Clotho: Transparent Data Versioning at the Block I/O Level

Michail D. Flouris
Department of Computer Science,
10 King's College Road,
University of Toronto,
Toronto, Ontario M5S 3G4, Canada
Email: flouris@cs.toronto.edu

Angelos Bilas
Institute of Computer Science
Foundation for Research and Technology - Hellas
Science and Technology Park of Crete
P.O.Box 1385
GR 711 10 Heraklion, Crete, Greece
Email: bilas@ics.forth.gr

Abstract

Recently storage management has emerged as one of the main problems in building cost effective storage infrastructures. One of the issues that contribute to management complexity of storage systems is maintaining previous versions of data. Up till now such functionality has been implemented by high-level applications or at the file system level. However, many modern systems aim at higher scalability and do not employ such management entities as filesystems.

In this paper we propose pushing the versioning functionality closer to the disk by taking advantage of modern, block-level storage devices. We present *Clotho*[1], a storage block abstraction layer that allows transparent and automatic data versioning at the block level. *Clotho* provides a set of mechanisms that can be used to build flexible higher-level version management policies that range from keeping all data modifications to version capturing triggered by timers or other system events.

Overall, we find that our approach is promising in offloading significant management overhead and complexity from higher system layers to the disk itself and is a concrete step towards building self-managed storage devices. Our specific contributions are: (i) We implement *Clotho* as a new layer in the block I/O hierarchy in Linux and demonstrate that versioning can be performed at the block level in a transparent manner to all higher system layers and provide simple techniques for capturing consistent versions during system operation. (ii) We investigate the impact on I/O path performance overhead using both microbenchmarks as well as SPEC SFS V3.0 over two real filesystems, Ext2FS and ReiserFS. We find that this can be negligible compared to non-versioned block devices, when there is no need for version compaction. (iii) We examine techniques that reduce the memory and disk space required for metadata information.

I. Introduction

Storage is currently emerging as one of the major problems in building tomorrow's computing infrastructure. Future systems will provide tremendous storage, CPU processing, and network transfer capacity in a cost-efficient manner and they will be able to process and store ever increasing amounts of data. The cost of managing these large amounts of stored data becomes the dominant complexity and cost factor for building, using, and operating modern storage systems. Recent studies [9] show that storage expenditures represent more than 50% of the typical server purchase price for applications such as OLTP (On-Line Transaction Processing) or ERP (Enterprise Resource

[1] Clotho, one of the Fates in ancient Greek mythology, spins the thread of life for every mortal.

Planning) and these percentages will keep growing. Furthermore, the cost of storage administration is estimated at several times the purchase price of the storage hardware [11], [33], [1], [4], [35], [32], [6]. Thus, building self-managed storage devices that reduce management-related overheads and complexity is of paramount importance.

One of the most cumbersome management tasks that requires human intervention is creating, maintaining, and recovering previous versions of data for archival, durability, and other reasons. The problem is exacerbated as the capacity and scale of storage systems increases. Today, backup is the main mechanism used to serve these needs. However, traditional backup systems are limited in the functionality they provide. Moreover they usually incur high access and restore overheads on magnetic tapes, they impose a very coarse granularity in the allowable archival periods, usually at least one day, and they result in significant management overheads [27], [4]. Automatic versioning, in conjunction with increasing disk capacities, has been proposed [27], [4] as a method to address these issues. In particular, magnetic disks are becoming cheaper and larger and it is projected that disk storage will soon be as competitive as tape storage [8], [4]. With the advent of inexpensive high-capacity disks, we can perform continuous, real-time versioning and we can maintain online repositories of archived data.

Moreover, *online storage versioning* offers a new range of possibilities compared to simply recovering users' files that are available today only in expensive, high-end storage systems:

- **Recovery from user mistakes.** Accidentally deleted or modified data can be recovered by rolling-back to a saved version. Online versioning allows frequent storage snapshots that reduce the amount of data lost compared to traditional backup systems. Moreover, recovery can be performed by the users themselves without administrator intervention.
- **Recovery from system corruption.** Online storage versioning can help with malicious attacks on computer systems that may corrupt stored data, such as through viruses, trojan horses, or cracker intrusions. With online versioning, system administrators can quickly and efficiently identify corrupted data as well as recover to a previous, consistent system state [30], [28]. Ideally, a self-managed system could automatically recover to a safe state when intrusion or corruption is detected.
- **Historical analysis of data modifications.** This can also be seen as storage auditing in cases where it is necessary to understand how a piece of data has reached a certain state. This is a particularly useful concept, for example when law imposes certain restrictions on data archival.

Our goal in this paper is to provide online storage versioning capabilities in commodity storage systems, in a *transparent* and *cost-effective* manner. Storage versioning has been previously proposed and examined purely at the filesystem level [26], [24] or at the block level [13], [17] but being filesystem aware. These approaches to versioning were intended for large, centralized storage servers or appliances. We argue that to build self-managed storage systems, versioning functionality should be pushed to lower system layers, closer to the disk to offload higher system layers. This is made possible by underlying technologies that drive storage systems. Disk storage, network bandwidth, processor speed, and main memory are reaching speeds and capacities that make it possible to build cost-effective storage systems with significant processing capabilities [10], [12], [22], [8] that will be able to both store vast amounts of information [16], [12] and to provide advanced functionality.

Our approach of providing online storage versioning is to provide all related functionality at the block level. This

approach has a number of advantages compared to other approaches that try to provide the same features either at the application or the filesystem level. First, it provides a higher level of transparency and in particular is completely filesystem agnostic. For instance, we have used our versioned volumes with multiple, third party, filesystems without the need for any modifications. Data snapshots can be taken on demand and previous versions can be accessed online simultaneously with the current version. Second, it reduces complexity in higher layers of storage systems, namely the filesystem and storage management applications [14]. Third, it takes advantage of the increased processing capabilities and memory sizes of active storage nodes and offloads expensive host-processing overheads to the disk subsystem, thus, increasing the scalability of a storage archival system [14].

However, block-level versioning poses certain challenges as well: (i) Memory and disk space overhead: Because we only have access to blocks of information, depending on application data access patterns, there is increased danger for higher space overhead in storing previous versions of data and the related metadata. (ii) I/O path performance overhead: It is not clear at what cost versioning functionality can be provided at the block-level. (iii) Consistency of the versioned data when the versioned volume is used in conjunction with a filesystem. (iv) Versioning granularity: Since versioning occurs at a lower system layer, information about the content of data is not available, as is, for instance, the case when versioning is implemented in the filesystem or the application level. Thus, we only have access to full volumes as opposed to individual files.

We design *Clotho*, a system that provides versioning at the block-level and addresses all above issues, demonstrating that this can be done at minimal space and performance overheads. First, *Clotho* has low memory space overhead and uses a novel method to avoid copy-on-write costs when the versioning extent size is larger than the block size. Furthermore, *Clotho* employs off-line *differential compression* (or diffing) to reduce disk space overhead for archived versions. Second, using advanced disk management algorithms, *Clotho*'s operation is reduced in all cases to simply manipulating pointers in in-memory data structures. Thus, *Clotho*'s common-path overhead follows the rapidly increasing processor-memory curve and does not depend on the much lower disk speeds. Third, *Clotho* deals with version consistency by providing mechanisms that can be used by higher system layers to guarantee that either all data is consistent or to mark which data (files) are not. Finally, we believe that volumes are an appropriate granularity for versioning policies. Given the amounts of information that will need to be managed in the future, specifying volume-wide policies and placing files on volumes with the appropriate properties, will result in more efficient data management.

We implement *Clotho* as an additional layer (driver) in the I/O hierarchy of Linux. Our implementation approach allows *Clotho* the flexibility to be inserted in many different points in the block layer hierarchy in a single machine, a clustered I/O system or a SAN. *Clotho* works over simple block devices such as a standard disk driver or more advanced device drivers such as volume managers or hardware/software RAIDs. Furthermore, our implementation provides to higher layers the abstraction of a standard block device and thus, can be used by other disk drivers, volume/storage managers, object stores or filesystems.

We evaluate our implementation with both microbenchmarks as well as real filesystems and the SPEC SFS 3.0 suite over NFS. The main memory overhead of *Clotho* for metadata is about 500 Kbytes per GByte of disk space and can be further reduced by using larger extents. Moreover, the performance overhead of *Clotho* for I/O operations is

minimal, however, it may change the behavior of higher layers (including the filesystem), especially if they make implicit assumptions about the underlying block device, e.g. the location of disk blocks. In such cases, co-design of the two layers, or system tuning maybe necessary to not degrade system performance. Overall, we find that our approach is promising in offloading significant management overhead and complexity from higher system layers to the disk itself and is a concrete step towards building self-managed storage systems.

The rest of this paper is organized as follows. Section II presents our design and discusses the related challenges in building block-level versioning systems. Section III presents our implementation. Section IV presents our experimental evaluation and results. Section V discusses related work, while section VI presents limitations and future work. Finally, Section VII draws our conclusions.

II. System Design

The design of *Clotho* is driven by the following high-level goals and challenges:

- Flexibility and transparency.
- Low metadata footprint and low disk space overhead.
- Low-overhead version and common I/O path operations.
- Consistent online snapshots.

Next we discuss how we address each of these challenges separately.

A. Flexibility and Transparency

Clotho provides to higher system layers versioned volumes. These volumes look similar to ordinary physical disks that can, however, be customized, based on user-defined policies to keep previous versions of the data they store. Essentially, *Clotho* provides a set of mechanisms that allow the user to add time as a dimension in managing data by creating and manipulating volume versions. Every piece of data passing through *Clotho* is indexed based not only on its location on the block device, but also on the time the block was written. When a new version is created, a subsequent write to a block will create a new block preserving the previous version. Multiple writes to the same data block between versions result in overwriting the same block. Using *Clotho*, device versions can be captured either on demand or automatically at prespecified periods. The user can view and access all previous versions of the data online, as independent block devices along with the current version. The user can compact and/or delete previous volume versions. In this work we focus on the mechanisms *Clotho* provides and we only present simple policies we have implemented and tested ourselves. We expect that systems administrators will further define their own policies in the context of higher-level storage management tools.

Clotho provides a set of primitives (mechanisms) that higher-level policies can use for automatic version management:

- `CreateVersion()` provides a mechanism for capturing the lower-level block device's state into a version.
- `DeleteVersion()` explicitly removes a previously archived version and reclaims the corresponding volume space.
- `ListVersions()` shows all saved version of a specific block device.

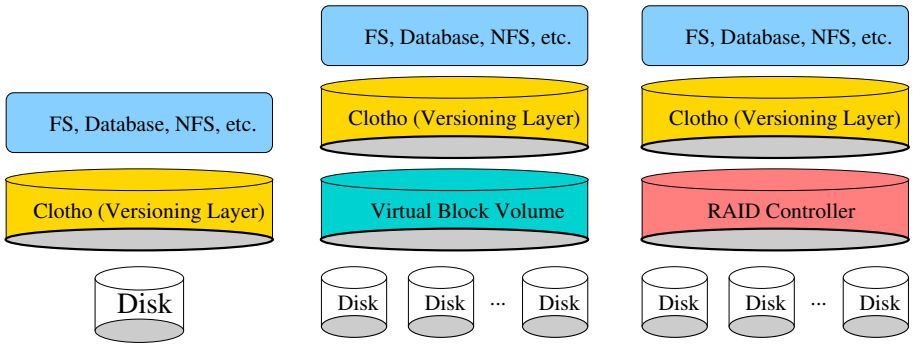

Fig. 1. *Clotho* in the block device hierarchy.

- `ViewVersion()` enables creating a virtual device that corresponds to a specific version of the volume and is accessible in read-only mode.
- `CompactVersion()` and `UncompactVersion()` provide the ability to compact and uncompact existing versions for reducing disk space overhead.

Versions of a volume have the following properties: Each version is identified by a unique *version number*, which is an integer counter starting from value 0 and increasing with each new version. Version numbers are associated with timestamps for presentation purposes. All blocks of the device that are accessible to higher layers during a period of time will be part of the version of the volume taken at that moment (if any) and will be identified by the same version number. Each of the archived versions exists solely in read-only state and will be presented to the higher levels of the block I/O hierarchy as a distinct, virtual, read-only block device. The latest version of a device is both readable and writable, exists through the entire lifetime of the *Clotho*'s operation and cannot be deleted.

One of the main challenges in *Clotho* is to provide all versioning mechanisms at the block level in a transparent and flexible manner. *Clotho* can be inserted arbitrarily in a system's layered block I/O hierarchy. This stackable driver concept has been employed to design other block-level I/O abstractions, such as software RAID systems or volume managers, in a clean and flexible manner [17]. The *input* (higher) layer can be any filesystem or other block-level abstraction or application, such as a RAID, volume manager, or another storage system. *Clotho* accepts block I/O requests (read, write, ioctl) from this layer. Similarly, the *output* (lower) layer can be any other block device or block-level abstraction. This design provides great flexibility in configuring a system's block device hierarchy. Figure 1 shows some possible configurations for *Clotho*. On the left part of Figure 1, *Clotho* operates on top of a physical disk device. In the middle, *Clotho* acts as a wrapper of a single virtual volume constructed by a volume manager, which abstracts multiple physical disks. In this configuration *Clotho* captures versions of the whole virtual volume. On the right side of Figure 1, *Clotho* is layered on top of a RAID controller which adds reliability to the system. The result is a storage volume that is both versioned and can tolerate disk failures.

Most higher level abstractions that are built on top of existing block devices assume a device of fixed size, with few rare exceptions such as resizable filesystems. However, the space taken by previous versions of data in *Clotho*, depends on the number and the amount of modified data between them. *Clotho* can provide both a fixed size block device abstraction to higher layers, as well as dynamically resizable block device abstractions, if the higher layers

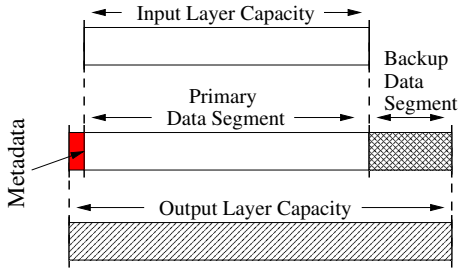

Fig. 2. Logical space segments in *Clotho*.

support it. At device initialization time *Clotho* reserves a configurable percentage of the available device space for keeping previous versions of the data. This essentially partitions (logically not physically) the capacity of the wrapped device into two logical segments as illustrated in Figure 2. The *Primary Data Segment* (PDS), which contains the data of the current (latest) version and the *Backup Data Segment* (BDS), which contains all the data of the archived versions. When BDS becomes full, *Clotho* simply returns an appropriate error code and the user has to reclaim parts of the BDS by deleting or compacting previous versions, or by moving them to some other device. These operations can also be performed automatically by a module that implements high-level data management policies. The latest version of the block device continues to be available and usable at all times. *Clotho* enforces this capacity segmentation by reporting as its total size to the input layer, only the size of the PDS. The space reserved for storing versions is hidden from the input layer and is accessed and managed only through the API provided by *Clotho*.

Finally, *Clotho*'s metadata need to be saved on the output device along with the actual data. Losing metadata used for indexing extents would render the data stored throughout the block I/O hierarchy unusable. This is similar to most block-level abstractions, such as volume managers, and software RAID devices. *Clotho* stores metadata to the output device periodically. The size of the metadata depends on the size of the encapsulated device and the extent size. In general, *Clotho*'s metadata are much less than the metadata of a typical filesystem and thus, saving them to stable storage is not an issue.

B. Reducing Metadata Footprint

The three main types of metadata in *Clotho* are the *Logical Extent Table* (LXT), the *Device Version List* (DVL), and the *Device Superblock* (DSB).

Logical Extent Table (LXT): *Clotho* presents to the input layer *logical* block numbers as opposed to the *physical* block numbers provided by the wrapped device. Note that these block numbers need not directly correspond to actual physical locations, if another block I/O abstraction, such as a volume manager (e.g. LVM [31]) is used as the output layer. *Clotho* uses the LXT to translate logical block numbers to physical block numbers.

Device Version List (DVL): *Clotho* maintains a list of all versions of the output device and makes them available to higher layers as separate block devices. For every existing version, it stores its version number, the virtual device it may be linked to, the version creation timestamp, and a number of flags.

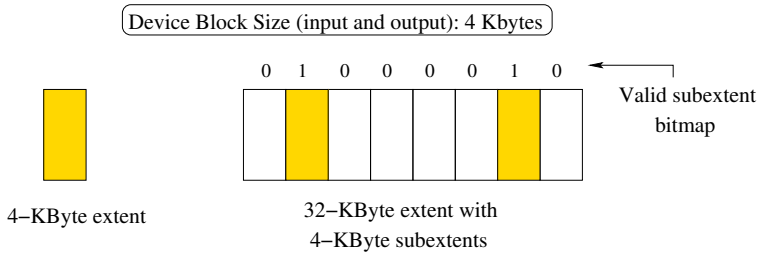

Fig. 3. Subextent addressing in large extents.

Device Superblock (DSB): The device superblock is a small table containing important attributes of the output versioned device. It stores information about the capacity of the input and output layer, the space partitioning, size of the extents, the output layer sector and block size, the current version counter, the total number of existing versions and several other usage counters.

The LXT is the most demanding type of metadata and is conceptually an array indexed by block numbers. The basic block size for most block devices varies between 512 Bytes (the size of a disk sector) and 8 KBytes. This results in large memory requirements. For instance, for 1 TByte of disk storage with 4-KByte blocks, the LXT has 256M entries. In the current version of *Clotho*, every LXT entry is 128-bits (16 bytes). These include 32 bits for block addressing and 32 bits for versions that allow for a practically unlimited number of versions. Thus, the LXT requires about 4 GBytes per TByte of disk storage. Note that a 32-bit address space, with 4 KByte blocks, can address 16 TBytes of storage.

To reduce the footprint of the LXT and at the same time increase the addressing range of LXT, we use *extents* as opposed to device blocks as our basic data unit. An extent is a set of consecutive (logical and physical) blocks. Extents can be thought as *Clotho*'s internal block size, which one can configure to arbitrary sizes, up to several hundred KBytes or a few MBytes. Similarly to physical and logical blocks, we denote extents as logical (input) extents or physical (output) extents. We have implemented and tested extent sizes ranging from 1 KByte to 64 KBytes. With 32-KByte extents and subextent addressing, we need only 500 MBytes of memory per TByte of storage. Moreover with a 32-KByte extent size we can address 128 TBytes of storage.

However, large extent sizes may result in significant performance overhead. When the extent size and the operating system block size for *Clotho* block devices are the same (e.g. 4KBytes), *Clotho* receives from the operating system the full extent for which it has to create a full version. When using extents larger than this maximum size, *Clotho* sees only a subset of the extent for which it needs to create a new version. Thus, it needs to copy the rest of the extent in the new version, even though only a small portion of it written by the higher system layers. This copy can significantly decrease performance in the common I/O path, especially for large extent sizes. However, large extents are desirable for reducing metadata footprint. Given that operating systems support I/O blocks of up to a maximum size (e.g. 4K in Linux), this may result in severe performance overheads.

To address this problem we use *subextent addressing*. Using a small (24-bit) bitmap in each LXT entry we need not copy the whole extent in a partial update. Instead we just translate the block write to a subextent of the same size and mark it in the subextent bitmap as valid, using just 1 bit. In a subsequent read operation we search for the valid

subextents in the LXT before translating the read operation. For a 32-Kbyte extent size, we need only 8 bits in the bitmap for 4-KByte subextents.

Another possible approach to reduce memory footprint is to store only a part of the metadata in RAM and perform swapping of active metadata from stable storage. However, this solution is not adequate for storage systems where large amounts of data need to be addressed. Moreover it is orthogonal to subextent addressing and can be combined with it.

C. Version Management Overhead

All version management operations can be performed at a negligible cost by manipulating in-memory data structures. Creating a new version in *Clotho* involves simply incrementing the current version counter and does not involve copying any data. When `CreateVersion()` is called, *Clotho* stalls all incoming I/O requests for the time required to flush all its outstanding writes to the output layer. When everything is synchronized on stable storage, *Clotho* increases the current version counter, appends a new entry to the device version list, and creates a new virtual block device that can be used to access the captured version of the output device, as explained later. Since each version is linked to exactly one virtual device, the (OS-specific) device number that sends the I/O request can be used to retrieve the I/O request's version.

The fact that device versioning is a low-overhead operation makes it possible to create flexible versioning policies. Versions can be created by external processes (user or admin) periodically or based on other system events. For instance, the user processes can specify that it requires a new version every 1 hour, or whenever all files to the device are closed or on every single write to the device. Some of the mechanisms to detect such events, e.g. if there are any open files on a device, may be (and currently are) implemented in *Clotho* but could also be provided by other system components.

In order to free backup disk space, *Clotho* provides a mechanism to delete volume versions. On a `DeleteVersion()` operation, *Clotho* traverses the primary LXT segment and for every entry that has a version number equal to the delete candidate, changes the version number to the next existing version number. It then traverses the backup LXT segment and frees the related physical extents. As with version creation, all operations for version deletion are performed in-memory adn can overlap with regular I/O. `DeleteVersion()` is provided to higher layer in order to implement *version cleaning policies*. Since storage space is finite, such policies are necessary in order to continue versioning without running out of backup storage. Finally, even if the backup data segment (BDS) is full, I/O to the primary data segment and the latest version of data can continue without interruption.

D. Common I/O Path Overhead

We consider the common path for *Clotho*, as the I/O path to read and write to the latest (current) version of the output block device, while versioning occurs frequently. Accesses to older versions are of less importance since they are not expected to occur as frequently as current version usage. Accordingly, we divide read and write access to volume versions in two categories, accesses to the current version and accesses to any previous version. The main technique to reduce common path overhead is to divide the LXT in two logical segments, corresponding to

the primary and backup data segments of the output device as illustrated in Figure 2. The primary segment of the LXT (mentioned as PLX in figures) has an equal number of logical extents as the input layer to allow a direct, 1-1 mapping between the logical extents and the physical extents of the current version on the output device. By using a direct, 1-1 mapping, *Clotho* can locate a physical extent of the *current version* of a data block with a *single lookup* in the primary LXT segment, when translating I/O requests to the current version of the versioned device. If the input device needs to access previous versions of a versioned output device, then multiple accesses to the LXT maybe required to locate the appropriate version of the requested extent.

To find the physical extent that holds the specific version of the requested block, *Clotho* first references the primary LXT segment entry to locate the current version of the requested extent (a single table access). Then it uses the linked list that represents the version history of the extent to locate the appropriate version of the requested block. Depending on the type of each I/O request and the state of the requested block, I/O requests can be categorized as follows:

Write requests can only be performed on the current version of a device, since older versions are read-only. Thus, *Clotho* can locate the LXT entry of a current version extent with a *single LXT access*. Write requests can be one of three kinds as shown in Figure 4:

a. Writes to new, unallocated blocks. In this case, *Clotho* calls its extent allocator module, which returns an available physical extent of the output device, it updates the corresponding entry in the LXT, and forwards the write operation to the output device. The *extent allocation policy* in our current implementation is a scan-type policy, starting from the beginning of the PDS to its end. Free extents are ignored until we reach the end of the device, when we rewind the allocation pointer and start allocating the free extents.
b. Writes to existing blocks that have been modified after the last snapshot was captured (i.e. their version number is equal to the current version number). In this case *Clotho* locates the corresponding entry in the primary LXT segment with a single lookup and translates the request's block address to the existing physical block number of the output device. Note that in this case the blocks are *updated in place*.
c. Writes to existing blocks that have not been modified since the last snapshot was captured (i.e. their version number is lower than the current version number). The data in the existing physical extent must not be over-written, but instead the new data should be written in a different location and a new version of the extent must be created. *Clotho* allocates a new LXT entry *in the backup segment* and *swaps* the old and new LXT entries so that the old one is moved to the backup LXT segment. The block address is then translated to the new physical extent address, and the request is forwarded to the output layer. This "swapping" of LXT entries maintains the 1-1 mapping of current version logical extents in the LXT which optimizes common-path references to a single LXT lookup.

This write translation algorithm allows for independent, fine grain versioning at the extent level. Every extent in the LXT is versioned according to its updates from the input level. Extents that are updated more often have more versions than extents written less frequently.

Read request translation is illustrated in Figure 5. First *Clotho* determines the desired version of the device by the virtual device name and number in the request (e.g. /dev/clt1-01 corresponds to version 1 and /dev/clt1-02 to version

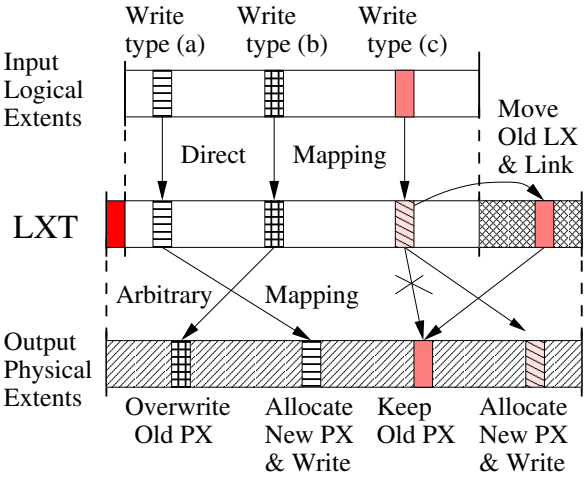

Fig. 4. Translation path for write requests.

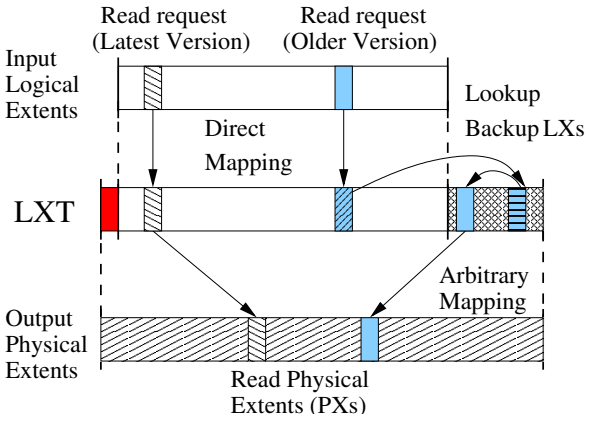

Fig. 5. Translation path for read requests.

2). Then, *Clotho* traverses the version list on the LXT for the specific extent or subextent and locates the appropriate physical block.

Finally, previous versions of a *Clotho* device appear as different virtual block devices. Higher layers, e.g. filesystems, can use these devices to access old versions of the data. If the device id of a read request is different from the normal input layer's device id, the read request refers to an extent belonging to a previous version. *Clotho* determines from the device id the version of the extent requested. Then, it traverses the version list associated with this extent to locate the backup LXT entry that holds the appropriate version of the logical extent. This translation process is illustrated in Figure 5.

E. Reducing Disk Space Requirements

Since *Clotho* operates at the block level, there is an induced overhead in the amount of space it needs to store data updates. If an application for instance, using a file modifies a few consecutive bytes in the file, *Clotho* will create a new version for the full block that contains the modified data. To reduce the space overhead in *Clotho* we provide a differential, content-based compaction mechanism, which we describe next.

Clotho provides the user with the ability to compact device versions and still be able to transparently access them online. The policy decision on when to compact a version is left to higher-layers in the system, similarly to all policy decisions in *Clotho*. We use a form of binary differential compression (or diffing) to only store the data that has been modified since the last version capture. When `CompactVersion()` is called, *Clotho* constructs a differential encoding (or delta) between the blocks that belong to a given version with corresponding blocks in its previous version. Although a lot of differential policies can be applied in this case, such as to compare the content of a specific version with its next version, or both the previous and the next version, at this stage we only explore diffing with the previous version. Furthermore, although versions can also be compressed independently of differential compression using algorithms such as Lempel-Ziv encoding [36] or Wheeler-Burrows encoding [2], this is beyond the scope of our work. We envision that such functionality can be provided by other layers in the I/O device stack.

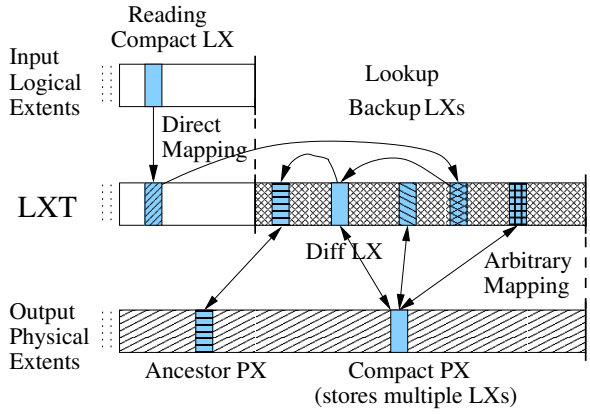

Fig. 6. Read translation for compact versions.

The differential encoding algorithm works as follows. When a compaction operation is triggered, the algorithm runs through the backup data segment of the LXT and locates the extents that belong to the version under consideration. If an extent does not have a previous version, it is not compacted. For each of the extents to be compacted the algorithm locates its previous version, diffs the two extents, and writes the diffs to a physical extent on the output device. If the diff size is greater than a threshold and diffing is not very effective, then *Clotho* discards this pair and proceeds with the next extent of the version to be compacted. In other words, *Clotho*'s differential compression algorithm works selectively on the physical extents, compacting only the extents that can be reduced in size. The rest are left in their normal format to avoid performance penalties necessary for their reconstruction.

Since the compacted form of an extent requires less size than a whole physical extent, the algorithm stores multiple diffs in the same physical extent, effectively, imposing a different structure on the output block device. Furthermore, for compacted versions, multiple entries in the LXT may point to the same physical extent. The related entries in the LXT and the ancestor extent are kept in *Clotho*'s metadata. Physical extents that are freed after compaction are reused for storage. Figure 6 shows sample LXT mappings for a compacted version of the output layer.

Data on a compacted version can be accessed transparently online as data on uncompacted volumes (Figure 6). *Clotho* follows the same path to locate the appropriate version of the logical extent in the LXT. To recreate the original, full extent data we need the differential data of the previous version of the logical extent. With this information *Clotho* can reconstruct the requested block and return it to the input driver. We evaluate the related overheads in Section IV.

Clotho supports recursive compaction of devices. The next version of a compacted version can still be compacted. Also, compacted versions can be uncompacted to their original state with the reverse process. A side-effect of the differential encoding concept is that it creates dependences between two consecutive versions of a logical extent, which affects the way versions are accessed, as explained next.

When deleting versions, *Clotho* checks for dependencies of compacted versions on a previous version and does not delete extents that are required for un-diffing, even if their versions are deleted. These logical extents are marked as `"shadow"` and are attached to the compacted version. It is left to higher-level policies to decide if keeping such blocks around increases the space overhead and it would be better to uncompact the related version and delete any

shadow logical extents.

F. Consistency

One of the main issues in block device versioning at arbitrary times is consistency of the stored data. There are three levels of consistency for online versioning:

System state consistency: This refers to consistency of system buffers and data structures that are used in the I/O path. To deal with this, *Clotho* flushes all device buffers in the kernel as well as filesystem metadata before version creation. This guarantees that the data and metadata on the block device correspond, let's say, to a valid snapshot of the filesystem at a point-in-time. That is, there are no consistency issues in internal system data structures.

Open file consistency: When a filesystem is used on top of a versioned device, certain files may be open at the time of a snapshot. *Clotho* provides two mechanisms to deal with this issue. First, higher level layers can query the system for open files on the particular device when creating a new version. This query is performed by *Clotho*'s user-level module. Second, if a version is created when files are open, *Clotho* creates a special directory with links to all open files and includes the directory in the archived version. Thus, when accessing older versions the user can find out which files were open at versioning time. We have also designed, but not implemented yet, a mechanism for merging subsequent volume versions to eliminate open files.

Application consistency: Applications using the versioned volume may have a specialized notion of consistency. For instance, an application may be using two files that are both updated atomically. If a version is created after the first file is updated and closed but before the second one is open and updated, then, although no files were open during version creation, the application data may still be inconsistent. This type of consistency is not possible to deal with transparently without application knowledge or support, and thus, is not addressed by *Clotho*.

III. System Implementation

We have implemented *Clotho* as a block device driver module in the Linux 2.4 kernel and a user-level control utility, in about 6,000 lines of C code. The kernel module can be loaded at runtime and configured for any output layer device by means of an `ioctl()` command triggered by the user-level agent. After configuring the output layer device, the user can manipulate the *Clotho* block device depending on the higher layer that they want to use. For instance, the user can build a filesystem on top of a *Clotho* device with `mkfs` or `newfs` and then `mount` it as a regular filesystem.

Our module adheres to the framework of block I/O devices in the Linux kernel and provides two interfaces to user programs. An `ioctl` command interface, and a `/proc` interface for device information and statistics. All operations described in the design section to create, delete, and manage version have been implemented through the `ioctl` interface and are initiated by the user-level agent. The `/proc` interface provides information about each device version through readable ASCII files. *Clotho* also uses this interface to report a number of statistics, including the times of creation, a version's time span, the size of modified data from the previous version and some specific information to compacted versions, such as the compaction level and the number of *shadow* extents.

The *Clotho* module uses the zero-copy mechanism of the `make_request_fn()` fashion that is used by LVM [31]. With this mechanism *Clotho* is able to translate the device driver id `(kdev_t)` and the sector address of a block request `(struct buffer_head)` and redirect it to other devices with minimal overhead. To achieve persistence of metadata, *Clotho* uses a kernel thread created at module load time, which flushes the metadata to the output layer at configurable (currently 30s) intervals, as well as when the module is unloaded.

The virtual device creation uses the partitionable block device concepts in the Linux kernel. A limit in the Linux kernel minor numbering is that there can be at most 255 minor numbers for a specific device and thus only 255 versions can be seen simultaneously as partitions of *Clotho*. However, the number of partitions supported by *Clotho* can be much larger. To overcome this limitation we have created a mechanism through an `ioctl` call that allows the user to link and unlink on demand any of the available versions to any of the 255 minor number partitions of a *Clotho* device. As mentioned, each of these partitions is read-only and can be used as a normal block device, e.g. can be mounted to a mount-point.

IV. Experimental Results

Our experimental environment consists of two identical PCs running Linux. Each system has two Pentium III 866 MHz CPUs, 768 MBytes of RAM, an IBM-DTLA-307045 ATA Hard Disk Drive with a capacity of 46116 MBytes (2-MByte cache), and a 100MBps Ethernet NIC. The operating system is Red Hat Linux 7.1, with the 2.4.18 SMP kernel. All experiments are performed on a 21-GByte partition of the IBM disk. With a 32-KByte extent, we need only 10.5 MBytes of memory for our 21-GByte partition.

Although there is a number of system parameters worth investigation, at this stage of our work, we evaluate *Clotho* with respect to two parameters: memory and performance overhead. We use two extent sizes, 4 and 32 KBytes. Smaller extent sizes have higher memory requirements. For our 21-GByte partition, *Clotho* with 4-KByte extent size uses 82 MBytes of in-memory metadata, the dirty parts of which are flushed to disk every 30 seconds. We evaluate *Clotho* using both microbenchmarks (Bonnie++ version 1.02 [3] and an in-house developed microbenchmark) and real-life setups with production-level filesystems. The Bonnie++ benchmark is a publicly available filesystem I/O benchmark [3]. For the real-life setup we run the SPEC SFS V3.0 suite on top of two well-known Linux filesystems, Ext2FS, and the high-performance journaled ReiserFS [20]. In our results we use the label *Disk* to denote experiments with the regular disk, without the *Clotho* driver on top.

A. Bonnie++

We use the Bonnie++ microbenchmark to quantify the basic overhead of *Clotho*. The filesystem we use in all Bonnie++ experiments is the Ext2FS with a 4-KByte extent size. The size of the file to be tested is 2 GBytes with block sizes ranging from 1 KByte to 64 KBytes. We measure accesses to the latest version of a volume with the following operations:

- *Block Write:* A large file is created using the `write()` system call.
- *Block Rewrite:* Each block of the file is read with `read()`, dirtied, and rewritten with `write()`, requiring an `lseek()`.

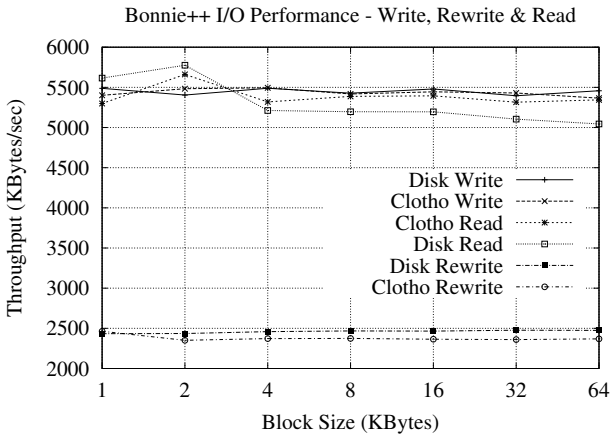

Fig. 7. Bonnie++ throughput for write, rewrite, and read operations.

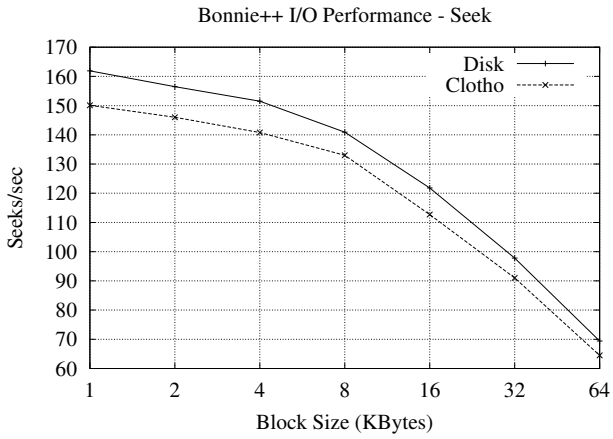

Fig. 8. Bonnie "seek and read" performance.

- *Block Read:* The file is read using a `read()` for every block.
- *Random Seek:* Processes running in parallel are performing `lseek()` to random locations in the file and `read()`ing the corresponding file blocks.

Figure 7 shows that the overhead in write throughput is minimal and the two curves are practically the same. In the read throughput case, *Clotho* performs slightly better than the regular disk. We believe this is due to the logging (sequential) disk allocation policy that *Clotho* uses. In the rewrite case, the overhead of *Clotho* becomes more significant. This is due to the random "seek and read" operation overhead, as shown in Figure 8. Since the seeks in this experiment are random, *Clotho*'s logging allocation has no effect and the overhead of translating I/O requests and flushing filesystem metadata to disk dominates. Even in this case however, the overhead observed is at most 7.5% of the regular disk.

B. SPEC SFS

We use the SPEC SFS 3.0 benchmark suite to measure NFS file server throughput and response time over *Clotho*. We use one NFS client and one NFS server. The two systems that serve as client and server are connected with a switched 100 MBit/s Ethernet network. We use the following settings: NFS version 3 protocol over UDP/IP, one NFS exported directory, `biod_max_read=2, biod_max_write=2,` and requested loads ranging from 300 to 1000 NFS V3 operations/s with a 100 increment step. Both warm-up and run time are 300 seconds for each run and the time for all the SPEC SFS runs in sequence is approximately 3 hours. As mentioned above, we report results for the Ext2FS and ReiserFS (with -notail option) filesystems [20]. A new filesystem is created before every experiment.

We conduct four experiments with SPEC SFS for each of the two filesystems: Using the plain disk, using *Clotho* over the disk without versioning, using *Clotho* and versioning every 5 minutes, and using *Clotho* with 10 minute versioning. Versioning is performed throughout the entire 3 hour run of SPEC SFS. Figures 9 and 10 show our throughput and latency results for 4-Kbyte extents, while Figures 11 and 12 show the results using 32-KByte extents with subextent addressing.

Our results show that *Clotho* outperforms the regular disk in all cases except ReiserFS without versioning. The

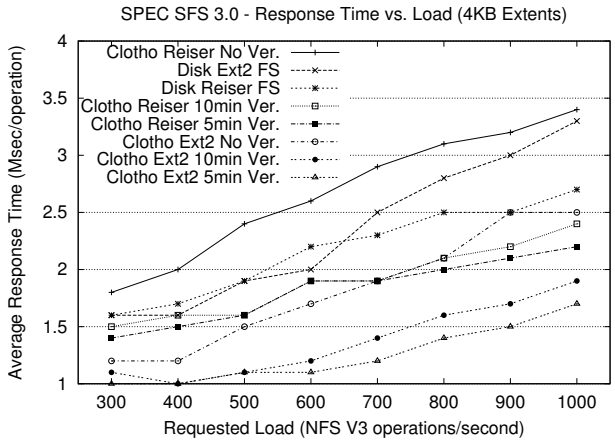

Fig. 9. SPEC SFS response time using 4-KByte extents.

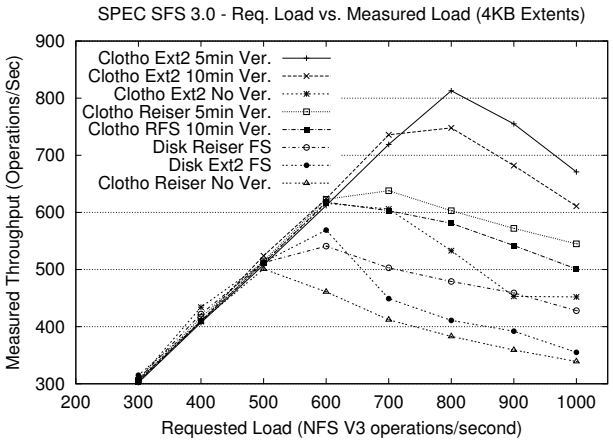

Fig. 10. SPEC SFS throughput using 4-KByte extents.

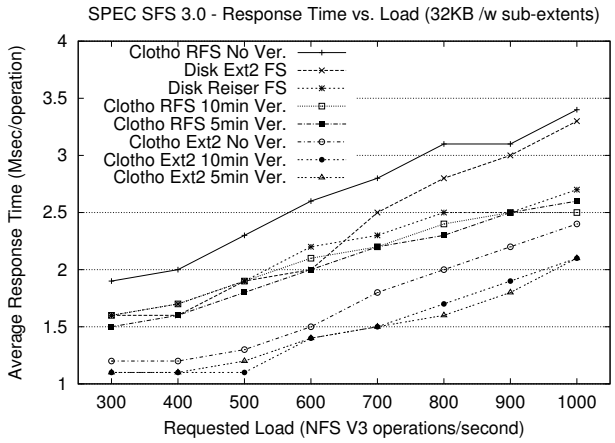

Fig. 11. SPEC SFS response time using 32-Kbyte extents with subextents (RFS denotes ReiserFS).

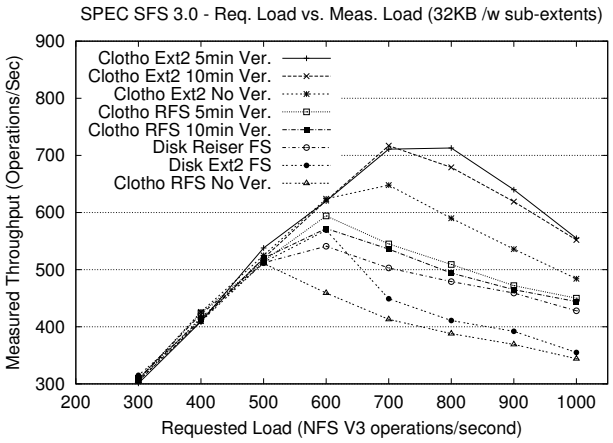

Fig. 12. SPEC SFS throughput using 32-Kbyte extents with subextents (RFS denotes ReiserFS).

higher performance is due to the logging (sequential) block allocation policy that *Clotho* uses. This explanation is reinforced by the performance in the cases where versions are created periodically. In this case, frequent versioning prevents disk seeks caused by overwriting of old data, which are now written to new locations on the disk in a sequential fashion. Furthermore, we observe that the more frequent the versioning, the higher the performance. The 32-KByte extent size experiments (Figures 11 and 12) show that even with much lower memory requirements, subextent mapping offers almost the same performance as the 4-KByte case. We attribute this small difference to the disk rotational latency, when skipping unused space to write subextents, while in the 4-KByte extent size, the extents are written "back-to-back" in a sequential manner.

Finally, comparing the two filesystems, Ext2 and ReiserFS, we find that the latter performs worse on top of *Clotho*. We attribute this behavior to the journaled metadata management of ReiserFS. While Ext2 updates metadata in place, ReiserFS appends metadata updates to a journal. This technique in combination with *Clotho*'s logging disk allocation appears to have a negative effect on performance in the SPEC SFS workload, compared to Ext2.

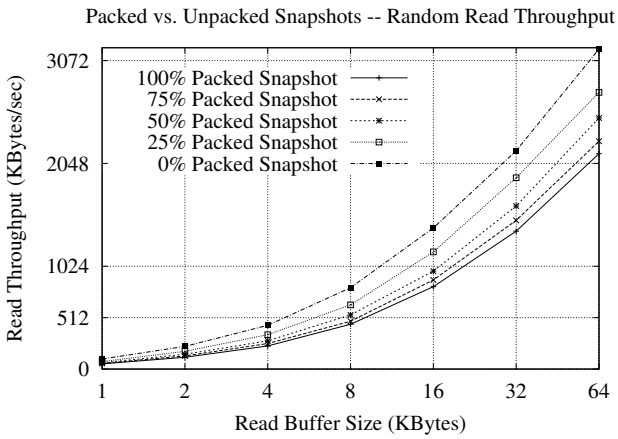

Fig. 13. Random "compact-read" throughput.

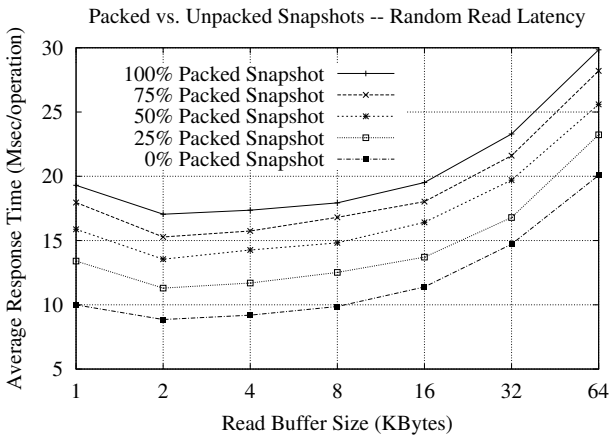

Fig. 14. Random "compact-read" latency.

C. Compact version performance

Finally, we measure the read throughput of compacted versions to evaluate the space-time tradeoff of diff-based compaction. Since old versions are only accessible in read-only mode, we developed a microbenchmark that performs only `read` operations. In the first stage, our microbenchmark writes a number of large files and captures multiple versions of the data through *Clotho*. In writing the data the benchmark is also able to control the amount of similarity between two versions, and thus, the percentage of space required by compacted versions. In the second stage, we `mount` a compacted version and our benchmark performs 2000 random read operations on the files of the compacted version. Before every run, the benchmark flushes the system's buffer cache.

Figures 13 and 14 present latency and throughput results for different percentages of compaction. For 100% compaction, the compacted version takes up minimal space on the disk, whereas in the 0% case compaction is not space-effective at all. The difference in performance is mainly due to the higher number of disk accesses per read operation required for compacted versions. Each such read operation requires two disk reads to reconstruct the requested block. One read to fetch the block of the previous version and one to fetch the diffs. In particular, with 100% compaction, each and every read results in two disk accesses and thus, performance is about half of the 0% compaction case.

V. Related Work

A number of projects have highlighted the importance and issues in storage management [15], [22], [12], [35]. Our goal in this work is to define innovative functionality that can be used in future storage protocols and APIs to reduce management overheads.

Block-level versioning was recently discussed and used in WAFL [13], a file system designed for Network Appliance's NFS appliance. WAFL works in the block-level of the filesystem and can create up to 20 snapshots of a volume and keep them available online through NFS. However, since WAFL is a filesystem and works in an NFS appliance, this approach depends on the filesystem. In our work we demonstrate that *Clotho* is filesystem agnostic by presenting experimental data with two production-level filesystems. Moreover, WAFL can manage a limited number

of versions (up to 20), whereas *Clotho* can manage a practically unlimited number. The authors in [13] mention that WAFL's performance cannot be compared to other general purpose file systems, since it runs on a specialized NFS appliance and much of its performance comes from its NFS-specific tuning. The authors in [14] use WAFL to compare the performance of filesystem- and block-level-based snapshots (within WAFL). They advocate the use of block-level backup, due to cost and performance reasons. However, they do not provide any evidence on the performance overhead of block-level versioned disks compared to regular, non-versioned block devices. In our work we thoroughly evaluate this with both microbenchmarks as well as standard workloads. SnapMirror [23] is an extension of WAFL, which introduces management of remote replicas in WAFL's snapshots to optimize data transfer and ensure consistency.

Venti [27] is a block-level network storage service, intended as a repository for backup data. Venti follows a write-once storage model and uses content based addressing by means of hash functions to identify blocks with identical content. Instead, *Clotho* uses differential compression concepts. Furthermore, Venti does not supper versioning features. *Clotho* and Venti are designed to perform complementary tasks, the former to version data and the latter as a repository to store safely the archived data blocks over the network. Distributed block-level versioning support was included in Petal [6]. Although the concepts are similar to *Clotho*, Petal also targets networks of workstations as opposed to active storage devices.

Since backup and archival of data is an important problem, there are many products available that try to address the related issues. However, specific information about these systems and their performance with commodity hardware, filesystems, or well-known benchmarks are scarce. LiveBackup [29] captures changes at the file level on client machines and sends modifications to a back-end server that archives previous file versions. EMC's SnapView [7] runs on the CLARiiON storage servers at the block level and uses a "copy-on-first-write" algorithm. However, it can capture only up to 8 snapshots and its copy algorithm does not use logging block allocation to speed up writes. Instead, it copies the old block data to hidden storage space on every first write, overwriting another block. Veritas's FlashSnap [34] software works inside the Veritas File System, and thus, unlike *Clotho*, is not filesystem agnostic. Furthermore it supports only up to 32 snapshots of volumes. Sun's Instant Image [17] works also at the block-level in the Sun StorEdge storage servers. Its operation appears similar to *Clotho*. However, it is used through drivers and programs in the Sun's StorEdge architecture, which runs only through the Solaris architecture and is also filesystem aware.

Each of the above systems, especially the commercial ones, uses proprietary customized hardware and system software, which makes comparisons with commodity hardware and general purpose operating systems difficult. Moreover, these systems are intended as standalone services within centralized storage appliances, whereas *Clotho* is designed as a transparent autonomous block-lever layer for active storage devices and appropriate for pushing functionality closer to the physical disk. In this direction, *Clotho* categorizes the challenges of implementing block-level versioning and addresses the related problems.

The authors in [5] examine the possibility of introducing an additional layer in the I/O device stack to provide certain functionality at lower system layers, which also affect the functionality that is provided by the filesystem. Other efforts in this direction, mostly include work in logical volume management and storage virtualization that

try to create a higher level abstraction on-top of simple block devices. The authors in [31] present a survey of such systems for Linux. Such systems usually provide the abstraction of a block-level volume that can be partitioned, aggregated, expanded, or shrunk on demand. Other such efforts [18] add RAID capabilities to arbitrary block devices. Our work is complementary to these efforts and proposes adding versioning capabilities to the block-device level.

Other previous work in versioning data has mostly been performed either at the filesystem layer or at higher layers. The authors in [26] propose versioning of data at the file level, discussing how the filesystem can transparently maintain file versions as well as how these can be cleaned up. The authors in [19] try to achieve similar functionality by providing mount points to previous versions of directories and files. They propose a solution that does not require kernel-level modification but relies on a set of user processes to capture user requests to files and to communicate with a back-end storage server that archives previous file versions. Other, similar efforts [21], [24], [25], [30], [28] approach the problem at the filesystem level as well and either provide the ability for checkpointing of data or explicitly manage time as an additional file attribute.

Self-securing storage [30] and its filesystem, CVFS [28] target secure storage systems and operate at the filesystem level. Some of the versioning concepts in self-securing storage and CVFS are similar to *Clotho*, but there are numerous differences as well. The most significant one is that self-securing storage policies are not intended for data archiving and thus, retain versions of data for a short period of time called *detection window*. No versions are guaranteed to exist outside this window of time and no version management control is provided for specifying higher-level policies. CVFS introduces certain interesting concepts for reducing metadata space, which however, are also geared towards security and are not intended for archival purposes. Since certain concepts in [30], [28] are similar to *Clotho*, we believe that a block-level self-secure storage system could be based on *Clotho*, separating the orthogonal versioning and security functionalities in different subsystems.

VI. Limitations and Future work

The main limitation of *Clotho* is that it cannot be layered below abstractions that aggregate multiple block devices in a single volume. and cannot be used with shared block devices transparently. If *Clotho* is layered below a volume abstraction that performs aggregation and on top of the block devices that are being aggregated in a single volume, policies for creating versions need to perform synchronized versioning accross devices to ensure data consistency. However, this may not be possible in a transparent manner to higher system layers. The main issue here is that it is not clear what are the semantics of versioning parts of a "coherent", larger volume. Furthermore, when multiple clients have access to a shared block device, as is usually the case with distributed block devices [6], [32], *Clotho* cannot be layered on top of the shared volume in each client, since internal metadata will become inconsistent accross *Clotho* instances. Solutions to these problems are interesting topics for future work.

Another limitation of *Clotho* is that it imposes a change in the block layout from the input to the output layer. *Clotho* acts as a filter between two block devices, transferring blocks of data from one layer to the next. Although this does not introduce any new issues with wasting space due to fragmentation (e.g. for files if a filesystem is used with *Clotho*), it alters significantly the data layout. Thus, it may affect I/O performance, if free blocks are scattered

over the disk or if higher layers rely on a specific block mapping, e.g. block 0 being the first block on the disk, block 1 the second, etc. However, this is an issue not only with *Clotho*, but with any layer in the I/O hierarchy that performs block remapping, such as RAIDs and some volume managers. Moreover, as I/O subsystems become more complex and provide more functionality, general solutions to this problem may become necessary. Since this is beyond the scope of this work, we do not discuss this any further here.

VII. Conclusions

Storage management is an important problem in building future storage systems. Online storage versioning can assist reduce these costs directly, by addressing data archival and retrieval costs and indirectly, by providing novel storage functionality. In this work we propose pushing versioning functionality closer to the disk and implementing it at the block-device level. This approach takes advantage of technology trends in building active self-managed storage systems to address issues related to backup and version management.

We present a detailed design of our system, *Clotho*, that provides versioning at the block-level. *Clotho* imposes small memory and disk space overhead for version data and metadata management by using large extents, sub-extent addressing and diff-based compaction. It imposes minimal performance overhead in the I/O path by eliminating the need for copy-on-write even when the extent size is larger than the disk-block size and by employing logging (sequential) disk allocation. It provides mechanisms for dealing with data consistency and allows for flexible policies for both manual and automatic version management.

We implement our system in the Linux operating system and evaluate its impact on I/O path performance with both microbenchmarks as well as the SPEC SFS standard benchmark on top of two production-level file systems, ExtFS and ReiserFS. We find that the common path overhead is minimal for read and write I/O operations when versions are not compacted. For compact versions, the user has to pay the penalty of double disk accesses for each I/O operation that accesses a compact block.

Overall, we believe that our approach is promising in offloading significant management overhead and complexity from higher system layers to the disk itself and is a concrete step towards building self-managed storage devices.

VIII. Acknowledgments

We thankfully acknowledge the support of Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario Innovation Trust, the Nortel Institute of Technology, Communications and Information Technology Ontario, and Nortel Networks.

References

[1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pages 175–188, Berkeley, CA, Jan. 28–30 2002. USENIX Association.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[3] R. Coker. Bonnie++. http://www.coker.com.au/bonnie++.

[4] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, Berkeley, CA, Dec. 9–11 2002. The USENIX Association.

[5] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of 14th SOSP*, pages 15–28, 1993.

[6] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of ASPLOS VII*, Oct. 1996.

[7] EMC. Snapview data sheet. http://www.emc.com/pdf/ products/ clariion/SnapView2_DS.pdf.

[8] S. C. Esener, M. H. Kryder, W. D. Doyle, M. Keshner, M. Mansuripur, and D. A. Thompson. Wtec panel report on the future of data storage technologies. 4501 North Charles Street, Baltimore, Maryland 21210-2699, June 1999. International Technology Research Institute. World Technology (WTEC) Division.

[9] GartnerGroup. Total Cost of Storage Ownership – A User-oriented Approach, Sept. 2000.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of the 8th ASPLOS*, Oct. 1998.

[11] G. A. Gibson and J. Wilkes. Self-managing network-attached storage. *ACM Computing Surveys*, 28(4es):209–209, Dec. 1996.

[12] J. Gray. What Next? A Few Remaining Problems in Information Technology (Turing Lecture). In *ACM Federated Computer Research Conferences (FCRC)*, May 1999.

[13] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the Winter 1994 USENIX Conference*, pages 235–246, 1994.

[14] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. Physical File System Backup. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Impl. (OSDI99)*, Feb. 1999.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*, November 2000.

[16] M. Lesk. How Much Information Is There In the World? http://www.lesk.com/ mlesk/ ksg97/ ksg.html, 1997.

[17] S. Microsystems. Instant image white paper. http://www.sun.com/ storage/ white-papers/ ii_soft_arch.pdf.

[18] Miguel de Icaza and Ingo Molnar and Gadi Oxman. The linux raid-1,-4,-5 code. In *LinuxExpo*, Apr. 1997.

[19] J. Moran, B. Lyon, and L. S. Incorporated. The Restore-o-Mounter: The File Motel Revisited. In *Proc. of USENIX '93 Summer Technical Conference*, June 1993.

[20] Namesys. Reiserfs. http://www.namesys.com.

[21] M. A. Olson. The design and implementation of the inversion file system. In *Proc. of USENIX '93 Winter Technical Conference*, Jan. 1993.

[22] D. Patterson. The UC Berkeley ISTORE Project: bringing availability, maintainability, and evolutionary growth to storage-based clusters. http://roc.cs.berkeley.edu, January 2000.

[23] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of FAST '02*. USENIX, Jan. 28–30 2002.

[24] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from bell labs. In *Proc. of the Summer UKUUG Conference*, 1990.

[25] W. D. Roome. 3dfs: A time-oriented file server. In *Proceedings of USENIX '92 Winter Technical Conference*, Jan. 1992.

[26] 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 17th SOSP*, Dec. 1999.

[27] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of FAST '02*, pages 89–102. USENIX, Jan. 28–30 2002.

[28] C. A. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST-03)*, Berkeley, CA, Apr. 2003. The USENIX Association.

[29] Storactive. Delivering real-time data protection & easy disaster recovery for windows workstations. http://www.storactive.com/files/ Storactive_Whitepaper.doc, Jan. 2002.

[30] 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-00)*, pages 165–180, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.

[31] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proceedings of USENIX 2001 Technical Conference*, June 2001.

[32] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 224–237, New York, Oct. 5–8 1997. ACM Press.

[33] A. C. Veitch, E. Riedel, S. J. Towers, and J. Wilkes. Towards Global Storage Management and Data Placement. In IEEE, editor, *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). May 20–23, 2001, Schloss Elmau, Germany*, pages 184–184, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001. IEEE Computer Society Press.

[34] Veritas. Flashsnap. http://eval.veritas.com/ downloads/ pro/ fsnap_guide_wp.pdf.

[35] J. Wilkes. Traveling to rome: Qos specifications for automated storage system m anagement. In *Proc. of the Int. Workshop on QoS (IWQoS'2001). Karlsruhe, Germany*, June 2001.

[36] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, May 1977.