Time-Travel for Closed Distributed Systems

Distributed Systems (cs7460) Project Report[*]

Anton Burtsev and Prashanth Radhakrishnan
{*aburtsev, shanth*}*@flux.utah.edu*

1 Introduction

We report the design and a premilinary implementation of a distributed time-travel system for the Emulab Network Testbed [22]. Time-travel is the ability to move backward and forward in time within the logged execution, reproducing it deterministically or non-deterministically an arbitrary number of times. Our goal is to provide the ability to time-travel and debug execution across different nodes, as Emulab experiments are usually distributed. We achieve this goal with logging and replay of the complete operating system along with user-level processes. To this end, we run the operating system over the Xen virtual machine monitor [3]. To improve the scalability and efficiency of the system, and to achieve state mutations during time-travel, we employ the techniques of cooperative logging and relaxed consistency (non-deterministic) during replay, respectively.

The report is laid out as follows. Section 2 talks about the related work in this field. Section 3 introduces the basic techniques in logging and replay, namely, deterministic logging, non-deterministic logging and cooperative logging. The subsequent three sections elaborates on these techniques. Section 7 has a detailed discussion on checkpointing and its use in time-travel. Section 8 lists some issues we have identified in implementing debugging capability over our time-travel system. Section 9 presents a preliminary implemenation of our system, where we have made some simplifying assumptions, and the evaluation results from the implementation are in section 10. We conclude with a discussion on the lessons learnt and future work in section 11.

2 Related work

Last year, the ability to log and replay a complete operating system execution was demonstrated by Chen et. al [12]. Chen's work was based on previous results in the area of fault-tolerant computing, where deterministic replay was used to reproduce exact the execution of the primary machine on the backup copies [5].

Chen et. al extended their initial work by developing a light-weight deterministic logging and checkpointing engine: ReVirt. In contrast to fault-tolerant solutions, where only the last checkpoint was needed to start replaying crashed execution, ReVirt was designed to support efficient navigation (time-traveling) between the chain of checkpoints in order to efficiently replay execution from any desired checkpoint.

Another important aspect dicussed by Chen is cooperative replay [4]. Cooperative logging is a technique where several communicating computers trust each other to rollback and replay execution together. This avoids logging messages sent by cooperated computers and relys on the fact that communicating

[*]Sections 9, 10 and 11 are the new additions since the proposal.

computers will resend the "same" messages during the replay. Chen's results were based on techniques used by message-logging recovery protocols surveyed and developed by Elnozahy et. al [8].

Chronous [21] uses time-traveling disk to locate critical changes in system configuration resulting in system misbehaviour. Chronous doesn't log complete system state, rather concentrates only on persistent changes committed to a disk. Xen Pervasive Debugger [11] currently supports only sequential debugging of the guest operating systemes running on a single Xen host, however authors promised to extend it with support of both time-traveling and multi-host debugging.

3 Introduction to logging and replay

3.1 Deterministic logging

Deterministic replay assumes the ability to recreate exact execution of the original run. Naturally that requires logging of all non-deterministic events (e.g. interrupts from external devices, memory mapped I/O, input from I/O ports etc.). Moreover, deterministic replay relies on the ordinary instruction assumption [5]. An instruction is ordinary if its result is determined by the initial system state and sequence of instructions that precede its execution on a processor. We hope that Intel CPUs provide ordinary instruction property, however some non-deterministic TLB placements or some other internal non-documented state of the CPU can potentially violate this property.

Note that Chen et. al used UMLinux [7] in their work, which is capable to provide full hardware emulation for the guest kernel. In our work we use Xen, which runs paravirtualized guests and uses frontend device drivers in guest domains. Frontend device drivers are no more than thin proxies communicating with the actual drivers run in device driver domains. Therefore, we will not allow to debug unmodified device drivers and paravirtualized parts of the guest kernel. We foresee the possibility of extending our approach by using processors with full virtualization support in hardware, however we do not discuss it in this proposal.

3.2 Non-deterministic logging

In order to reduce the overhead of deterministic logging we will investigate the possibility of relaxing log consistency and replay execution non-deterministically.

We will try to explore the assumption that some use model of our time-travelling architecture can tolerate certain degree of non-determinism. We understand that it's rarely the case when the goal of the investigation is debugging. However, it's highly probable that some statistical characteristics of distributed systems can be successfully explored without use of deterministic time-travelling.

As a solution lying in between the mentioned extremes, we suggest *deterministic surrounding*. In other words, if the user wants to debug a crash encountered during the non-deterministic logging, we will replay execution from an earlier point, log deterministically, and hope that the error appears in this execution. If so, user will be able to debug the error by replaying last part of execution deterministically.

3.3 Cooperative logging

Cooperative logging assumes that several nodes will cooperate to recreate previous execution together. In case of a networked environment, nodes rely on the fact that other participants will resend messages.

Cooperative replay seems to be especially attractive for closed or almost closed Emulab environment. Most of the log information will be recreated by participants.

A reasonable extension to the above approach lies in combining both cooperative and non-cooperative logging. Thus we will do cooperative logging between nodes which we trust and log everything coming from external (non-cooperative) world.

4 Deterministic Logging

4.1 Sources of non-determinism

To be able to deterministically recreate original execution we have to first define sources of non-determinism, log them and replay. We discuss below the typical processor events and classify them as deterministic, those that do not require any assistance to be recreated, and non-deterministic, those that have to be logged and replayed.

Moreover, it's also useful to further classify deterministic events as synchronous and asynchronous. Asynchronous events (e.g. disk reads) consist of two parts: request and response. Usually time of response is not deterministic and has to be logged like any other non-deterministic event.

4.1.1 Deterministic and non-deterministic events

Internal CPU exceptions (interrupts 0-31).

We attribute the 32 exceptions defined by the Intel architecture to deterministic events. Exceptions pass control to the hypervisor, and although it can deliver them asynchronously, guest will be frozen and observe exceptions as synchronous.

Nevertheless, there are processor exceptions that require at least some clarification about their determinism:

- Page faults

  Page fault exceptions can occur either due to the internal guest memory management or due to the external paging of the guest memory performed by hypervisor. In both cases page faults are deterministic. If guest swapped out own pages, it will do that deterministically again in subsequent replays. If on the other hand guest's address space was managed by the virtual machine monitor, hypervisor will handle page fault transparently without violating replay.

  Note, however that popular ballooning technique [20] tries to manage guest memory relying at the same time on the assistance from the guest. Therefore, ballooning has to be replayed during subsequent executions as any other external interaction with the guest. Obviously that makes little sense. Thus, we will prohibit ballooning of logged guests.

- Debug exceptions (exceptions 1 and 3)

  Debug exceptions can be used either by guest or by the hypervisor. Therefore, hypervisor always checks who placed breakpoint and if it was placed by the hypervisor it will not be delivered to the guest, and therefore will not disturb replay.

Exceptions from external devices

Exceptions from external devices are usually known as IRQs. They are the major source of non-determinism in our architecture. Therefore, we have to log and replay them carefully.

Some of the IRQs are handled directly in the Xen microkernel, some require assistance from the device driver domains. In a later section we will discuss how logging of these events is implemented.

- Timer IRQ

  Timer events are handled in the Xen microkernel.

- Console

  Console interrupts are delivered to the device driver domain (most of the time it's one privileged domain called *domain0*).

- Disk drives

  Interrupts from disk drives are handled in driver domains.

- Network cards

  Network interrupts are delivered to the driver domain.

- APIC IRQs

  Inter-processor interrupts (IPIs) are used in case of an SMP guest. They are handled directly in the hypervisor. We will not consider the SMP guest in our discussion.

Since we are targeting server environment in this work, we will concentrate on the devices enumerated above and omit discussion of other devices: graphic cards, CD-ROM drives, pointing devices, floppy drives, USB devices, etc.

Xen hypercalls

Xen hypercalls use interrupt xx, and are also deterministic. But hypercalls that see external state (for example, read time of the day hypercall) should be logged and should return the same value during replay.

4.1.2 Paravirtualized events

Note that paravirtualization allows us to avoid handling many complex events considered in other works, for example in [12].

In our work guests use only paravirtualized device drivers. In other words guest device drivers communicate only with the device driver domains by means of Xen IPC. Therefore, a paravirtualized non-privileged guest will never legally perform any of the following events: in/out instructions, memory-mapped I/O, device DMA.

4.2 Xen integration

In this section we discuss how logging mechanisms can be implemented in Xen virtual machine monitor. Xen [10] is a small microkernel providing simple exception handling, scheduling, memory management and IPC mechanisms. Device drivers, IP stack, file systems reside in guest virtual machines. Usually only one privileged virtual machine hosts all device drivers. Other virtual machines have only frontend device drivers, which communicate by means of Xen IPC with device driver domains in order to place requests to devices and receive responses. Requests from the guests are scheduled by the backend device driver to be sent further to the physical device driver.

4.2.1 Logging

In the previous section we noted that in order to reproduce exact execution we have to log and replay all non-deterministic events. By identifying these events we came to the conclusion that only IRQs from external devices are non-deterministic and have to be logged.

In the Xen architecture, non-deterministic hardware events are either processed directly by the Xen microkernel or delivered to the device driver domains. Backend drivers communicate with the frontend drivers to deliver information to the guests. Therefore, a paravirtualized guest never actually sees IRQs from disk or network devices. All events are delivered to the guest by means of two forms of Xen IPC: event channels and shared memory rings.

An attractive approach to logging implementation would be to log all communication at the level of event channels. Unfortunately, such an approach introduces unacceptable overhead. In order to deliver a small network packet Xen microkernel uses page sharing and therefore passes the whole memory page between the device driver domain and the guest. A blind logging of the whole page can easily deplete log space.

In order to avoid logging unnecessary information we need to be aware of the semantics of communication (i.e. what exactly was sent on the page). The perfect place to extract this information would be a thin layer below backend drivers in the device driver domain. At that level we can mediate the guest communication, and extract required information from the shared page by using the same structures, which are used by the backend driver.

At the same time in order to replay event deterministically, we not only need the data, but also the state of the guest at the time of event delivery. This state is accessible only from the Xen microkernel. Therefore we split our log operation in two stages:

- Log data: during this stage, logging operates in the device driver domain and logs the actual data which has to be delivered to the quest.
- Log guest state: during this stage logging operates in the Xen microkernel and saves the state of the guest at the time of event delivery. Note, that in order to pass data to the actual log we do an upcall from the Xen microkernel to the device driver domain.

Events which do not require interaction of the device driver domain are logged directly from the Xen microkernel. In order to log backend driver information we are planning to implement driver aware logging components. Currently they will log only console, network and disk I/O.

To remove disk writes from the critical path we separate operations of logging and flushing the log to persistent storage. We implement logging mechanism as a kernel thread which will accumulate log data and flush it to persistent storage periodically.

4.3 Deterministic replay

Deterministic replay of the event requires stopping the execution of the running guest exactly at the time, where event occurred during the original run and injecting the event to the guest. To stop the guest in the desired place of execution we use hardware branch counters provided by conventional Intel processors. Intel processor can count the number of branches encountered during execution and raise exception when the counter overflows. To employ this fact we log the number of branches between events during the original run. Upon replay we set the branch counter to overflow before the block of code where we have to inject the event. After that we place software breakpoint and proceed execution up to the desired instruction pointer position.

Note however that, while employing the above approach we have to be careful about the following complications:

- We have to carefully maintain branch counters upon VM switches and hypervisor exceptions.
- Some Intel models can deliver branch counter overflow exceptions within the average delay of 5 cycles. Therefore, we have to overflow the branch counter earlier and step through the code after that.
- Intel IA-32 architecture allows interrupt string copy operation. Therefore, we have to log `ecx` register value.

We have also carefully consider and investigate all possible violations of *Ordinary Instruction Assumption* [5]. Some examples deserving consideration are: possible non-deterministic TLB replacement policy, effects of the MSR and MRR registers on the CPU behaviour, effects of hyper-threading, or coprocessor interactions.

5 Non-Deterministic Logging

Most probably, a naive approach to a non-determinisic logging should not assume any logging at all except required by consistent distributed checkpoint protocol.

However, it's natural to extend the non-deterministic logging with delivery of external events. We believe that for many time-travelling experiments it will be sufficient to replay external events in approximately the same time, when they arrived during the original run. Note, that we assume that time of event delivery should be a virtual machine time.

In order to maintain some reasonably consistent state among distributed modes, we will rely on the physical clock synchronization, and resume execution on different nodes at approximately the same time.

Note, that we also suggest to use deterministic surrounding as was suggested in Section 3.2 and switch from the non-deterministic logging to the deterministic in case we need to debug program crashes.

6 Cooperative logging

Similarly to the non-cooperative logging, non-cooperating logging can be performed either deterministically or non-deterministically.

6.1 Cooperative logging in the deterministic case

In the deterministic case we rely on the fact that all participants during replay will deterministically resend all messages and cooperatively recreate distributed execution. However, even if a node receive messages from the original run, which are resend by cooperative parties, it still requires precise time of the message delivery in order to replay execution deterministically.

Combining the two ideas above, we introduce *pseudo* log entries. Pseudo entry stores only the state of the guest upon message delivery and the digest of the message needed to verify message integrity. The actual content of the message will be recreated upon message arrival. Note, that the guest can be frozen waiting for arrival of the message from a remote participant.

Note also, that we should consider the following reliability issue: generally we cannot assume reliabile message channels. Therefore messages can be lost. In that case we have to detect the loss, and undertake some steps to resolve the situation. Our current approach is to rollback the execution by one checkpoint and replay in the hope that the message will not be lost during the second run.

Note, that pseudo entries are used not only for network I/O but also for any other asynchronous events. For example, disk, which is usually considered to be totally deterministic, is only cooperative in our model. Thus, we have to add a pseudo entry to the log describing the state of the guest at the point when it received reply from the disk.

6.2 Cooperative logging in the non-deterministic case

In the non-deterministic case, cooperative logging can be reduced to the problem of guest synchronisation at the point of starting replay. In that case, we don't log anything except external events.

We propose to synchronize guests in a very simple fashion: synchronize the clock of the physical machines, agree on the time of replay allowing a large enough delay to let all guests to prepare, and start replaying at the scheduled time.

7 Checkpointing

Logging and replaying by themselves are capable of recreating the state of the guest at any point of execution history. Unfortunately, recreation of the state by only logging can be very inefficient [12]. Therefore, logging is most often combined with periodic checkpointing.

7.1 Distributed Checkpointing

Co-ordinated distributed checkpointing is the process of saving a consistent global state of the distributed system on-the-fly, with co-ordination from the individual nodes. A global state or checkpoint of a distributed system constitutes the states of the individual nodes and those of the communication channels. A consistent global state is one in which, if a node reflects a message receipt, then the state of the corresponding sender reflects sending of the message [6]. There has been vast prior research in the various techniques of taking distributed checkpoints [23, 13].

To meet the global consistency requirement, the co-ordinated checkpointing protocol should prevent nodes from receiving messages that could make the checkpoint inconsistent. Such messages constitute the channel state of the checkpoint. Depending on the mechanism used for identifying these messages [23], the checkpointing protocols can be broadly classified under two categories: control-message-based and time-based.

Control-message-based protocols rely on explicit control messages (markers) or piggybacked information to trigger checkpoint or identify messages received from senders in earliar checkpoint intervals. Some control-message-based protocols have restrictions on the nature of the communication channel. Some protocols assume a FIFO message delivery [6, 19, 18], while others assume causal message ordering [2, 1]. These protocols do not suit our design goals, since we try to avoid assumptions about message delivery guaranties. Further, we restrict ourselves to non-blocking checkpointing protocols, which do not block communications while the checkpointing protocol executes [8].

The control-message-based protocols incur a lot of communication overhead for checkpoint co-ordination. If we could assume a synchronized global clock, checkpoint co-ordination can be done efficiently [8]. All nodes can decide to take their local checkpoints at a fixed point in (global) time ('t'). All messages with timestamps smaller than 't' and received after a node after took its local checkpoint constitute the state of the channel. It would probably be tough to checkpoint all the guests before time 't' and also ensure that the checkpointed guests do not send any messages before 't' (so that they don't violate the constraint that messages before 't' belong to the older checkpoint interval), without blocking the guest operation. Thus, we don't plan to go along this path.

We propose to explore the applicability of the control-message-based protocols by Li et al [15] and Elnozahy [9], that use checkpoint indices piggybacked with messages and have no channel restrictions. When a node receives a message with a higher index than its local checkpoint index, guest checkpointing is triggered. Note that checkpointing will not be triggered on a node that is not communicating with other nodes. To ensure progress of the protocol, we might have to insert control messages with checkpoint indices. Since these should not reach the guest, our logging module would have to filter them off.

In these protocols, we need to tag a sequence number along with all the messages that are sent. Note that this tagging increases the packet size and could affect the packet fragmentation compared to the real run!

Also note that messages constituting the channel state of the checkpoint will not be resent if we replay execution from that checkpoint. So, we would have to log these messages along with their contents irrespective of the type of logging employed (Ofcourse, in the case of non-co-operative logging, incoming message contents are always stored).

7.2 Checkpointing a single guest

Checkpointing a guest involves saving the current content of the guest's memory and disk. In addition to this, we should log the contents of incoming messages from guests in previous checkpoint intervals. These messages in addition to the memory and disk contents constitute the checkpoint state on a guest.

Naturally, saving the entire memory or disk contents at each checkpoint is neither time nor space efficient. So, we try to store only minimal data required to identify state at a checkpoint.

7.2.1 Disk Checkpointing

Checkpointing a guest's disk entails the ability to regenerate a view of the entire disk as it were at the time of a checkpoint. There are two common approaches. The first approach is to do Copy-on-Write of blocks changed during a checkpoint interval and track mappings of guest disk blocks to their actual physical location for every checkpoint. The second approach is to track the changes in a redo/undo log, so that reconstructing the state at a checkpoint will involve traversing the undo (or redo) logs to move backward (or forward) in time. Doing time travel between checkpoints could be an expensive operation in the latter approach. Whereas, the first approach has the CoW runtime penalty.

Chen et al [12] use an optimized redo/undo log based approach, by avoiding copy of data blocks into redo/undo logs and instead creating a new block on first write after checkpoint and tracking the guest-host disk block mappings in the undo/redo logs. This approach still incurs the penalty of having to traverse the undo/redo logs to reconstruct the disk state during time travel.

We believe that the first approach, that is typically used by existing versioning storage systems [16, 14], would avoid this log traversal. And to avoid the CoW penalty, we will explore implementing (or adding to existing systems) the Create-on-First-Write functionality (that Chen et al use) in versioning systems.

Given that a Xen guest's disk can either be a disk partition or a file (in domain-0), we have the choice between disk block-level [14] and fileystem-level [16] versioning systems. It needs to be seen how the relative performance of the two approaches is. Although, in theory, file-level approach seems more of an overhead.

7.2.2 Memory Checkpointing

Support for efficient memory checkpointing is more complicated than disk. In general we plan to follow the approach adopted by Chen et al, where each checkpoint has redo and undo logs containing those pages modified in the previous and subsequent checkpoint intervals respectively. Similar to disk checkpointing, Chen et al's approach suffers from the log traversal overhead (especially when time travelling to distant checkpoints).

We propose storing the changed pages in versioning storage, similar to disk checkpointing. In addition to this we also propose to use undo/redo logs which would contain pointers to disk locations storing the pages (rather than complete page contents). We can use the redo/undo logs to identify the specific pages to be loaded to time travel to a checkpoint. This way we hope to get the best of both worlds- Time-travel to near checkpoints will use redo/undo logs followed by paging-in of specific pages. Time-travel to distant checkpoints, instead of traversing the logs, will load the checkpoint's memory snapshot into memory in totality. It is only our intuition that loading the entire memory would be more efficient than a long log traversal, but we would have to find out if it really holds true.

Chen et al's implementation uses CoW to save pages modified since last checkpoint. Their approach needs to copy the original page to store it in the undo log of the previous checkpoint. Since we use versioning storage, we can identify the contents of the page in the previous checkpoint and thus can avoid this memory copy overhead.

To identify changed pages, we propose to maintain a bit per page that gets set upon a write to the page. Setting write-protect bit on each page and having the first write access after the checkpoint trap into the hypervisor, is a straightforward, but inefficient, way.

7.2.3 Co-ordinating disk and memory checkpointing

Similar to checkpointing of multiple guests, disk and memory checkpoints should be co-ordinated. Suppose that there was a disk request in flight when memory and disk got checkpointed. This disk request should be saved and reissued to the disk upon replay.

We may need to apply distributed snapshotting techniques (in this case, the channel could be assumed to be FIFO) by considering disk and memory as the two entities in a distributed system. So, disk requests and responses too would be piggybacked with checkpoint indices.

7.2.4 Checkpoint deletion issues

To save disk space we propose to adopt exponential deletion, wherein we delete successive checkpoints for distant times in the past, such that the checkpoint intervals exponentially increase with increasing time in the past.

Further, we propose to maximize space saving by deleting the checkpoint with largest amount of changes among the consecutive checkpoints.

It is important to note that when log overflow happens and we overwrite a single log entry, we lose the ability to time travel to any checkpoint prior in time to that log entry. If such a thing happens, we should garbage collect those "inaccessible" checkpoints.

8 Debugging

8.1 Local debugging

Local debugging is hardened by the fact that in order to maintain determinism of the replay we have to operate externally with respect to the debugged operating system. We are not able to rely on any assistance from the guest. At the same time, we need some knowledge about guest operating system state to place breakpoints intelligently.

Some authors [12, 11] claim that they can easily debug the guest kernel and that the only complexity lies in distinguishing user-level processes of the guest. We claim that such an approach is in general incorrect and that external guest debugging is a nontrivial problem. Complexity of debugging stems from the fact that both guest kernel and user level processes possess enough inherent dynamism to prevent straightforward approaches to the analysing guest state needed to place debug breakpoints. Below we consider some related problems, which can help clarify the situation.

A guest kernel can load and unload modules, therefore any attempt to place breakpoint on the module requires, in general, analysis of guest kernel structures, which describe guest address space layout and modules present in memory. We can assume that it's possible to possess enough information to parse guest kernel structures to undertake this analysis. Unfortunately, the situation is more complex, since kernel structures required to undertake this analysis may have been partially swapped out. The next natural step is to try to parse guest swap file system in order to place breakpoint directly on the swapped page, or analyse whether the required module is actually in memory. Unfortunately, even this aggressive assumption doesn't solve the problem completely. The swapped page can be in the middle of transition between the swap file system and the memory and reside somewhere in a cumbersome disk I/O buffer cache layer.

Debugging of user level processes basically faces the same problems. Moreover, they are even more realistic. In kernel case, one can claim that the kernel rarely loads and unloads modules, or can require all kernel pages to be present in memory. Obviously, it's not true for user-level processes, which are created and swapped often, load shared libraries dynamically.

The most promising approach that we have figured out so far is to rely on guest assistance in placing breakpoints. Upon placing breakpoint we propose to checkpoint the guest and branch in a non-deterministic execution. While trying to place breakpoint, the guest operating system will find the virtual page for us wherever it is — in the memory, I/O cache, or swapped out. We in turn, track location of the page, rollback to the point of branching and place breakpoint in the same place where page was found.

8.2 Distributed debugging

Distributed time-travel debugging requires ability to step forward and backward in execution, jump between nodes, place forward and backward breakpoints.

Note that so far we have not logged the sending of a message. In order to improve navigating backward in time, especially in order to move backward along with the causal order induced by messages, we can extend the log with events describing the sending of messages.

8.3 Automatic crash detection

Time-travel debugging is especially useful for debugging long running applications. Therefore, it is desirable to provide support for an automatic crash detection. We foresee this support in two forms: passive and active.

In passive form, we propose to monitor system activity and if it changes unexpectedly, assume system misbehavior. For an active monitoring we propose to use a user-provided application, which will check some system invariant, and if it's no longer held report a crash.

8.4 Branches in the log

The natural use model of time-travelling assumes that user can modify logged execution and create branches. Therefore, there are two possible ways to evolve execution along the branch: proceed execution in the real world immediately after the branch or create the branch and continue replay of the log. Although the first approach seems to be much more natural, we feel that there is some rationality behind the second approach as well.

For example, during the non-deterministic replay, replaying the old log after the branch can help to explore some statistical characteristics of the system and their change with respect to the modification in execution.

Note, that there are also two ways to change execution at the point of branch: change data in the log or change the application code.

8.5 Version file system to support branches

From our survey, none of the current disk or filesystem level versioning solutions support branching, where one can go back to an older version and fork off from there. We plan to explore implementing branching on top of an existing solution like ext3cow [16] or handle it by managing the versions explicitly.

9 Implementation

We have implemented non-deterministic time-travel with consistent distributed checkpointing. Our implementation has made two simplifying assumptions. First, we assume that the time-travel system operates in a "closed world". This may seem restrictive at first glance, but is actually not so. Emulab, our target environment, is typically "closed", with the nodes in an experiment rarely interacting with the external world. The only external interaction for the time-travel system is with the Emulab "control plane" and our current implementation ignores this. Second, we assume that the communication channel is non-reliable and thus expect distributed applications to gracefully handle packet losses.

The implication of the first assumption is that our logging and checkpointing modules need not handle network packets originating from non-time-travelling nodes in any special way. The second assumption obviates the need for packet logging during consistent checkpointing. For instance, packets from the previous checkpoint epoch *i-1* received during the current checkpoint epoch *i* need not be logged and delivered during replay from checkpoint *i*.

Implementation Overview. Time-travel has been implemented as a loadable Linux kernel module in *domain0*. If the time-travel module is not loaded, Xen's default functionality is retained. We have added hooks in Xen's backend disk and network drivers for implementing logging and checkpointing. In accordance with our design decision to keep user domains transparent to time-travel, we have made no time-travel changes to *domainU* kernel or userland.

Time-Travel Kernel Module. Our time-travel module constitutes a checkpointing thread and a logging thread per time-travelling user domain on a physical machine. The logging thread periodically dumps the log buffer to the disk. The log buffer is filled in by the Xen backend disk and network drivers, with disk or network packets that are to be logged. As a result of implementing non-deterministic time-travel and the assumptions mentioned above, the logging functionality, although implemented, is currently unused.

The checkpointing thread's only job in life is to invoke memory and disk checkpointing when woken up. The checkpoint thread is woken up for three different reasons. First, from the backend network driver when it receives a packet tagged with a checkpoint epoch that is higher than that of the user domain's current checkpoint epoch. This achieves consistent checkpointing. Second, from the timer callback, for periodic checkpointing. Third, upon writing the user domain ID to the procFS entry */proc/tt-checkpoint*, for user applications to initiate checkpointing.

Ethernet Encapsulation. The above discussion on achieving consistent checkpointing requires that each outgoing network packet be tagged with a checkpoint epoch ID. We do so by ethernet level encapsulation. We introduce a new ethernet type and add a time-travel header that contains a checkpoint epoch ID (Figure 1). This header is added by the backend network driver at the sender. The backend network driver at the packet receiver strips this header and delivers a proper ethernet packet to the user domain. As a result the MTU visible to the user domain is reduced by the time-travel header size.

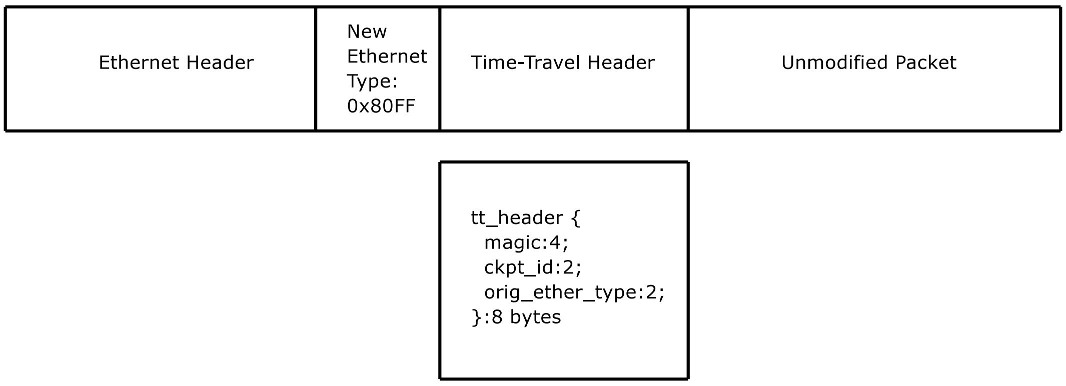

Figure 1: Ethernet Packet Encapsulation

Memory Checkpointing. Currently, we have a naive implementation of memory checkpointing that leverages on Xen's *save* and *resume* operations. *Save* halts the user domain and dumps its memory image to a file, while *resume* restarts a user domain from a file representing its memory image. While this was quite trivial to use, we were impeded by a bug (causing user domain disk I/O to hang upon a *resume*) that we eventually fixed.

Disk Checkpointing. We use the Copy-on-Write instant snapshot feature of Linux LVM for disk checkpointing. The user domain is initially booted off an LVM volume. Disk checkpointing involves simply taking a snapshot of the original LVM volume.

Replay. Time-travel to a previous checkpoint requires three operations. First, the time-travel module is to be disabled because we currently cannot take checkpoints during time-travel runs (which run off LVM snapshots). This is due to the limitation of Linux LVM that disallows taking snapshot of a snapshot. Second, the LVM snapshot corresponding to the checkpoint is chosen. Finally, a *resume* operation is done off the memory image corresponding to the chosen checkpoint.

10 Evaluation

In this section we present some evaluation results of our implementation on a simple setup consisting of two physical machines running a user domain each (with Linux Fedora Core 4) and connected by a 1 Gb network. We have verified that active *ssh* connections and in-progress *scp* operations between two user domains are uninterrupted across checkpointing and time-travel to previous checkpoints. We have also observed that checkpointing (including memory and disk) takes about 2-3 seconds.

Encapsulation Overhead. We compared the *scp* speed while copying a 1 GB file in three scenarios. First, between two user domains with ethernet encapsulation being done on all network packets. Second, between two user domains without ethernet encapsulation. Third, between two *domain0s*, that do not incur any virtualization overhead. The results are presented in Figure 2. As one would expect, the *scp* speed in the domain0 case is much higher. Interestingly, with time, the speed of the non-encapsulated user domain case becomes worse than the encapsulated user domain case. We do not have a specific reason for this, but suspect that it may be due to some transient network conditions. Nevertheless, these results indicate that encapsulation overhead is not appreciably high.

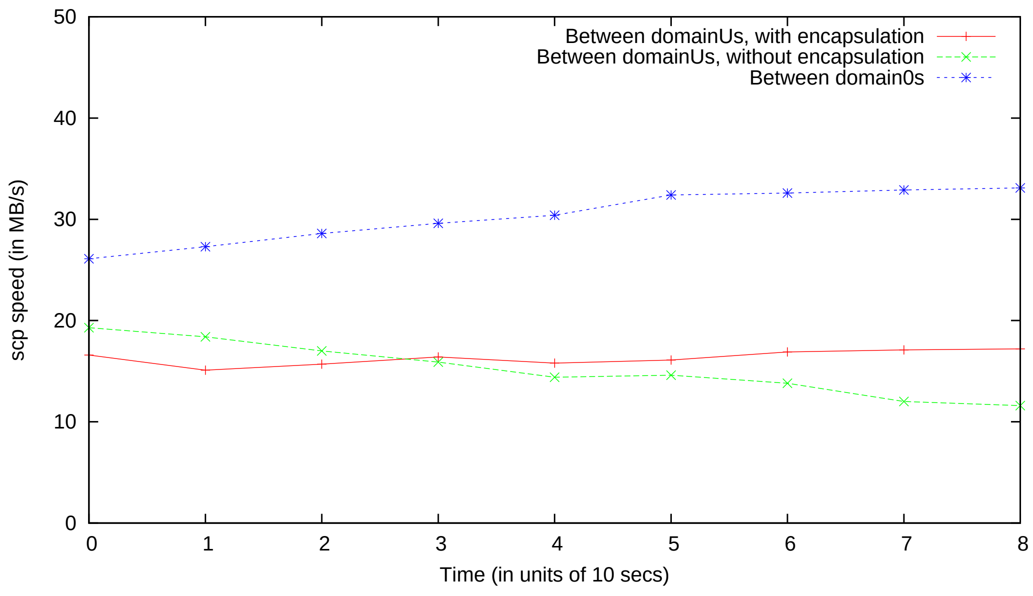

Figure 2: Encapsulation Overhead

Checkpointing and Replay Overheads. We compared the *scp* speed between the original and replay runs for over 8 checkpoint intervals (Figure 3). The results indicate that the speed of the original run decreases exponentially with increase in number of checkpoints. The reason for such poor performance lies in the way LVM handles its snapshots. Every new write to the original LVM volume pushes the previous contents of the volume to all the previous snapshots. This overhead, due to multiple-copy-on-writes, is the cause for performance degradation in the original run. Beyond the fourth checkpoint interval, the *scp* speed of the original run stabilizes. This is probably because earlier snapshots do not need copy-on-writes.

In contrast, the replay speed is almost constant because all writes during time-travel runs happens on (single-level) LVM snapshot volumes. In this case, the *scp* speed is dictated by the write performance

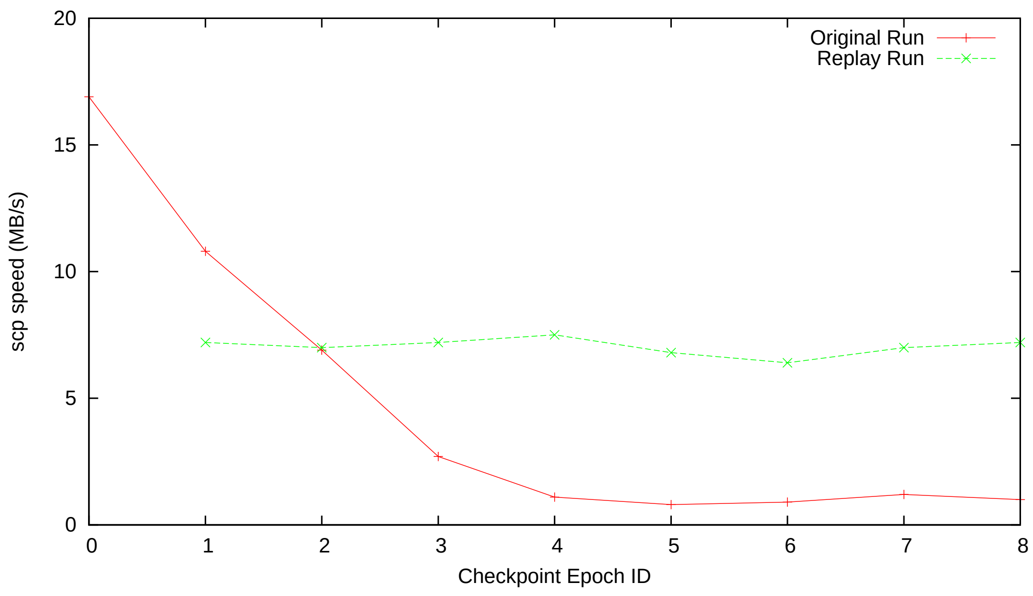

Figure 3: Checkpointing and Replay Overheads

on snapshot volumes, which seems to be less than half of that on original LVM volume without any snapshots, i.e. original run speed at checkpoint number 0.

11 Lessons Learnt and Future Work

Our implementation was mightily simplified compared to our proposed design. Nevertheless, we believe that it has provided useful insights and directions for future work.

Specifically, we have the following observations from our implementation effort:

- Distributed applications (atleast the ones we have evaluated) are highly resilient to packet losses. This gives us motivation to further pursue relaxed consistency in replay.
- Even with a naive implementation of memory checkpointing, the total checkpointing time is about 2-3 seconds. This gives us reason to believe that an optimized implementation could achieve checkpointing times almost imperceptible to user domain.
- LVM as a disk checkpoint solution is a bad idea. The write performance, both on original volumes with few snapshots and on snapshots, in general, is unacceptable.

As future work, we would like to pursue the following:

- **Implement deterministic checkpointing.**
- **Make memory checkpointing more efficient.**
- **Optimize disk checkpointing.** We propose to explore using ideas from the recent reasearch work on vitualization aware filesystem [17] that claims to handle versioning and branching efficiently.

References

[1] A. Acharya and B. R. Badrinath. Recording distributed snapshots based on causal order of message delivery. *Information Processing Letters*, 44(6):317–21, 28 December 1992.

[2] Alagar and Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. *IPL: Information Processing Letters*, 50, 1994.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[4] Murtaza Basrai and Peter M. Chen. Cooperative revirt: Adapting message logging for intrusion analysis. Research Report CSE-TR-504-04, University of Michigan, 2004.

[5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *SOSP*, pages 1–11, 1995.

[6] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, February 1985.

[7] Jeff Dike. A user-mode port of the Linux kernel. In USENIX, editor, *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, pages ??–??, pub-USENIX:adr, 2000. pub-USENIX.

[8] Elnozahy, Alvisi, Wang, and Johnson. A survey of rollback-recovery protocols in message-passing systems. *CSURV: Computing Surveys*, 34, 2002.

[9] E. N. Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[10] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, August 2004.

[11] Alex Ho, Steven Hand, and Timothy L. Harris. PDB: Pervasive debugging with xen. In *GRID*, pages 260–265, 2004.

[12] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*, April 2005.

[13] A D Kshemkalyani, M Raynal, and M Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4):224–233, 1995.

[14] A. J. Lewis. http://www.tldp.org/howto/lvm-howto.

[15] H. F. Li, Thiruvengadam Radhakrishnan, and K. Venkatesh. Global state detection in non-FIFO networks. In *ICDCS*, pages 364–370, 1987.

[16] Zachary N. J. Peterson and Randal C. Burns. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system, June 18 2003.

[17] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *3rd Symposium of Networked Systems Design and Implementation (NSDI)*, May 2006.

[18] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *ICDCS*, pages 382–388, 1986.

[19] S. Venkatesan. Message-optimal incremental snapshots. In *ICDCS*, pages 53–60, 1989.

[20] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[21] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, pages 77–90, 2004.

[22] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA, December 2002.

[23] Z. Yang and T. A. Marsland. Global snapshots for distributed debugging: An overview. In *Technical Report TR-92-03, Computer Science Department, University of Alberta.*