Time travel in the virtualized past: cheap fares and first class seats

Liuba Shrira
Brandeis University
liuba@cs.brandeis.edu

Catharine van Ingen
Microsoft
vaningen@microsoft.com

Ross Shaull
Brandeis University
rshaull@cs.brandeis.edu

1 Abstract

"Time travel" in the storage system is accessing past storage system states. Legacy application programs could run transparently over the past states if the past states were virtualized in a form that makes them look like the current state. There are many levels in the storage system at which past state virtualization could occur. How do we choose? We think that past state virtualization should occur at a high storage system buffer manager level, such as database buffer manager. Everything above this level can run legacy programs. The system below can manage the mechanisms needed to implement the virtualization. This approach can be applied to any kind of storage system, ranging from traditional databases and file systems to the new generation of specialized storage managers such as Bigtable. Granted that time travel is a desirable feature, this position paper considers the design axis for virtualizing past states for time travel, and asks what amounts to the question, can we sit in first class and still have cheap fares?

2 Virtualized past states

Cheap disk means a storage system can retain past application states and keep them around for a long time. Back-in-time execution (BITE) is the ability of the storage system to run read-only application programs on snapshots of the past storage system states in addition to the current state. BITE provides what is popularly called time travel. Traditionally snapshots have been used to run queries over past states (snapshot isolation [1] queries) to avoid interfering with rapidly evolving current state, and for rapid light weight recovery by copy-paste or query, particulary to cope with inadvertent user errors. BITE in addition allows to audit and mine persistent past states in real-time, performing after-the-fact analysis of past states, possibly using methods that were unavailable at the time. Such abilities are increasingly demanded by modern applications.

We want to be able to provide BITE in storage systems which support legacy applications. Therefore we want to virtualize the snapshots of the past states in a form that looks like the current state so that (unmodified) programs that run on the current state can run on snapshots. This approach is in contrast, to temporal databases that introduce an explicit time coordinate, virtualizing the current state to look like the past state.

Figure 1 depicts how an unmodified application program issueing get and put operations to the storage system,

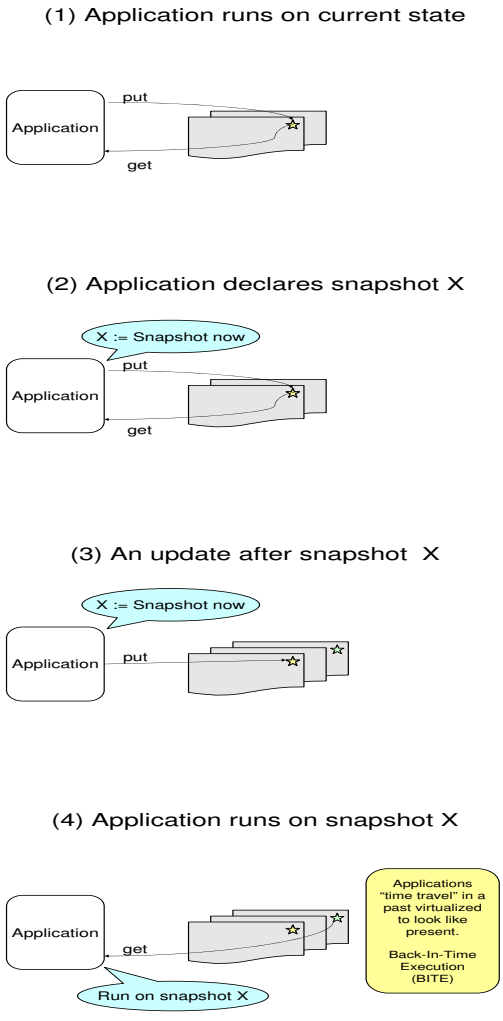

Fig. 1: Snapshot interface

interacts with snapshots. First, in (1) the get and put operations access the current state, (2) snapshot X is declared, (3) snapshot system retains the state of an object, before a put operation modifies the object first time following snapshot X declaration, (4) application now runs on snapshot X, get operation observes the object state retained for snapshot X.

Legacy applications expect to see a consistent state of a storage system. To this end, our virtualized past states will provide *crash consistency*. Crash consistency corresponds to the state the application might see if the storage system

would have crashed and recovered. This consistency condition, of course, depends on the consistency guarantees of the original storage system without snapshot virtualization.

At any point in the system execution, an application may *declare a snapshot*, establishing a specific *named* storage system state that can be revisited subsequently in a BITE. We think it is important for BITE to support application controlled snapshots rather than periodic snapshots, because only applications are in the position to recognize which states are important. In contrast, backup systems use periodic snapshots, as do systems that run queries on snapshots (snapshot isolation [1] queries) to avoid interference with updates. Some applications may require frequent snapshots. Support for frequent snapshots means that taking a snapshot should be "non-disruptive", that is, should not unduly delay other on-going activities in the storage system. Moreover, snapshots should be *long-lived*, meaning that they should be available for BITE as long as needed.

3 Design choices

There are many levels in the storage system at which past state virtualization could occur. We contend that BITE is best supported by integrating the snapshot system with a storage system buffer manager. The buffer cache is well-suited to redirecting BITE code to access past block states, and legacy code above this level can run unmodified. This approach can be applied to any kind of storage system, from databases to file systems.

The buffer manager exposes the current storage system state in the form of groups of blocks or database pages; a buffer manager that virtualizes past states for BITE does the same, but can also expose past states at snapshot points.

When a snapshot is declared, the states included in a snapshot are not only states on disk but also the states of the dirty pages in the buffer at the declaration point, possibly backed by a recovery log. The virtualization design must therefore deal with organizing versions of blocks in the cache and on disk, choosing when to write those versioned blocks to disk, and providing a way to find them again when BITE is later requested.

A storage stack contains many buffer managers; for example, there may be write caches in a disk controller, file system, and database in a single system, layered one over the other. Integrating snapshots at the level nearest the applications requesting snapshots provides the best performance. Snapshot crash consistency condition assumed by the application can be enforced with less overhead because capturing snapshot states does not require immediate (synchronous) flushing to lower-levels. The virtualization may be able to keep or efficiently reconstruct past states in the buffer cache, avoiding reading past states back from disk as we explain below.

Here are the questions one could ask when evaluating an implementation of past state virtualization. How fast will the BITE applications run? What impact will past state virtualization have on normal application performance? The relative importance of BITE and normal operation performance will affect the design choices.

Caching Snapshot Blocks An update to a block following a snapshot declaration could overwrite a past state that belongs to a snapshot. Writing the snapshot block to disk before overwriting it preserves the needed snapshot state but could interfere with normal application performance when snapshots are frequent. Virtualizaton at the buffer manager level can manage a "versioned" buffer cache that holds multiple versions of the same block. This enables defering snapshot block writes and reduce the impact to normal applications at the expense of the extra buffer cache memory to hold the versioned blocks. The cached versioned blocks could be lost in a crash. In the buffer cache a snapshot system can take advantage of the storage system recovery system to recover the snapshot blocks needed to insure snapshot crash consistency.

In a database buffer cache, snapshot crash consistency can be guaranteed simply by writing snapshot declaration points into the database recovery log and maintaining a write ordering invariant that guarantees that past block states needed for a snapshot are written to disk and become durable before they are overwritten in the current database state on disk by a later update. This snapshot write ordering invariant dovetails the write-ahead-log invariant, and is called *write-ahead-snapshot invariant* (WAS, for short). WAS insures that the database recovery log containing the needed snapshot states is not garbage collected before the needed snapshot blocks have been written to the snapshot disk. This way, if there is a crash, any unwritten past states of a block that belong to a crash consistent snapshot can be (re)captured in the buffer cache during the recovery of the unwritten current state, in the same way they are captured during normal operation. This assumes, of course, that the recovery log is replayed "as is", without pre-processing.

Figure 2 depicts a sequence of steps showing how a snapshot virtualization layer, integrated within a buffer cache, manages a versioned cache and enforces the WAS invariant. The snapshot system depicted in the figure stores the snapshot blocks separately from the current state blocks, on a different disk. We discuss the pros and cons of co-locating current state blocks and snapshot blocks at the end of this section. We assume, the sequence of steps in the figure is immediately preceeded by a snapshot X declaration. At this point in the execution, (1) the current state of the cached pages P, Q belongs to snapshot X, (2) snapshot versions of P, Q are being retained in the cache when pages P, Q are updated, (3) the snapshot versions of P, Q are being written to the snapshot disk (background writes), enforcing WAS invariant, that is, ahead of writing in the step (4) the latest versions of P, Q to the database disk (back-

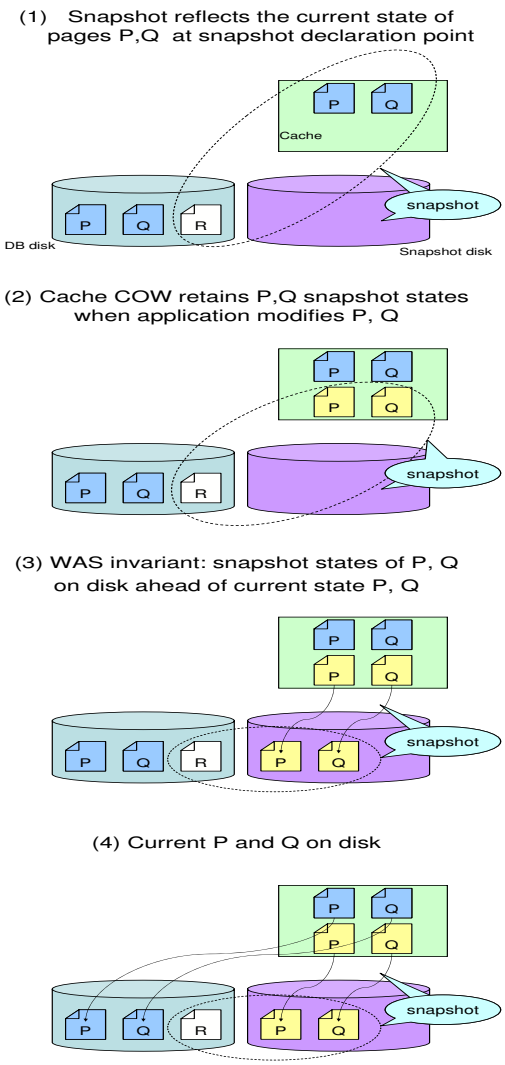

Fig. 2: Versioned cache and WAS invariant

ground writes).

Same approach can be applied in a file system, exploiting file system recovery mechanism for low-cost capture of consistent snapshots. Not all applications manage their own page cache; but, many utilize the file system interface. The highest-level buffer manager relative s to such applications is in the file system, so we describe the technique for virtualizing snapshots inside the file system cache. Like adding snapshots to a database buffer cache, we make the file system cache *versioned*. Unlike a database, which provides durability guarantees for all committed data, a file system generally only promises metadata consistency; so, if multiple versions of a block are in the cache, then snapshot data blocks may be lost after a crash. We must ensure that no snapshot is inconsistent (that is, unreadable by file system tools) and that, while data may be lost from a snapshot, newer data created after a snapshot declaration never becomes part of that snapshot.

A file system that keeps a metadata journal must serialize snapshot declarations into the journal. This way, if there is a crash, past versions of the meta-data blocks needed for snapshots can be re-captured during meta-data recovery. In a file system that maintains meta-data consistency using a soft update technique [5], that is, by carefuly ordering defered writes to disk, the careful ordering must be extended to encompass snapshot declarations and snapshot block writes. The same holds for the generalization of the soft update technique that aims to defer any storage system disk updates, without violating crash consistency, by tracking general dependencies on deferred disk updates and making sure not to expose system effects dependent on deferred updates before the updates become durable [6].

In these dependency tracking systems, maintaining the WAS invariant, that is, making sure that a page P that belongs to a snapshot becomes durable on the snapshot disk, ahead of it becoming unrecoverable from the storage system disk version, allows the snapshot system to capture the needed snapshot states during storage system recovery in a way that preserves snapshot consistency guarantees. Note that a past state of page P in the database becomes unrecoverable as a snapshot state, when a storage system overwrites the page needed to recover the snapshot state and garbage collects the recovery log.

Without WAS, it might be possible for BITE to not find any past versions of a block, and so use the current-state block, which could be newer than the snapshot. By writing snapshot states first, we ensure that, snapshots consist of blocks that would be valid post-recovery if a crash occurred right after the snapshot was declared.

The write ordering requirement can be relaxed if the versioned write cache is located in non-volatile storage, since past states become durable once they are scheduled for write and will not have to be recovered after a crash.

Copying Snapshot Blocks to Disk Snapshot state can physically reside on disk together with the current state, or apart from the current state on a different disk, possibly in a different host. Snapshot state can also have different physical representations. It can be represented as a complete materialized replica of the past storage system state, or a log of changes, or a combination of both.

A common snapshot implementation approach is copy-on-write. There can be two types of copy-on-write. The type of copy-on-write determines whether the past is stored apart or together with the current state. A copy-on-write that copies the older data into a snapshot log before the new data is written, results in snapshots that are *split* from the current state. Examples of this approach are the volume snapshots in VSS [9], DBMS snapshots in SNAP [13], file system snapshots [17, 16], and controller CDP snapshots in [3], to mention a few. A copy-on-write that uses a *no-overwrite* approach, copying the new data elswhere and leaving the older data in place, results in snapshots that re-

side together with the current state. The commercial product NetApp filer [2] employs the WAFL file system to take no-overwrite snapshots, and virtualizes past states to allow mounting of snapshots for time-travelling access by legacy applications. Examples from the research community include CVFS [15] and ext3cow [7].

In the split approach, every application write may incur up to two overhead i/o operations. A read operation may be needed to retrieve the old data and a write operation may be needed to write the old data to the snapshot log. The extra disk i/o may impact normal application performance. The extra write cost can be mitigated if the snapshots are stored on a different disk, the writes to the snapshot log may occur in parallel with updating the current state, partially "hiding" the impact of snapshots. A buffer cache that can reference multiple block versions, can also reduce or eliminate the need to read old data from disk by keeping old versions in the cache (possibly in a compressed form) until they are written to the snapshot log.

The no-overwrite snapshot approach does not require extra i/o to store the past disk block states. However, it raises a different performance isssue. The past states residing with the current state decluster the current state on disk causing performance degradation. Regaining the performance requires periodic re-clustering in the current state, similar to cleaning (garbage collection) in log-structured file systems. Studies show [10] that re-clustering can be disruptive, unless done during downtime.

Compared to retaining a full copy (mirroring) of the current state for a snapshot, the copy-on-write approach reduces the total disk space consumed for a collection of snapshots. The storage optimization, however, comes at a cost as it reduces the performance of BITE. With *really* cheap disk, mirroring (full copy) might at first blush seem to be more attractive. Because the reads can be striped, the load at the disk is actually less than without mirroring. The wrong thinking is that the cost of the disk isn't just the purchase price, but also the power consumption and rack or machine box space; full copies remain expensive.

Indexing snapshot Blocks The snapshot system has to find the blocks that belong to a given snapshot to run BITE. In the no-overwrite snapshot approach, the logical to physical block map for the current state that has existed at the snapshot declaration point can be used by BITE to find the blocks as the past states remain in place [2]. In contrast, in the split snapshot approach, the snapshot block location is determined over time as blocks that belong to a given snapshot move to the snapshot log from the current state, so the block location has to be tracked dynamically [13, 7, 17].

4 VSS and SNAP

A snapshot system that virtualizes past states for BITE has many similarities with a snapshot system for backup, but there are important differences. We briefly consider how the difference in requirements between BITE and backup translate into different design choices in two systems: a commercial high-performance backup snapshot service VSS and an experimental snapshot system SNAP. Both VSS and SNAP provide snapshots in existing storage systems that update data in place and therefore use the split snapshot approach.

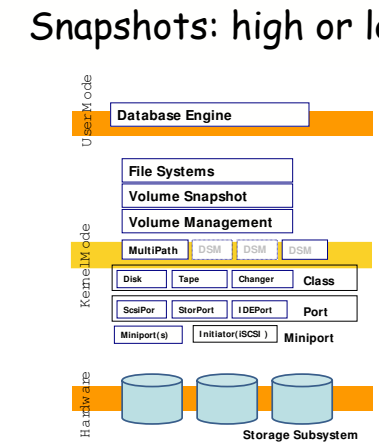

Fig. 3: Storage stack layers

4.1 VSS

VSS (Virtual Shadow Copy Service) is a commercial snapshot implementation in the Microsoft Windows operating system. The VSS service coordinates the actions of a *requestor*, one or more *providers*, one or more *writers*, and the NTFS file system. The *requestor* is an application requesting creation of a snapshot. A *provider* actually creates and maintains the snapshot either by copy on write or mirroring. Providers may be implemented as software (the operating system includes a copy-on-write volume filter driver) or external storage hardware. A *writer* is an application such as SQL Server or operating system component such as the registry that wishes to participate in snapshot creation. VSS is implemented below the NTFS buffer manager, that is, below the highest buffer manager for these applications. In the storage stack depicted in In Figure 3 VSS corresponds to the Volume Snapshot layer.

The VSS coordination ensures that the snapshots are consistent. Critical application, operating system, and file system metadata are flushed to disk prior to the split of the snapshot by the provider. The coordination is optimized to minimize the realtime nature of that flush, minimizing the time that all writes are frozen by the operating system.

4.2 SNAP

SNAP [13, 14] is an experimental split snapshot system for a transactional storage system Thor [4]. SNAP supports real-time ad-hoc BITE-based analysis of long-lived high-frequency snapshots. The system retains consistent snapshots, virtualizing past states at the highest buffer manager

level, residing in the top layer of the storage stack in Figure 3. SNAP manages a versioned buffer cache so that snapshot pages can be captured without the need to quiesce the database, avoiding the disruption of the database in the short-run. The cache stores the versioned modifications in a compact form (modified objects) reducing the memory overhead for versioned blocks to below 10%. The snapshot system incurs a 4% performance penalty in the storage system on standard workloads [13].

SNAP needs an efficient way to find the blocks that belong to a given snapshot that performs well even when snapshots are frequent and long-lived (multi-year). Some split snapshot systems [8] use the recovery log as the snapshot store and "roll back" to a consistent snapshot. Such solution is acceptable in a short-lived snapshot system but would be inefficient in a long-lived system. SNAP, instead, uses a new approach. It indexes the snapshot blocks at low-cost by writing the *mappings* of the snapshot blocks into a sequential log as it copies snapshot blocks to the snapshot store.

Scanning block mappings to find a block is faster than scanning a recovery log but a mapping scan can be still slow if some database blocks are modified infrequently, since the scan has to pass over many repeated mappings of the frequently modified blocks before finding mappings for infrequent ones. Yet, both infrequently and frequently modifed data is common. The code running on a snapshot in real-time has to wait for the infrequently modified blocks. SNAP uses a new indexing method called Skippy [12] that hierarchically builds condensed persistent summaries of the mapping log, with duplicate entries removed, and links the summaries into the mapping log at regular intervals. Slow scans can now proceed faster over these lower-resolution summaries. Skippy brings down the delays for code running over long-lived snapshots in real-time to become comparable to short-lived snapshots. It is efficient both in theory and in practice, showing a significant (19 fold) performance improvement in the SNAP system [11].

BITE snapshots differ from backup snapshots in their long-term storage costs. Storing long-lived high-frequency snapshots can become costly because, while disk is cheap, power, space, and administration costs for high volume of snapshots over long duration can be high. Moreover, some snapshots are less important that others to applications. SNAP allows applications to specify the relative importance of snapshots and exploits the copying of past states in split snapshots to organize on-the-fly the snapshot storage according to importance. This organization enables low-cost storage management functions useful in long-lived systems, such as eventual selective garbage collection of snapshots, that have minimal impact on the normal storage system performance [14].

5 Conclusion

Virtualizing the past states of a storage system as they are viewed by the *high-level* buffer manager (such as a database buffer manager) has several benefits. Snapshot consistency can be maintained by caching snapshot blocks in memory, exploiting the recovery mechanisms in the buffer manager that insure storage systems crash consistency. BITE is possible because the buffer manager can virtualize snapshots to provide past state access identical to current state access. These benefits come at the cost of implementation effort for different high-level buffer managers, and less generality. A short-lived snapshot system such as VSS, intended for backup, implemented below the file system buffer manager, with the goal of serving many applications, is more general.

Long-lived, low-impact snapshots with high-performance BITE are not a free ticket. However, the benefit of time traveling with legacy (and new!) programs in the distant and near past over interesting, application-determined snapshots suggests that we can provide a first class flight for a coach price even in systems that generate large numbers of long-lived snapshots - if we virtualize past state inside the high-level buffer manager.

References

[1] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. In *Proceedings of SIGMOD '95* (San Jose, CA, 1995).

[2] HITZ, D., LAU, J., AND MALCOM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference* (San Francisco, CA, January 1994).

[3] LADEN, G., TA-SHMA, P., YAFFE, E., AND FACTOR, M. Architectures for controller based cdp. In *Proceedings of FAST '07* (San Jose, CA, Februrary 2007).

[4] LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. Providing persistent objects in distributed systems. *Lecture Notes in Computer Science 1628* (1999).

[5] MCKUSICK, M. K., AND GANGER, G. R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. pp. 1–17.

[6] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync (awarded best paper!). In *OSDI* (2006), pp. 1–14.

[7] PETERSON, Z. N. J., AND BURNS, R. C. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system, 2003.

[8] ROMERO, A., AND ASHDOWN, L. *Oracle Database Backup and Recovery Basics*, 10g release 2 (10.2) ed. Oracle Corporation, Redwood Shores, CA, 2005,

ch. Oracle Flashback Technology: Alternatives to Point-in-Time Recovery.

[9] SANKARAN, A., GUINN, K., AND NGUYEN, D. Volume shadow copy service. *Power Solutions* (March 2004).

[10] SELTZER, M. I., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. N. File system logging versus clustering: A performance comparison. In *USENIX Winter* (1995), pp. 249–264.

[11] SHAULL, R., SHRIRA, L., AND XU, H. Skippy technical report. http://www.cs.brandeis.edu/ rshaull/skippy/skippy-tr.pdf, 2007.

[12] SHAULL, R., SHRIRA, L., AND XU, H. Skippy: a new indexing method for copy-on-write snapshots. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering (ICDE'08), Poster* (2008).

[13] SHRIRA, L., AND XU, H. Snap: Efficient snapshots for back-in-time execution. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)* (Washington, DC, USA, 2005), IEEE Computer Society.

[14] SHRIRA, L., AND XU, H. Thresher: An efficient storage manager for copy-on-write snapshots. In *USENIX '06: Proceedings* (Berkeley, CA, USA, 2006), Advanced Computer Systems Association.

[15] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), USENIX Association.

[16] VERITAS. Flashbackup 3.4 system administrator's guide unix. *VERITAS NetBackup 3.4 Documentation* (2000). http://docs-pdf.sun.com/875-3244-10/875-3244-10.pdf.

[17] WIRES, J., AND FEELEY, M. J. Secure file system versioning at the block level. In *Proceedings of EuroSys '07* (Lisbon, Portugal, March 2007).