Virtual Machine Time Travel Using Continuous Data Protection and Checkpointing

Paula Ta-Shma Guy Laden Muli Ben-Yehuda Michael Factor
paula@il.ibm.com laden@il.ibm.com muli@il.ibm.com factor@il.ibm.com

IBM Haifa Research Lab

ABSTRACT

Virtual machine (VM) time travel enables reverting a virtual machine's state, both transient and persistent, to past points in time. This capability can be used to improve virtual machine availability, to enable forensics on past VM states, and to recover from operator errors. We present an approach to virtual machine time travel which combines Continuous Data Protection (CDP) storage support with live-migration-based virtual machine checkpointing. In particular, we present a novel approach for CDP which enables efficient reverts of the storage state to past points in time and makes it possible to *undo* a revert, and this is achieved using a simple branched-temporal data structure. We also present a design and implementation of a simple live-migration-based checkpointing mechanism in Xen.

1. INTRODUCTION

Many hypervisors support checkpointing the state of a running virtual machine [20, 7, 25]. A checkpoint typically captures the transient memory and system state of the virtual machine, writing this state to a file. The virtual machine (VM) can later be quickly restarted from the point in time (PiT) when the checkpoint was created by reloading its state from a file. If the original VM is left running when the checkpoint file is loaded and restarted, we have *cloned* the VM at a prior point in time [27, 15].

While some checkpoint implementations account for the state of internal persistent storage [26], this is atypical. More generally, checkpoints do not address the state of external (e.g., SAN or NAS attached) storage. A virtual machine can be resumed from a checkpoint only if the persistent state upon which it depends was not modified in a manner inconsistent with the VM's transient state after the checkpoint was created.

Continuous Data Protection [9] (CDP) is a relatively recent storage technology that enables fast revert of storage state to any prior point in time. CDP at the extreme keeps a history of the persistent disk state after every modification. More generally, CDP may have a given granularity, such as every second or every minute, for which it keeps a consistent copy of the persistent storage state. Taking frequent snapshots can be viewed as a degenerate form of CDP.

For some applications, reverting the persistent disk state may suffice to quickly reset the virtual machine to a prior state— performing a reboot of the VM after reverting the disk state is similar to a boot after an unexpected power outage. However, such a reboot loses the VM's transient state. For some applications the loss of the transient state is unacceptable. Application restart may be very expensive, entailing significant application recovery. A restart of the virtual machine may also lead to the loss of critical information, e.g., information required to diagnose a misbehaving virtual machine, that only exists in the virtual machine's transient memory.

By combining a storage technology such as CDP, for recording the persistent storage state at any prior point-in-time, with VM checkpointing technology, we can build a mechanism for inexpensive *virtual machine time travel*. In VM time travel, we quickly revert the *coordinated* transient and persistent states back to a prior point in time. To enable this time travel, we take VM checkpoints at points in time that are consistent with the CDP history, and we record the corresponding point in time in the CDP history together with each VM checkpoint. To revert a VM to a prior point in time, the VM's transient state is loaded from the checkpoint file, and the persistent state is reset to the corresponding time using the CDP history. The time to perform time travel is now roughly the time it takes to read the VM checkpoint from disk, and no reboot or recovery is needed. The cost of this approach is the overhead of periodically saving VM checkpoints to disk. The more often one checkpoints a VM's memory state, the more points in time are available to revert to.

Our main contribution is demonstrating a generic VM time travel service through coordinated CDP and VM checkpointing which respectively capture the persistent and transient states of a virtual machine. We have prototyped such a service on Linux and Xen [3]. Our implementation provides time travel to Xen VMs running any supported OS and does not require any changes to be made to the VM.

Our implementation consists of two main components: a storage subsystem with CDP support and a checkpointer component which non-disruptively saves VM memory state to disk. Periodically, mutually consistent versions of VM memory state and storage state are saved.

The rest of the paper is organized as follows. We describe several applications of VM time travel in section 2. Sections 3 and 4 describe our CDP storage and the checkpointer, respectively. Related work is presented in section 5, and con-

clusions and further work appear in section 6.

2. APPLICATIONS

Possible applications of VM time-travel include:

Improving availability Historically, a prime motivator for checkpointing (both with and without virtualization) has been to recover from failures in long running computations [20]. Traditionally such recovery has either used ad hoc methods to coordinate the persistent state or addressed applications without dependencies upon external persistent state. With our coordinated time travel, we have a generic means of recovering a VM's transient and persistent states after a failure. It is possible to quickly recover from serious corruptions of a VM, such as viruses or administration errors, using time travel. Fast time travel is especially important if determining the appropriate PiT to recover to requires performing multiple trial reverts, e.g., to determine when a corruption occurred.

Forensics After a failure of a VM, one can revert the VM to a point in time prior to the occurrence of the problem and then run forward, e.g., after attaching a debugger, to determine the cause. In addition, combining time travel with VM cloning enables performing root cause analysis while the production system is running.

System Administration Combining VM time travel with VM cloning enables fast testing of upgrades, patches and configuration modifications without affecting the production system.

Boot speedup As a faster alternative to a reboot in some scenarios, a checkpoint taken immediately after boot could be reverted to.

We note time travel is problematic when used with most transactional applications. Backing out of a committed transaction will cause problems for clients since it violates durability.

3. STORAGE SUPPORT

CDP capabilities can be provided by storage controllers, network appliances or by software running on the target host. We opt for a host-based solution that integrates with a hypervisor to provide CDP services to VMs. This integration point allows an OS-agnostic and storage-agnostic solution.

Based on the applications we foresee for virtual machine time travel, we designed our storage support with the following requirements in mind:

Protection granularity: Protection granularity controls the number of points in time available for reverting back to. The system should support coarse granularity protection, avoiding needless overheads in this case, while scaling well to fine granularity protection such as "once every second" or finer.

Robust revert support: The system should support a fast revert operation. It is important that the operation does not have a long latency before responding to I/O's on the reverted volume. A reverted volume is now the production volume and it should continue to provide full time-travel capabilities. Revert needs to be a reversible operation since it is important to allow mistakes to be backed out of.

Space management: Automated space reclamation is required. CDP implementations typically define an *accessibility window* which specifies the period of time to allow reverts to. This is usually on the order of days or weeks. Data outside the accessibility window is to be reclaimed.

Performance: The performance impact of having time-travel support turned on should be minimal.

Support for clones: The system should support creating clones which are writable copies of a volume at any previous PiT. This allows experimenting to find the best PiT before reverting the production volume, as well as forensics and system administration applications. For some applications it may be important to be able to create a large number of clones. Clone creation should be fast and thin provisioning of clones should be supported. Finally, clones should be first-class citizens, enabling both further revert and clone operations.

In this paper we focus on time travel support for the storage layer, although the mechanisms described here can be generalized in a straight forward way to support cloning.

The rest of this section is organized as follows. In section 3.1 we introduce the general architecture of our CDP system. In section 3.2 we describe how CDP metadata can be organized to support the above requirements. In section 3.3 we describe the software architecture of our prototype implementation and in section 3.4 we describe how the metadata is put to use in the context of the whole system.

3.1 The Checkpointing CDP Architecture

In a recent paper we described several broad storage architectures for time travel support [16]. We focus here on the Checkpointing CDP architecture. This architecture separates the current version of a volume (its *production device*) from the historical versions (contained in its *repository device*). Such a separation helps provide fast read access since there is no indirection involved in accessing the production device. Locality is also maintained in the production device, ensuring sequential read performance is not affected. The price to be paid is that data needs to be written or copied to both the production device and the repository device. A block is copied to the repository just before it is about to be overwritten in the production device by an incoming write I/O (i.e., copy-on-write, COW).

When the user wishes to revert the system to a previous PiT, the production device must be updated to contain the correct data. We perform these copies in the background and do not block I/O's during this process: writes go to their destination in the production device (possibly causing COW to be performed) and reads are serviced directly from the repository if necessary. To complete the illusion of quick

revert, the system supports performing another revert while the background copy from a previous one is still in progress. This results in a system where reverts appear instantaneous to the user, although performance is degraded somewhat while the background copy is in progress.

3.2 Repository Metadata

In this section we show how to use metadata to organize the data in the repository device in order to support time travel of a volume. A straight forward extension also supports cloning of volumes, but we omit the details. A main contribution of our work is a metadata framework which supports *fast* and *reversible* revert, where the revert operation returns the state of a volume to that of potentially any previous point in time.

In addition to the Checkpointing architecture, our repository metadata organization is also applicable to the other CDP architectures described in our earlier work [16]. We show how our metadata framework is integrated into the Checkpointing architecture in section 3.4.

3.2.1 Metadata Indexes

In order to support fast and reversible revert, our metadata associates a logical *timestamp*, a physical address and a *branch* with every block written to the repository device. The timestamp is a counter associated with each volume that is increased in accordance with the protection granularity: less often for coarse protection and more often for fine granularity protection—potentially as fine as "every write". The physical address is the location of the block in the repository device. Each revert operation introduces a new branch, which inherits from a parent branch at the time being reverted to. Each volume is associated with a single *current branch* which is the branch incoming writes for that volume are associated with. We give a concrete example in Section 3.2.2.

Timestamps, physical addresses and branches for blocks in the repository are stored in two metadata indexes, which can be implemented using standard B-Trees [4].

1. The **branch table** represents the branch hierarchy. They key is <branch-id> and the data is <start-time, end-time, parent-id, revert-to-time>.
2. the **lpmap** (logical to physical map) maps from key <logical address, timestamp> to the physical location in the repository <physical address>.

These indexes are used to service read, write and revert requests to a volume, where read/write requests generate lookup and insert operations on the indexes.

A final metadata structure is used to manage the space allocation within the repository device. This structure need not be time-travel aware and our current implementation consists of a B-Tree indexing the free extents in the repository. We point out that it is not critical to layout space in the repository to give optimal sequential read performance, because we only read from the repository immediately after a revert. The emphasis in our context is on speeding up

```
lookup(l)
 b = current branch, t = current time
  while (b != NULL) {
   find the latest lpmap entry with prefix <l>
     whose timestamp is between start-time(b) and t
   if one exists
     return associated physical address
   else
     t = revert-to-time(b)
     b = parent(b)
 }
 return NULL
```

Figure 1: Pseudo-code for lookup(l) where l is a logical address: Look for the last write on the path from the current branch to its ancestors.

free space allocations and writes to the repository, somewhat similar to LFS [18].

We implement the following metadata API:

insert (logical address, timestamp, physical address): Inserts the corresponding entry to the lpmap.

lookup (logical address): This operation indicates the location of the last write to the given logical address. Since branches inherit writes from their ancestors, this requires a recursive algorithm whose pseudo-code is described in Figure 1. It returns <physical-address>.

revert (revert-to-time): Create a new branch which starts at the current time and whose parent is the ancestor of the current branch defined at revert-to-time. Mark the end-time of the current branch, and set the newly created branch to be the current branch.

Note that branches do not temporally overlap—there is exactly one branch that contains any given timestamp. This means the parent-id column of the branch table can be deduced from the revert-to-time column—the parent is the branch whose start-time and end-time contain the revert-to-time. We omit the parent-id column from the example in Section 3.2.2.

We show how the above API is used in the Checkpointing CDP storage architecture in section 3.4.

We expect the typical branch table size to be orders of magnitude smaller than the corresponding lpmap, and to be able to easily cache it in memory even when massive branching (thousands of branches) is present. We should mention that as long as the history of writes to a logical address is short (e.g., can be stored within a single B-Tree disk page), then there is an efficient implementation which avoids multiple B-Tree accesses to implement lookup at that address. The use of timestamps in addition to branches allows us to scale well to "every write" protection granularity. The alternative of introducing a new branch for every write would unnecessarily explode the size of the branch table.

3.2.2 Example

Figure 2 depicts the branch table (right), and the subset of the lpmap for logical address 12 (left). It shows the evolution from time 0 to 120, where revert operations took place at time 80 (reverting to time 35) and at time 100 (reverting to time 70). To lookup address 12 at time 105 we first look for the latest lpmap entry in the interval [100,105). Since there is none, we then look for the latest lpmap entry in the interval [35,80), which is at time 60 (physical address d). Note that the entry for time 60 was not accessible after the first revert to time 35, although a second revert allowed us to change the target revert-to-time. Since there is no bound on the allowed number of reverts, this allows a search for the best time to revert to, for example, the latest time before the onset of a virus, which is typically not known in advance.

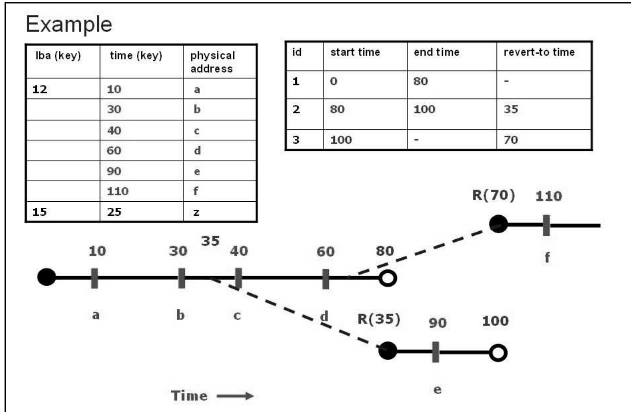

Figure 2: Metadata indexes for a particular sequence of inserts and reverts: lpmap (left) and branch table (right).

3.2.3 Space reclamation

CDP systems enable specifying an *accessibility window* which bounds the window of time they wish the system to provide reverts to. Given this bound, our system performs automated space reclamation of unneeded metadata and user data in the repository.

Space reclamation is done by scanning the lpmap structure, one lba at a time. For each lba, a mark and sweep type scan of all its entries is performed and any blocks that cannot possibly be accessed are reclaimed. Obviously any block with a timestamp within the accessibility window can not be reclaimed. Blocks with older timestamps can only be reclaimed if they are not *visible* within the CDP window, taking into consideration writes performed at later timestamps and the branching structure. Garbage collection of branch table entries must also be performed.

3.3 System Architecture

We implement a hypervisor based block CDP layer on Linux and Xen which provides time-travel support to Xen guest VMs. The prototype is in C and uses the Berkeley DB library [24] for the persistent metadata B-Trees.

Under Xen, virtual machine I/O's are served from a privileged domain (called dom0) running Linux. Xen provides an extensible block I/O interception framework called *blktap* (block tap) [28] which forwards VM block I/O requests to a user-space process running in dom0. For performance, I/O requests are aggregated and typically arrive in large batches. Integrating our storage system with blktap allowed us to provide guest VMs with CDP-enabled volumes transparently.

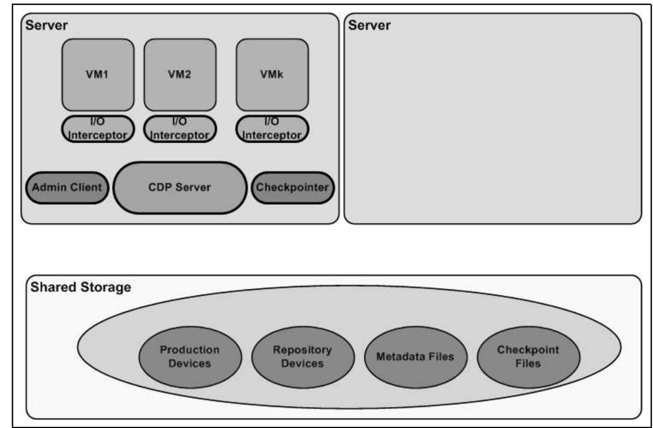

Figure 3: System architecture and storage organization.

In order to make most of our code *hypervisor-agnostic* we split our storage subsystem in two parts—a hypervisor-specific I/O *interceptor* and a separate *CDP server* process. All metadata management, background tasks, management operations, I/O's to the repository, etc., are handled by the CDP server process, while the interceptor process performs the data I/O's to the production device on behalf of the VM. Our Xen interceptor runs within the blktap user-space process and we run one such process per storage volume. There is one CDP Server process per physical machine.

The interceptor communicates with the CDP server using IPC. The IPC traffic has relatively low overhead as it consists of the I/O *requests* and not of their contents. It provides a way for the CDP server to redirect requests or to prepare the production or repository devices prior to access by the interceptor. The communication occurs once per batch of I/O requests. Batches consisting entirely of reads avoid this overhead entirely if there is no active revert background process on the volume.

The CDP server supports a simple management protocol. A command-line client enables administrators to provision CDP-enabled volumes and to revert them to previous points in time.

Our storage subsystem is *event-based*, meaning the volume logical timestamp is increased under control of an external program via the management protocol. The timestamp is returned to the caller which can use it to later revert the storage to that particular PiT. For fine granularity protection it is easy to have the storage system periodically increment its counters automatically.

Figure 3 shows the basic software components as well as the storage layout. The figure depicts multiple VMs run-

ning on a physical machine, whose I/O's are intercepted via the I/O Interceptor component which communicates with the CDP server. Administrative commands such as event marking and revert are handled via an administration client. The memory checkpointer can be invoked at the desired frequency, and is discussed in detail in Section 4. Each CDP enabled storage volume has an associated production device, repository device, and metadata files (Berkeley DB indexes). Each VM has associated checkpoint files. If live migration between physical machines needs to be supported, then all data should reside on shared storage.

No caching of VM I/O data traffic is performed by our storage subsystem. Write I/O's reach the interceptor once the virtual machine's operating system's buffer-cache has flushed them, at which point the OS expects disk-like semantics. Reads have already had the benefit of the VM buffer-cache and there is likely no point second guessing it. CDP Server metadata on the other hand *can* be cached, and it is cached by Berkeley DB in our implementation.

A *block size* defines the alignment of actual I/O's performed to both the production and history volumes. VM I/O requests of any size are supported, however non block-aligned requests may require an extra I/O. The larger the block size, the less repository metadata is needed. Typically setting a block size which is identical to the VM's buffer-cache page size (4KB) is a good idea. For VM's running databases, the best block size would be the database page size.

3.4 Use of Metadata in the Checkpointing CDP Architecture

We give a brief outline of the actions taken by the CDP server for each of the read, write and revert operations.

Performing a revert involves invoking the metadata **r**evert operation which will create a new branch and associate it with the volume. A revert operation starts a background task in the CDP server, transparent to the guest VM, which copies data from the repository device to the production device. We say a volume has an *active* revert if this copy process is in progress.

If no revert is active (this is the common case) reads typically do not involve metadata lookup. If a revert is active then the data to be read may reside in the repository device. This can happen for example if the background process has not copied it yet. In this case the CDP server performs a metadata **lookup** to locate the required data and then copies it from the repository device to the production device. Once this is done, the interceptor can read the data from the production device, as though no revert had been active.

Writes whose previous versions need to be copied to the repository (COW) involve a space allocation followed by a metadata **insert** operation. Note that this occurs only for the first write to a particular logical address after an event was marked. Whether this is the case can be determined by maintaining an in memory bitmap or performing a metadata lookup operation.

Space reclamation continuously runs as a background task in the CDP server. We note that our space reclamation only deals with metadata, unlike Rosenblum and Ousterhout's Log-Structured Filesystem cleaner processes [18].

4. CHECKPOINTER

Xen did not have integral support for virtual machine checkpointing at the time of this work (although such support was merged into the xen-unstable tree recently). We therefore implemented a "good enough" checkpointer component which makes use of available Xen features and does not require any changes to Xen itself. Xen supports two operations related to checkpointing: *save* and *restore*, and *VM live migration* [11, 22, 6]. Saving a VM causes it to become suspended; restoring it causes it to resume execution from the point at which it was suspended. Unfortunately, checkpointing in Xen cannot be trivially implemented as a combination of *save* followed by *restore*, since *save* is disruptive to the domain being saved. As part of the saving process Xen suspends the domain, severing all open network connections.

Xen's support for live migration [6], on the other hand, enables a running VM to be migrated to a different physical machine with minimal disruption (i.e., network connections are not severed). We therefore implemented checkpointing by migrating a VM to the same hypervisor. During the migration, a copy of the migration bit stream is written to a file. Our checkpointer intercepts the migration traffic between the source and target machines, masquerading as the target machine to the original source machine and as the source machine to the original target machine.

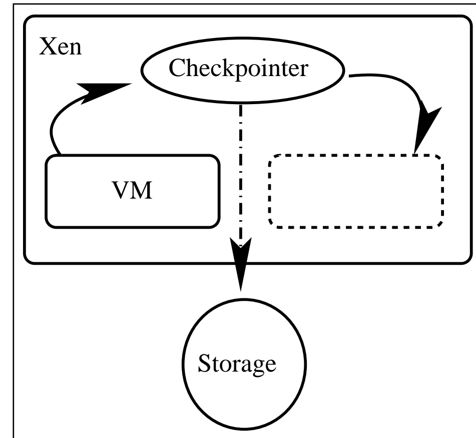

Figure 4: Checkpointing via localhost live migration.

When the checkpointer is started, it creates a listening socket on a well known port and waits for incoming migration requests. To create a VM checkpoint, a Xen live migration of the VM is directed to that port on the same physical machine where the VM is executing. The checkpointer performs the migration protocol with the initiator, and initiates a migration back to Xen on the same physical machine. As the migration data begins to flow through, the checkpointer intercepts it, writing the VM's transient state to disk as well as forwarding it back to Xen. A unique feature of *live* migration is that the VM continues running while most of its state is transferred. Its execution is only paused for the bare

minimum of time needed to get a consistent view of the last few dirty frames of memory and the CPU registers state.

It is during the small window of time when the VM is paused that the checkpointer causes the memory checkpointed to be synchronized with the disk state, by marking a CDP event on the VM's storage devices. To revert a VM to a previous PiT, given the checkpoint file and associated CDP events, one would first revert each of the VM's storage devices to the appropriate event using the CDP administration client, and then run the standard Xen `xm restore` command with the checkpoint file as a parameter. The VM would then resume from the requested point in time.

Our approach to checkpointing, based on Xen's live migration support, is fast enough, transparent and unobtrusive. Other approaches are certainly possible and more efficient approaches may be worthwhile for very frequent checkpoints or for specific workloads. Our checkpointer was implemented initially in a few hundred lines of Python for quick prototyping and then in (a few hundred more) lines of C for the production version. It should be noted that since this work was done, Xen has gained integrated checkpointing support in the xen-unstable tree, via the "xm save –checkpoint" command [7, 8]. We are in the process of evaluating its suitability for our purposes.

5. RELATED WORK

Point in time copies (snapshots) of storage [2] are a common feature, supported by storage controllers, appliances, logical volume managers and filesystems. In the context of time-travel, snapshot support often has limitations such as a relatively small number of allowed snapshots on a volume, or overheads relating to snapshots which preclude very frequent snapshotting. Another issue using these systems for time-travel is that revert of a volume is often a slow synchronous operation during which I/O's can not be performed. It is also common that reverting to a snapshot destroys intermediate snapshots, making reverts non reversible.

Versioned filesystems [23, 21, 17] keep track of updates to files and enable access to historical versions. The unit of protection in these filesystems is a file—the user may access a historical version of a specific file—while in block-based CDP the unit of protection is an entire LUN. However since typical files and directories are much smaller than LUN's and have far fewer updates than a LUN, the design of the metadata structures for managing file and directory history is usually not as scalable. Typically only read-only access to old versions is offered and no revert operation is supported.

TRAP-Array [32] is a system which supports fine-granularity CDP. The focus of the system is on reducing the space consumption of the CDP history data. For fine-granularity protection they achieve good compression of the version history of each block in the repository by XOR'ing consecutive versions of the block. One cost of such a system is that it becomes necessary to retrieve the entire history of a block in order to access any particular version. The system does not support performing revert in the background while accepting new I/O's.

An example of putting time-travel to work is given by Brown and Patterson [5]. Time-travel is used there for supporting recovery from operator errors in an email-store. They provide time-traveling storage based on NetApp filer snapshot support. They had to overcome a limitation of this support which they call 'no forward time travel'. This means that once a volume is reverted to a snapshot, all the snapshots between the current time and the snapshot are lost. This loss of history at revert makes it impossible to 'undo an undo', a capability they think important enough that they developed a workaround. The proposed workaround is slow and cumbersome—they resort to implementing revert by copying all files from the old snapshot to the current volume using the filers support for constant-time copying of a file from an old snapshot. Another limitation they mention is that only 31 snapshots were supported. This is overcome by using a log of all incoming email traffic (which they have to maintain for other reasons) to roll the system forward from the closest available snapshot. They measure these workarounds to be about two orders of magnitude slower than native disk-level time travel support for their application. It is clear that better time travel support in the disk layer would make a big difference for their intended application.

A different example of an application enabled by VM time travel functionality is presented by Whitaker et al. [30]. A system is described which performs VM time travel as part of an automated search for the point in time when a VM transitioned into a failed state, for example, due to operator error. In their system only disk state time travel is supported and VMs need to be rebooted from the reverted disk state. Their time-travel disk (TTDISK) works by logging all writes and maintaining an lpmap like data structure. However a TTDISK has no notion of branches and read-write access to a previous PiT requires a separate COW disk with its own meta data to be mounted over the TTDISK. It is not clear how time travel of this COW disk can be supported.

VM time travel is put to use in the context of facilitating OS debugging by King et al. [14]. Their system enables fine grained logging of VM actions to enable exact replay of VM execution from past points in time. This involves logging all non-deterministic events that can affect VM execution. To reduce log replay overhead, periodic checkpoints of memory and disk state are taken and the log is replayed relative to a checkpoint. The work was done in the context of the UML virtualizer [10]. Their memory checkpointer implements the optimization of only saving pages that were changed since the previous checkpoint. The storage support for time travel is based on a no-overwrite scheme. An in-memory structure maps logical addresses to physical locations. This structure reflects the current version only. To support time travel a log of changes to this structure is maintained separately. Performing time travel involves synchronously performing log replay/undo of these metadata changes.

A system for Xen VM fault tolerance is presented by Vallée et al. [26]. The system is based on checkpoint/restart of VM disk and memory states. Similar to our checkpointing approach, their memory checkpointing is based on existing Xen features (save/restore and migration). For support in saving disk state they make use of unionFS [31] running *in the guest VM* and seem to have to shut down the guest at every disk-state snapshot. In addition taking a disk-state

snapshot in their system is an expensive operation involving a significant amount of data copying.

Our approach to structuring repository metadata is related to work done on temporal database access methods [19]. Most temporal access methods however do not support branching which is important for enabling fast, reversible reverts and clones. One exception is Jiang et al.'s BT-tree [13] which is a *branched-temporal* structure. The idea of a separate small branch-table was inspired by their work. For the following reasons, in our context we can afford to use a simpler data-structure which can be mapped onto a standard B-Tree. First, our lpmap structure does not support fast queries along the time dimension. This enables us to avoid duplicating entries—a certain key (lba,branch,timestamp) appears in our lpmap at most once. The BT-tree uses duplication to balance the tree by both time and key dimensions which increases the size of the metadata and can complicate space-reclamation. Another simplification is that since CDP windows tend to be relatively small and given support for space-reclamation, we felt it reasonable to have lookup time at a particular address proportional to the size of the lpmap for that address.

The WAFL [12] and ZFS [1] filesystems can support a large number of snapshots. By frequently creating snapshots, CDP-like functionality could be supported. These filesystem adopt a no-overwrite policy for both data and metadata. The filesystem (both user data and metadata) is logically organized as a large tree. Creating a snapshot creates a new tree root which shares all children with existing nodes. The first write to any logical block will now cause an entire tree-path of metadata to change. This should be contrasted with an update-in-place metadata structure such as the lpmap where a B-Tree insert operation is performed. In the context of fine granularity protection the update-in-place approach to metadata has smaller overheads.

Parallax [29] is a proposed storage subsystem for clusters of Xen virtual machines. Being able to perform VM memory checkpointing coupled with disk snapshots every 30 seconds is mentioned as a design goal. A radix tree is proposed for the mapping between from logical block address to physical address and like in WAFL designs, branching is to be supported by creating a new (overlapping) tree for each branch.

6. CONCLUSIONS AND FURTHER WORK

We have described a system which supports VM time travel by taking VM checkpoints which are coordinated with CDP at the storage layer. Our system supports fast, reversible revert which speeds the return to VM availability after a VM failure. Storage cloning is also needed to complement time travel for some applications such as system administration and forensics, and clones also have applications in areas such as VM image management. Our infrastructure can be extended to clones in a straight forward way, and we would like to fully incorporate clones into our system as first class citizens. We also plan to perform a detailed performance analysis of our work at the storage layer, and are interested in adding support for the *SplitStream* CDP architecture [16] and comparing the performance of the various CDP architectures defined in our recent paper [16] empirically.

Acknowledgments

We thank Dan Smith and Andrew Ball from IBM for their insights regarding the checkpointing solution. We thank the past and current members of the IBM/HRL CDP team: Uri Braun, Eitan Yaffe, Doron Chen, Yaron Orenstein, Ari Yakir and Ealan Henis. We thank Alain Azagury for first suggesting the coupling of VM checkpointing and CDP to us. Finally, we thank our managers Dalit Naor, Yaron Wolfsthal and Kalman Meth for their support.

7. REFERENCES

[1] ZFS: The last word in file systems. http://www.sun.com/2004-0914/feature/.

[2] A. Azagury, M. E. Factor, J. Satran, and W. Micka. Point-in-Time Copy: Yesterday, Today and Tomorrow. In *Proceedings of the 10th NASA Goddard and 19th IEEE Symposium Conference on Mass Storage Systems and Technologies (MSST'02)*, pages 259–270, April 2002.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP 03)*, pages 164–177, 2003.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[5] A. Brown and D. A. Patterson. Undo for Operators: Building an undoable E-mail store. In *Proceedings of USENIX Annual Technical Conference, San Antonio, TX*, June 2003.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA*, May 2005.

[7] B. Cully. Virtual machine checkpointing. Xen Summit 2007. http://www.xensource.com/files/xensummit_4/talk_Cully.pdf.

[8] B. Cully and A. Warfield. Secondsite: disaster protection for the common server. In *Proceedings of the 2nd conference on Hot Topics in System Dependability (HOTDEP'06)*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[9] J. Damoulakis. Continuous protection. *Storage, June 2004*, 3(4):33–39, 2004.

[10] J. Dike. A user-mode port of the linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, 2000.

[11] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004.

[12] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter'94 USENIX Technical Conference*, pages 235–246, 1994.

[13] L. Jiang, B. Salzberg, D. B. Lomet, and M. Barrena. The BT-tree: A Branched and Temporal Access Method. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000), September 10-14, 2000, Cairo, Egypt*, pages 451–460.

Morgan Kaufmann, 2000.

[14] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of USENIX Annual Technical Conference*, April 2005.

[15] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. VMPlants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society.

[16] G. Laden, P. Ta-Shma, E. Yaffe, M. Factor, and S. Fienblit. Architectures for controller based CDP. In *Proceedings 5th USENIX Conference on File and Storage Technologies (FAST '07)*, Feb 2007.

[17] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.

[18] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured filesystem. *ACM Transactions on Computer Systems*, pages 26–52, 1992.

[19] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[20] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Workshop 18*, page 300.2, Washington, DC, USA, 2005. IEEE Computer Society.

[21] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Otir. Deciding when to forget in the Elephant file system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP 99)*, 1999.

[22] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[23] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–34, 1985.

[24] M. I. Seltzer. Berkeley DB: A Retrospective. *IEEE Data Eng. Bull.*, 30(3):21–28, 2007.

[25] E. Skoglund, C. Ceelen, and J. Liedtke. Transparent orthogonal checkpointing through user-level pagers. In *Proceedings of the 9th International Workshop on Persistent Object Systems*, pages 201–215, Lillehammer, Norway, Sept. 6–8 2000.

[26] G. Vallée, T. Naughton, H. Ong, and S. L. Scott. Checkpoint/restart of virtual machines based on xen. In *High Availability and Performance Computing Workshop (HAPCW'06)*, Santa Fe, New Mexico, USA, Oct. 2006. Held in conjunction with LACSI 2006.

[27] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 39(5):148–162, 2005.

[28] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, pages 379–382, 2005.

[29] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *USENIX Hot Topics in Operating Systems (HOTOS)*, 2005.

[30] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI 2004), San Francisco, CA*, December 2004.

[31] C. P. Wright, J. Dave, P.Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and unix semantics in a fan-out unification file system. Technical Report FSL-04-01b, Computer Science Department, StonyBrook University, October 2004.

[32] Q. Yang, W. Xiao, and J. Ren. TRAP-Array: A disk array architecture providing timely recovery to any point-in-time. In *Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA'06)*, pages 289–301, Washington, DC, USA, 2006. IEEE Computer Society.