DeltaFS

Lonnie Princehouse

May 9, 2009

Abstract

DeltaFS combines a read-only network filesystem with a mechanism for storing local changes. It is intended for use on limited capacity devices with good net connectivity, such as netbooks, mobile devices, and virtual machines. The local footprint of the filesystem is proportionate to the volume of changes made; so long as most files are unmodified, the local user can be given the illusion of having access to a very large drive.

1 Introduction

In October 2007, computer manufacturer ASUS released the Eee PC — an extremely small, extremely inexpensive laptop. In the months that followed, the Eee proved to be a surprise success, and the netbook (as these small laptops have come to be known) market was born. The original Eee came equipped with a 2GB or 4GB solid-state hard disk. As of March 2009, most major PC manufacturers have launched their own versions of the netbook; some use traditional laptop hard drives of up to 160GB, but the most affordable still ship with 4GB or 8GB solid-state disks. With 512MB or 1GB RAM and 1.6GHz Intel Atom CPUs, these netbooks are two generations behind the performance of the recent full-size laptops, but have quite sufficient horsepower for tasks like web browsing and word processing. Limited hard disk space, however, becomes a problem. The disk footprints of operating systems and applications have been growing steadily for years, unconstrained by rapidly growing disk capacities. A bare installation of Windows Vista with SP1 occupies upwards of 8GB. Ubuntu Linux 8.10 does better, using just over 2GB. Add to this several hundred megabytes of applications, and a netbook has precious little space left for user data. Netbook users cope with this by using older operating systems, such as Windows XP (which also runs better than Vista considering limited RAM and GPU), and by installing stripped down Linux-based operating systems like Ubuntu Mobile Edition. Even with such an operating system installed, we conjecture that netbook power-users will quickly fill their disks to the point that installing new software becomes a hassle.

With DeltaFS, we help to ameliorate this problem by providing a filesystem which can store infrequently used data, such as operating system and application files, in the cloud, allowing space-confined systems such as netbooks, smaller mobile devices, and virtual machines to use more of their local disks for storage of user data. DeltaFS consists of two pieces: The Base, a read-only network filesystem, and the Delta, a locally-stored record of modifications made to the filesystem. When a DeltaFS volume is created, it is initially identical to the Base, which could contain a (potentially very large) system image. Even though the Base is read-only, DeltaFS gives the users the illusion of having write access, with the caveat that writes only take affect locally and are not visible to other machines using the same Base. When a read request is made, DeltaFS downloads the relevant blocks from the Base and then applies any modifications logged in the Delta. For performance, frequently downloaded blocks are cached locally. A cache on the order of one or two hundred megabytes should make almost all reads local during normal operation (for example, Ubuntu Linux performs 70MB of reads

from the time it starts until the time it arrives at a user-ready desktop). With DeltaFS, the local footprint of a complete (not stripped) operating system and set of applications is equal to the size of the cache plus the Delta, which stores only changes made by the user.

Another benefit of DeltaFS is that to duplicate a volume requires only that the Delta be duplicated, since the read-only Base is the same. This may be very useful in the world of virtual machines, where cloning a machine that hasn't made many local changes could be done very quickly. It also provides data de-duplication, as many virtual machines could share the same Base.

DeltaFS includes a mechanism to "freeze" the Delta. This combines the Base and Delta, publishing the current view of the filesystem as a new read-only cloud filesystem that can be used as a new Base. By freezing the filesystem, a user can commit her changes to the network and free up space used by the local Delta.

While DeltaFS does demand a fast and reliable network connection, a properly filled cache could allow limited operation during connection outages.

2 Design

DeltaFS stores persistent data in two places:

The Base, a read-only network filesystem.

The Delta, a local log-structured filesystem. The Delta stores modifications to the Base, and caches data read from the Base.

2.1 Handling reads and writes

Any event that modifies the state of the filesystem is stored in the log. To determine the current state of the filesystem, the log must be replayed; for individual filesystem requests, which only concern one or two i-nodes, it is sufficient to replay only those log events which modify the affected i-nodes to determine their current state. Replaying these log events for every request would perform poorly, so DeltaFS caches *summaries* of i-node changes in memory. The current implementation keeps a summary of every modified i-node in memory, but a more sophisticated implementation might generate summaries on demand and keep a fixed-size cache according to some eviction policy. Upon receiving a read request, DeltaFS consults the relevant i-node's summary. Summaries store i-node attributes such as size and permissions, directory entries, file data chunk pointers (the data itself is stored on disk). If the requested information is in the summary, then DeltaFS fills the request quickly. Otherwise, DeltaFS fetches the information from the Base.

To handle write requests, DeltaFS writes i-node change events and new data chunks to the log. The current implementation replays these new log events immediately to update in-memory summaries.

2.2 Freezing

If the Base implementation supports it, the user may send a "freeze" command to DeltaFS. This merges the original base filesystem with changes stored in the Delta, and produces a new base. The transition to this new base is seamless; it is not necessary to unmount DeltaFS. Data chunks stored locally become cached copies of data residing at the new base, and the log is truncated. To the user, the DeltaFS filesystem is unchanged, except that more free space is available. The new base will ideally share unchanged data with the old base, but this is dependent on the Base implementation.

2.3 The Base

DeltaFS's design is largely independent of the design of the Base. However, the properties resulting from a DeltaFS instance depend greatly on the Base. The current implementation supports two Base backends: S3ROBase, a self-verifying read-only cloud filesystem stored in Amazon S3, and POSIXBase, which uses any directory on the Linux filesystem as a base. Each of these bases deliver different properties: S3ROBase supports freezing. When data is frozen to S3RO, the user benefits from S3's durability guarantees: freezing periodically becomes a viable backup strategy, in

addition to a way of freeing up local disk space. The space available on S3 is also virtually infinite, so this becomes a way to store a filesystem of ever-increasing size, even though it may get expensive. The Linux filesystem base allows users to have the illiusion of, for example, writing to a mounted CD-ROM, TAR or ZIP archive. It also allows speculative I/O to be performed which can later be "rolled back", since the Base directory is unchanged. The Linux filesystem implementation does not yet support freezing.

Each Base backend must implement its own Freezer module. A Freezer module combines the event log with the base to produce a new base.

2.4 Delta Structure

DeltaFS is intended to work well on solid-state disks (SSDs). SSDs are divided into sectors (typically 128 KB), each of which have a limited number of writes but can be read an unlimited number of times without penalty. It is therefore desirable to spread filesystem writes as evenly as possible across all sectors, to avoid exhausting any one sector lifespan while others still have many writes remaining. Writing any data to a sector invariably writes the entire 128 KB, so it is wise to batch writes into consecutive 128 KB pieces that fall on the sector boundaries.

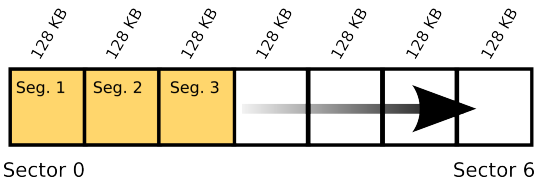

Figure 1: Physical sectors being filled with logical segments

DeltaFS batches log writes into 128 KB segments. Segments are logical entities, and each new segment bears an ever-increasing segment ID. Segments are written to consecutive disk sectors, cycling around the disk so that the write load to all sectors is balanced. Initially, all sectors are initialized to contain segment number zero. This is the null-segment. Eventually, once the log has cycled through all sectors, it will encounter sectors which contain non-null segments. At this point, garbage collection is need to determine which objects within a segment are obsolete, and to consolidate segments that are partially utilized to free up sectors. This is *log cleaning* is a typical log-structured filesystem problem. The current DeltaFS implementation does not implement cleaning, but it does offer the ability to freeze, thereby moving data to a remote filesystem and creating local free space. A garbage-collector would need to determine which log events and data objects are reachable from the current filesystem state; DeltaFS does provide a reachability graph, but time on the project was too short to design a proper garbage collector.

Segments contain a number of *objects* of variable size. These objects may be either log events or data objects, the difference being that the ordering of log events is significant (because they must be replayed to reconstruct current filesystem state), but data objects have no notion of ordering. Both kinds of objects have a unique identifier and an associated type; there are many types of log objects, but only one type of data object. Objects are packed into a segment starting from the least significant byte, and at the same time a *segment header table* is created at the most-significant end of the segment. The segment header table is an inventory of all objects in the segment.

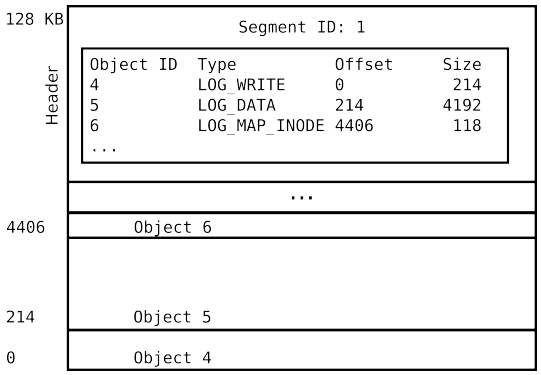

Figure 2: Segment Layout

When a DeltaFS is mounted, the entire disk is scanned, and the segment header tables read from

every segment. DeltaFS builds a map that associates each object with its segment, and another map from logical segments to physical sectors. This is scan is relatively quick for small disks, but it makes DeltaFS impractical for disks larger than a few gigabytes. This is an aspect of the design that needs improvement; the fundamental challenge is that we cannot designate any fixed sector(s) to store this meta-information without violating the policy of balanced sector writes.

2.5 Log Structure

The Log is a sequence of events that, when replayed, reconstruct the state of the filesystem. It is a logical log, meaning that the semantics of an event are preserved (e.g., "create a directory entry named README pointing to i-node number 109332"). This is opposed to a physicial log, which would store the above change as a new set of bits representing the modified directory i-node. This decision of logical versus physical log is discussed in greater depth in the Conclusions section.

In addition to the obvious types of log events — read, write, delete, create file, etc. — there are events that modify the data structures DeltaFS uses to track the mapping between the Base and the filesystem as seen by the user. For example, the Linux VFS demands that every i-node have a unique and persistent number. DeltaFS could use the same i-node numbers as the Base filesystem, but then how could we assign numbers to newly created files and directories, especially without reading the entire Base first to compile a set of already-used i-node numbers? Instead, DeltaFS stores a mapping between Base i-node numbers and DeltaFS i-node numbers. DeltaFS numbers are assigned on-demand, either when a new i-node is created, or when an i-node is read from the Base for the first time. This allows DeltaFS to store a mapping only for i-nodes it knows about, rather than all i-nodes in the system. This i-node mapping is logged in a LOG_MAP_INODE event; replaying this log event will update the i-node mapping kept by DeltaFS, even though it does not directly affect the state of any file or directory.

When DeltaFS is first initialized, the log contains two events: a Base event (LOG_LINUX_BASE or LOG_S3RO_BASE) followed by a LOG_MAP_INODE to give the root-directory i-node a DeltaFS i-node number. When a freeze occurs, LOG_ZERO is written, followed by a new BASE event and relevant LOG_CACHE events.

2.6 Summary Structure

Directory and file summaries store pointers to a struct stat filesystem meta-information structure. If this pointer is NULL, then requested stat information will be read from the Base.

Directory summaries store a directory entry diff map, which associates entry names with either a DeltaFS i-node ID, or the special value DELETED, indicating that the named entry has been deleted from the Base. If an entry lookup request for a named entry is made and the directory summary has no such entry, then it is fetched from the Base or an error is returned if the Base i-node does not contain the entry. A request to list all entries must download the Base i-node entry table, update it with the entry diff map, and present the result.

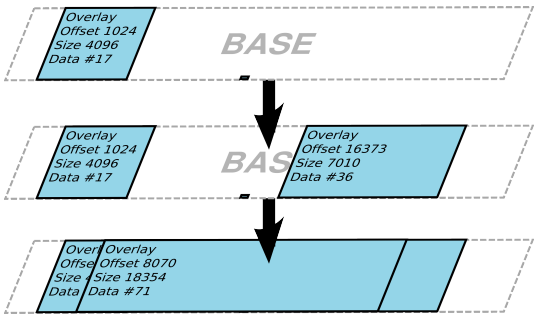

Figure 3: Successive write overlays show how a file summary evolves over time

File i-node summaries store modified data as a sorted list of *overlays*. An overlay consists of a start offset, a size, a data object ID, and an offset into the data object. Overlays cannot overlap, but there may be gaps between them. If a read request falls into

Event Type	Description
LOG_MAP_INODE	Associate a Base i-node ID with a DeltaFS i-node ID
LOG_CREATE_REGULAR_FILE LOG_CREATE_DIRECTORY	Create file and directory i-nodes
LOG_SETATTR	Set the stat() meta-information for an i-node
LOG_SET_DIRENT	Create a directory entry, or delete if the special child id DELETE is used
LOG_WRITE	Write data from a given data object to a regular file
LOG_CACHE	Similar to LOG_WRITE, but only creates a weak link to a data object. If a data object is only linked to by LOG_CACHE events, it can be safely garbage-collected. When a freeze occurs, a flurry of LOG_CACHE events is used to mark all data objects as cache.
LOG_S3RO_BASE	Initialize DeltaFS with the S3RO Base specified by the root directory stored in S3 object bucket/root_hash and using the supplied credentials
LOG_LINUX_BASE	Initialize DeltaFS with the Linux filesystem Base
LOG_ZERO	This is a meta-event that invalidates all previous log events. When DeltaFS is mounted, it replays log events starting from the most recent LOG_ZERO event; in this way, a freeze can invalidate all prior events without physically deleting them from the disk.

Table 1: Log Event Types

such a gap, then data is fetched from the Base. As successive writes occur, individual overlays may be truncated and split to accomodate newer overlays. If an overlay is complete covered by a new overlay, it becomes unreachable, and the event that created it may be garbage collected. If that, in turn, causes the data object to become unreachable, then the data object may also be garbage collected.

3 Related Work

The DeltaFS Base is a read-only, self-verifying network filesystem. This type of filesystem was pioneered by SFSRO [1] , which stores file data and metadata as blocks in a simple key-value storage server. A cryptographic hash of the block is used as the block's key; upon fetching a block, an SFSRO client verifies the block's authenticity by recomputing its hash and comparing it to the key used to fetch the block. SFSRO meta-data blocks link to other blocks via these cryptographic hashes, so a malicious server has no opportunity to corrupt any block in the filesystem. The root block is signed by the publisher's private key and indexed by a hash of the public key. This allows the publisher to re-publish updated filesystems with the same key. Several systems build on SFSRO: CFS distributes and de-centralizes the filesystem among many servers. [fixme: what else?] DeltaFS could conceivably be built atop any read-only filesystem, but an SFSRO-style filesystem is attractive because it is immutable, and the correctness of DeltaFS will depend on the Base not changing. The Base in DeltaFS will also attempt to aggregate small files into larger blocks, as is done for Cumulus [2] . By doing so, Cumulus reduces overhead in several ways: compression, encryption, and Amazon S3 transaction expenses are all reduced by aggregation.

The Delta in DeltaFS is a locally-stored record of local changes made on top of the read-only Base. Logically, the Delta describes how two filesystems differ; applying the changes recorded in Delta to the Base yields the current view of the local filesystem. This kind of filesystem "diff" has long been a part of incremental backup systems. The UNIX tar [3] and rsync [4] utilities compute such a diff for the purposes of backup and network synchronization, respectively. More sophisticated backup systems maintain several old versions of a filesystem, called snapshots. rdiff-backup [5] stores older snapshots snapshots as diffs

with respect to the most recent snapshot, which is a full backup. The primary difference between these backup diffs and those used by DeltaFS are the performance characteristics: The data structures used for backup diffs are optimized for full filesystem restores, not random access. Thus, they would be a poor choice for use in a filesystem. Some revision control systems, such as Subversion [6] , are also able to store filesystem diffs. However, a Subversion repository would be strained with the sheer volume of reads and writes made to a primary filesystem, and would not perform well if used as our Delta. Perhaps the closest related work to the Delta is the use of logs as a means to record incremental changes to a filesystem, as is done with log-structured filesystems [7] . These logs are kept in full log form for correctness, but for efficiency, a cache of the most recent version of changed files is kept. We expect to do the same with DeltaFS. Logs are also used as a means of tamper-proofing filesystems; when combined with signatures, they can record an un-repudiable chain of events. This is of greater concern for network filesystems, as data may be stored on untrusted hosts; such logs are used by Antiquity [8] and SUNDR [9] , among others.

DeltaFS is intended to perform well enough to serve as the root filesystem for a machine, containing the operating system and installed software. In this scenario, it would be highly desirable to pre-fill a local cache with those blocks from the read-only Base that are most likely to be needed to boot the operating system and are most frequently used during the course of normal operation. Because DeltaFS aggregates small files, the question arises of how to do aggregation such that files likely to be used at the same time are grouped together in the same block, thus minimizing block downloads. This is closely related to a large body of work that strives to optimize hard disk performance by grouping data together that are likely to be accessed sequentially, thus reducing the movement of the disk's head. One method for achieving this is to predict access patterns based on file access traces of previous executions of the operating system [10] . However, this kind of optimization is beyond what we expect to accomplish for DeltaFS in the context of a class project.

4 Implementation

Together, DeltaFS and S3RO are implemented in 11 lines of Python and over 4300 lines of C++. Two Bases are implemented, S3ROBase and a Linux filesystem Base, and new Bases can be implemented with relative ease thanks to a flexible C++ design. Other Base designs that were considered include a local BerkeleyDB block store for S3RO (thus avoiding S3 network problems), a Linux filesystem base capable of freezing, and a decentralized Base such as could be implemented using CFS [**?**]. Although the implementation does work, there are several features not implemented because of time constraints. Notably, log-structured filesystem cleaning is not implemented, so freezing is the only way to reclaim disk space. There are many known bugs regarding correct filesystem behavior — not all error cases are handled correctly; for instance, running out of memory or disk space will cause DeltaFS to crash — so the system in its current state is not suitable for any kind of real usage.

4.1 S3RO

S3RO is a self-verifying read-only cloud filesystem written for use with DeltaFS. It is a simplified implementation of SFS[1], the main difference being that it uses Amazon S3 as a data store. S3RO stores i-nodes and data chunks in a block store, indexed by the SHA1 hash of the block. In this way, data cannot be forged by a malicious server or middleman, because requested blocks must match their keys. Unlike SFS, S3RO makes no provision to use keys based on a cryptographic signature. This is intentional. With SFS and other subsequent systems, this capability was used to publish new versions of a filesystem. We do not want this capability in DeltaFS; immutability of the Base filesystem guarantees it will not change while we are storing change information.

S3RO breaks large files into 65KB data chunks, but does not do so for i-nodes: a directory with many thousands of entries could translate into a large S3 object. Directory and file i-nodes are each stored in their own S3 object; file i-nodes contain a list of all data chunks in the file.

S3RO verifies the content hashes of all blocks it downloads. Rudimentary client-side caching is done: to improve performance, the most recently used 200 S3 objects are cached in memory. However, overall performance of S3RO is very bad, because it is not currently able to make parallel requests to S3 so a great deal of time is wasted on latency. Additionaly, the implementation is unreliable unless being run on an EC2 instance; the S3 connection regularly times out, and the current S3RO implementation wastes much time recovering from these problems.

5 Results

DeltaFS remains too buggy for robust benchmarks, so quantitative results are left as future work. Anecdotal results suggest that performance of DeltaFS is very good for data stored locally, but the S3RO Base filesystem is quite slow, making network traffic a show-stopping bottleneck. It remains to be seen how performance degrades after a long period of use without freezing — a full disk of log events and highly fragmented file overlay summaries will certainly not be good for performance , but it is unknown just how bad performance will be under these conditions.

6 Conclusions

6.1 Logical vs. Physical Log

The current design uses a logical log to store events. The alternative would be a physical log, which would store on the Delta a direct record of bitwise diffs for the i-node and file data. A physical log would be simpler, in that it would not require the many different event types listed in Table 1. However, a physical log retains more information about the intent behind modifications, and this could be useful for future improvements; for example, in resolving freeze conflicts in a hypothetical multi-user version of DeltaFS. The logical log may also be smaller, as small logical events may translate into large phyiscal changes; then again, it might not, since it also contains many more events than a physical log would.

6.2 Future Work and Improvements

A number of improvements could be made to the DeltaFS design.

LBFS [11] and Cumulus [2] use Rabin fingerprints to break files into chunks with movable, content-based chunk boundaries. This allows more efficient diffs to be computed for files when bytes are inserted or deleted, and could be applied to S3RO to make freezing to S3RO use less bandwidth, storage space, and time.

DeltaFS could also explore the idea of speculation and reverting back to original base files. One way to do this would be to overload the filesystem namespace with special directives for DeltaFS. For example, if the regular file README has been modified, DeltaFS could be modified to interpret the path "README/original" as a request for the original file from the Base. This would make it possible to diff against an original version ("diff README README/original") and to revert to the original ("cp README/original README").

The DeltaFS log is currently stored in a block device or fixed-size file meant to simulate a block device. However, there may be uses to storing this log in a regular file that grows as the log grows. This would allow per-directory speculation, where changes to a directory could be stored temporarily in a DeltaFS volume file, and then replayed to commit changes to the actual directory. It also opens the door to nested DeltaFS instances — that is, storing a DeltaFS volume file inside another mounted DeltaFS filesystem.

However, the immediate and obvious future work is much more mundane. The bugs discussed in the Implementation section need to be fixed, and log cleaning should be implemented, in order to make it an attractive system for actual work.

References

[1] Kevin Fu, M. Frans Kaashoek, and David Mazieres. Fast and secure distributed filesystem In *Proc. 2nd USENIX OSDI*, December 2000

[2] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem Backup to the

Cloud In *Proc. 7th USENIX FAST*, February 2009

[3] Preston, W. C. *Backup and Recovery. OReilly*, 2006.

[4] A. Tridgell Efficient Algorithms for Sorting and Synchronization *PhD thesis, Australian National University*, February 1999

[5] Escoto, B. rdiff-backup. http://www.nongnu.org/rdiff-backup/

[6] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato *Version Control with Subversion, O'Reilly*, 2008

[7] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System *ACM Trans. on Computer Systems*, February 1992.

[8] H. Weatherspoon, P. Eaton, B. G. Chun, J. Kubiatowicz Antiquity: exploiting a secure log for wide-area distributed storage In *EuroSys '07*, March 2007

[9] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha Secure untrusted data repository In *Proc. 6th USENIX OSDI*, December 2004.

[10] Chris Ruemmler and John Wilkes Unix Disk Access Patterns *Hewlett-Packard Tech Report HPL-92-152*, December 1992

[11] A Muthitacharoen, B Chen, D Mazieres A low-bandwidth network filesystem *Proc. 18th ACM SOSP*, 2001