodes in the system subscribe to the bus, and any node can publish a notification in order to trigger an action on all nodes.

To support event-driven checkpoints, nodes use the notification bus directly, sending a "checkpoint now" notification. For scheduled checkpoints we use the same mechanism, and also synchronize the clocks on all nodes across the network. Scheduling a checkpoint consists of sending a "checkpoint at time *t*" notification. The time is far enough in the future to allow for propagation and processing of the notifications. Upon receiving the notification, nodes schedule their checkpoints locally. Accurate local timers and clock synchronization algorithms ensure precise checkpoint synchronization across the network. At the end of a checkpoint, we resynchronize all nodes so that they resume simultaneously. This is done using a barrier to detect when all nodes have finished checkpointing, followed by a "resume" notification.

To synchronize clocks we rely on the Network Time Protocol (NTP) [Mills 1991]. Under perfect LAN conditions, NTP provides clock synchronization with an error of 200 μs. To ensure such conditions, we run NTP over the Emulab control network. We chose NTP for two reasons. First, we do not want Emulab to depend on hardware synchronization devices [IEEE 2004, Micheel 2001], as it will increase cost and restrict further deployment of the testbed. Second, we are not sure that existing implementations of TSC-based protocols [Veitch 2004] perform well in the face of varying CPU temperature and dynamic-frequency scaling.

4.4 Transparency of the network core

To transparently handle in-flight packets generated due to network delays, we leverage Emulab's delay nodes. As described in Section 2, Emulab can shape a network link (set bandwidth, latency, and loss characteristics) by interposing a delay node on the link. Thus, instead of implementing a delay-accurate replay mechanism on all nodes in a system under test, we need only checkpoint the network core—i.e., the set of delay nodes. This lets us capture most of the in-flight packets in the network. Because the links between a delay node and the endpoints are zero-delay links, only packets that are physically in flight need to be logged at the receiving node. This reduces the size of the replay log to a number bounded by the synchronization error.

There are two possible methods for checkpointing delay nodes. One is to run delay nodes as virtual machines under the hypervisor and rely on our node checkpoint mechanism. The other is to implement a new checkpoint mechanism for the traffic-shaping subsystem. In Emulab, this subsystem is the FreeBSD Dummynet module [Rizzo 1997]. We chose the latter approach for two reasons.

First, the overhead of virtualization seems to be prohibitive for implementing an accurate, high-speed delay emulation as required in Emulab. Xen's network path has been shown to become CPU-bound under high loads [Cherkasova 2005, Santos 2008].

Second, Xen's time architecture provides no guarantees about the accuracy of timer-interrupt frequency or jitter. Periodic timer interrupts are critical for Dummynet, which relies on the system clock to implement delay scheduling. Furthermore, to reduce the overhead of time management, Xen limits the resolution of a guest timer's interrupt to 1 ms.

Our delay-node checkpoint mechanism saves the state associated with Dummynet. This state consists of a hierarchy of pipes, router queues, and the packets queued in those pipes and queues. For the checkpoint, we implement functions serializing and deserializing the state of this hierarchy.

Similar to what we did for Xen, we implemented a live-checkpoint mechanism within the Dummynet module. During a checkpoint we suspend Dummynet and serialize the state non-destructively. After the checkpoint completes, we resume execution by unblocking Dummynet and virtualizing time to account for the time spent in the checkpoint.

5. Stateful Swapping

The demand for physical resources in Emulab often exceeds its capacity. To improve resource utilization, Emulab provides a weak form of time-sharing: *swap-out* of inactive experiments. Upon swap-out, the run-time state of an experiment—i.e., the memory and disk state of experiment nodes—is freed and lost. If the experiment is later swapped in again, a user must manually recover his or her previous "node-local" state.

Using our transparent checkpoint mechanism, we extended Emulab with the ability to swap-out experiments without losing their node-local state. Our implementation ensures that the entire period of inactivity—the swapped-out time—is transparent to the experiment. Effectively, we extended Emulab with a coarse-grained scheduling mechanism similar in nature to OS process scheduling.

There are two significant challenges associated with making this mechanism practical in Emulab. The first is efficient handling of potentially large experiment state, due to the large local disks on nodes. The second is dealing with experiment connections to the "external world," including Emulab infrastructure and the Internet, which are not part of the checkpointed environment.

5.1 Disk state

To be practical, stateful swapping has to be fast. The run-time state of an experiment typically consists of tens of gigabytes of data, mostly on node disks. The way in which disk state is saved and restored at swap time is critical to the performance of the system, since naive approaches might require tens of minutes or even hours to transfer this state to and from persistent storage. We employ multiple optimizations to reduce the swapping time.

Several observations help to reduce the amount of state that must be transferred upon swap-out. First, an experiment changes relatively little disk state between swap-in and swap-out. Thus, it is useful to save only the changes generated since the last swap-in. Second, nodes within and across experiments use a relatively small set of base filesystem images, which can be cached on the experimental nodes and shared across experiments.

Based on these observations, we split the virtual disk of a guest system into three components (Figure 3, left side): an immutable base file system image ("Golden image"), the aggregated disk changes made with respect to the base image for all previous swap-ins ("Aggregated delta"), and all disk changes since the current swap-in ("Current delta"). These

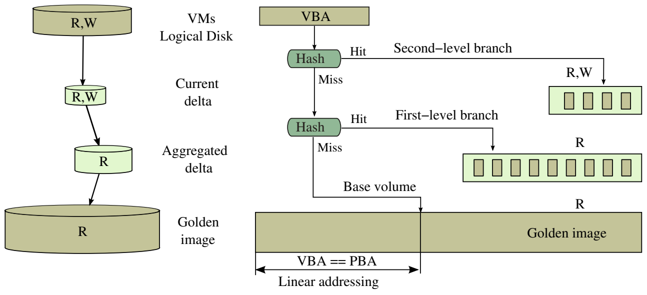

Figure 3. Three-level branching storage with a corresponding virtual block address translation scheme

parts are stitched together to provide a logical disk by using a copy-on-write storage mechanism.

This partitioning has several benefits. First, saving an experiment's state during swap-out requires saving only the small current delta. Second, the golden image can be cached in advance, requiring only that the aggregated delta—which is typically much smaller than the golden image—be transferred during the swap-in. Finally, because the golden image is immutable, it can be shared across virtual machines that reside on a common physical node.

To further reduce the size of the current delta, we eliminate most of the blocks freed by the guest filesystem. The need for, and complexity of, this optimization stems from the semantic gap between the hypervisor and the guest system. Xen virtualizes disks at the block level. Thus, filesystem-specific information is not available to the swapping system. We verified that this optimization is crucial by running a `make` followed by `make clean` command on a Linux kernel source tree. Free-block elimination reduces the delta size from 490 MB to 36 MB. We eliminate free blocks by implementing filesystem-specific plugins to snoop on writes at the level below the guest system. A plugin constructs a free-block metadata map that is consistent with respect to the data blocks on the disk. We have implemented free block elimination for the Linux ext3 filesystem.

To reduce the latency of a "context switch" between two experiments (i.e., swap-out one, swap-in the other), we rely on background disk data transfer and pipeline swap-out and swap-in. During a swap-in, the virtual machine is resumed as soon as its memory image is downloaded. The entire disk state is either paged on demand (individual disk blocks copied to local disk on first reference) or lazily copied (continuous background copying of the entire disk state) to the experiment node. Similarly, during the swap-out we eagerly begin copying the delta image to persistent storage before the guest's execution is suspended.

5.2 The external world

If an experiment interacts with the external world, a checkpoint may distort execution of the experiment in two significant ways. First, since time is suspended during swap-out, time inside the experiment progresses slower than outside. An experiment and the external world can confuse each other with their differing notions of time. Second, if an interaction with the external world is stateful, any "outside" component of that state will not be preserved across swap-out, leading to a mismatch when swapped back in. While it is impossible to provide a general solution for supporting all possible types of external world interactions, it is possible to solve the problem for the Emulab control interface.

With the exception of the event system, dealing with these issues is relatively straightforward. To conceal time differences between an experiment and Emulab-provided services, we rely on the knowledge of the DNS, NTP and NFS protocols to *transduce* timestamps embedded in protocol messages. We convert timestamps found in the inbound packets to the guest system's virtual time, and those in the outbound packets to the actual time. We implement this technique for NFS by filtering NFS commands containing timestamps. Fortunately, most Emulab control services are stateless: DNS, NTP, and NFS version 2 are stateless by design, so there are no issues related to state saving.

Emulab's event service is both stateful and time-aware. The per-experiment event scheduler runs on an Emulab server and dispatches events to experiment nodes at the appropriate times, optionally receiving notifications when events complete. When an experiment is swapped out, the event scheduler could be suspended, but this does not address the fact that connections to event agents can be stateful TCP connections. Our solution is to move the event scheduler into the closed world of the experiment. Although we have not yet implemented this, there is no need for the scheduler to run on an Emulab server; it is strictly historical.

5.3 Implementation status

We have implemented most parts of the stateful swapping system and demonstrated it in a non-production version of Emulab. Details of the implementation can be found in [Radhakrishnan 2008]; we summarize the work here.

Our copy-on-write disk storage is based on the snapshots provided by the Linux Logical Volume Manager (LVM) [Redhat 2006]. We modified LVM to provide the semantics of branching storage, i.e., support for a tree-like structure with recursive immutable snapshots and any number of mutable branches. Furthermore, we optimized LVM to achieve an order-of-magnitude improvement in COW write performance. We continue to work on improving the read performance for deeply nested snapshot trees.

To implement background data transfer, we take advantage of LVM mirror volumes, a facility that allows RAID1-style redundancy. By locating half of a mirror volume on a remote machine across NFS, we get automatic remote redirection of reads and remote mirroring of writes. The original implementation of LVM mirror volumes synchronizes data aggressively, causing a significant impact on VM performance and thus affecting the realism of experimentation.

To address this, we added a rate-limiting function that slows synchronization activity relative to normal system I/O.

We optimize our branching storage for a three-level store. We implement the copy-on-write portion as a redo log, i.e., redirecting writes to the log and keeping the original disk intact. The log is optimal for our case as it avoids writes to the immutable aggregate delta and base filesystem images. It also allows us to rely on locality of access provided natively by the base system image, without getting into the complexity of handling defragmentation typical in write-anywhere approaches. When performing a logical to physical block address translation, writes incur the cost of a single hash lookup to index into the log (Figure 3, right side). Reads incur the cost of two lookups: first in the write log and then in the aggregated delta. If both fail, reads are redirected to the base filesystem image, which uses linear addressing.

To optimize write speed, we ensure that the filesystem block size is always a multiple of an LVM block size. Thus, copy-on-write is always a complete overwrite and never requires a read-before-write.

We further optimize reads by preserving data locality in the base filesystem and the aggregated delta. Allocation of data in the on-disk delta representing a COW branch happens on a first-write basis, i.e., when a block is written for the first time since the creation of the branch. Over the course of several swap-outs and swap-ins, the aggregated delta is repeatedly merged with a disk delta. Over time, data locality in these branches may be lost, resulting in poor read performance due to excessive disk seeks. Thus, when we merge the disk and aggregated deltas offline after a swap-out, we reorder blocks in the aggregated delta to restore locality.

6. Time-Travel System

Another compelling use of transparent checkpointing in Emulab is experiment time-travel. The ability to navigate and inspect the execution history of a system is a powerful aid in debugging and analyzing complex networked systems. Backward navigation unrolls the execution of a system to a state in its past. Forward navigation replays a system run, reproducing its execution deterministically or non-deterministically. Designing and implementing a facility to make this possible for realistic, large, and long-running distributed systems was an initial motivation for our work.

Time-travel in Emulab allows a user to preserve the execution of an experiment and later, if desired, play it forward from any point in time that the experimenter deems interesting. For example, if a feature begins to develop only after a certain period of time or develops randomly within the run, time-travel would allow continuation of the experiment around the points of interest. Similarly, if a phenomenon is observed mid-way through an experiment run and the user wants to understand the circumstances under which it occurs, it is possible to restart the run from a point just before the appearance of the phenomenon. The user could revisit the point of appearance many times, each time with a different set of environmental conditions.

Using our distributed checkpoint mechanism, we have implemented a prototype for time-travel with non-deterministic replay. The prototype captures the original run of an experiment by frequent checkpointing during its execution. Backward navigation is implemented by restarting the experiment from a particular checkpoint. The closed-world assumption and the control mechanisms described in Section 5 ensure that experiments are not broken during replay due to external dependencies. The control mechanisms also allow the user some limited experiment control from the outside.

The transparency of checkpoints is essential to providing frequent checkpointing without affecting the realism of experimentation. This allows a user to introspect, or debug via replay, any unexpected execution without the need to recreate the faulty situation "with debugging turned on."

After time-travelling to a point in the past, intentional state mutation or non-determinism may change the behavior of the replayed execution relative to the original run. As such, every replay run creates a new branch in the execution history of a system. The result is that time-travel sessions form a tree, with internal nodes representing checkpoints and leaves representing checkpoints or active executions. This is in contrast to deterministic replay, where state mutations are disallowed, and checkpoints are arranged in a linear chain.

To support iterating over a single point of execution under different experimental conditions, we rely on our branching storage system. We use one of the two local disks available on most Emulab nodes to store disk and memory snapshots. This allows us to store time-travel trees with thousands of nodes and provide support for various system analyses. For example, a model checker could branch from past execution checkpoints to test unexplored states.

Our current and future work focuses on support for varying degrees of determinism during replay and turning our time-travel system into a platform for different types of system analyses. During replay we plan to allow relaxed determinism, which permits a user to mutate the system state and perturb selected system inputs. For instance, users should be able to skew interrupt delivery times, reorder packets, and dilate system time for the purposes of automated bug finding [Tucek 2007, Qin 2005], model checking [Yang 2006, Killian 2007], delta debugging [Zeller 2002], capacity testing [Gupta 2006; 2008], performance debugging, and other analyses. In this way, the time-travel system could present non-determinism as a "knob" that could be turned up to increase perturbation during replay, or turned down to enforce deterministic replay. To provide instruction-level accurate replay, we plan to work on a deterministic replay mechanism similar to TTVM [King 2005] and SMP-ReVirt [Dunlap 2008]. Deterministic replay will provide support for distributed system debugging [Geels 2007] and forensic intrusion analysis [Dunlap 2002].

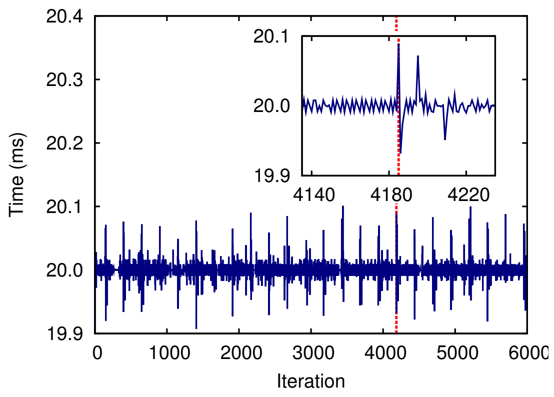

Figure 4. Periodic checkpointing of a microbenchmark executing a 10 ms sleep in a loop

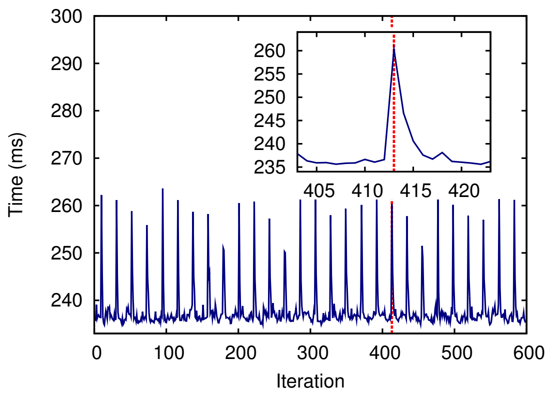

Figure 5. A microbenchmark executing a CPU-intensive job in a loop

7. Evaluation

In this section we evaluate our implementations of transparent checkpointing and stateful swapping. All tests were run in Emulab on experiments configured to use "pc3000" nodes connected via 1 Gbps Ethernet experiment links. The pc3000 nodes are Dell PowerEdge 2850 rackmount PCs, each configured with a single 3.0 GHz 64-bit Xeon processor with hyperthreading support, 2 GB RAM, and two 146 GB, 10,000 RPM SCSI disks. On these we load 6 GB virtual machine disk images containing a 32-bit Linux Fedora Core 4 system configured on top of an ext3 root filesystem. Each virtual machine is configured with 256 MB of memory. All machines are connected to the Emulab control network, a dedicated 100 Mbps Ethernet LAN.

7.1 Transparent checkpointing

Our first tests were designed to evaluate the transparency of the checkpointing mechanism. To do this we quantify what effect checkpointing has with respect to four measurable metrics: time, CPU allocation, network I/O, and disk I/O, as observed by the system under test.

Time. We evaluated our time virtualization mechanisms by running a synthetic microbenchmark that measures the passage of wall-clock time. The benchmark invokes the Linux `usleep` function in a loop, sleeping for 10 ms in each loop. After every sleep, we read the system time directly using the `gettimeofday` function to measure the actual time of each iteration. The measured iteration time for a system with no checkpointing is 20 ms. We then run the benchmark while performing a checkpoint every 5 seconds. The measured time for each iteration during the checkpointed run is shown in Figure 4, with the inset figure showing detail of a 100-iteration interval around a single checkpoint. (The time of that checkpoint is shown by the dashed vertical line.)

While this figure shows a detectable pattern of spikes corresponding to checkpoints, the impact is minor. During normal intra-checkpoint execution, for 97% of the iterations the timer is accurate to within 28 μs. The inset plot shows

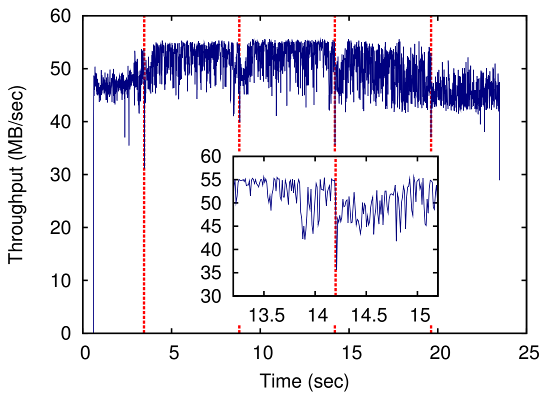

Figure 6. Iperf running on a 1 Gbps network

that a checkpoint briefly increases measurement error to $\pm 80\,\mu$s. This defines an empirical limit on the transparency of time for local checkpoint in our implementation.

CPU allocation. The previous benchmark shows that we effectively virtualize time as seen directly by user code. Another concern is how well the temporal firewall works for concealing the checkpoint from a CPU-intensive task. We evaluated the transparency of a checkpoint with respect to CPU time allocation using another synthetic microbenchmark. This benchmark executes a CPU-intensive workload in a loop. For every iteration, we measure how long it took to complete the work. The measured time for a system with no checkpointing is 236.6 ms. For 90% of the iterations the work was completed within 9 ms of this average value. We then run the benchmark while performing a checkpoint every 5 seconds. The measured time for each iteration during the checkpointed run is shown in Figure 5, with the inset figure showing detail of a 20-iteration interval around a single checkpoint (again, indicated by a dashed vertical line).

Once again there is a detectable pattern of spikes at each checkpoint. Here proper time virtualization and atomicity provided by the temporal firewall ensure allocation of CPU

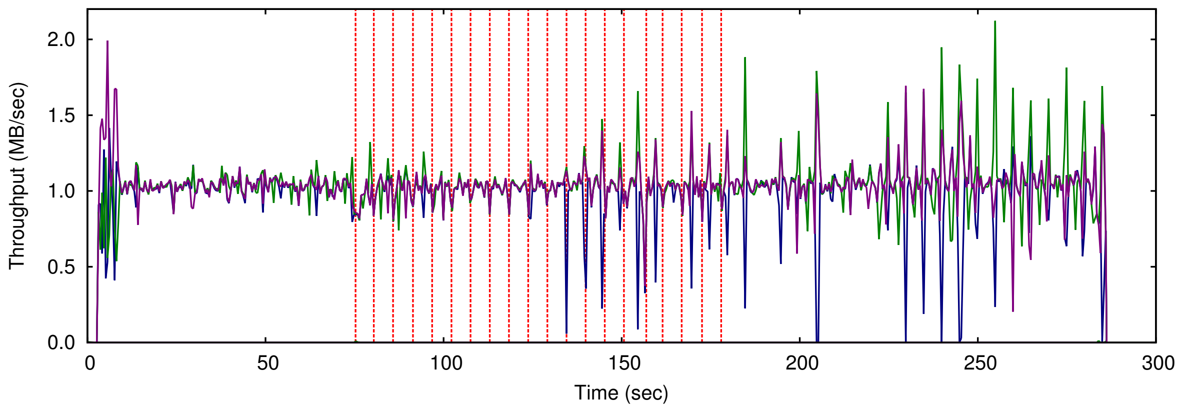

Figure 7. Four-node BitTorrent experiment

time to within 27 ms of the expected value. Note that this is considerably less accurate than demonstrated in the previous benchmark. As this benchmark is extremely sensitive to background activity, this discrepancy is explained by residual checkpoint-related activity. To verify this hypothesis, we ran the same benchmark, but instead of performing checkpoints, we periodically ran different jobs in the the Xen privileged domain. Even the simplest operations affected performance of the benchmark. Listing the contents of the root directory with the `ls` command, computing a checksum of the Linux kernel binary with `sum`, and simple invocation of the Xen daemon (`xm list`) increased the time of the benchmark by 5–7 ms, 13–17 ms, and 130 ms respectively.

Network I/O: Microbenchmark. To verify transparency of a checkpoint with respect to network behavior, we constructed an experiment with two nodes communicating over a 1 Gbps network. In this experiment we ran iperf, sending a TCP stream in one direction between the nodes, and checkpointed the iperf session every 5 seconds. Network measurements were obtained by capturing a packet trace on the receiving node.

Figure 6 shows the throughput of a checkpointed session observed over a period of 25 seconds; dashed lines show the checkpoint events. The plotted throughput values are averages taken over 20 ms intervals throughout the run. The inset graph shows detail of a two-second window surrounding a checkpoint. The slight drop in throughput immediately following the checkpoint is attributable to interference between background checkpoint activity and the guest system.

Analyzing the packet trace from the session, we observed that the first four checkpoints cause inter-packet arrival delays of 5801 μs, 816 μs, 399 μs, and 330 μs respectively, for the four intervals that span the checkpoint boundaries. This is in contrast to an average 18 μs inter-packet arrival time observed throughout the run. The packet delays are the result of a fundamental limitation on the transparency of our checkpoint implementation, which is defined by the accuracy of the clock synchronization algorithm. We inspected the packet trace to confirm that checkpoints caused no retransmissions, double acknowledgements, or changes of window size for the TCP session.

Network I/O: Real application. To evaluate how distributed checkpoint performs on a more realistic application, we conducted a multi-node BitTorrent experiment. BitTorrent is a popular peer-to-peer program for cooperatively downloading large files. Peers act as both clients and servers: once a peer has downloaded a part of a file, it serves that part to other peers. To get more predictable behavior, we modified BitTorrent to use a static tracker.

The experiment was configured with four Emulab nodes—one seeder and three clients—on a 100 Mbps LAN. The clients all download the same 3 GB file, initially present only on the seeder node. We started checkpointing of the experiment 70 seconds into the run, giving BitTorrent time to reach a steady state. We then took periodic checkpoints every 5 seconds for another 100 seconds, at which point we stopped checkpointing and let the system run for another 100 seconds. Network measurements were obtained by capturing a packet trace on the seeder node. Figure 7 plots the outgoing throughput as observed on the seeder node. The three lines in the plot, which largely overlap with one another, represent traffic to the three different clients.

The graph shows that each of the clients has an average throughput around 1 MB/sec, although the application is bursty. Each checkpoint causes a small drop in throughput around the checkpoint event as we observed in our previous experiment; however, this disturbance is small. Moreover, repeated checkpointing does not change the obvious "center line" that runs through the graph.

Disk I/O: Copy-on-write performance. Since a copy-on-write storage system is an integral part of the checkpoint implementation, we evaluate its effect on the fidelity of experimentation. To do this, we configured the popular Bonnie++ benchmark [Coker 2003] to operate on a 512 MB file—twice the size of the guest system's memory. We then ran Bonnie++ on three configurations of a guest system: a

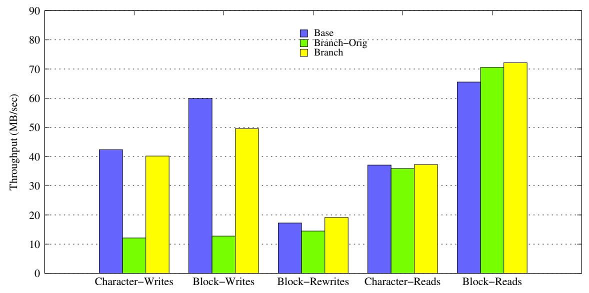

Figure 8. Comparison of a copy-on-write storage against native disk speed

raw disk partition (Base), a two-level branching storage implemented by LVM (Branch-Orig), and a two-level branching storage with our modifications (Branch). The results are shown in Figure 8.

On a freshly created disk, sequential block writes to a branch incur 17% overhead compared to a native disk. This is explained by additional seeks performed by the branch device to update the on-disk metadata regions that are distributed over the entire disk. We conducted additional tests to check how this behavior changes as a disk ages and the metadata regions are filled up. Not surprisingly, we found that metadata overhead disappears and branch storage performs within 2% of the native disk.

Block writes to the original LVM are 74% slower than writes to our modified branch device. This is a result of our optimization eliminating the read-before-write overhead described in Section 5. As the disk ages and more writes go directly to a branch, read-before-write overhead disappears, and Branch-Orig and Branch perform equally well.

Disk I/O: Background data transfer. Background data transfers also play a big role in out stateful swap-out implementation, so we studied their effect on experiment disk I/O. We did this by copying a large file to simulate a disk-intensive workload, while measuring throughput to disk at one-second intervals. We ran the benchmark in three scenarios: with no swap activity occurring, during swap-in (lazy copy-in), and during swap-out (eager copy-out). Figure 9 plots the write throughput for these three cases. For the swap-out case, the swap is triggered 60 seconds into the run. In the graph, this eager copy-out case looks very similar to the run with no swapping, with only a 9% increase in execution time. However, the lazy copy-in case has a more noticeable overhead, resulting in a 19% increase in execution time and a 45% drop in throughput. This is caused by a limitation of our rate-limiting feature, leading to more aggressive prefetching of data compared to the swap-out case.

7.2 Stateful swapping

Finally, we offer some preliminary insight into the performance of the stateful swapping implementation. To measure the swapping performance, we used a single node experiment that we swapped in, and then out, four times consecutively. During every swapped-in session, the experiment generated 275 MB of disk data. When swapping out, node state was transferred over the control network to the Emulab file server. Although use of the 100 Mbps control network is clearly a bottleneck and not ideal for large data transfers (as compared to a 1 Gbps link or a SAN), Emulab is intentionally designed not to depend on "high end" infrastructure in order to lower the barrier for its deployment at other sites.

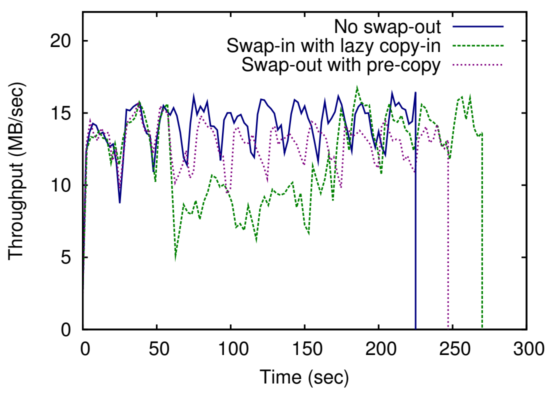

Figure 9. Effect of a background data transfer on disk throughput

The initial swap-in took eight seconds when the base system image was cached on the physical VM-hosting node. If that image was not cached, it took an additional 60 seconds to download it. Subsequent swap-ins required downloading the aggregated delta image that increased in size each swap cycle. Without the lazy swap-in optimization, swap-in times increased to over 150 seconds by the fourth iteration. With the optimization, swap-in times were constant at approximately 35 seconds.

For swap-out, since the experiment was producing the same amount of new data at each iteration, the times were constant at 60 seconds. To simulate worst-case behavior for the eager swap mechanism, we re-ran the tests with a disk intensive workload running in the experiment. This resulted in a 20% slowdown for swap-out times, attributable to two factors. First, blocks overwritten during pre-copy may be sent more than once. Second, we intentionally limit the rate of pre-copy with a rate-limiting function to reduce interference with the guest system's execution.

8. Related Work

A classic work by Elnozahy [2002] provides a comprehensive survey of distributed checkpointing protocols. Traditional approaches concentrate on consistency of the checkpoint. In contrast to these approaches, our focus is systems-level transparency to software under test.

Multiple systems optimize downtime of the checkpoint. For instance, Remus implements an efficient checkpoint capable to checkpoint a Xen virtual machine 40 times per second [Cully 2008]. Remus heavily relies on the background copy-on-write state tracking for both disk and memory; essentially, the guest system never leaves the checkpoint mode. Furthermore, Remus delays external I/O until checkpoint commits. Both background state-saving and buffered I/O may harm realism of experimentation. Zap [Osman 2002] and Flashback [Srinivasan 2004] provide lightweight, process-level checkpointing. In contrast, our approach checkpoints systems from the OS up.

Xen implements live migration of virtual machines without stopping interactive services [Clark 2005]. While Xen's goal was disallowing disruption during migration, we have a stricter requirement of preventing the VM's perception of a checkpoint.

Replay debugging [King 2005], delta debugging [Qin 2005, Zeller 2002], and model checking systems [Yang 2006, Killian 2007] implicitly rely on transparency of the checkpointing and can directly benefit from our work.

Copy-on-write and branching storage systems have a long history. ZFS [Sun Microsystems, Inc. 2008] is a promising production system, but was unavailable as open-source at the time we started this project. Parallax [Meyer 2008] is a recent implementation of a block-level copy-on-write storage optimized to support large number of snapshots. It uses a radix tree for block address translation. We believe our simpler indexing scheme, ability to leverage linear addressing, and data locality for the base system image better match needs of the three-level storage used by stateful swap-out.

Recent work by Gupta [2006; 2008] virtualizes time to slow down the passage of time from a virtual machine's perspective. The basic goal of this work is to subject the guest system to network speeds much higher than what is physically possible.

9. Conclusion

We have implemented a novel, transparent checkpointing facility for distributed systems in the Emulab network testbed. Emulab seeks to provide its users with test environments that are both realistic and highly controllable—and until now, in order to maintain realism, Emulab has not provided checkpointing to its users. In this paper we have reconciled realism and control through the design and implementation of checkpointing facility that is transparent to systems under test within Emulab. Transparency is essential to make checkpointing a precise research tool, capable of providing fidelity of distributed systems analysis. Transparency is achieved through the design and use of a "temporal firewall," which we use to ensure the atomicity of suspending time and execution. This firewall can be used as a general mechanism to change the perception of time for the system under test and conceal various external events.

Based on transparent checkpointing, we have implemented two powerful means of control over experiment execution. The first, stateful swap-out, provides a way to truly time-share the resources of a computer network. The second, time travel, creates a platform for experimenting with various debugging, testing, and model-checking techniques. Each of these facilities advances our notion of Emulab as an operating system for a computer network. Furthermore, each enhances Emulab as a basis for performing sound and effective distributed systems research.

Acknowledgments

We thank Kevin Atkinson for helping us with many of experiments described in Section 7. We also thank Eric Eide for many fruitful discussions about this work, and for providing significant editing and typesetting help. Finally, we thank the anonymous EuroSys reviewers and our shepherd, Danny Dolev. Their comments helped us to improve this paper greatly. This material is based upon work supported by the National Science Foundation under Grant No. 0524096.

References

[Barham 2003] Paul Barham et al. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, Bolton Landing, NY, 2003.

[Chandy 1985] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[Cherkasova 2005] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proc. USENIX*, pages 387–390, Anaheim, CA, 2005.

[Clark 2005] Christopher Clark et al. Live migration of virtual machines. In *Proc. NSDI*, pages 273–286, Boston, MA, May 2005.

[Coker 2003] Russell Coker. Bonnie++, 2003. `http://sourceforge.net/projects/bonnie/`.

[Cully 2008] Brendan Cully et al. Remus: high availability via asynchronous virtual machine replication. In *Proc. NSDI*, pages 161–174, San Francisco, CA, 2008.

[Dunlap 2002] George W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, pages 211–224, Boston, MA, December 2002.

[Dunlap 2008] George W. Dunlap et al. Execution replay for multiprocessor virtual machines. In *Proc. VEE*, pages 121–130, Seattle, WA, March 2008.

[Elnozahy 2002] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34 (3):375–408, 2002.

[Geels 2007] Dennis Geels et al. Friday: Global comprehension for distributed replay. In *Proc. NSDI*, pages 285–298, Cambridge, MA, April 2007.

[Gupta 2006] Diwaker Gupta et al. To infinity and beyond: time-warped network emulation. In *Proc. NSDI*, pages 87–100, San Jose, CA, May 2006.

[Gupta 2008] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. DieCast: Testing distributed systems with an accurate scale model. In *Proc. NSDI*, pages 407–421, San Francisco, CA, April 2008.

[Hibler 2003] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with Frisbee. In *Proc. USENIX*, pages 283–296, San Antonio, TX, June 2003.

[IEEE 2004] IEEE. IEEE 1558 standard for a precision clock synchronization protocol for networked measurement and control systems, September 2004.

[Killian 2007] Charles Killian et al. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. NSDI*, pages 243–256, Cambridge, MA, April 2007.

[King 2005] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX*, pages 1–15, Anaheim, CA, April 2005.

[Lagar-Cavilla 2007] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. VMM-independent graphics acceleration. In *Proc. VEE*, pages 33–43, San Diego, CA, 2007.

[Meyer 2008] Dutch T. Meyer et al. Parallax: virtual disks for virtual machines. In *Proc. EuroSys*, pages 41–54, Glasgow, Scotland, March–April 2008.

[Micheel 2001] Jörg Micheel, Stephen Donnelly, and Ian Graham. Precision timestamping of network packets. In *Proc. 1st ACM SIGCOMM Workshop on Internet Measurement (IWM)*, pages 273–277, San Francisco, CA, November 2001.

[Mills 1991] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Trans. Comm.*, 39:1482–1493, 1991.

[Osman 2002] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: a system for migrating computing environments. In *Proc. OSDI*, pages 361–376, Boston, MA, May 2002.

[Qin 2005] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proc. SOSP*, pages 235–248, Brighton, UK, October 2005.

[Radhakrishnan 2008] Prashanth Radhakrishnan. Stateful-swapping in the Emulab network testbed. Master's thesis, University of Utah, August 2008.

[Redhat 2006] Redhat. LVM2 Resource Page, 2006. `http://sourceware.org/lvm2/`.

[Ricci 2003] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM Comput. Commun. Rev.*, 33(2):65–81, April 2003.

[Ricci 2007] Robert Ricci et al. The Flexlab approach to realistic evaluation of networked systems. In *Proc. NSDI*, pages 201–214, Cambridge, MA, April 2007.

[Rizzo 1997] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[Santos 2008] Jose Renato Santos et al. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proc. USENIX*, pages 29–42, Boston, MA, 2008.

[Srinivasan 2004] Sudarshan M. Srinivasan et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. USENIX*, pages 29–44, Boston, MA, June–July 2004.

[Sun Microsystems, Inc. 2008] Sun Microsystems, Inc. ZFS, June 2008. `http://www.opensolaris.org/os/community/zfs/`.

[Swift 2004] Michael M. Swift et al. Recovering device drivers. In *Proc. OSDI*, pages 1–16, San Francisco, CA, December 2004.

[Tucek 2007] Joseph Tucek et al. Triage: Diagnosing production run failures at the user's site. In *Proc. SOSP*, pages 131–144, Stevenson, WA, October 2007.

[Veitch 2004] Darryl Veitch, Satish Babu, and Attila Pàsztor. Robust synchronization of software clocks across the Internet. In *Proc. 4th ACM SIGCOMM Conf. on Internet Measurement (IMC)*, pages 219–232, Taormina, Italy, October 2004.

[White 2002] Brian White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Boston, MA, December 2002.

[Yang 2006] Junfeng Yang et al. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4): 393–423, November 2006.

[Zeller 2002] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proc. FSE*, pages 1–10, Charleston, SC, November 2002.