

An Incremental File System Consistency Checker for Block-Level CDP Systems

Maohua Lu* Tzi-cker Chiueh* Shibiao Lin†

* *Department of Computer Science, Stony Brook University, {mlu,chiueh}@cs.sunysb.edu*
 † *Google Inc., shibiao@gmail.com*

Abstract

A block-level continuous data protection (CDP) system logs every disk block update from an application server (e.g., a file or DBMS server) to a storage system so that any disk updates within a time window are undoable, and thus is able to provide a more flexible and efficient data protection service than conventional periodic data backup systems. Unfortunately, no existing block-level CDP systems can support arbitrary point-in-time snapshots that are guaranteed to be consistent with respect to the metadata of the application server. This deficiency seriously limits the flexibility in recovery point objective (RPO) of block-level CDP systems from the standpoint of the application servers whose data they protect. This paper describes an incremental file system check mechanism (iFSCK) that is designed to address this deficiency for file servers, and exploits file system-specific knowledge to quickly fix an arbitrary point-in-time block-level snapshot so that it is consistent with respect to file system metadata. Performance measurements taken from a fully operational iFSCK prototype show that iFSCK can turn a 10GB point-in-time block-level snapshot to be file-system consistent in less than 1 second, and takes less than 25% of the time required by the Fscck utility for vanilla ext3 under relaxed metadata consistency requirements.

1. Introduction

Conventional data backup systems take a snapshot of the file/storage system periodically, so that the file/storage system can be restored 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, hence its associated recovery time cannot be bounded because additional manual repair may be required after a programmatic roll-back. Continuous data protection (CDP) greatly improves the RTO and RPO of a data backup solution by keeping the before image of every update operation against a file/storage system for a period of time. Because every update is undoable, CDP supports arbitrary point-in-time roll-back.

A block-level CDP system [1], [2], [3], [4], [5] applies CDP at the disk access interface, i.e., every disk write operation from an application server (e.g. a file or DBMS server) to its backend storage system within a *data protection window* (a data protection window is the time period within which every update is undoable, and is typically on the order of a week or a month) is logged and thus undoable. A key advantage of block-level CDP is that it can protect the data of arbitrary application servers without any modification to them.

Although existing block-level CDP systems can roll back the protected storage's image to arbitrary points in time in the past, none of them is able to guarantee that these images are consistent with respect to the meta-

data of the application servers whose data they protect. 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. As a result, even though block-level CDP supports flexible RTO for block-level images, it does not support flexible RTO for file-level images when the application server is a file server. This is a serious limitation of existing block-level CDP systems because it substantially decreases their practical utility and appeal.

Different application servers have different types of metadata, typically transparent to block-level CDP systems. To support arbitrary point-in-time metadata-consistent snapshots, exploiting application server-specific knowledge is inevitable. When the application server is a file server, one could apply a standard file system checker to block-level snapshots returned by a block-level CDP systems and transform them into a file system-consistent state. However, such checkers are too slow for some legacy file systems. This paper describes the design, implementation and evaluation of an *incremental* file system consistency checker called *iFSCK* that leverages file system-specific knowledge to convert a point-in-time snapshot into one that is guaranteed to be consistent with respect to file system metadata, i.e., file system-consistent.

2. Consistent block-level snapshotting

2.1. Access to block-level snapshots

A typical set-up that uses a block-level CDP system consists of an application server such as an NFS file server, an iSCSI storage server, and a block-level CDP server that logs every iSCSI write from the application server to the iSCSI storage server. If the application server needs to roll its disk image, which is backed by the iSCSI storage server, back to an earlier point in time, it contacts the CDP server for the corresponding point-in-time disk image snapshot. This subsection describes how an end user accesses a point-in-time snapshot under NFS. Similar snapshot access mechanisms can be developed for other network file access protocols such as CIFS.

In a typical NFS environment, the end user accesses files from an NFS client, which uses the NFS protocol to communicate with an NFS server, whose data we assume is protected by a block-level CDP server. To access a specific directory of a particular point-in-time snapshot of her file system, the user specifies the directory's pathname (P) and the target timestamp (T) in a request, which is sent to a dedicated daemon running on the NFS server. Upon receiving such a request, the daemon requests the associated block-level CDP server to create a disk image snapshot at T as an iSCSI target, instructs the NFS server to bind this iSCSI target to a local iSCSI initiator device, mounts a local directory on the iSCSI initiator device, and exports this local directory. Finally, another daemon on the NFS client mounts the target directory P within the NFS server's

exported directory to a local directory it creates. The client-side daemon communicates with the server-side daemon through a proprietary control protocol. The above snapshot access procedure interoperates with the standard NFS/iSCSI protocol with no modification.

2.2. Guaranteeing file system metadata consistency

Due to file system caching, file system-level updates do not immediately trigger block-level operations. Therefore, when a user asks for a point-in-time snapshot of a storage volume at time T , the returned snapshot may not capture all file system-level updates that take place before T , and more importantly it may not be even file system-consistent.

In addition to user data blocks, a file system also includes a set of metadata for managing its disk storage space and the relations and attributes of its files. When a user application modifies a file system object, the modify operation may trigger multiple updates to the file system's metadata. In theory, a file system update operation and the file system metadata updates it triggers should be performed *atomically*, as if they are batched into a transaction, so that the file system state is always consistent. However, for performance reasons, existing legacy file systems, e.g., the *ext2* file system under Linux, choose not to implement these updates as transactions. Instead, they resort to periodic buffer flushing to amortize the disk I/O cost of making file system metadata persistent. Under such file systems, how to quickly convert a point-in-time storage volume snapshot into its corresponding file system-consistent snapshot is a technical challenge for block-level CDP systems.

Given a disk image snapshot for a timestamp T , *iFSCK* is designed to identify all disk-level updates after T that correspond to file system-level update operations before and at T , and replay them against the snapshot to ensure that all file system-level updates before T are completed successfully. The key implementation challenge of this approach is how to accurately correlate block-level disk updates with their associated file system-level updates. *iFSCK* solves this problem by assuming that it knows the disk locations of file system metadata and their internal structures. Given a timestamp T and its corresponding disk snapshot V , *iFSCK* transforms V into a file system-consistent snapshot by using the following algorithm, assuming the host file system is an *ext2* file system:

- 1) *iFSCK* scans disk block updates that took place within a time window $[T - LB, T + UB]$, where $T - LB$ is the lower bound of the time window and $T + UB$ is the upper bound of the time window, and classifies them into the following types: *Block Bitmap* updates, *Inode Bitmap* updates, *Inode* updates and *Data Block* updates. By examining Inode updates in more detail, one can further sub-divide *Data Block* updates into *User Data Block* updates and *Directory Block* updates. Each of the file system metadata updates listed above modifies a distinct range of V 's block address space. From *Directory Block* updates, *iFSCK* identifies all file/directory creation, deletion and renaming operations within $[T - LB, T + UB]$.
- 2) *iFSCK* extracts from each disk block update individual metadata update operations triggered by file-level update operations. In concrete, for

each updated block, *iFSCK* retrieves its previous version, and performs a byte-by-byte comparison to determine which part of the block and which metadata entries (e.g. Inodes or bitmap entries) in that block are modified. Therefore, even if multiple file system-level updates result in a single block update, *iFSCK* can correctly identify each of them. In addition, *iFSCK* can also identify modifications to indirect blocks, which are no different than normal data blocks, because whenever *iFSCK* recognizes an Inode update, it follows the Inode to track down its indirect, doubly indirect and triply indirect blocks, and checks if they appear in the list of updated blocks within $[T - LB, T + UB]$.

- 3) *iFSCK* partitions the metadata update operations within $[T - LB, T + UB]$ to groups, each of which corresponds to a file-level update operation, according to a pre-computed table (shown in Table I) that lists the set of metadata updates for file deletion, file renaming, file truncation, file write, and file append. For example, when a new file is created, a new Inode is allocated to the file (Inode Bitmap update), this Inode is properly initialized (Inode Table update), some data blocks are allocated to the file (Block Bitmap update) and modified (Data Block update), the directory holding this new file is modified (parent Dentry update), and so is the directory's Inode (Inode Table update). *iFSCK* detects file truncation by examining the file length field in Inodes.
- 4) For every group that has at least one metadata update operation occurring between $T - LB$ and T , *iFSCK* includes into the *redo* list all the group's constituent metadata update operations that appear after T , replays the final *redo* list to the snapshot at T , and eventually produces a file system-consistent snapshot that is after and closest to T .

In Step (3), we make the assumption that a file system update operation that logically starts before T completes all its disk-level updates before $T + UB$. Because *pdflush* wakes up every 5 seconds, we assume the disk updates associated with a file system update operation span at most 5 seconds. Therefore, for a file system update operation that occurs exactly at time T , the latest disk update operations associated with it must occur before $T + 5$, and these disk updates must be flushed to disk before $T + 35$. However, because the *pdflush*'s wake-up timing may be mis-aligned, in the worst case, they must be flushed before $T + 40$. Therefore, in general UB is set to be $T_{wake-up} + T_{flush} + T_{span}$, where $T_{wake-up}$ corresponds to the periodic wake-up interval of *pdflush* (5 seconds in Linux), T_{flush} to the flushing threshold (30 seconds in Linux) and T_{span} to the time span of a file system update's disk-level update operations (assumed to be 5 seconds). On the other hand, LB has to be large enough so that for every file system update some of whose disk-level updates occur before $T - LB$, there is at least one of its disk-level updates takes place in $[T - LB, T]$. This prevents the anomaly that even though some disk-level updates associated with a file system update occur before T , *iFSCK* mistakes it for one whose first disk-level update occurs after T . As in UB , LB is also set to $T_{wake-up} + T_{flush} + T_{span}$.

Figure 1 illustrates how *iFSCK* extracts metadata *redo* list from the metadata updates within a period. R

File Operation	Related Updates
Creation	Inode Bitmap update, Inode Table update(new Inode and its parent Inode), parent Dentry update, Group Descriptor update(free Inode count), Superblock update(free Inode count)
Deletion	Inode Bitmap update, Inode Table update(deleted Inode and its parent Inode), Parent Dentry update, Block Bitmap update, Group Descriptor update(free Inode count, free block count), Superblock update(free Inode count, free block count)
Renaming	Inode Table update(parent Inode), parent Dentry update
Truncation	Inode Table update, Data Bitmap update, Data Block update, Group Descriptor update(free block count), Superblock update (free block count)
File Write	Inode Table update, Data Block update
File Append	Inode Table update, Data Bitmap update, Data Block update, Group Descriptor update(free block count), Superblock update(free block count)

TABLE I. File operations and the corresponding related metadata updates.

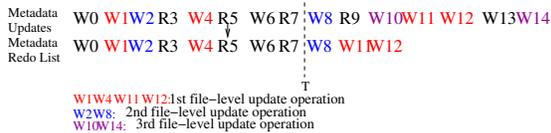


Fig. 1. An example showing how *iFSCK* extracts a metadata redo list from the metadata updates within a time period. T is the target timestamp. W_i is block-level operations triggered by file-level updates.

and W stand for metadata read and metadata write, respectively. *iFSCK* includes file system metadata update operations that are part of file-level update operations and occur before the target time stamp T . In this case, the first and second file-level update operations occur before T and therefore should be redone, whereas the third file-level update operation occurs after T and therefore should be undone.

iFSCK supports three file system consistency levels, each incurring a different performance overhead. The strongest consistency level (level=1) guarantees consistency of all file system metadata, including Block bitmap, Inode bitmap, directory contents, etc. This consistency level is also known as crash consistency as used in standard file system checkers, and is useful in creating a read/write snapshot. The second strongest consistency level (level=2) guarantees consistency only for a selected subset of file system metadata (e.g. excluding allocation bitmap information) and directory contents. This level is useful for creating a read-only snapshot. The weakest consistent level (level=3) provides the same consistency guarantee as the second strongest level except that its scope is restricted to a particular directory rather than the entire file system. For the weakest consistency level, *iFSCK* iterates through each directory entry in the target directory, making sure that the Inode number corresponding to each directory entry refers to a valid in-use Inode, i.e. the Inode is within a valid range, is allocated and the Inode link count is not zero, etc. The performance cost of weaker consistency levels is lower because disk updates associated with certain file system metadata updates (e.g. Block and Inode bitmap) can be ignored.

Although the above algorithm is designed for ext2 file systems, its underlying principle can be easily applied to other Unix file systems (e.g. Solaris UFS

that have similar file system structure to ext2's. For journaling file systems such as ext3 and NTFS, the journal already explicitly contains the file system metadata updates and their grouping information, so most of the analysis in *iFSCK* is no longer necessary. For example, ext3's file system consistency check tool first applies its metadata journal, and then continues with a ext2-style full file system check if the file system's superblock indicates that further checking is required. We have ported *iFSCK* to ext3 so as to derive the redo list directly from its journal rather than from scanning disk updates within $[T - LB, T + UB]$. In the case that an ext3 file system's journal is corrupted, *iFSCK* invokes its own incremental checker rather than ext2's file system checker.

3. Performance evaluation

In this section, we evaluate the effectiveness and performance of *iFSCK*'s incremental file system check. For *iFSCK*, we use the number of blocks in a point-in-time snapshot that need to be mended to make it file system-consistent as the evaluation metric for its efficiency. In addition to performance overhead, we also verify the correctness of *iFSCK* by comparing their results with the ground truth.

The testbed consists of an NFS client node, an NFS server node, and a file update logging node based on a block-level CDP implementation called Mariner [5], all of which are connected by a Netgear GS508T 8-port Gigabit Ethernet switch. All these nodes are a Dell PowerEdge 600SC machine with an Intel 2.4 GHz CPU, 768 MB memory, a 400 MHz front-side bus, an embedded Gigabit Ethernet Card, and up to five ATA/IDE hard disks, each of which is a 80-GB IBM Deskstar DTLA-307030 disk. The operating system is Fedora Core 3 with Linux kernel 2.6.11. The file system is an ext2 file system unless specified otherwise. We ran the following four sets of workloads to create historical images on the CDP node, and use these images to evaluate *iFSCK*. All experiment runs are 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. So an instance of the synthetic workload is characterized by four parameters: N , D , C and K .

- **Postmark:** Postmark [6] is a file system benchmark that emulates a heavy small file workload. The benchmark creates a specified number of files/sub-directories, performs various file update operations on them and eventually deletes all of them. The read/write ratio is always set to 1, but the number of transactions and other parameters are different from run to run.
- **Lair trace:** The Lair trace is an NFS trace collected from the EECS NFS server of Harvard University over two months [7]. The EECS workload is dominated by metadata updates with a read/write ratio of less than 1. The EECS trace grows by 2GB every day. We use the Trace-Based file system Benchmarking Tool (TBBT) [8] to replay this trace against the NFS server on the testbed.
- **SPECsfs:** SPECsfs is a general-purpose benchmark for NFS servers [9]. SPECsfs bypasses the NFS client and accesses the NFS server directly. SPECsfs has a default NFS request mix, where 12% of the requests update the file system and the remaining requests are read-only. We ran SPECsfs with 1 server process, 1 NFS client and set the operation rate to 500 to age the file system image.

3.1. Correctness evaluation of *iFSCK*

We evaluate the correctness of *iFSCK* by comparing the restored result from *iFSCK*(level=1) and from ext3-FSCK for a set of snapshot images with an ext3 file system. We ran the Postmark workload for 1,426 seconds with the following parameters: 1,000 files, 1,000 subdirectories, 1,000 transactions, and the inter-transaction interval set to 1 second. For the populated file system, we ended up setting up 650 snapshots to discover all file/directory versions during the run, 242 snapshot images are not file system-consistent and need to be "fixed". We invoked *iFSCK* (level = 1) against these 242 snapshot images to restore them to a consistent state. After that, we ran the vanilla ext3-FSCK against *iFSCK*'s restored images to determine if there is any residual inconsistency. None of these restored images require any additional fixes from ext3-FSCK. Moreover, all the files/directories within those images could be correctly read. This experiment demonstrates that *iFSCK*(level=1) indeed achieves the same file system consistency level as that of standard file system checkers.

3.2. Synthetic workload experiment

We evaluate both ext2 and ext3 file systems for the synthetic workload. Figure 2(a) shows the elapsed time of *iFSCK* when operating under 3 consistency levels as the number of disk block updates within $[T - LB, T + UB]$ is varied from 0 to 2500, where T is the user-specified target timestamp and both LB and UB are 40 seconds. As expected, the stronger the consistency

level is, the more time-consuming the corresponding *iFSCK* version is.

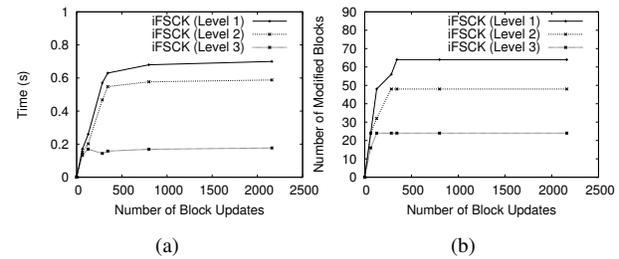


Fig. 2. (a) Elapsed time comparison of *iFSCK* operating under three different consistency levels when the number of disk block updates in the CDP node's log between $[T - LB, T + UB]$ is varied, where T is the user-specified target timestamp and both LB and UB are set to 40 seconds. (b) Number of file system metadata blocks *iFSCK* modifies to support different file system consistency levels when the number of disk block updates in the CDP node's log between $[T - LB, T + UB]$ is varied.

iFSCK(level=3) takes about 150 msec to check a point-in-time snapshot regardless of the number of disk block updates within $[T - LB, T + UB]$, because it only focuses on a particular directory and the number of disk block updates related to the directory remains virtually independent of the file-level update rate of the synthetic workload. The synthetic workload always creates the same number of subdirectories inside a parent directory. On the other hand, the elapsed time of both *iFSCK*(level=1) and *iFSCK*(level=2) increases proportionally to the number of block updates within $[T - LB, T + UB]$ when it is smaller than 300. As the number of disk block updates grows beyond 300, their elapsed times level off, because the same block gets updated multiple times and for each block *iFSCK* only needs to examine the latest update before T and the oldest update after T . The performance difference between *iFSCK*(level=2) and *iFSCK*(level=3) originates from their different scopes of consistency maintenance, a directory vs. the entire file system. On the other hand, the performance difference between *iFSCK*(level=1) and *iFSCK*(level=2) is attributed to a difference in the number of file system metadata checks they perform, e.g. *iFSCK*(level=1) checks the consistency of Inode and Block bitmaps while *iFSCK*(level=2) does not.

Figure 2(b) shows the number of file system metadata blocks that *iFSCK* modifies to support different file system consistency levels as the number of disk block updates within $[T - LB, T + UB]$ is varied from 0 to 2500, and correlates very well with Figure 2(a). This list of metadata block updates correspond to the redo list described in Section 2.2.

Because *iFSCK* only needs to focus on a small number of file system metadata block updates around the snapshot timestamp, the number of disk reads and writes it incurs is much smaller than a vanilla file system checker. Figure 3(a) shows the elapsed time comparison between *iFSCK*(level=1) and ext2's file

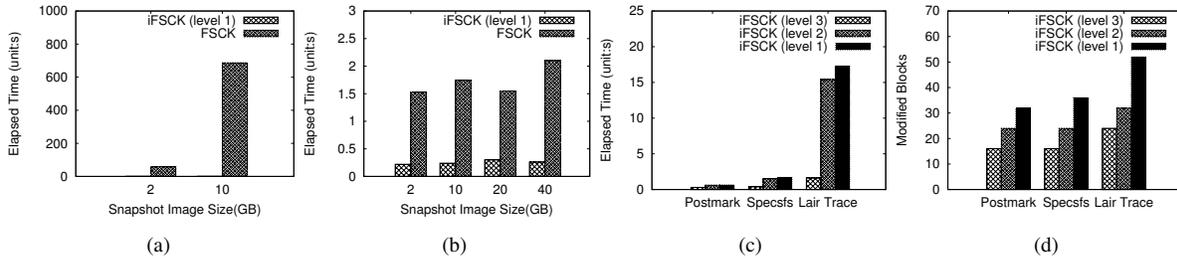


Fig. 3. (a) Elapsed time comparison between *iFSCk* and vanilla FSCk for an ext2 file system with different snapshot image sizes. When the snapshot image size is 10GB, *iFSCk*(level=1) is about three orders of magnitude faster than vanilla FSCk. (b) Elapsed time comparison between *iFSCk* and vanilla FSCk for an ext3 file system with different snapshot image size. (c) Elapsed time of *iFSCk* when operating at different file system consistency levels for the Postmark, SPECsfs and the Lair trace. (d) Number of file system metadata blocks *iFSCk* needs to modify to support different file system consistency levels for the Postmark, SPECsfs and the Lair trace.

system checker for two snapshot image sizes with size 2GB and 10GB. The elapsed time of *iFSCk*(level=1) remains virtually the same when the snapshot image size is increased from 2GB to 10GB, because the number of metadata block updates around the snapshot timestamp is not affected by the snapshot image size. In contrast, the elapsed time of ext2's file system checker grows proportionally with the snapshot image size because it scans the entire image. When the snapshot image size is 10GB, *iFSCk*(level=1) takes only 0.61 second, which is more than 100 times faster than that of ext2's file system checker.

Figure 3(b) shows the elapsed time comparison between *iFSCk*(level=1) and ext3's file system checker for snapshot images of different sizes. By leveraging the metadata journal, which is similar in functionality to *iFSCk*'s redo list, ext3's file system checker can complete a file system check transaction using roughly the same amount of time regardless of the snapshot image size, as is the case of *iFSCk*. In all cases, *iFSCk* still outperforms ext3's native file system checker, because the latter checks more global metadata than *iFSCk*(e.g., total Inode count, total block count).

3.3. Standard benchmark experiments

Figure 3(c) shows the elapsed time of *iFSCk* when operating under different file system consistency levels for the Postmark and SPECsfs benchmarks and the Lair trace. *iFSCk* takes less time under the Postmark benchmark than under the SPECsfs benchmark because Postmark's disk write pattern is sparser than SPECsfs's. Unlike Postmark and SPECsfs, disk writes in the Lair trace are quite bursty in some short periods and become very sparse for the rest of the trace. Therefore, for snapshots created in the sparse-write periods, *iFSCk* returns almost immediately (less than 10 msec) because there are very few writes (in most cases, it is 0) in $[T - LB, T + UB]$. However, for snapshots created in the bursty write periods, *iFSCk*(level=1) takes more than 17 seconds to complete, whereas vanilla FSCk takes more than an hour to do the same. In this test, *iFSCk* needs to examine 35,000 disk block updates, which appear in the 80-second interval of $[T - LB, T + UB]$, and 8878 directories. Figure 3(d) shows the number of file system metadata blocks *iFSCk* needs to modify under these workloads. For most runs, *iFSCk* needs to modify fewer than 56 blocks to bring a snapshot image to the strongest file system consistency

level. The numbers of blocks modified under these three workloads are roughly comparable suggests most of the performance overhead of *iFSCk* under the Lair trace comes from extracting a small number of file system metadata updates from a large number of disk block updates in $[T - LB, T + UB]$.

4. Related work

4.1. Continuous data protection

CDP (*Continuous Data Protection*) 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. Letting alone the implementation complexity, the performance overhead of file-level CDP is significant compared with file systems that do not support file versioning [10], [11]. Moreover, in-kernel update logging makes file-level CDP less portable.

Network-level CDP logs updates at the NFS or CIFS level. UCDP (User-level Continuous Data Protection) [12] 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.

Block-level is a natural place to implement CDP because of its simple interface: block read and write. Self-securing storage(S4) [13], [14] is a network-attached object store that maintains previous versions of storage objects in a way transparent to both the operating system and applications. Compared with S4, *Mariner* [5] also implements CDP at the block level but without modifying the block level interface.

Venti [15] uses a cryptographic hash of a disk block's contents as the disk block's unique ID for reads and writes, and supports a write-once policy that never overwrites or deletes data. Future block-level CDP systems can employ this technique to reduce the storage space requirement of their block update logs.

4.2. FSCk

Val Henson [16] proposed a file system-wide dirty bit to indicate whether the file system is being actively modified when the system crashes. When a file system

crashes with the dirty bit not set, FSCK knows that it does not have to do a full FSCK when the system reboots. In addition, they proposed a technique called linked writes to identify a list of dirty Inodes to limit the scope of FSCK to checking only those dirty Inodes. Linked writes can also be viewed as a form of journaling where the journal entries are scattered across the disk and linked by the on disk dirty Inode list. It needs to link the Inode to the on disk dirty Inode list before the actual operation on this Inode takes place. Similarly, *iFSCK* scans updated blocks within a time window to limit the scope of FSCK. The difference is that *iFSCK* leverages file system-specific knowledge to identify block updates that modify file system metadata and therefore works without any file system modification.

Other file systems, for example the ext3 file system under Linux, use a journaling architecture to give a transaction semantics to file system metadata updates through redo logging; it is relatively straightforward to identify disk updates that modify file system metadata because each of them appears in a separate log. In addition, the ext3 file system readily tags disk updates associated with their corresponding file system update operations. Therefore, its crash recovery code does not need to parse the disk updates to correlate them with their associated file system update operations.

CIMStore [17] exposes the history of all data for search or browsing. CIMStore marks the state of the system as being “quiescent” to speed up the recovery of the state of a storage application (e.g. a file system) and provides consistent point-in-time information. These “quiescent” points are located either by understanding high level data structures such as the “unmounted cleanly” flag of the superblock of a file system, or explicit notification from a client-side application monitoring storage application. This “quiescent” point detection is currently done manually in CIMStore. In contrast, *iFSCK* needs no “quiescent” points and transparently restore the storage volume to a consistent state.

J. Kent Peacock et al. [18] developed a fast consistency checking technique for Solaris file system. The fast FSCK provided with Netra NFS keeps track of the active portions of the file system and limits the scope of the file-system check to only that “working set.” They need to modify the file system structures to add the state information and perform logging of certain transactions such as directory update. In contrast, *iFSCK* combines the write log from the CDP node and the file system-specific knowledge to infer the working set without modifying the file system itself.

5. Conclusion

Continuous data protection (CDP) enables undo of updates within a data protection window. Block-level CDP [1], [2], [4] in particular has emerged as a key building block of the data protection solution used in modern enterprises, and has the potential to displace existing periodic data backup systems because of its flexible RTO and RPO and its ability to drastically simplify storage administration in large organizations. However, owing to the lack of application server-specific knowledge, existing block-level CDP systems share one common deficiency: The RTO flexibility in terms of snapshots that are consistent with respect to application server metadata is much less than that in terms of snapshots without such consistency requirements. When the application server whose data is

protected by a block-level CDP system is a file server, this deficiency means that the point-in-time snapshots returned by the CDP system are not necessarily file system-consistent, and thus in many cases not directly usable.

To overcome this limitation, this paper describes the design, implementation and evaluation of *iFSCK*, an incremental file system consistency checker that exploits file system-specific knowledge to efficiently turn point-in-time block-level snapshots into file system-consistent. Although the design of *iFSCK* is targeted at legacy file systems such as ext2, it can also effectively leverage any metadata journal if the underlying file system is a journaling file system, such as ext3 or NTFS. Overall the performance overhead of *iFSCK* is pretty modest: It takes less than 1 second to turn a 10GB point-in-time block-level snapshot into file system-consistent.

References

- [1] “InfiniView,” <http://www.mendocinosoft.com/technology.htm>, 2007, Mendocino Software Inc.
- [2] “RecoveryPoint,” http://software.emc.com/products/software_az/recoverpoint.htm, 2007, EMC Inc.
- [3] “Veritas NetBackup,” http://www.symantec.com/business/products/overview.jsp?pcid=2244&pvid=2_1, 2007, Symantec Inc.
- [4] “InMage,” <http://www.inmage.net/features-and-benefits.html>, 2007, InMage Inc.
- [5] M. Lu, S. Lin, and T. Chiueh, “Efficient Logging and Replication Techniques for Comprehensive Data Protection,” *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pp. 171–184, 2007.
- [6] J. Katcher, “PostMark: A New File System Benchmark,” *Technical report TR-3022*, 1997, Network Appliance Inc.
- [7] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, “Passive NFS Tracing of Email and Research Workloads,” *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 203–216, 2003.
- [8] N. Zhu and J. Chen and T. Chiueh and D. Ellard, “TBBT: Scalable and Accurate Trace Replay for File Server Evaluation,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 392–393, 2005.
- [9] “SPEC SFS (System File Server) Benchmark,” <http://www.spec.org/osg/sfs97/>, 1997, Standard Performance Benchmark Corporation.
- [10] C. Soules, G. Goodson, J. Strunk, and G. Ganger, “Metadata Efficiency in Versioning File Systems,” *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 43–58, 2003.
- [11] Z. Peterson and R. Burns, “Ext3cow: a Time-Shifting File System for Regulatory Compliance,” *ACM Transactions on Storage*, vol. 1, no. 2, pp. 190–212, 2005.
- [12] N. Zhu and T. Chiueh, “Portable and Efficient Continuous Data Protection for Network File Servers,” *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 687–697, 2007.
- [13] J. Wylie, M. Bigrigg, and J. Strunk, “Survivable Information Storage Systems,” *Computer*, vol. 33, no. 8, pp. 61–68, 2000.
- [14] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger, “Self-Securing Storage: Protecting Data in Compromised Systems,” *Proceedings of the 4th Symposium on Operating System Design and Implementation*, pp. 165–180, 2000.
- [15] S. Quinlan and S. Dorward, “Venti: A New Approach to Archival Data Storage,” *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pp. 89–101, 2002.
- [16] V. Henson, Z. Brown, T. Y. Tso, and A. Ven, “Reducing FSCK Time for Ext2 File Systems,” *Ottawa Linux Symposium*, 2006.
- [17] III Charles B. Morrey, *Cimstore: content-aware integrity maintaining storage*, Ph.D. thesis, Boulder, CO, USA, 2006, Adviser-Dirk Grunwald.
- [18] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal, “Fast Consistency Checking for the Solaris File System,” *Proceedings of the Annual Technical Conference on USENIX Annual Technical Conference*, pp. 7–21, 1998.