Scalable Archival Data and Metadata Management in Object-based File Systems

Sage A. Weil
sage@cs.ucsc.edu

CMPS 290s Project, Spring 2004
University of California, Santa Cruz

Abstract

Online archival capabilities like snapshots or checkpoints are fast becoming an essential component of robust storage systems. Emerging large distributed file systems are also shifting to object-based storage architectures that decouple metadata from file I/O operations. As the size of such systems scale to petabytes of storage, it is critically important that file system features continue to operate efficiently. We present a flexible mechanism for archiving file system state that allows the creation of *checkpoints* for arbitrarily sized subtrees of the hierarchy. Checkpoints are managed in a distributed fashion while maintaining efficient utilization of system resources.

1 Introduction

One of the most significant advances in file system usability in the last decade has been the popularization of online data archiving. Although early research systems like Plan 9 explored mechanisms for preserving past state within the standard file system interface, it wasn't until WAFL [3] was introduced that the concept of file system "snapshots" became popular.[1] Archival systems have become increasingly common in research (Elephant [7], Venti [6]) and commercial systems, and archiving facilities are rapidly becoming an indispensable feature of robust storage systems.

Another paradigm shift is taking place in the area of distributed storage as object-based storage architectures have become increasingly attractive building blocks for large file systems. Object-based storage allows low-level block allocation to be encapsulated by semi-intelligent network-attached disks, effectively distributing allocation management across hundreds of thousands of devices. Object-based storage devices (OSDs) facilitate a model in which metadata transactions are decoupled from file read and write operations, allowing greater system scalability.

Research into and experience with object-based systems is still relatively limited, and few existing systems have integrated archival features. In particular, the issue of scalability in such systems is particularly important as storage systems grow in size and traditional forms of backup become increasingly impractical.

In this paper we present a strategy for integrating online archiving of file system state into a large-scale object-based distributed file system. Our architecture is based on the concept of file system *checkpoints* or consistency points, which can be created at any time to preserve the current state of an arbitrary subtree of the file system. In particular, we investigate a system in which checkpoints can be later discarded, in contrast to some systems that seek to preserve archival data for all time.

2 Background

We examine the design and integration of checkpoints into a file system architecture being developed by the Storage Systems Research Group at UC Santa Cruz. Our system is based on object-based storage devices, decoupled metadata management, and is designed to scale to petabytes in size.

2.1 System Architecture

In this a system a client will consult a metadata server (MDS) cluster, responsible for maintaining the file system namespace, to receive permission to open a file and information specifying the location of its content. Subsequent reading or writing takes place independent of the MDS cluster by communicating directly with one or more Object-based Storage Devices (OSDs), which intelligently manage their own on-disk storage and enforce security policies. Although the size of metadata is relatively

[1] Is this accurate? I don't think I'm not old enough.

small compared to the overall size of the system, metadata operations may consist of 50% to 70% of all file system operations [5], making the performance of the MDS cluster and the impact of archival subsystem integration of critical importance.

2.2 Data Layout

File data is stored in a large array of OSD devices—potentially tens of thousands in a petabyte-scale system. Data within each file is striped across a large number of objects for good read/write performance, and data is replicated within the system for safety.

2.2.1 Distribution

A motivating feature of object-based storage is the ability to ignore allocation details. To further escape the task of mapping files to large sets of objects, we utilize the Replication Under Scalable Hashing (RUSH) algorithm [4] to distribute file data across the OSD array. Given a single small input (seed), RUSH generates a sequence of pseudo-random disk identifiers across which file data may be distributed while providing mechanisms for efficient (optimal) data re-balancing when disks are added or removed. The algorithm is designed so that this mapping can be generated by client accessing the system without interaction with the MDS cluster or any central allocation authority, while still providing probabilistic guarantees of a balanced data distribution.

2.2.2 Replication

For safety, data in the system is also replicated on multiple OSDs to protect against disk failure. The FARM architecture [11] uses RUSH to first map file data into collections, augments them with either replicas or erasure-codes for safety, and then distributes collections to disks for storage (see Figure 1). This combination of FARM and RUSH distributes data in a probabilistically balanced fashion while additionally guaranteeing that replicas (or erasure codes) for any given item will always be stored on different disks.

2.3 Metadata Management

File system metadata in this system is dynamically partitioned across a small cluster (tens) of metadata servers (MDSs) by assigning partial subtrees of the directory hierarchy to different nodes. Subtree partitioning results in efficient MDS cache utilization and prefetching capability while allowing workload to be dynamically redistributed to adapt to current client demands. A critical aspect of the dynamic metadata management architecture is that file

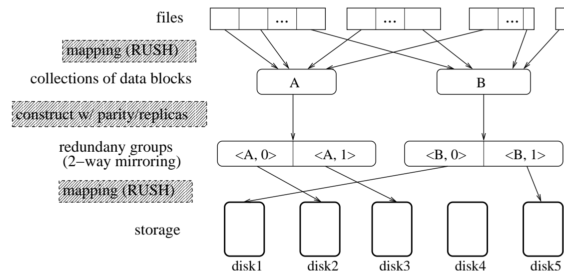

Figure 1: File blocks are distributed to collections, replicated, and then distributed to disk such that a balanced distribution is maintained and no two replicas for any item are stored on the same OSD. (Figure lifted from [11].)

metadata (inodes) are embedded with directory entries, allowing metadata to be stored in chunks consisting of entire directories. This general strategy (originally proposed with the C-FFS file system [2]) streamlines typical access patterns and facilitates prefetching and other mechanisms for exploiting workload locality.

2.3.1 Storage Framework

Metadata updates in the file system must be quickly flushed to disk for safety, while metadata reads due to cache misses should be as efficient as possible to maximize performance. To facilitate short-term high bandwidth write demands and efficient data layout for later read performance, MDSs utilize a two-tiered storage framework. Each MDS maintains a sequential log for committing updates to stable storage. When items fall off the end of the log or are retired from the MDSs in-memory cache, they are committed to long-term storage, where metadata is organized into directories and written to an object store in a manner similar to other file content.

2.3.2 Anchors

One consequence of embedding inodes in directories is that a global inode table or index is no longer necessary. This avoids difficulties associated with managing a large, distributed and sparsely populated index in a consistent and efficient fashion. However, the lack of globally addressable inodes also requires that inodes referenced by multiple hard links (file names) be handled differently. A distributed hierarchy makes this process non-trivial, as changes in the hierarchy may affect path names of entire subtrees of items.

Instead of indexing all inodes (that vast majority or which have only a single reference), we propose the creation of global table to locate *anchors*, a generalized locater that can be attached to (file or directory) inodes of particular interest in the file system. The table consists of records of the form <inode, parent_inode,

<inode, parent, ref>	Filename	Effective Location
<1,-,1>	(root)	/
<12,1,2>	foo	/foo
<223,12,1>	bar	/foo/bar
<532,12,1>	baz	/foo/baz

<inode, parent, ref>	Filename	Effective Location
<1,-,1>	(root)	
<52,1,1>	**quux**	**/quux**
<12,52,2>	**foo**	**/quux/foo**
<223,12,1>	bar	/quux/foo/bar
<532,12,1>	baz	/quux/foo/baz

Figure 2: Anchor table state before and after `mv /foo /quux/foo`. Only bolded rows are updated, even though (in general) an unbounded number of anchors are potentially shifted in the hierarchy. The triple on the left reflects table data, possibly including the file name in the center column, while the right column is a comment indicating effective path names (which not stored in the table). Anchored items can be located by inode number by recursively identifying parent directories.

`ref_count>`, which are inserted for each anchored inode with an initial ref_count of 1. A record is also inserted for every parent_inode referenced, with a ref_count indicating its reference count. Thus, given an inode number, a series of recursive lookups can be performed until a known inode is identified (in the case of an empty cache, the root inode), at which point the necessary directories can be opened to locate the record in question. Rename or move operations can preserve the accuracy of the table by making the necessary changes whenever the inode being relocated exists in the table. This will require at most n + m operations, where n is the current and m is the new depth of the item in the directory hierarchy. Depending on the in-memory directory representations, it may be necessary to include a filename in anchor records to facilitate inode location within a given directory. Figure 2 contains a sample anchor table before and after moving directory `/foo` to `/quux/foo`, demonstrating that only two updates are necessary even though multiple anchors are shifted in the hierarchy.

Anchors facilitate hard links by allowing a multiply-referenced inode to be anchored and stored in the directory it is most often referenced from. Less used directory entries simply reference a unique inode number and use the anchor table to locate the anchored inode. Anchors can be used to allow the efficient location any metadata record of interest (not just multiply-linked inodes) in the file system with a minimally sized global table and small overhead associated with hierarchy changes. In contrast, C-FFS indexed the all directories, even though most entries were unnecessary.

2.4 OSD Architecture

The ANSI T10 Technical Committee is currently working on a draft specification for an OSD command set and interface. Although it is a work in progress, it is highly suggestive of the kinds of interfaces that will be used by hardware devices when manufactured OSDs finally hit the market. As such, it is a useful point of reference and consideration for designing new file systems, both in terms of making file system implementation practical and influencing the direction of the standard to meet file system requirements. We specifically consider what elements of the standard are particularly useful, and what additions or modifications would further facilitate efficient archival features.

3 Related Work

Although *snapshots* or *checkpoints* were originally introduced as a mechanism for managing databases, snapshots were popularized in file systems by the WAFL file system [3]. At any point in time a snapshot of a volume can be created with minimal cost to preserve the current state of the file system indefinitely. Subsequent modifications utilize copy-on-write techniques to preserved snapshotted data, and allocated storage is not freed until snapshots referencing data are removed. The implementation of snapshots in WAFL is particularly efficient as it follows naturally out of underlying file system design. Most modern commercial storage systems include snapshot features as well, with varying degrees of elegance in their implementations.

Snapshot techniques have also been implemented at the block level by volume managers such as the Logical Volume Manager (LVM). After a snapshot is taken, LVM copies any block that is about to be modified to a specially reserved region of disk and maintains an appropriate index such that the effective snapshotted state of the entire block device can be preserved. This allows snapshots to be taken in a file system independent fashion, so long as metadata and data are stored on the same logical block device and the snapshot is taken synchronously (with all data flushed to the block device). The Venti archival system [6] similarly facilitates the preservation of past file system state at the block level by preserving all stored blocks for all time. Because such systems are based on block devices and block-level allocation, their specific approaches to archival consistency is only indirectly relevant to an object-based system.

The S4 and CVFS self-securing storage systems [10, 9] provide versioning at the OSD level by preserving all file data written to the device for some period of time. This comprehensive versioning approach allows OSD (and thus file system) state to be determined at any point within a given time period (the intrusion detection window in the context of system security). The Lustre distributed storage system [1, 8] similarly delegates copy-on-write semantics to the object storage layer and leverages that facility for

preserving snapshotted data. In contrast to S4 and CVFS, which archive at a very fine grain, Lustre creates snapshots only when explicitly instructed to do so, and always for the entire file system.[2] Unlike both systems, we are interested in facilitating archival features at a variable level of granularity by allowing consistency points to be created for any subtree of the file system hierarchy.

4 Archival Framework

The motivating capability examined in this paper is the ability to create *consistency points* or *checkpoints* (CPs) for objects in the file system and cause them to be preserved in the face of future modifications until later discarded. Checkpoints should be able to be created for individual files or (more generally) arbitrary subtrees of the directory hierarchy. In contrast, snapshot features present in existing file systems are often restricted entire volumes.

There are a few performance and efficiently requirements that follow from this basic capability:

- Checkpoint creation should be fast, such that current online and future performance is not significantly affected. In particular, the checkpointed data can't simply be copied at CP creation, since CPs may affect arbitrarily large subtrees of the file system hierarchy.

- Space utilization should be related to the amount of data modified since CP creation and not total data encompassed by the checkpoint.

- Deletion of past checkpoints need not be immediate, but it should be reasonably efficient in terms of computational and data I/O demands.

- Implementation details should ideally mesh well with emerging OSD standards. In instances where they do not, requirements should be well justified such that proposed modifications to the standard can be defended.

Given these basic requirements, we present a flexible architecture for facilitating the creation of arbitrary checkpoints in a large, distributed object-based file system. The concept of checkpoints embody greater flexibility than most existing file systems, and the architecture proposed is believed to be scalable and efficient.

[2] I think this is true; I made an inquiry on one of the Lustre discussion lists to verify this but haven't received a response.

4.1 Epochs

Checkpoints are initially defined and created based on a given point in time and the root of a subtree of the file system (*i.e.* `<current_time,subtree_root>`). In order to simplify underlying storage management, we simplify checkpoint definition and identification by dividing time into a sequence of *epochs* such that only one CP is created per epoch, and CPs can thus be identified by the epoch in which the occur. Similarly, the life span of a particular file data fragment (object) can be partially described by a pair of epoch numbers specifying the time interval in which it is defined.

4.2 Realms

When checkpoints are created, they typically refer to large sets of items in the system. Each set of items that uniformly belong the same set of CPs comprise a *realm*. The life span of a particular item (inode or data object) is fully defined by the realm it belongs do, and the corresponding list of CPs that reference that data or metadata. (The epoch time interval mentioned above is a partial description, and is simply the first and last checkpoint for that realm.) Grouping objects into realms allows object references to be efficiently specified. The root of each realm is stored and specified at its root. The realm for any particular item can be identified by walking up the directory hierarchy until a realm root is found.

When CPs are created over dynamic hierarchies of data, however, realm definition becomes a bit complex. If a new CP is created with the same root as an existing realm, the CPs epoch can simply be added to the list. When the CP and realm roots do not correspond, however, a new realm is created. At that point, the more deeply nested root of the new checkpoint forms a new realm that recursively includes the contents of the outer realm in addition to the new checkpoint's epoch. Things are further complicated when files or directories within one realm are moved into another realm. Although their new position in the hierarchy might otherwise imply a different set of CPs and likely inclusion in a different set of future CPs, the past CPs for which a realm's objects are defined remains the same and must be preserved. Moves between realms thus also induce the creation of new realms rooted at the relocated item, which recursively include the subset of the previous realm prior to the epoch in which they are moved.

The recursively defined structure of realms forms a directed graph of nodes, whereby links between nodes indicate the inclusion of interval subsets of other realms corresponding to time periods during which realms were nested (and subsequently referenced by the same CPs). Realms are identified by the inode number they are rooted at. Be-

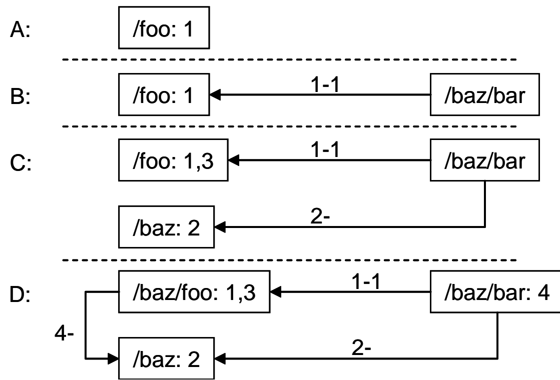

Figure 3: Realms and their corresponding sets of checkpoints are defined by nested sets, represented as a directed graph where individual edges imply the inclusion of a interval subset of the target realm. The realm graph is shown at different points in time, interspersed with checkpoint creation and the movement of subtrees of the directory hierarchy. Checkpoints rooted at each realm are shown within the boxes, while the intervals included from other realms are shown next to the graph edges.

cause realms may be referenced outside of their subtree of the hierarchy, their root inodes must be anchored as soon as a child realm is created, and a reference count should be maintained for effective cleanup when checkpoints are eventually removed or expire.

Figure 3 illustrates how a realm graph would evolve based on a typical sequence of operations. Initially, a checkpoint of `/foo` is taken in epoch 1 (A). An `mv /foo/bar /baz/bar` causes a new realm to be created that includes epoch 1 from the previous ancestor (B). A checkpoint on `/baz` defines an additional CP for all nested items including `bar` (C). Finally, a checkpoint is created for `/foo` in epoch 3, an `mv /foo /baz/foo` relocates an existing realm in the hierarchy, and finally a checkpoint on `baz` defines a new CP in epoch 4 for all nested items (D). It should be noted that although movement of directories might appear to potentially create a graph with cycles, this is not actually the case: edges are actually referencing intervals of realms, which can never be contained within each other cyclically at any single point in time.

5 File Data

The fundamental goal of checkpoints is to preserve the state of files at some particular point in time. Implementing a checkpoint mechanism in an object-based storage system presents a unusual challenge in this area because file I/O occurs in a parallel and distributed fashion: although clients interact with the metadata server to obtain capabilities, once permission is obtained they can write to OSDs directly with no single point of control. In particular, unlike most existing systems, there is no single point in the system where the CP creation can cause past state to be suddenly preserved instead of overwritten.

The fundamental mechanism of efficient archival data preservation is copy-on-write, whereby some sort of metadata indicates that a block of data is referenced multiple times (in this case, by current and/or past epochs) but data is not duplicated until subsequent writes require it. The granularity of copy-on-write is another critical question in an object-based storage system based on objects instead of blocks as the basic unit of storage.

5.1 Granularity

There is an underlying trade-off related to the granularity of any allocation mechanism, including the granularity of a copy-on-write strategy: finer grained allocation results in better space utilization at the expense of metadata and software complexity, while coarse allocation wastes underlying storage while retaining simplicity. The motivating purpose of object-based storage is to push allocation into the OSD and free the file system from concerning itself with low-level storage. In the context of such a system, sub-object-based copy-on-write semantics make little sense. The wide range of file sizes supported in large systems also make coarse-grained per-file approaches prohibitively wasteful.

Semantically, a system should thus manage archival duplication at the same level as allocation: on a per-object basis. When archived data (data belonging to some checkpoint) is modified, the containing objects must be replicated such that the new data does not clobber the old. For this process to occur efficiently and atomically (which are important for performance and consistency), the OSD interface should support either an atomic RENAME or a DUPLICATE command.

Although semantically we would like to identify data at the granularity of objects, that does not preclude efficient and intelligent allocation at lower levels. If copy-on-write mechanisms are implemented within the object storage layer, an OSD device need only be made aware of multiple references to duplicate data in order to manage low-level allocation efficiently and intelligently. Thus, although a RENAME would be sufficient for implementing copy-on-write at the object level (clients could rename existing objects and then write new data to a new object), a DUPLICATE command allows the file system to communicate useful information to the OSD. Notably, the OSD is not required to use that information: it is free to simply make an on-disk copy of the object. An intelligent OSD, however, might reap significant performance and

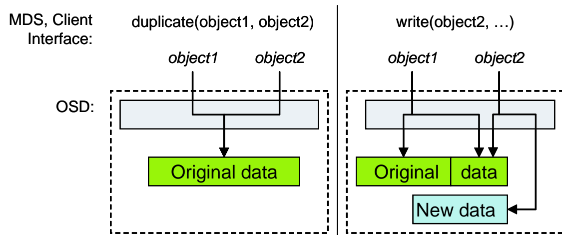

Figure 4: An intelligent OSD can implement finer-grained copy-on-write semantics (*e.g.* at the block level) while presenting a purely object-based external interface. The *duplicate(object1, object2)* operation allows additional information to be communicated to the OSD, in contrast to creating a writing identical information to a new object. The gray bars indicate allocation choices internal to the OSD.

storage utilization benefits by only duplicating data when it is about to be overwritten (as illustrated in Figure 4).

The current draft of the T10 OSD command set does not include support for either a RENAME or DUPLICATE command; only CREATE, READ, WRITE, and REMOVE are currently supported. Although it is possible to build copy-on-write behavior on top of this interface, it would be significantly less efficient than implementations based on only slightly richer interfaces that include DUPLICATE or (to a somewhat lesser extent) RENAME.

5.2 Object Nomenclature

The file data object distribution and replication strategy employed by RUSH and FARM guarantee a balanced pseudo-random distribution of file data across the OSD cluster and the placement of all replicas of any particular object on separate disks. On any given OSD or for any given file, however, individual objects need to be uniquely identified within some namespace such that the file they belong to and portion of that file are distinct for all objects. Conceptually, objects are uniquely identified by <file_id, block_id>; the RUSH/FARM object distribution guarantees make an object's collection and replica ids object *attributes* instead of *identifiers*.

In an archiving file system, it is additionally necessary to identify different checkpointed versions of a given piece of a file data. An object containing file data may define either current "live" file state or only a particular archived interval of time. As such, although a given object may not correspond to an single <file_id, block_id, epoch>, conceptually any such tuple should uniquely identify a single object (if it exists). Because the set of checkpoints for which an object's data might be defined is metadata that is readily available for any given file (based on its place in the hierarchy), identifying object lifetime intervals by either the first or last epoch might be a sufficient naming strategy (*e.g.* <file_id, block_id, last_epoch>).

Naming objects within an actual OSD is, unfortunately, more difficult in practice than in theory because the object name must be reduced to a unique bit string. Further, one of the critical motivating elements of the RUSH/FARM architecture is the lack of cumbersome and inefficient allocation tables, a design choice that relies on a sufficiently large namespace to allow objects to be mapped into a namespace without danger of collision.

Current OSD interface proposals identify objects by 64 or 128 bit strings, depending on the feature set desired. In the case of the ANSI T10's draft specification, a 128-bit identifier is divided into two 64-bit parts: a partition id and object id. Although the ability to subdivide the namespace into logical partitions may be desirable in practical environments, the semantics as currently defined prevent the entire 128-bit namespace from being seamlessly used by a single application because collections (indexed sets of objects) cannot span partition boundaries. Unfortunately, collections currently appear to be critical for supporting the replication and reliability subsystems of our architecture, effectively leaving us with a 64-bit object namespace (in the absence of further trickery).

For a file system on the scale of petabytes, a 64-bit namespace is likely sufficient for identifying objects in the absence of checkpoints. A 64-bit <file_id, block_id) with a 48-bit file_id would allow almost 3 trillion files with the remaining 16-bits providing 65,000 objects per file and a corresponding 68 TB max file size (with 1 MB objects). Larger files or a sparser file id namespace are possible if file data is striped across sets of objects instead of storing stripe units in individual objects.

Adding epochs to object identifiers is more difficult because the epochs for any given object will generally not be sequential (unless the object is included in all checkpoints). Further, it is not immediately clear how to map sparse epoch numbers into a narrow bit string in an efficient manner that does not impose undue burden on the hierarchical management of realm metadata. If such a mapping is feasible, striping across sets of objects were employed (*e.g.* 1 MB stripe units but 256 stripe units per object), and an upper limit on the number of checkpoints for a given file were acceptable, then a 64-bit namespace may prove sufficient. Even so, it is a tight fit.

If slower performance for archived data is acceptable, an alternative approach is to identify objects initially by <file_id, block_id> and then locate archived versions based on attributes of the currently live object. Such an approach would allow archived objects (those below the dotted line in Figure 5) to live in a separate namespace (such as a different partition on the OSD) while effec-

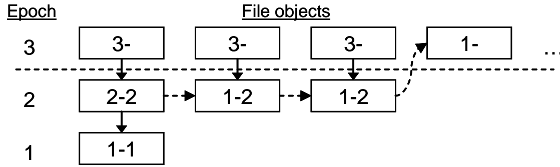

Figure 5: Storage of data objects for a single file after 4, 1, and then 3 blocks are rewritten (without truncation) during epochs 1, 2, and 3, respectively. Objects above the dotted line comprise the current "live" file state with well defined names, while items below the line may be stored in a secondary partition and referenced only by object attributes of the live objects.

tively distributing the lookup table for locating archived data across all OSDs. (This strategy would benefit from an OSD rename operation that permitted objects to move between partitions.)

5.3 Client Synchronization

A fundamental challenge in an object-based file system is that checkpoints are initiated by the metadata server cluster, but one of their basic goals is to affect data I/O. Because clients are performing I/O with the OSDs directly, they become responsible for properly preserving previously written data to achieve the basic copy-on-write behavior. Furthermore, when files are opened for writing at the time of consistency point creation, for instance, data written before and after the checkpoint should be disambiguated such that the checkpointed data will properly reflect the state of the file system at a single point in time.

If the basic copy-on-write strategy is formulated as an overwrite-or-copy-then-write dichotomy, then client synchronization becomes a very difficult issue in a system with potentially hundreds of thousands of clients and disks. The permission capability infrastructures controlling client access to OSDs are generally quite robust (as proposed) and should facilitate capability revocation to enforce potentially system-wide coordination, but this process is likely to be prohibitively expensive whether it is affected through client callbacks or OSD capability revocation. This is probably the case even though synchronization is only necessary for current writers to open files.

An alternative approach is for clients to *always* perform copy-on-write and preserve previous file data. At the time a file is closed, the MDS can decide if the previous file data should be retained for a recently created checkpoint (after the file was opened) or simply discarded. Individual objects within the file might have different fates depending on whether they were written before or after a particular point in time. Although this approach should be quite efficient and generally work quite well, the checkpoint time granularity is coarse; data written to the same object block before or after a point in time cannot be inexpensively disambiguated without the client having prior knowledge of the timestamp of interest or retaining a significant amount of information about write behavior. However, given that in practice checkpoint time granularity is already affected by the rate at which data is written through client caches and that in practice administrators will explicitly sync all data to disk when integrity is of particular concern (*e.g. for backups*), fine-grained ordering may not be of great concern.

5.4 Garbage Collection

One critical element of a checkpointing file system is the ability to expire old file system state and reclaim storage. The efficiency of garbage collection in particular is important in a large system, although not necessarily the time period over which it occurs. Each object in the store is referenced by the current state of the file system and/or some set of checkpoints. When all references are broken (the data object is no longer in the live file system and all relevant checkpoints are removed) the object should be discarded.

Garbage collection can be performed in a distributed fashion with minimal interaction with the metadata cluster by utilizing the collection facilities present in current OSD interface proposals. Collections occupy the same namespace as object and define (potentially overlapping) sets of zero more more objects. Objects can belong to zero or more collections, provided they exist within the same partition.

For each epoch, we define two collections: `top_n` and `bottom_n`, where `n` is an epoch number. When an object is removed from the active file system (because the file is modified or removed), the archived object is added the two collections corresponding to the first and last CPs for which it is defined. The identifier for the realm the file belongs to is also specified as an object attribute such that for each archived object in the store, the list of referencing CPs can be retrieved from the MDS cluster.

When a checkpoint is removed (say, `i`), then all objects directly affect can be efficiently identified by enumerating the relevant top and bottom collections. In particular, all objects belonging to both `top_i` and `bottom_i` can be immediately removed as they are referenced by a single, newly removed checkpoint (object *c* in Figure 6). For the remaining objects, the list of valid CPs referencing them is retrieved (by looking up each object's realm in the MDS cluster) and their collection membership is adjusted based on the new first and last CPs referencing them. This process can proceed in a distributed fashion, affected by any agent with sufficient capability to remove archived objects and adjust collection membership on the OSD and (ide-

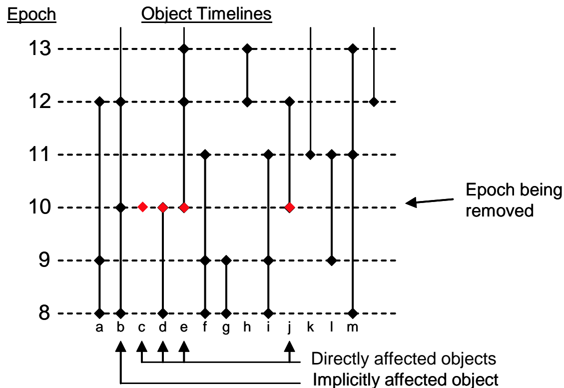

Figure 6: Each OSD object is defined for some set of checkpoints and possibly the current file state based on the realm it belongs to and its life interval (shown as a vertical line). When CP 10 is removed, objects contained in `top_10` or `bottom_10` are either removed (*c*) or have their collection membership adjusted (*d, e, j*). Other objects (*b*) may be only implicitly affected by the CP removal in that they are no longer semantically defined at that point, a fact reflected by the non-existence of that epoch in the object's realm.

ally) enough memory to effectively cache the epoch lists for most realms.

6 Metadata Management

The relatively simple semantics of file I/O make file data easy enough to deal with. Although metadata management of an evolving file hierarchy is a bit more difficult to keep consistent, the lack of a constraining OSD interfaces leaves greater flexibility in its management.

6.1 Checkpoint Creation

We've already described checkpoint realms, each of which defines a list of checkpoints that uniformly apply to the content of a subtree of the file hierarchy. Realms are associated with and thus uniquely identified by the inode at the root of the realm's subtree. Collectively realms form a directed graph denoting their recursive definition. Because the hierarchy in which they live is dynamically changing, inodes at the root of each realm need to be anchored such that they can be located in the future.

When a checkpoint is created, its epoch identifier is immediately added to the checkpoint's realm and the global epoch counter is incremented. Because parent directories are already implicitly involved in all metadata transactions (for POSIX security checks), the MDS can recognize (with no addition cost) when a new checkpoint has been created for any inode nested beneath that point and initiate the appropriate operations to preserve archived file system state.

6.2 Inodes and Directories

The underlying metadata storage strategy is to keep inodes within a given directory together as a unit, stored together on disk. The simplest metadata that must be archived are changes to inodes. Metadata in individual inodes changes when file data is modified (mtime, file size may be updated) or when explicitly altered (*e.g.* chmod). Thus, for any given inode, there may be a sequence of past revisions that are referenced by checkpoints and must be preserved.

The directory entries (file names and inode pointers) that comprise directory metadata must also be preserved for different points in time. The number, location, and name of directory entries referring to individual inodes may change over the life of a file, and must be accurately preserved for each checkpoint. The directory view for each checkpoint should be well-defined for each directory, and the location of inodes that directory entries refer to should be well-defined, as their location may change over time as files are moved in the hierarchy.

On fundamental question is whether all revisions for a particular inode should be stored together. Doing so would require the equivalent of hard links to be created (using anchors to locate distant inodes by number) when inodes appear in different directories in different checkpoints, at some additional cost related to the increased size of the anchor table. Alternatively, different revisions of inodes could be stored in the directories that contain them at that point in time. Because the file allocation specification is concise and constant (only a single input value is needed to seed the RUSH algorithm) this is possible, although it may make inode number namespace management intractable. In a system without a global inode index, inode number reclamation will likely depend on accurately recording inode deletion; allowing multiple revisions of inodes to appear in different parts of the file system likely precludes doing so efficiently. For this reason, we propose that inode revisions always be maintained within a single directory (the inode's current or last home) and anchors be used locate inodes for past checkpointed directory entries (as illustrated in Figure 7).[3]

The specific on-disk structure of directories is not of critical importance so long as the referential integrity of the past and present file system state is preserved. In particular, this means:

[3] It should be possible to fragment an inode's history in multiple directories if both fragments are anchored. This incurs a higher (anchor table related) cost than anchoring only the most recent inode location, but could conceivably be desirable for particularly large inode logs.

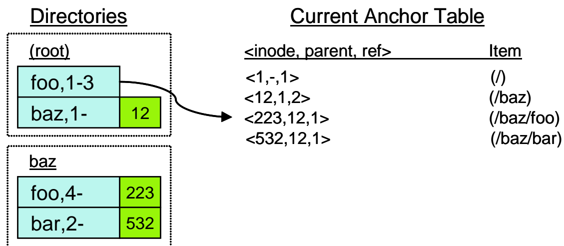

Figure 7: Directory contents for the root directory and `baz` are shown along with the current contents of the anchor table. The file `foo` originally appeared in the root directory for epochs 1-3 and later moved to `baz`. The old directory entry links to the anchor table to locate the recently moved inode. Each inode retains a log (conceptually) of past states over the checkpoints in which it is defined.

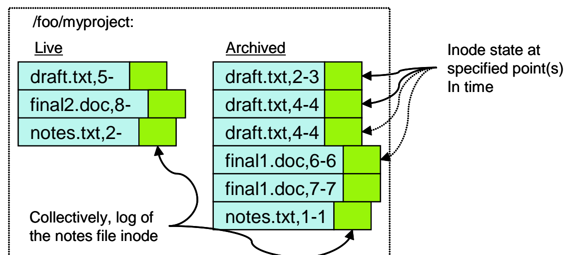

Figure 8: Live and archived contents of directory stored in two multiversion tables, one for the live file system and one or archived metadata. Collective, both tables retain past and present directory entry and inode state over some set of checkpoints.

- Inode revisions are kept together (in conceptual inode logs) such that inode numbers can be reclaimed effectively.
- Inode logs are attached to the directories that contain the inodes in a distributed fashion (no central tables).
- When inodes change directories between checkpoints, inode contents and logs follow the most recent and/or actively used referring directory entry. The inode is added to the anchor table so that other references can find the distant inode by its inode number.
- Directory entry state is preserved for all checkpoints for which they should be defined.

Prior research by Soules and Strunk with the S4 and CVFS file systems [10, 9] demonstrated that both metadata journaling and multiversion B-trees are practical for archiving inode and namespace metadata, although the two strategies correspond to a trade-off between time and space efficiency. Soules concluded that journaling for file metadata and multiversion B-trees for directories were most efficient for common workloads. Because checkpoints will be less frequent than in the comprehensive versioning file system he was investigating (which implicitly checkpoints every modification), and because file metadata in our system does not include block allocation information, a multiversion index structure alone may be sufficient. Santry's work with the Elephant [7] file system, which does not comprehensively archive state but is still archives relatively aggressively, also treated inode and directory metadata differently (similarly using logs and a multiversion directory index, respectively).

Given that checkpoints in our system will be less frequent than CVFS or Elephant, we believe that inode and directory entry state can be efficiently retained in one or more multiversion prefix B+-trees per directory. The multiversion tree keys entries on both the original key (a filename) and the time interval over which they are valid. Prefix B+-trees attempt to use partial substrings of key values in index nodes for space efficiency, a strategy that should be particularly effective for sorted file names, particularly when multiple revisions of identical names may be present. In this context, the conceptual inode log might be spread over a set of entries in the structure. Like Elephant, archived entries can be moved into an auxiliary structure to keep the B-tree size for live file system data small and quick. Two such tables' contents (for live and archived metadata) are shown in Figure 8, which shows a sample directory containing a series of file modified and checkpointed over a period of time.

6.3 Anchor Table

The state of the anchor table in any defined epoch must be recoverable in order for data within past checkpoints to be usable. There are two general possibilities for accomplishing this. A naive solution might be to make a copy of the records in the table relevant to a given consistency point. If that CP applies to a small portion of the hierarchy, it may be the case that none of its inodes are represented (that is, the CP root is not in the table when the checkpoint is created). If the checkpoint root *is* in the table, records relevant to the subtree could be recursively identified and copied to an auxiliary structure attached to the CP root inode and associated with the given epoch. However, if the checkpoint applies to a large subtree (or even the entire file system), the number of records that have to be copied may be very large and make checkpoint creation prohibitively expensive.

Instead, the relative uniformity of records in the anchor table makes it easy to extend record keys to include creation and deletion epochs, allowing archived items to be retained as long as they are relevant to checkpoints still in use by the system. A multiversion B-tree seems well-

suited for this task, particularly given the small size and simplicity of anchor table records.

6.4 Garbage Collection

When a checkpoint is removed from the file system, the affected portions of the file system are specified by the structure of the realm digraph. In particular, each checkpoint is associated with a single realm in which it is rooted, and may additionally apply to other realms that were nested beneath it at that time (indicated by edges specifying intervals which include that checkpoint epoch). Because affected realms must be enumerated by a reverse graph traversal, reverse edges need to be recorded such that we can enumerate all of the realms referring to a particular realm. When all of the CPs in a realm are removed, the realm itself can be destroyed and the references to it expired.

File system metadata archived within individual directories should be eventually discarded as well. Because the metadata partition is primarily subtree-based, the garbage collection process can proceed independently on MDS nodes authoritative for the elements of the hierarchy affected. In particular, the intervals for keys in multiversion B-trees should be adjusted in a fashion analogous to the `top_n` and `bottom_n` collections for OSD objects. When all referencing checkpoints have been expired, keys can be removed entirely.

As with objects, metadata garbage collection does not have strong time constraints. Once the realm definitions are updated (a near-instantaneous process) it is immediately clear at any point in the hierarchy what metadata is useful and what is extraneous. Furthermore, because metadata is particularly small, the primary benefit of expiring old metadata is performance-related in that simpler directories translate into slightly lower I/O and CPU demands. However, because the system architecture makes it likely that metadata servers will perform this cleanup, there is little motivation to do so proactively. Expiration can be delayed indefinitely until a directory must be read or rewritten for other reasons, resulting in lower amortized I/O and CPU overhead. Proactive metadata expiration will still result in reduced storage utilization sooner, but other resource constraints will likely only make it worthwhile during relatively idle periods, if at all.

7 Future Work

Although the basic infrastructure and metadata strategies necessary to facilitate archiving checkpoints in a large distributed file system are outlined here, the implementation details are still relatively ambiguous. For the most part they are not of direct importance to the high-level architecture, but in many cases choices depend on performance characteristics of different strategies. As we move closer to a working implementation of this system, it will be important to revisit many of the assumptions about what will and will not be most efficient or relevant to overall system performance.

This paper examines the preservation of archived file system state in the context of explicitly initiated checkpoints, synchronously applied to entire hierarchies of the file system. It may be desirable for future systems to employ a strategy more like that of Elephant, in which heuristics control what data is preserved in the system at the granularity of individual files, at the time of modification. In this context, the some of the generalizations—in particular, the simplification of reducing time to a series of epochs—may not be appropriate.

8 Conclusions

We present an architecture for efficiently managing archived checkpoints for arbitrary subtrees of a file system hierarchy in a large object-based distributed file system. Mechanisms for managing storage allocation and garbage collection after checkpoint removal are defined, and the fundamental data structures and strategies necessary for managing file system and archival metadata are described.

The collection mechanism in current OSD interface proposals are particularly useful for efficient garbage collection in archival systems. However, the lack of even a RENAME command will significantly affect system performance built on existing proposals. Furthermore, the addition of an OSD DUPLICATE operation could vastly improve efficiency of such a system (both in time and space).

References

[1] P. J. Braam. The Lustre storage architecture, 2002.

[2] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17. USENIX Association, Jan. 1997.

[3] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

[4] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[5] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, pages 15–24, Dec. 1985.

[6] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In D. D. E. Long, editor, *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[7] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, Dec. 1999.

[8] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[9] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 43–58, San Francisco, CA, 2003.

[10] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, Oct. 2000.

[11] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Honolulu, HI, June 2004.