File Versioning for Block-Level Continuous Data Protection

Maohua Lu* Tzi-cker Chiueh*#
Stony Brook University* Symantec Research Labs#
{mlu,chiueh}@cs.sunysb.edu

Abstract

Block-level continuous data protection (CDP) logs every disk block update so that disk updates within a time window are undoable. Standard file servers and DBMS servers can enjoy the data protection service offered by block-level CDP without any modification. Unfortunately, no existing block-level CDP systems can provide users a file versioning view on top of the block versions they maintain. As a result, the data they maintain cannot be used as an an extension to the on-line system with which users routinely interact. This paper describes a name-based *user-level file versioning system called* UVFS *that is designed to reconstruct file versions from disk block versions maintained by a block-level CDP.* UVFS *reconstructs file versions by following the last modified time of files and directories, a common file metadata supported by almost all modern file systems, and therefore does not require any modification to the host file system that a block-level CDP system protects. In addition,* UVFS *incorporates a file system-specific incremental consistency check mechanism to quickly convert an arbitrary point-in-time block-level snapshot to a file system-consistent one. Performance measurements taken from a fully operational* UVFS *prototype show that the average end-to-end elapsed time required to discover a file version is under 50 msec from the perspective of an NFS client serviced by an NFS server backed by a block-level CDP system.*

1. Introduction

As modern enterprises rely on an increasing amount of digital data for continuous and effective operation, strong data integrity and availability becomes a critical requirement for enterprise-class file/storage systems. Conventional data backup systems periodically take a snapshot of the file/storage system, so that the file/storage system can be restored back to one of these snapshots in case it is corrupted. However, this approach is limited in terms of recovery point objective (RPO) and recovery time objective (RTO). That is, it cannot roll the file/storage system back to arbitrary points in time, and as a result its associated recovery time cannot be bounded because additional manual repair may be required after a programmatic rollback. One effective way to improve the RTO and RPO of a data backup solution is continuous data protection (CDP), which keeps the before image of every disk/file update operation for a period of time. Because CDP allows every update operation to be undoable, it supports arbitrary point-in-time rollback without incurring excessive recovery time.

A block-level CDP system [2]–[5], [16] is one that applies CDP at the level of disk access, i.e. every disk write operation within the *data protection window* [1] is logged. The key advantage of block-level CDP is that its data protection service is immediately accessible to different file systems without any modifications. However, because the high-level file system or DBMS operations that trigger individual disk writes are completely opaque to block-level CDP systems, the data they maintain are mainly used for data recovery purpose, rather than as an extension of the on-line file service whose data they protect.

This lack of knowledge of the high-level context behind each disk write imposes two limitations on existing block-level CDP systems, which substantially decreases their practical appeal. First, when a user of a network file server protected by a block-level CDP system takes a point-in-time snapshot, the file system metadata in the returned snapshot may not be consistent with one another. This means that even though existing block-level CDP systems can support arbitrary point-in-time disk image snapshots within the data protection window, these snapshots are not necessarily file system-consistent. Second, none of the existing block-level CDP systems provide the same file versioning functionality as a versioning file system, because they lack the necessary file system metadata required to map disk blocks to files. As a result, a block-level CDP system cannot answer such questions as "what is the last version of `/a/b` before T", "how many versions of the file `/a/b` exist in $[T1, T2]$", etc.

We have developed a user-level file versioning system specifically designed to address these two limitations of existing block-level CDP systems. This system consists of a file versioning subsystem called *UVFS* (User-level Versioning File System) and an incremental file system consistency checker called *iFSCK* [15], which exploits file system-specific knowledge to convert a point-in-time disk image snapshot to be file system-consistent. This paper focuses specifically on the design, implementation and evaluation of *UVFS*, which augments a block-level CDP system with a file versioning capability that is portable across all main-stream operating systems (Linux, Solaris, and Windows XP).

1. The data protection window is the time period within which every update is undoable. Typical data protection window length is a week or a month.

File Operation	Incarnation/Version Changes
Creation	Start of a new incarnation
Deletion	End of an existing incarnation
Renaming	End of an existing incarnation and start of a new incarnation
Truncation	New version
File Write	New version
File Append	New version

Table 1: *Incarnation/version modifications associated with each type of file update operations.*

2. Portable File Versioning

2.1. Overview

In *UVFS*, a file system object is uniquely defined by its pathname, rather than by its internal representation, such as an Inode. Because of hard links, an Inode can have multiple pathnames, each of which still corresponds to a unique file system object. If a file system object is renamed, it becomes a different file system object. An $incarnation$ of a pathname corresponds to a file system object with that pathname from its creation to its deletion (including rename). If a file system object is created and deleted multiple times, it has multiple incarnations. Different incarnations of a file system object can use different Inodes. An incarnation can have multiple $versions$, each corresponding to a distinct modification within the incarnation. Logically, versions associated with one incarnation are unrelated to versions associated with another incarnation with the same pathname. The second column in Table 1 shows the file incarnation/version modifications associated with different file update operations.

Given a point-in-time disk snapshot, *UVFS* leverages the original host file system, from which the snapshot is taken, to properly interpret its contents, and makes only two assumptions about the file system: (1) support for *last modify time* field and a system call to access it such as $stat$ in Linux and (2) support for a system call that accesses the contents of a directory file such as $readdir$ in Linux. Because all main-stream operating systems, including Linux [7], BSD Unix [18], Solaris [17], AIX and Windows XP/Vista, support these two features, and that *UVFS* is implemented completely at the user level, *UVFS* is portable across different operating systems. However, for ease of exposition, the following description assumes an ext2/ext3 file system. Because the time resolution of the $stat$ system call on Linux kernel 2.6 is 1 second, the current *UVFS* prototype cannot distinguish file versions in the same second.

UVFS provides users the following file version query operations that are commonly supported by existing versioning file systems:

- *File/directory snapshot access:* accessing a particular snapshot described by a time point and a pathname.
- *Searching for versions associated with an incarnation within a time range:* listing all the versions associated with an incarnation with a given file pathname within a specified time range.
- *Searching for incarnations within a time range:* listing all incarnations with a given file pathname within a specified time range.
- *Version search across incarnations within a time range:* listing all versions associated with all incarnations with a given file pathname within a specified time range.
- *Searching for all file/directory versions under a directory within a time range:* listing all versions of files and subdirectories that ever existed under a given directory within a specified time range.

2.2. Version Query Processing Algorithms

When an application on an NFS client issues a file version query, a *UVFS* agent on the client services the query by executing the following algorithms against the NFS server and the block-level CDP server. The basic primitives used in the query processing are (a) set-up/tear-down of snapshot images, (b) traversal of the file systems associated with snapshots, and (c) internal processing in the form of comparison of directory contents or timestamps.

2.2.1. Versions Associated with an Incarnation

When a new version of a file incarnation is created, the *last modify time* field of the incarnation's Inode must be modified. Therefore, to discover the versions of an incarnation that exist within a time range, one just needs to identify the time points at which the incarnation's *last modify time* field is modified, and to access the incarnation's snapshots at these time points. More concretely, given a pathname P and a time range $[T1, T2]$, *UVFS* first accesses P's snapshot corresponding to the time point $T2 - \delta$ and retrieves that snapshot's *last modify time*, say T. If $T \geq T1$, *UVFS* repeats the same procedure to locate the version immediately prior to the version corresponding to T, etc. If $T < T1$, then *UVFS* has found all the versions of the incarnation P within $[T1, T2]$ and the process stops. The parameter δ is chosen in such a way to ensure that whatever modifications to the file system before and at $T2$ should already be reflected to the snapshot at time $T2$ - δ.

2.2.2. Incarnations Associated with a File

A file pathname may refer to multiple incarnations within a time period. Each of these incarnations corresponds to a pair of creation and deletion of a file system object with that file pathname. When an incarnation with a particular pathname is created or deleted, the immediate parent directory containing the file pathname must be modified, as is its *last modify time* field. To discover all incarnations of a given pathname P within a time period $[T1, T2]$, *UVFS* first extracts the pathname for P's immediate parent directory, say Q, identifies all versions of Q within $[T1, T2]$, and compares adjacent versions of Q to determine if the difference between them is related to the creation or deletion of P. Every time a new instance of P appears in a new Q version, a new incarnation of P is created; every time an existing instance of P disappears in a new Q version, the corresponding incarnation of P is considered over.

However, identifying all versions of Q within $[T1, T2]$ itself is non-trivial, because it requires identifying all incar-

nations of Q within $[T1, T2]$; this in turn requires identifying all versions and incarnations of Q's immediate parent directory within $[T1, T2]$, all versions and incarnations of the immediate parent directory of Q's immediate parent directory within $[T1, T2]$, etc. Fortunately, this recursive process eventually stops because by definition, there is only one incarnation for the root directory of every file system.

2.2.3. Versions Associated with a File

This operation is simply built upon the above two operations. Given a pathname P and a time period $[T1, T2]$, *UVFS* first discovers all incarnations of P within the specified time range, and then extracts all the versions associated with each of these incarnations.

2.2.4. All File Versions Under a Directory

To service this type of file versioning queries, *UVFS* first locates all versions of the specified directory, then extracts all versions of every pathname that ever appears in any version of the specified directory, and finally outputs a union of all the file and subdirectory versions found.

2.3. Optimizations

To service a file versioning query, a *UVFS* agent needs to set up snapshots, traverse file systems associated with these snapshots, and perform some internal processing such as timestamp or directory content comparison. Typically the performance cost of internal processing is negligible. The cost of setting up snapshot consists of two components: (a) establishing NFS and iSCSI connections and (b) invoking *iFSCK* to fix an established disk snapshot. To reduce the cost of (a), *UVFS* reuses NFS/iSCSI connections and virtual devices created at the block-level CDP server that are set up for accesses to subsequent snapshots. To reduce the cost of (b), *UVFS* takes an optimistic approach by assuming that most point-in-time disk snapshots are consistent, and invokes *iFSCK* lazily, specifically only when either $readdir$ or $stat$ returns with an unexpected error during file system traversal. In addition, when *UVFS* does invoke *iFSCK*, it applies consistency check only to the path from the root to the target file or directory, and focuses only on their last modified times and directory contents while ignoring other types of file system metadata.

To reduce the performance cost associated with file system traversal, *UVFS* employs various forms of caching to reuse efforts invested in previous snapshot accesses. *UVFS* caches the *last modify times* for files and directories in a snapshot, and reuses them for subsequent file versioning queries that need to access the same snapshot. Moreover, to exploit the significant redundancy among the snapshots that are established during the service of a file versioning query, *UVFS* adds a simple caching mechanism on the CDP server that caches disk blocks which are recently accessed and associated with previously established snapshots. This disk block caching mechanism is meant to reduce the disk I/O cost associated with traversal of temporally adjacent snapshots that overlap with each other significantly.

3. Performance Evaluation

In this section, we evaluate the effectiveness and performance of *UVFS*'s file version searching capability. From the perspective of an end user, the ability to interactively navigate through historical versions of files enables her to quickly zoom into file versions of interest. We use the elapsed time for servicing a file versioning query as the metric for evaluating *UVFS*'s performance.

3.1. Methodology

The testbed used in this study consists of an NFS client node, an NFS server node, and a file update logging node based on an experimental block-level CDP system called Mariner [16], all of which are connected by a Netgear GS508T Gigabit Ethernet switch. The CDP server is a Dell PowerEdge 600SC machine with an Intel 2.4 GHz CPU, 768 MB memory, a Gigabit Ethernet card, and five IBM Deskstar ATA/IDE hard disks, four of which are log disks and one of which holds data. Other testbed nodes are Dell PowerEdge SC1425 servers with an Intel 2.8 GHz CPU, a Gigabit Ethernet card, and 1024 MB memory. The operating system is Fedora Core 3 with Linux kernel 2.6.11. The test file system used is an ext3 file system unless specified otherwise. When caching of disk data blocks on the CDP server is turned on, the amount of memory dedicated to caching is 4 Mbytes. We ran the following workloads to create historical images on the CDP node, and used the resulting images to evaluate *UVFS*. All experiment runs were conducted on machines with cold cache.

Synthetic Workload: The synthetic workload starts with a file system with only an empty root directory. It creates N subdirectories under the root directory, picks one of the subdirectories, say `/a`, and creates N subdirectories under it, picks one of the subdirectories of `/a`, say `/a/b`, and creates N subdirectories under it, and recursively applies the same set of operations until it creates a directory of a certain depth (D). At that point, the workload creates N files under one of the most recently created batch of directories (called the *leaf directory*, e.g. `/a/b/c/d/e` for $D = 5$), and for each file creates C incarnations, each of which in turn has K versions. Then it deletes all files in the leaf directories, all subdirectories in the leaf directory's parent directory, all subdirectories in the parent directory of the leaf directory's parent directory, and recursively upwards until the file system becomes an empty root directory again. So an instance of the synthetic workload is characterized by four parameters: N, D, C and K.

Lair Trace: The Lair trace is an NFS trace collected from the EECS NFS server (EECS) of Harvard University over two months [9]. The EECS trace grows by 2GB every day. We use the Trace-Based file system Benchmarking Tool [28] to replay this trace against the NFS server on the testbed.

SPECsfs: SPECsfs is a general-purpose benchmark for NFS servers [1]. SPECsfs bypasses the NFS client and accesses the NFS server directly. We ran SPECsfs with 1

server process, 1 NFS client and set the operation rate at 100 OPS to age the file system image.

Three types of file versioning queries are used in this performance study: a *version search* query asking for all the versions of all the incarnations associated with a given pathname, an *incarnation search* query asking for all the incarnations associated with a given pathname, and a *directory search* query asking for all the versions of all the incarnations associated with all pathnames under a given directory. Each file versioning query is serviced by a special agent on an NFS client, which accesses historical snapshots on a block-level CDP server through an NFS server.

3.2. Correctness of File Versioning Algorithm

To verify the correctness of *UVFS*, we ran the Postmark workload and compared the versions discovered by *UVFS* with those that were derived from a comprehensive file-level update trace collected during the run. To collect this trace, we instrumented the source code of Postmark to record every file-level update operation, including $open$, $write$, $close$, $unlink$, $mkdir$, $rmdir$, etc. The file system was mounted with the *dirsync* and *sync* flag. During each run, we turned off all file caching, including Postmark's own buffering, to ensure that all file-level updates are propagated immediately down to the block level.

After a Postmark run with 1,000 files, 1,000 subdirectories and 10,000 transactions, we deduced from the resulting file-level update trace that the run produced in total 16,637 versions of 5,900 files/directories. Because the temporal resolution of the file-level update trace is millisecond, we consolidated all updates to the same file block within the same second into one update, so as to match the temporal resolution of *UVFS*. It takes *UVFS* 305 seconds to complete servicing a *directory search* query starting from the root directory, and the result it returns matches exactly with those derived from the file-level update trace.

3.3. Synthetic Workload

We ran the synthetic workload with the *dirsync* flag turned on to force to disk synchronously every file system metadata update, including Inode bitmap, Inodes, directory entries, etc. Under this configuration, *UVFS* never needs to invoke *iFSCK* because every snapshot it accesses is always file system-consistent. As a result, the main performance cost associated with snapshot access comes from NFS and iSCSI connection set-up.

The performance cost associated with file system traversal mainly comes from the set of NFS operations used in the traversal. The performance cost of each NFS operation in turn is determined by the number of associated remote procedure calls (RPC). Table 2 lists the number of RPCs required by each type of NFS operation used in file version searching under a synthetic workload characterized by $\langle N, D, C, K\rangle$. The actual performance overhead of each RPC is dominated by the disk accesses it requires.

Assume the iSCSI/NFS connections required by the service of a file versioning query are pre-established and therefore their set-ups incur zero cost, the query processing cost is dominated by the file system traversal cost, which in turn is determined by the number of RPCs required. Processing of a *version search* query for a file f starts with an *incarnation search* of f to find all incarnations of f, which in turn triggers a *version search* of f's parent directory. This recursive procedure continues until it reaches the root directory. Therefore, the total number of RPCs required by a version search query is thus

NFS Op	getattr	lookup	readdir	access	readdirplus
Cost	1	D	N	1	N

Table 2: *The cost of each type of NFS operation used in file version searching in terms of numbers of RPCs required.*

$$VS_{RPC} = \sum_{i=1}^{D-1} 2*N*(i+2) + C*K*(D+2) \quad (1)$$

Accordingly, the per-version discovery time of a *version search* query is $\frac{VS_{RPC}*AvgRPCTime}{C*K}$.

Similarly, the total number of RPCs required by an *incarnation search* query is:

$$IS_{RPC} = \sum_{i=1}^{D-1} 2*N*(i+2) + C*(D+2) \quad (2)$$

Therefore, the per-incarnation discovery time of an *incarnation search* query is $\frac{IS_{RPC}*AvgRPCTime}{C}$.

A *directory search* of a directory d first finds all versions associated with d. A $readdir$ is then issued to read all directory entries of each version of d. For each file f under each version of d, a *version search* is initiated to locate all versions of f. Therefore, the total number of RPCs required by a *directory search* query is:

$$DS_{RPC} = \sum_{i=1}^{D-1} 2*N*(i+2) + N + N*C*K*(D+2) \quad (3)$$

and the per-version discovery time of a *directory search* query is $\frac{DS_{RPC}*AvgRPCTime}{N*C*K}$.

If the iSCSI/NFS connections required by the service of a file versioning query are not pre-established, the number of historical snapshots needed during a file versioning query plays an important role in the query processing cost. The numbers of snapshots required in an *incarnation search*, *version search*, and *directory search* are $2*N*D$, $2*N*D+C*K$, and $2*N*D+C*K*N$, respectively. As an example, if $D=4$, $C=2$, $K=2$, $N=10$, and the target file is `/a/b/c/f1`, a *version search* needs to examine all 20 versions of `/` (10 directory creations and 10 directory deletions) to determine that there is only one incarnation of `/a`, all 20 versions of `/a` (10 directory creations and 10 directory deletions) to determine that there is only one incarnation of `/a/b`, all 20 versions of `/a/b` (10 directory creations and 10 directory deletions) to determine that there is only one incarnation of `/a/b/c`, and all 40 versions of `/a/b/c` (20 files creations and 20 file deletions) to determine that there are two incarnations of `/a/b/c/f1`, from which it then locates the two versions of each incarnation based on its *last modify time*. In total, it needs to set up 100 (20 + 20 + 20 + 40) snapshots.

We set $N=10, C=2, K=2$, and varied the file system tree depth parameter D from 0 to 16 for the *directory search* query, or from 1 to 16 for the *version search* query and the *incarnation search* query. Figure 1(a) shows the

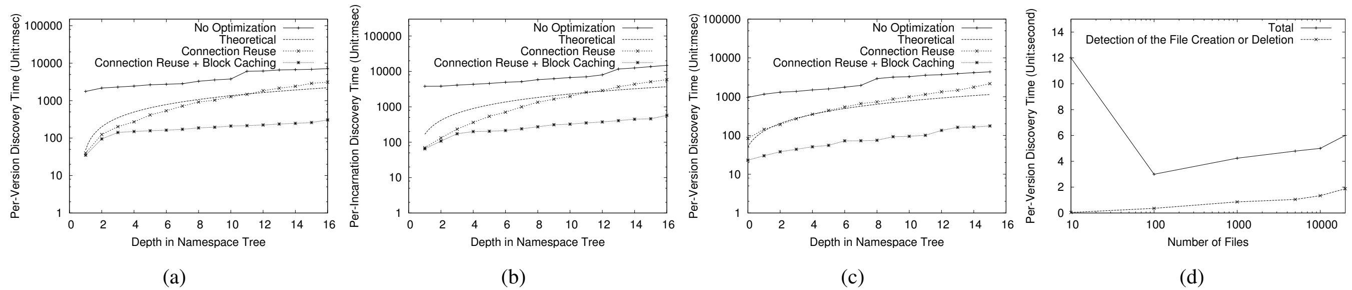

Figure 1: **(a)** *The average per-version discovery time for a* version search *query against a file in the leaf directory when N=10, C=2, K=2, and D varied from 1 to 16. The* **no optimization** *curve corresponds to a vanilla* UVFS *implementation without any optimization, in which iSCSI/NFS connections are set up on demand. The* **theoretical** *curve is derived from Equation (1) with* $AvgRPCTime = 16$ *msec and connection reuse turned on. The* **connection reuse** *curve corresponds to an implementation with connection reuse optimization, in which iSCSI/NFS connections are reused and thus pre-established in most cases. The* **block caching+connection reuse** *curve corresponds to an implementation with both connection reuse and block caching optimizations, in which disk block caching on the CDP server is enabled. The Y axis is in log scale.* **(b)** *The average per-incarnation discovery time for an* incarnation search *query against a file in the leaf directory. Other parameters are the same as those in* **(a)**. **(c)** *The average per-version discovery time for a* directory search *query against the leaf directory when N=10, C=2, K=2, and D varied from 0 to 16. Other parameters are the same as those in* **(a)**. **(d)** *The average per-version discovery time for a* directory search *query against the root directory and the portion related to detection of incarnation deletion and creation when D=0, C=1, K=2 and N is varied from 10 to 20,000. No optimization is enabled. The X axis is in log scale.*

average per-version discovery time of a *version search* query whose target is a randomly selected file f in the leaf directory. In each run, four file versions are returned to each *version search* query. Figure 1(b) shows that the average per-incarnation discovery time of an *incarnation search* query whose target is a randomly selected file in the leaf directory. In each run, two incarnations are returned to each *incarnation search* query. Figure 1(c) shows the average per-version discovery time of a *directory search* query whose target is a randomly selected leaf directory. In each run, 40 file versions are returned to each *directory search* query. There are four curves in each figure, corresponding to the theoretical performance cost when NFS/iSCSI connections are pre-established, the measured performance cost when NFS/iSCSI connections are pre-established, the measured performance cost when NFS/iSCSI connections are set up on demand, and the measured performance cost when NFS/iSCSI connections are pre-established and disk block caching on the CDP server is turned on.

Regardless of whether optimizations are enabled, the per-version or per-incarnation discovery time generally increases with the tree depth of the target file, because pathname lookup cost is a significant component, and NFS decomposes the lookup of a pathname of length N into N individual lookups, each of which takes roughly the same amount of time assuming there is no client-side caching. As shown in Figure 1(a) and (b), the average per-version discovery time of a *version search* is smaller than the average per-incarnation discovery time of an *incarnation search* because an *incarnation search* of a file involves a *version search* of the file's parent directory. However, the per-version discovery time of a *version search* is not necessarily longer or shorter than the per-incarnation discovery time of an *incarnation search* against the same target file system object. For example, if a file `/a/b` is created at $T1$ and deleted at $T2$ with no intermediate version, a version search and an *incarnation search* of `/a/b` between $T1$ and $T2$ take the same amount of time. However, if multiple versions of `/a/b` are created between $T1$ and $T2$ and the cost of discovering a new version by invoking $stat$ is larger than that to discover an incarnation, the per-version discovery time of a *version search* is larger than the per-incarnation discovery time of an *incarnation search* in this case. On the other hand, if the number of files under `/a` is very large (e.g. 8,000), the cost to discover an incarnation is larger than the cost to discover a new version, and the per-version discovery time of a *version search* is going to be smaller than the per-incarnation discovery time of an *incarnation search* .

In the calculation of the theoretical curve, we set $AvgRPCTime$ to a constant, 16 msec. However, in practice $AvgRPCTime$ varies with D. Larger D tends to increase the RPC cost due to less locality in data accesses during RPC processing. Thereafter, the theoretical curve does not always fit perfectly with the corresponding empirical results.

Being able to reuse NFS and iSCSI connections makes a big difference on the response time of the file version search queries, with the impact ranging from two orders of magnitude to a factor of 2 across all D values. If iSCSI/NFS connections need to be set up on demand, a large fixed overhead due to iSCSI/NFS connection set-up is added to the per-version or per-incarnation discovery time. That's why the slope of the curve for the no-connection-reuse case tends to be smaller or flatter than that for the connection-reuse case. Finally, iSCSI connection set-up takes much longer than NFS connection set-up, and one iSCSI connection set-up can support up to 255 snapshot accesses. Therefore, whenever servicing a file version search query needs more than 255 snapshots, an additional iSCSI connection set-up overhead will show up, for example, between D=10 and D=11 in Figure 1(a), between D=12 and D=13 in Figure 1(b), and

between D=7 and D=8 in Figure 1(c).

When the CDP server turns on disk block caching, the per-version discovery time of a file version search query is further reduced by a factor of 2 to 9, beyond what can be achieved with the network connection reuse optimization when $D = 16$. Specifically, this caching drastically cuts down the number of disk accesses associated with $readdir$ and $lookup$ operations in file version query processing. Accordingly, the effectiveness of this caching technique increases with D because the relative weight of file system traversal cost in a file version search query increases with D. In contrast, the effectiveness of the network connection reuse optimization decreases with D because the relative weight of network connection set-up cost in a file version search query decreases with D.

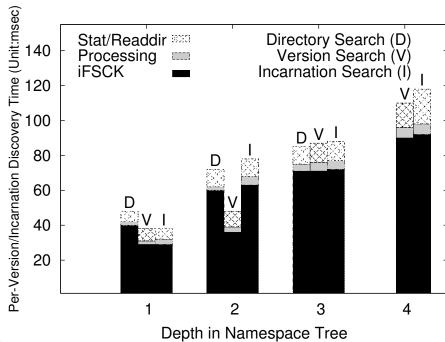

Figure 2: *The average per-version or per-incarnation discovery time and its breakdown into the Stat/Readdir, internal processing and* iFSCK *components for an* incarnation search, version search *and* directory search *against file system snapshots generated by SPECsfs, with the target object's depth in the file system namespace varied from 1 to 4.Because no directory with a namespace depth of 4 exists, there is no measurement for* directory search *with* $D = 4$.

Figure 1(d) shows the average per-version discovery time of a *directory search* query and the portion of it related to detection of incarnation deletion and creation with $D = 0, C = 1, K = 2$ and N varied from 10 to 20,000. We modified the synthetic workload so that consecutive creations or deletions of files occur one immediately after another. When servicing a *directory search* query, *UVFS* needs to set up snapshot images and compare contents of the adjacent versions of a directory to detect creation or deletion of a file incarnation inside that directory. Because the current *UVFS* implementation organizes each directory's content as an ordered list of file pathnames, it takes $O(NlogN)$ pathname comparisons to detect creation and deletion of file incarnations in a directory, where N is the number of files in the directory. The lower curve in figure 1(d), which corresponds to the average time required to detect changes in the directory's content indeed increases in a logarithmic fashion. When N is greater than 100, the directory content comparison time is a significant component and the average per-version discovery time of a *directory search* query increases with N. However, when N is less than 100, the directory content comparison time is negligible and the time needed to set up snapshots dominates, and the average per-version discovery time decreases when N is increased from 10 to 100 because the snapshot set-up cost is amortized over a larger number of versions as N increases.

Configuration	Number of Distinct File and Directories	Versions Discovered	Elapsed Time (Unit: msec)
Ext3+CDP+UVFS	28,693	29,674	849,378
Ext3cow - 60s	28,693	29,456	13,892
Ext3cow - 5s	28,693	29,667	14,415

Table 3: *Results of finding all file versions of a file system aged by the Lair trace using* $Ext3 + CDP + UVFS$*, and using* $Ext3cow$ *with two snapshot frequencies, 5 and 60 seconds.*

3.4. SPECsfs

We use the default setting of the SPECsfs benchmark: 12% write requests with the rest as read requests. A SPECsfs run creates a directory CL_i for the ith client, and another directory named *validatedir* for validation purpose. CL_i has in total N (the number of runs) $testdir_j$ directories, which hold the generated directories and files. For each test run, the corresponding $testdir_j$ contains up to 700 files/subdirectories. Processing a *directory search* query against the $testdir_0$ directory requires setting up 365 snapshots and only 10 of them need to invoke *iFSCK*.

We measured the per-version and per-incarnation discovery time for *version search*, *incarnation search* and *directory search* by issuing these queries against all files/directories at a particular file system name-space tree depth and computed the average of them. Figure 2 shows the per-version and per-incarnation discovery times of the three queries and their breakdowns under the SPECsfs benchmark. They are similar to those under the synthetic workload because SPECsfs creates only a small number of files/subdirectories and the name space hierarchy it creates is quite regular.

3.5. Comparison with Ext3cow

Ext3cow [20] is a file system that provides its users with file versioning, snapshotting and a time-shifting interface to navigate through the file versions. In this section, we compare Ext3cow with a vanilla Ext3 file system coupled with *UVFS* on top of a block-level CDP, in terms of the accuracy and performance of file version query processing. We used the Lair trace to age the test file system and measured the elapsed time required to locate all file versions under the root directory for each of the three configurations: Ext3cow on a local file system with the snapshotting frequency set to 5 seconds, Ext3cow on a local file system with the snapshotting frequency set to 60 seconds, and Ext3 on a local file system backed by *UVFS*, an NFS server and a block-level CDP server, with both NFS/iSCSI connection reuse and disk block caching turned on. The Lair trace used in this experiment is a one-hour trace starting from 10am on Oct 21, 2001. For Ext3cow, we use the time-shifting interface to retrieve all file versions under the root directory, starting from a cold file system.

Table 3 shows that $Ext3 + CDP + UVFS$ finds 0.7% more file versions than $Ext3cow$ with a 60-second snapshot-

ting frequency, and only 7 more file versions than $Ext3cow$ with a 5-second snapshotting frequency. This result shows that CDP plus *UVFS* indeed can capture some file versions that are missed by periodic snapshotting systems such as Ext3cow, although the marginal value of these missed versions depends on user and application requirements. As for performance overhead, $Ext3+CDP+UVFS$ needs to set up 372 snapshots to find all file versions, and as a result is 60 times slower than $Ext3cow$. This performance difference is attributed to two factors. First, $Ext3 + CDP + UVFS$ involves three parties over the network whereas $Ext3cow$ only requires local processing. Second and more importantly, $Ext3+CDP+UVFS$ does not require any modification to the host file system, whereas $Ext3cow$ builds file versioning directly into the file system itself.

4. Related Work

4.1. Versioning File Systems

Versioning file systems [11], [13], [19], [21], [23], [27] maintain versions of files at the file system level. File systems in Plan-9 [21], AFS [13], and WAFL [11] checkpoint the whole file systems periodically to support file versioning. For these systems, the temporal resolution of file versions is the checkpoint interval. Some file systems such as Elephant [23] and Versionfs [19] supports finer-grained versioning granularity and more flexible data retention policy. As in Elephant, a new file version is created only on file close but the versioning policy is flexible. VersionFS also provides a friendly interface for users to access old versions and to customize the versioning policies. VersionFS still incurs non-negligible performance overhead - about 100% when measured by the Postmark benchmark [12]. Neither Elephant nor VersionFS can distinguish between file updates that occur between a file open and file close operation.

Ext3cow [20] is a comprehensive versioning file system [6], [8], [20], [24] that does not require any modification to ext3's interface of the kernel. It implements a $time-shifting$ interface in ext3 file system and employs a copy-on-write scheme to avoid polluting the file system cache. In ext3cow, only versions that are propagated to disk are retained. Block-level CDP systems take the same approach and create a new block version only when an updated block reaches the storage server. However, in ext3cow, individual file versions are attached to snapshots and file versions between consecutive snapshots are invisible to end users.

4.2. Continuous Data Protection

CDP (*Continuous Data Protection*) [14], [22], [25], [26], [29] is generally used to protect data on critical servers such as database and email servers. In terms of *Recovery Point Objective(RPO)*, CDP provides the finest RPO granularity because CDP logs every data update. There are three types of CDP implementations available on the market place: 1) file-level, 2) network-level and 3) block-level.

File-level CDP logs updates at the local file system level and requires changes to the kernel. Apart from the implementation complexity, the performance overhead of file-level CDP is significant compared with file systems that do not support file versioning [20], [24]. UCDP (User-level Continuous Data Protection) [29] is a transparent NFS proxy that logs NFS requests and responses, and is shown to impose minimal performance overhead. UCDP is portable across all platforms that support the NFS protocol. Controller-based CDP [14] studied four CDP design alternatives and evaluated the tradeoff between the CDP granularity and the overhead in terms of space and extra disk I/Os for these CDP architectures. Mariner [16] adopted the *logging* architecture among four CDP design architectures but focused on improving the space utilization and write latency at the same time. The write latency can be as low as 0.5 msec while the space utilization is more than 70%.

4.3. Block-level Versioning

Clotho [10] pushed the versioning capability down to the block level and exposes several versioning-related primitives to the upper-layer applications, either file systems or database management systems. Upper-layer applications control the creation or deletion of block versions through those primitives. As a result, not all block versions are captured and upper-layer applications need to be modified to leverage the versioning capability. In contrast, CDP can capture all block versions and no modification of upper-layer applications is required. In Clotho, the consistency of file systems is enforced by flushing all dirty pages before creating a snapshot, which can degrade the performance greatly if snapshots are taken frequently. Instead, *iFSCK* [15] is developed for *UVFS* to shift the performance burden from the run time to the time when the snapshot is accessed.

5. Conclusion

Commercial block-level CDP products [2]–[4] have emerged as a critical building block in the set of data backup tools used in modern enterprise data centers, and have the potential to replace most of existing periodic data backup systems because of their flexible RTO and RPO and their ability to simplify storage administration. However, existing block-level CDP systems exhibit two weaknesses. First, the point-in-time snapshots they create are not necessarily file system-consistent, or more generally do not guarantee any metadata consistency for the application servers whose data they protect. Second, they do not provide a high-level versioning view of the block versions they maintain that is more user-friendly and easier to use. Specifically, they do not provide the kind of file versioning view that standard versioning file systems readily support. Both limitations are related to the fact that block-level CDP systems are designed to be transparent to the application servers whose data they monitor and protect. As a result, existing block-level CDP systems are limited to data backup and cannot be used as an extension of the on-line storage system.

This paper describes the design, implementation and evaluation of *UVFS*, which reconstructs a file versioning view similar to that provided by a versioning file system based on block versions maintained by block-level CDP systems, and for the first time demonstrates that it is possible to use a block-level CDP system as a seamless extension of the on-line filing system. Because *UVFS* relies only on the *last modify time* field of files/directories, it is portable across all main-stream operating systems, including Linux, Solaris, Windows XP and Windows Vista. To discover file versions or incarnations, point-in-time snapshots need to be "fixed" so that they are file system-consistent. *UVFS* incorporates an incremental file system checker called *iFSCK* for this purpose. Overall the performance overhead of *UVFS* is pretty modest. It takes on average 0.3 second to discover each version of a file that is 16 levels below the root, and on average 50 msec for a file that is 3 levels below the root.

UVFS was originally developed in conjunction with a high-performance block-level CDP system called Mariner [16] , and is currently being incorporated into a commercial block-level CDP product from Symantec.

References

[1] SPEC SFS (System File Server) Benchmark. http://www.spec.org/osg/sfs97/, 1997.

[2] InfiniView. www.mendocinosoft.com/technology.htm, 2007.

[3] InMage. www.inmage.net/features-and-benefits.html, 2007.

[4] RecoveryPoint. http://software.emc.com/products /software_az/recoverpoint.htm, 2007.

[5] Veritas NetBackup. http://www.symantec.com/business /products/overview.jsp?pcid=2244&pvid=2_1, 2007.

[6] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[7] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly & Associates, Inc., 2002.

[8] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-Level Versioning File System for Linux. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 27–37, Berkeley, CA, USA, 2004. USENIX Association.

[9] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 203–216, Berkeley, CA, USA, 2003. USENIX Association.

[10] M. D. Flouris. Clotho: Transparent Data Versioning at the Block I/O Level. In *In Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004*, pages 315–328, 2004.

[11] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[12] J. Katcher. PostMark: A New File System Benchmark. In *Technical report TR-3022*, Sunnyvale, CA, 1997. Network Appliance Inc.

[13] M. L. Kazar. Synchronization and Caching Issues in the Andrew File System. In *USENIX Winter Conference Proceedings*, pages 27–36, 1988.

[14] G. Laden, P. Ta-Shma, E. Yaffe, M. Factor, and S. Fienblit. Architectures for Controller-Based CDP. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 21–36, Berkeley, CA, USA, 2007.

[15] S. Lin, M. Lu, and T. cker Chiueh. An Incremental File System Consistency Checker for Block-Level CDP Systems. In *SRDS'2008: 27th International Symposium on Reliable Distributed Systems*, pages 132–140, 2008.

[16] M. Lu, S. Lin, and T. cker Chiueh. Efficient Logging and Replication Techniques for Comprehensive Data Protection. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 171–184, San Diego, CA, USA, 2007.

[17] R. McDougall and J. Mauro. *Solaris(TM) Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Inc., 2006.

[18] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[19] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, Berkeley, CA, USA, 2004.

[20] Z. Peterson and R. Burns. Ext3cow: a Time-Shifting File System for Regulatory Compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[21] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The Use of Name Spaces in Plan 9. *SIGOPS Operating Systems Review*, 27(2):72–76, 1993.

[22] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 7, Berkeley, CA, USA, 2002.

[23] 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*, pages 110–123, New York, NY, USA, 1999.

[24] C. A. N. Soules and G. R. Ganger. Metadata Efficiency in Versioning File Systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, Berkeley, CA, USA, 2003.

[25] 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 *OSDI'2000: Proceedings of the 4th Symposium on Operating System Design and Implementation*, pages 165–180, 2000.

[26] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable Information Storage Systems. *Computer*, 33(8):61–68, 2000.

[27] E. Zadok and J. Nieh. FIST: a Language for Stackable File Systems. *SIGOPS Operating Systems Review*, 34(2):38, 2000.

[28] N. Zhu, J. Chen, T. cker Chiueh, and D. Ellard. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):392–393, 2005.

[29] N. Zhu and T. cker Chiueh. Portable and Efficient Continuous Data Protection for Network File Servers. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 687–697, Washington, DC, USA, 2007.