Tracking Back References in a Write-Anywhere File System

Tracking Back References in a Write-Anywhere File System

Peter Macko
Harvard University
pmacko@eecs.harvard.edu

Margo Seltzer
Harvard University
margo@eecs.harvard.edu

Keith A. Smith
NetApp, Inc.
keith.smith@netapp.com

Abstract

Many file systems reorganize data on disk, for example to defragment storage, shrink volumes, or migrate data between different classes of storage. Advanced file system features such as snapshots, writable clones, and deduplication make these tasks complicated, as moving a single block may require finding and updating dozens, or even hundreds, of pointers to it.

We present *Backlog*, an efficient implementation of explicit *back references*, to address this problem. Back references are file system meta-data that map physical block numbers to the data objects that use them. We show that by using LSM-Trees and exploiting the write-anywhere behavior of modern file systems such as NetApp® WAFL® or btrfs, we can maintain back reference meta-data with minimal overhead (one extra disk I/O per 102 block operations) and provide excellent query performance for the common case of queries covering ranges of physically adjacent blocks.

1 Introduction

Today's file systems such as WAFL [12], btrfs [5], and ZFS [23] have moved beyond merely providing reliable storage to providing useful services, such as snapshots and deduplication. In the presence of these services, any data block can be referenced by multiple snapshots, multiple files, or even multiple offsets within a file. This complicates any operation that must efficiently determine the set of objects referencing a given block, for example when updating the pointers to a block that has moved during defragmentation or volume resizing. In this paper we present new file system structures and algorithms to facilitate such dynamic reorganization of file system data in the presence of block sharing.

In many problem domains, a layer of indirection provides a simple way to relocate objects in memory or on storage without updating any pointers held by users of the objects. Such virtualization would help with some of the use cases of interest, but it is insufficient for one of the most important—defragmentation.

Defragmentation can be a particularly important issue for file systems that implement block sharing to support snapshots, deduplication, and other features. While block sharing offers great savings in space efficiency, sub-file sharing of blocks necessarily introduces on-disk fragmentation. If two files share a subset of their blocks, it is impossible for both files to have a perfectly sequential on-disk layout.

Block sharing also makes it harder to optimize on-disk layout. When two files share blocks, defragmenting one file may hurt the layout of the other file. A better approach is to make reallocation decisions that are aware of block sharing relationships between files and can make more intelligent optimization decisions, such as prioritizing which files get defragmented, selectively breaking block sharing, or co-locating related files on the disk.

These decisions require that when we defragment a file, we determine its new layout in the context of other files with which it shares blocks. In other words, given the blocks in one file, we need to determine the other files that share those blocks. This is the key obstacle to using virtualization to enable block reallocation, as it would hide this mapping from physical blocks to the files that reference them. Thus we have sought a technique that will allow us to track, rather than hide, this mapping, while imposing minimal performance impact on common file operations. Our solution is to introduce and maintain *back references* in the file system.

Back references are meta-data that map physical block numbers to their containing objects. Such back references are essentially inverted indexes on the traditional file system meta-data that maps file offsets to physical blocks. The challenge in using back references to simplify maintenance operations, such as defragmentation, is in maintaining them efficiently.

We have designed Log-Structured Back References,

or *Backlog* for short, a write-optimized back reference implementation with small, predictable overhead that remains stable over time. Our approach requires no disk reads to update the back reference database on block allocation, reallocation, or deallocation. We buffer updates in main memory and efficiently apply them *en masse* to the on-disk database during file system consistency points (checkpoints). Maintaining back references in the presence of snapshot creation, cloning or deletion incurs no additional I/O overhead. We use database compaction to reclaim space occupied by records referencing deleted snapshots. The only time that we read data from disk is during data compaction, which is an infrequent activity, and in response to queries for which the data is not currently in memory.

We present a brief overview of write-anywhere file systems in Section 2. Section 3 outlines the use cases that motivate our work and describes some of the challenges of handling them in a write-anywhere file system. We describe our design in Section 4 and our implementation in Section 5. We evaluate the maintenance overheads and query performance in Section 6. We present related work in Section 7, discuss future work in Section 8, and conclude in Section 9.

2 Background

Our work focuses specifically on tracking back references in *write-anywhere* (or *no-overwrite*) file systems, such as btrfs [5] or WAFL [12]. The terminology across such file systems has not yet been standardized; in this work we use WAFL terminology unless stated otherwise.

Write-anywhere file systems can be conceptually modeled as trees [18]. Figure 1 depicts a file system tree rooted at the *volume root* or a *superblock*. Inodes are the immediate children of the root, and they in turn are parents of indirect blocks and/or data blocks. Many modern file systems also represent inodes, free space bitmaps, and other meta-data as hidden files (not shown in the figure), so every allocated block with the exception of the root has a parent inode.

Write-anywhere file systems never update a block in place. When overwriting a file, they write the new file data to newly allocated disk blocks, recursively updating the appropriate pointers in the parent blocks. Figure 2 illustrates this process. This recursive chain of updates is expensive if it occurs at every write, so the file system accumulates updates in memory and applies them all at once during a *consistency point* (CP or *checkpoint*). The file system writes the root node last, ensuring that it represents a consistent set of data structures. In the case of failure, the operating system is guaranteed to find a consistent file system state with contents as of the last CP. File systems that support journaling to stable storage

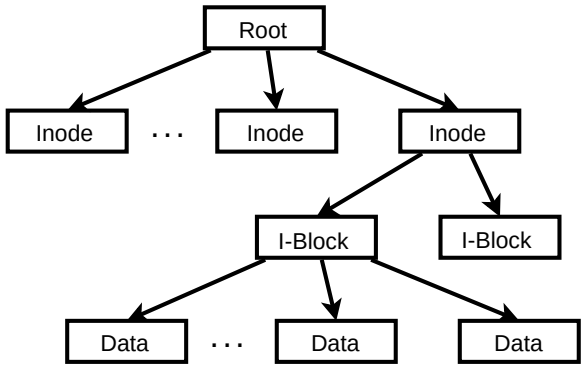


Figure 1: **File System as a Tree**. The conceptual view of a file system as a tree rooted at the *volume root* (superblock) [18], which is a parent of all inodes. An inode is a parent of data blocks and/or indirect blocks.

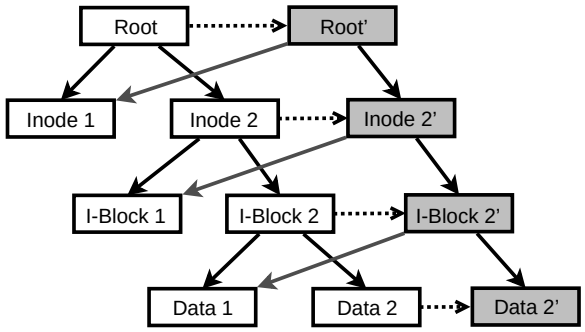


Figure 2: **Write-Anywhere file system maintenance**. In write-anywhere file systems, block updates generate new block copies. For example, upon updating the block "Data 2", the file system writes the new data to a new block and then recursively updates the blocks that point to it – all the way to the volume root.

(disk or NVRAM) can then recover data written since the last checkpoint by replaying the log.

Write-anywhere file systems can capture *snapshots*, point-in-time copies of previous file system states, by preserving the file system images from past consistency points. These snapshots are space efficient; the only differences between a snapshot and the live file system are the blocks that have changed since the snapshot copy was created. In essence, a write-anywhere allocation policy implements copy-on-write as a side effect of its normal operation.

Many systems preserve a limited number of the most recent consistency points, promoting some to hourly, daily, weekly, etc. *snapshots*. An asynchronous process typically reclaims space by deleting old CPs, reclaiming blocks whose only references were from deleted CPs. Several file systems, such as WAFL and ZFS, can create writable *clones* of snapshots, which are useful especially in development (such as creation of a writable

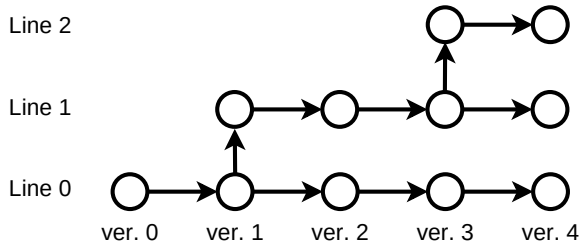


Figure 3: **Snapshot Lines**. The tuple (line, version), where *version* is a global CP number, uniquely identifies a snapshot or consistency point. Taking a consistency point creates a new version of the latest snapshot within each line, while creating a writable clone of an existing snapshot starts a new line.

duplicate for testing of a production database) and virtualization [9].

It is helpful to conceptualize a set of snapshots and consistency points in terms of *lines* as illustrated in Figure 3. A time-ordered set of snapshots of a file system forms a single *line*, while creation of a writable clone starts a new line. In this model, a (line ID, version) pair uniquely identifies a snapshot or a consistency point. In the rest of the paper, we use the global consistency point number during which a snapshot or consistency point was created as its version number.

The use of copy-on-write to implement snapshots and clones means that a single physical block may belong to multiple file system trees and have many meta-data blocks pointing to it. In Figure 2, for example, two different indirect blocks, *I-Block 2* and *I-Block 2'*, reference the block *Data 1*. Block-level deduplication [7, 17] can further increase the number of pointers to a block by allowing files containing identical data blocks to share a single on-disk copy of the block. This block sharing presents a challenge for file system management operations, such as defragmentation or data migration, that reorganize blocks on disk. If the file system moves a block, it will need to find and update all of the pointers to that block.

3 Use Cases

The goal of Backlog is to maintain meta-data that facilitates the dynamic movement and reorganization of data in write-anywhere file systems. We envision two major cases for internal data reorganization in a file system. The first is support for bulk data migration. This is useful when we need to move all of the data off of a device (or a portion of a device), such as when shrinking a volume or replacing hardware. The challenge here for traditional file system designs is translating from the physical block addresses we are moving to the files referencing those blocks so we can update their block pointers. Ext3, for example, can do this only by traversing the entire file system tree searching for block pointers that fall in the target range [2]. In a large file system, the I/O required for this brute-force approach is prohibitive.

Our second use case is the dynamic reorganization of on-disk data. This is traditionally thought of as defragmentation—reallocating files on-disk to achieve contiguous layout. We consider this use case more broadly to include tasks such as free space coalescing (to create contiguous expanses of free blocks for the efficient layout of new files) and the migration of individual files between different classes of storage in a file system.

To support these data movement functions in write-anywhere file systems, we must take into account the block sharing that emerges from features such as snapshots and clones, as well as from the deduplication of identical data blocks [7, 17]. This block sharing makes defragmentation both more important and more challenging than in traditional file system designs. Fragmentation is a natural consequence of block sharing; two files that share a subset of their blocks cannot both have an ideal sequential layout. And when we move a shared block during defragmentation, we face the challenge of finding and updating pointers in multiple files.

Consider a basic defragmentation scenario where we are trying to reallocate the blocks of a single file. This is simple to handle. We find the file's blocks by reading the indirect block tree for the file. Then we move the blocks to a new, contiguous, on-disk location, updating the pointer to each block as we move it.

But things are more complicated if we need to defragment two files that share one or more blocks, a case that might arise when multiple virtual machine images are cloned from a single master image. If we defragment the files one at a time, as described above, the shared blocks will ping-pong back and forth between the files as we defragment one and then the other. A better approach is to make reallocation decisions that are aware of the sharing relationship. There are multiple ways we might do this. We could select the most important file, and only optimize its layout. Or we could decide that performance is more important than space savings and make duplicate copies of the shared blocks to allow sequential layout for all of the files that use them. Or we might apply multi-dimensional layout techniques [20] to achieve near-optimal layouts for both files while still preserving block sharing.

The common theme in all of these approaches to layout optimization is that when we defragment a file, we must determine its new layout in the context of the other files with which it shares blocks. Thus we have sought a technique that will allow us to easily map physical blocks to the files that use them, while imposing minimal performance impact on common file system operations.

Our solution is to introduce and maintain *back reference* meta-data to explicitly track all of the logical owners of each physical data block.

4 Log-Structured Back References

Back references are updated significantly more frequently than they are queried; they must be updated on every block allocation, deallocation, or reallocation. It is crucial that they impose only a small performance overhead that does not increase with the age of the file system. Fortunately, it is not a requirement that the meta-data be space efficient, since disk is relatively inexpensive.

In this section, we present Log-Structured Back References (Backlog). We present our design in two parts. First, we present the conceptual design, which provides a simple model of back references and their use in querying. We then present a design that achieves the capabilities of the conceptual design efficiently.

4.1 Conceptual Design

A naïve approach to maintaining back references requires that we write a back reference record for every block at every consistency point. Such an approach would be prohibitively expensive both in terms of disk usage and performance overhead. Using the observation that a given block and its back references may remain unchanged for many consistency points, we improve upon this naïve representation by maintaining back references over ranges of CPs. We represent every such back reference as a record with the following fields:

- `block`: The physical block number
- `inode`: The inode number that references the block
- `offset`: The offset within the inode
- `line`: The line of snapshots that contains the inode
- `from`: The global CP number (time epoch) from which this record is valid (i.e., when the reference was allocated to the inode)
- `to`: The global CP number until which the record is valid (exclusive) or ∞ if the record is still alive

For example, the following table describes two blocks owned by inode 2, created at time 4 and truncated to one block at time 7:

block	inode	offset	line	from	to
100	2	0	0	4	∞
101	2	1	0	4	7

Although we present this representation as operating at the level of blocks, it can be extended to include a `length` field to operate on extents.

Let us now consider how a table of these records, indexed by physical block number, lets us answer the sort of query we encounter in file system maintenance. Imagine that we have previously run a deduplication process and found that many files contain a block of all 0's. We stored one copy of that block on disk and now have multiple inodes referencing that block. Now, let's assume that we wish to move the physical location of that block of 0's in order to shrink the size of the volume on which it lives. First we need to identify all the files that reference this block, so that when we relocate the block, we can update their meta-data to reference the new location. Thus, we wish to query the back references to answer the question, "Tell me all the objects containing this block." More generally, we may want to ask this query for a range of physical blocks. Such queries translate easily into indexed lookups on the structure described above. We use the physical block number as an index to locate all the records for the given physical block number. Those records identify all the objects that reference the block and all versions in which those blocks are valid.

Unfortunately, this representation, while elegantly simple, would perform abysmally. Consider what is required for common operations. Every block deallocation requires replacing the ∞ in the `to` field with the current CP number, translating into a read-modify-write on this table. Block allocation requires creating a new record, translating into an insert into the table. Block reallocation requires both a deallocation and an allocation, and thus a read-modify-write and an insert. We ran experiments with this approach and found that the file system slowed down to a crawl after only a few hundred consistency points. Providing back references with acceptable overhead during normal operation requires a feasible design that efficiently realizes the conceptual model described in this section.

4.2 Feasible Design

Observe that records in the conceptual table described in Section 4.1 are of two types. *Complete records* refer to blocks that are no longer part of the live file system; they exist only in snapshots. Such blocks are identified by having `to` $< \infty$. *Incomplete records* are part of the live file system and always have `to` $= \infty$. Our actual design maintains two separate tables, `From` and `To`. Both tables contain the first four columns of the conceptual table (`block`, `inode`, `offset`, and `line`). The `From` table also contains the `from` column, and the `To` table contains the `to` column. Incomplete records exist only in the `From` table, while complete records appear in both tables.

On a block allocation, regardless of whether the block is newly allocated or reallocated, we insert the corre-

sponding entry into the From table with the from field set to the current global CP number, creating an incomplete record. When a reference is removed, we insert the appropriate entry into the To table, completing the record. We buffer new records in memory, committing them to disk at the end of the current CP, which guarantees that all entries with the current global CP number are present in memory. This facilitates pruning records where from = to, which refer to block references that were added and removed within the same CP.

For example, the Conceptual table from the previous subsection (describing the two blocks of inode 2) is broken down as follows:

From:

block	inode	offset	line	from
100	2	0	0	4
101	2	1	0	4

To:

block	inode	offset	line	to
101	2	1	0	7

The record for block 101 is complete (has both From and To entries), while the record for 100 is incomplete (the block is currently allocated).

This design naturally handles block sharing arising from deduplication. When the file system detects that a newly written block is a duplicate of an existing on-disk block, it adds a pointer to that block and creates an entry in the From table corresponding to the new reference.

4.2.1 Joining the Tables

The conceptual table on which we want to query is the outer join of the From and To tables. A tuple F ∈ From joins with a tuple T ∈ To that has the same first four fields and that has the smallest value of T.to such that F.from < T.to. If there is a From entry without a matching To entry (i.e., a live, incomplete record), we outer-join it with an implicitly-present tuple T′ ∈ To with T′.to = ∞.

For example, assume that a file with inode 4 was created at time 10 with one block and then truncated at time 12. Then, the same block was assigned to the file at time 16, and the file was removed at time 20. Later on, the same block was allocated to a different file at time 30. These operations produce the following records:

From:

block	inode	offset	line	from
103	4	0	0	10
103	4	0	0	16
103	5	2	0	30

To:

block	inode	offset	line	to
103	4	0	0	12
103	4	0	0	20

Observe that the first From and the first To record form a logical pair describing a single interval during which the block was allocated to inode 4. To reconstruct the history of this block allocation, a record from = 10 has to join with to = 12. Similarly, the second From record should join with the second To record. The third From entry does not have a corresponding To entry, so it joins with an implicit entry with to = ∞.

The result of this outer join is the Conceptual view. Every tuple C ∈ Conceptual has both from and to fields, which together represent a range of global CP numbers within the given snapshot line, during which the specified block is referenced by the given inode from the given file offset. The range might include deleted consistency points or snapshots, so we must apply a mask of the set of valid versions before returning query results.

Coming back to our previous example, performing an outer join on these tables produces:

block	inode	offset	line	from	to
103	4	0	0	10	12
103	4	0	0	16	20
103	5	2	0	30	∞

This design is feasible until we introduce writable clones. In the rest of this section, we explain how we have to modify the conceptual view to address them. Then, in Section 5, we discuss how we realize this design efficiently.

4.2.2 Representing Writable Clones

Writable clones pose a challenge in realizing the conceptual design. Consider a snapshot (l, v), where l is the line and v is the version or CP. Naïvely creating a writable clone (l', v') requires that we duplicate all back references that include (l, v) (that is, C.line $= l \wedge$ C.from $\leq v <$ C.to, where C ∈ Conceptual), updating the line field to l' and the from and to fields to represent all versions (range $0 - \infty$). Using this technique, the conceptual table would continue to be the result of the outerjoin of the From and To tables, and we could express queries directly on the conceptual table. Unfortunately, this mass duplication is prohibitively expensive. Thus, our actual design cannot simply rely on the conceptual table. Instead we implicitly represent writable clones in the database using structural inheritance [6], a technique akin to copy-on-write. This avoids the massive duplication in the naïve approach.

The implicit representation assumes that every block of (l, v) is present in all subsequent versions of l', unless explicitly overridden. When we modify a block, b, in a new writable clone, we do two things: First, we declare the end of b's lifetime by writing an entry in the To table recording the current CP. Second, we record the alloca-

tion of the new block b' (a copy-on-write of b) by adding an entry into the From table.

For example, if the old block $b = 103$ was originally allocated at time 30 in line $l = 0$ and was replaced by a new block $b' = 107$ at time 43 in line $l' = 1$, the system produces the following records:

From:

block	inode	offset	line	from
103	5	2	0	30
107	5	2	1	43

To:

block	inode	offset	line	to
103	5	2	1	43

The entry in the To table *overrides* the inheritance from the previous snapshot; however, notice that this new To entry now has no element in the From table with which to join, since no entry in the From table exists with the line $l' = 1$. We join such entries with an implicit entry in the From table with from $= 0$. With the introduction of structural inheritance and implicit records in the From table, our joined table no longer matches our conceptual table. To distinguish the conceptual table from the actual result of the join, we call the join result the *Combined* table.

Summarizing, a back reference record C $\in$ Combined of (l, v) is implicitly present in all versions of l', unless there is an overriding record C$'$ $\in$ Combined with C.block $=$ C$'$.block $\wedge$ C.inode $=$ C$'$.inode $\wedge$ C.offset $=$ C$'$.offset $\wedge$ C$'$.line $=$ 1$'$ $\wedge$ C$'$.from $= 0$. If such a C$'$ record exists, then it defines the versions of l' for which the back reference is valid (i.e., from C$'$.from to C$'$.to). The file system continues to maintain back references as usual by inserting the appropriate From and To records in response to allocation, deallocation and reallocation operations.

While the Combined table avoids the massive copy when creating writable clones, query execution becomes a bit more complicated. After extracting initial result from the Combined table, we must iteratively expand those results as follows. Let Initial be the initial result extracted from Combined containing all records that correspond to blocks $b_0, \ldots, b_n$. If any of the blocks b_i has one or more override records, they are all guaranteed to be in this initial result. We then initialize the query Result to contain all records in Initial and proceed as follows. For every record R $\in$ Result that references a snapshot (l, v) that was cloned to produce (l', v'), we check for the existence of a corresponding override record C$'$ $\in$ Initial with C$'$.line $= l'$. If no such record exists, we explicitly add records C$'$.line $\leftarrow l'$, C$'$.from $\leftarrow 0$ and C$'$.to $\leftarrow \infty$ to Result. This process repeats recursively until it fails to insert additional records. Finally, when the result is fully expanded we mask the ranges to remove references to deleted snapshots as described in Section 4.2.1.

This approach requires that we never delete the back references for a cloned snapshot. Consequently, snapshot deletion checks whether the snapshot has been cloned, and if it has, it adds the snapshot ID to the list of *zombies*, ensuring that its back references are not purged during maintenance. The file system is then free to proceed with snapshot deletion. Periodically we examine the list of zombies and drop snapshot IDs that have no remaining descendants (clones).

5 Implementation

With the feasible design in hand, we now turn towards the problem of efficiently realizing the design. First we discuss our implementation strategy and then discuss our on-disk data storage (section 5.1). We then proceed to discuss database compaction and maintenance (section 5.2), partitioning the tables (section 5.3), and recovering the tables after system failure (section 5.4). We implemented and evaluated the system in fsim, our custom file system simulator, and then replaced the native back reference support in btrfs with Backlog.

The implementation in fsim allows us to study the new feature in isolation from the rest of the file system. Thus, we fully realize the implementation of the back reference system, but embed it in a simulated file system rather than a real file system, allowing us to consider a broad range of file systems rather than a single specific implementation. Fsim simulates a write-anywhere file system with writable snapshots and deduplication. It exports an interface for creating, deleting, and writing to files, and an interface for managing snapshots, which are controlled either by a stochastic workload generator or an NFS trace player. It stores all file system meta-data in main memory, but it does not explicitly store any data blocks. It stores only the back reference meta-data on disk. Fsim also provides two parameters to configure deduplication emulation. The first specifies the percentage of newly created blocks that duplicate existing blocks. The second specifies the distribution of how those duplicate blocks are shared.

We implement back references as a set of callback functions on the following events: adding a block reference, removing a block reference, and taking a consistency point. The first two callbacks accumulate updates in main memory, while the consistency point callback writes the updates to stable storage, as described in the next section. We implement the equivalent of a user-level process to support database maintenance and query. We verify the correctness of our implementation by a utility program that walks the entire file system tree, reconstructs the back references, and then compares them with the database produced by our algorithm.

5.1 Data Storage and Maintenance

We store the `From` and `To` tables as well as the precomputed `Combined` table (if available) in a custom row-oriented database optimized for efficient insert and query. We use a variant of LSM-Trees [16] to hold the tables. The fundamental property of this structure is that it separates an in-memory *write store* (WS or C_0 in the LSM-Tree terminology) and an on-disk *read store* (RS or C_1).

We accumulate updates to each table in its respective WS, an in-memory balanced tree. Our `fsim` implementation uses a Berkeley DB 4.7.25 in-memory B-tree database [15], while our btrfs implementation uses Linux red/black trees, but any efficient indexing structure would work. During consistency point creation, we write the contents of the WS into the RS, an on-disk, densely packed B-tree, which uses our own LSM-Tree/Stepped-Merge implementation, described in the next section.

In the original LSM-Tree design, the system selects parts of the WS to write to disk and merges them with the corresponding parts of the RS (indiscriminately merging all nodes of the WS is too inefficient). We cannot use this approach, because we require that a consistency point has all accumulated updates persistent on disk. Our approach is thus more like the Stepped-Merge variant [13], in which the entire WS is written to a new RS run file, resulting in one RS file per consistency point. These RS files are called the Level 0 runs, which are periodically merged into Level 1 runs, and multiple Level 1 runs are merged to produce Level 2 runs, etc., until we get to a large Level N file, where N is fixed. The Stepped-Merge Method uses these intermediate levels to ensure that the sizes of the RS files are manageable. For the back references use case, we found it more practical to retain the Level 0 runs until we run data compaction (described in Section 5.2), at which point, we merge all existing Level 0 runs into a single RS (analogous to the Stepped-Merge Level N) and then begin accumulating new Level 0 files at subsequent CPs. We ensure that the individual files are of a manageable size using horizontal partitioning as described in Section 5.3.

Writing Level 0 RS files is efficient, since the records are already sorted in memory, which allows us to construct the compact B-tree bottom-up: The data records are packed densely into pages in the order they appear in the WS, creating a Leaf file. We then create an Internal 1 (I1) file, containing densely packed internal nodes containing references to each block in the Leaf file. We continue building I files until we have an I file with only a single block (the root of the B-tree). As we write the Leaf file, we incrementally build the I1 file and iteratively, as we write I file, In, to disk, we incrementally build the I$(n+1)$ file in memory, so that writing the I files requires no disk reads.

Queries specify a block or a range of blocks, and those blocks may be present in only some of the Level 0 RS files that accumulate between data compaction runs. To avoid many unnecessary accesses, the query system maintains a Bloom filter [3] on the RS files that is used to determine which, if any, RS files must be accessed. If the blocks are in the RS, then we position an iterator in the Leaf file on the first block in the query result and retrieve successive records until we have retrieved all the blocks necessary to satisfy the query.

The Bloom filter uses four hash functions, and its default size for `From` and `To` RS files depends on the maximum number of operations in a CP. We use 32 KB for 32,000 operations (a typical setting for WAFL), which results in an expected false positive rate of up to 2.4%. If an RS contains a smaller number of records, we appropriately shrink its Bloom filter to save memory. This operation is efficient, since a Bloom filter can be halved in size in linear time [4]. The default filter size is expandable up to 1 MB for a `Combined` read store. False positives for the latter filter grow with the size of the file system, but this is not a problem, because the `Combined` RS is involved in almost all queries anyway.

Each time that we remove a block reference, we prune in real time by checking whether the reference was both created and removed during the same interval between two consistency points. If it was, we avoiding creating records in the Combined table where `from` = `to`. If such a record exists in `From`, our buffering approach guarantees that the record resides in the in-memory WS from which it can be easily removed. Conversely, upon block reference addition, we check the in-memory WS for the existence of a corresponding `To` entry with the same CP number and proactively prune those if they exist (thus a reference that exists between CPs 3 and 4 and is then reallocated in CP 4 will be represented with a single entry in Combined with a lifespan beginning at 3 and continuing to the present). We implement the WS for all the tables as balanced trees sorted first by `block`, `inode`, `offset`, and `line`, and then by the `from` and/or `to` fields, so that it is efficient to perform this proactive pruning.

During normal operation, there is no need to delete tuples from the RS. The masking procedure described in Section 4.2.1 addresses blocks deleted due to snapshot removal.

During maintenance operations that relocate blocks, e.g., defragmentation or volume shrinking, it becomes necessary to remove blocks from the RS. Rather than modifying the RS directly, we borrow an idea from the C-store, column-oriented data manager [22] and retain a *deletion vector*, containing the set of entries that should not appear in the RS. We store this vector as a B-tree in-

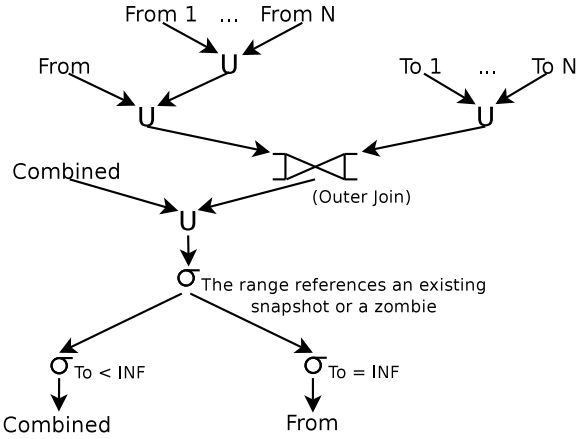


Figure 4: **Database Maintenance**. This query plan merges all on-disk RS's, represented by the "From N", precomputes the `Combined` table, which is the join of the `From` and `To` tables, and purges old records. Incomplete records reside in the on-disk `From` table.

dex, which is usually small enough to be entirely cached in memory. The query engine then filters records read from the RS according to the deletion vector in a manner that is completely opaque to query processing logic. If the deletion vector becomes sufficiently large, the system can optionally write a new copy of the RS with the deleted tuples removed.

5.2 Database Maintenance

The system periodically compacts the back reference indexes. This compaction merges the existing Level 0 RS's, precomputes the `Combined` table by joining the `From` and `To` tables, and purges records that refer to deleted checkpoints. Merging RS files is efficient, because all the tuples are sorted identically.

After compaction, we are left with one RS containing the complete records in the `Combined` table and one RS containing the incomplete records in the `From` table. Figure 4 depicts this compaction process.

5.3 Horizontal Partitioning

We partition the RS files by block number to ensure that each of the files is of a manageable size. We maintain a single WS per table, but then during a checkpoint, we write the contents of the WS to separate partitions, and compaction processes each partition separately. Note that this arrangement provides the compaction process the option of selectively compacting different partitions. In our current implementation, each partition corresponds to a fixed sequential range of block numbers.

There are several interesting alternatives for partitioning that we plan to explore in future work. We could start with a single partition and then use a threshold-based scheme, creating a new partition when an existing partition exceeds the threshold. A different approach that might better exploit parallelism would be to use hashed partitioning.

Partitioning can also allow us to exploit the parallelism found in today's storage servers: different partitions could reside on different disks or RAID groups and/or could be processed by different CPU cores in parallel.

5.4 Recovery

This back reference design depends on the write-anywhere nature of the file system for its consistency. At each consistency point, we write the WS's to disk and do not consider the CP complete until all the resulting RS's are safely on disk. When the system restarts after a failure, it is thus guaranteed that it finds a consistent file system with consistent back references at a state as of the last complete CP. If the file system has a journal, it can rebuild the WS's together with the other parts of the file system state as the system replays the journal.

6 Evaluation

Our goal is that back reference maintenance not interfere with normal file-system processing. Thus, maintaining the back reference database should have minimal overhead that remains stable over time. In addition, we want to confirm that query time is sufficiently low so that utilities such as volume shrinking can use them freely. Finally, although space overhead is not of primary concern, we want to ensure that we do not consume excessive disk space.

We evaluated our algorithm first on a synthetically generated workload that submits write requests as rapidly as possible. We then proceeded to evaluate our system using NFS traces; we present results using part of the EECS03 data set [10]. Next, we report performance for an implementation of Backlog ported into btrfs. Finally, we present query performance results.

6.1 Experimental Setup

We ran the first part of our evaluation in `fsim`. We configured the system to be representative of a common write-anywhere file system, WAFL [12]. Our simulation used 4 KB blocks and took a consistency point after every 32,000 block writes or 10 seconds, whichever came first (a common configuration of WAFL). We configured the deduplication parameters based on measure-

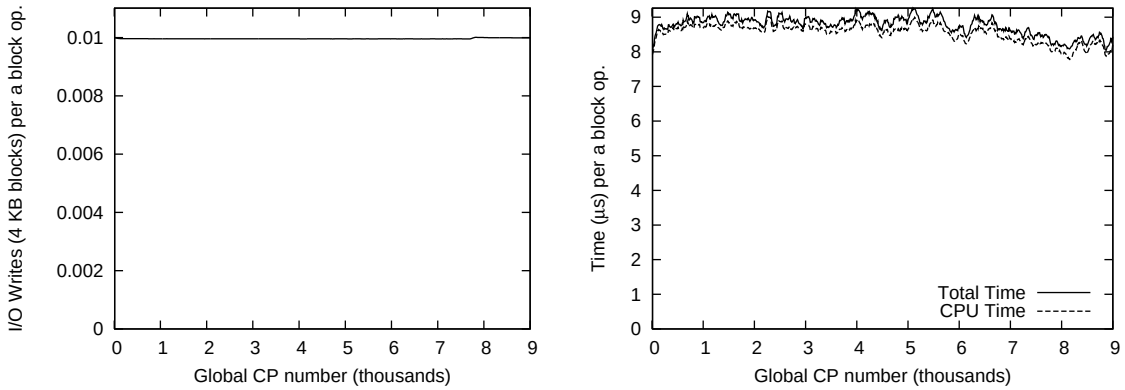


Figure 5: **Fsim Synthetic Workload Overhead during Normal Operation**. I/O overhead due to maintaining back references normalized per persistent block operations (adding or removing a reference with effects that survive at least one CP) and the time overhead normalized per block operation.

ments from a few file servers at NetApp. We treat 10% of incoming blocks as duplicates, resulting in a file system where approximately 75 – 78% of the blocks have reference counts of 1, 18% have reference counts of 2, 5% have reference counts of 3, etc. Our file system kept four hourly and four nightly snapshots.

We ran our simulations on a server with two dual-core Intel Xeon 3.0 GHz CPUs, 10 GB of RAM, running Linux 2.6.28. We stored the back reference meta-data from `fsim` on a 15K RPM Fujitsu MAX3073RC SAS drive that provides 60 MB/s of write throughput. For the micro-benchmarks, we used a 32 MB cache in addition to the memory consumed by the write stores and the Bloom filters.

We carried out the second part of our evaluation in a modified version of btrfs, in which we replaced the original implementation of back references by Backlog. As btrfs uses extent-based allocation, we added a `length` field to both the `From` and `To` described in Section 4.1. All fields in back reference records are 64-bit. The resulting `From` and `To` tuples are 40 bytes each, and a `Combined` tuple is 48 bytes long. All btrfs workloads were executed on an Intel Pentium 4 3.0 GHz, 512 MB RAM, running Linux 2.6.31.

6.2 Overhead

We evaluated the overhead of our algorithm in `fsim` using both synthetically generated workloads and NFS traces. We used the former to understand how our algorithm behaves under high system load and the latter to study lower, more realistic loads.

6.2.1 Synthetic Workload

We experimented with a number of different configurations and found that all of them produced similar results, so we selected one representative workload and used that throughout the rest of this section. We configured our workload generator to perform at least 32,000 block writes between two consistency points, which corresponds to the periods of high load on real systems. We set the rates of file create, delete, and update operations to mirror the rates observed in the EECS03 trace [10]. 90% of our files are small, reflecting what we observe on file systems containing mostly home directories of developers – which is similar to the file system from which the EECS03 trace was gathered. We also introduced creation and deletion of writable clones at a rate of approximately 7 clones per 100 CP's, although the original NFS trace did not have any analogous behavior. This is substantially more clone activity than we would expect in a home-directory workload such as EECS03, so it gives us a pessimal view of the overhead clones impose.

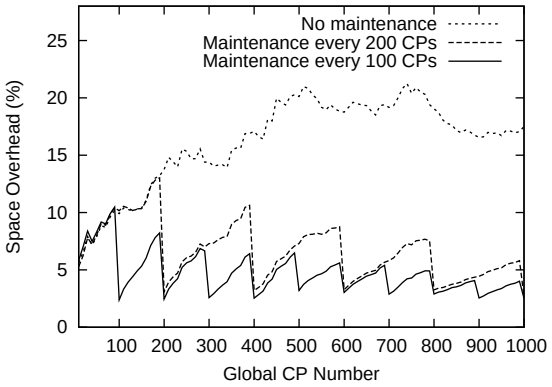
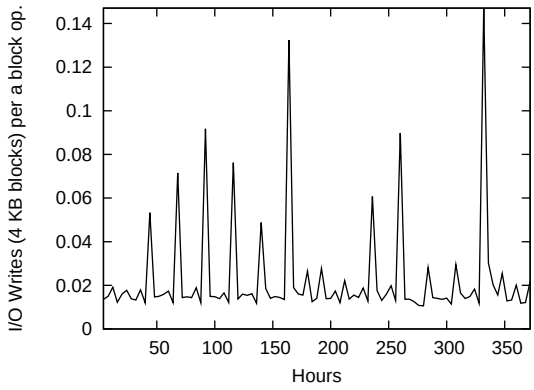


Figure 6: **Fsim Synthetic Workload Database Size**. The size of the back reference meta-data as a percentage of the total physical data size as it evolves over time. The disk usage at the end of the workload is 14.2 GB after deduplication.

Figure 5 shows how the overhead of maintaining back



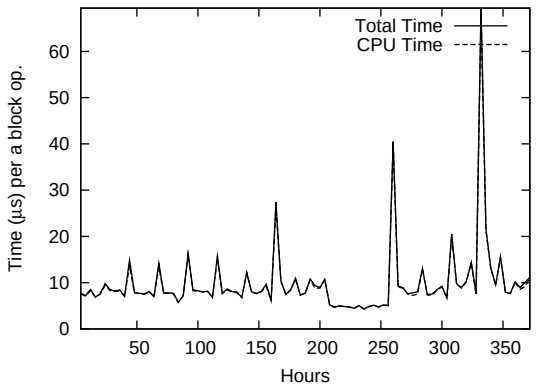


Figure 7: **Fsim NFS Trace Overhead during Normal Operation**. The I/O and time overheads for maintaining back references normalized per a block operation (adding or removing a reference).

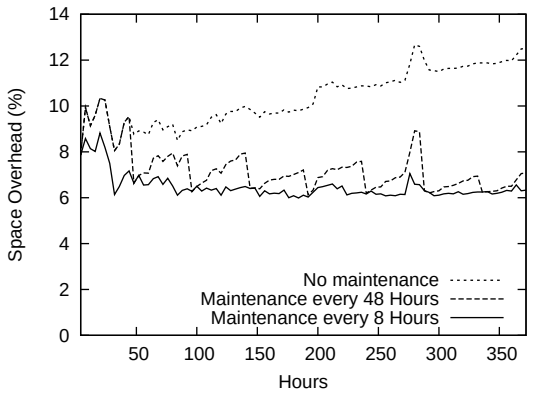


Figure 8: **Fsim NFS Traces: Space Overhead**. The size of the back reference meta-data as a percentage of the total physical data size as it evolves over time. The disk usage at the end of the workload is 11.0 GB after deduplication.

references changes over time, ignoring the cost of periodic database maintenance. The average cost of a block operation is 0.010 block writes or 8-9 μs per block operation, regardless of whether the operation is adding or removing a reference. A single copy-on-write operation (involving both adding and removing a block from an inode) adds on average 0.020 disk writes and at most 18 μs. This amounts to at most 628 additional writes and 0.5–0.6 seconds per CP. More than 95% of this overhead is CPU time, most of which is spent updating the write store. Most importantly, the overhead is stable over time, and the I/O cost is constant even as the total data on the file system increases.

Figure 6 illustrates meta-data size evolution as a percentage of the total physical data size for two frequencies of maintenance (every 100 or 200 CPs) and for no maintenance at all. The space overhead after maintenance drops consistently to 2.5%–3.5% of the total data size, and this low point does not increase over time.

The database maintenance tool processes the original database at the rate 7.7 – 10.4 MB/s. In our experiments, compaction reduced the database size by 30 – 50%. The exact percentage depends on the fraction of records that could be purged, which can be quite high if the file system deletes an entire snapshot line as we did in this benchmark.

6.2.2 NFS Traces

We used the first 16 days of the EECS03 trace [10], which captures research activity in home directories of a university computer science department during February and March of 2003. This is a write-rich workload, with one write for every two read operations. Thus, it places more load on Backlog than workloads with higher read-/write ratios. We ran the workload with the default configuration of 10 seconds between two consistency points.

Figure 7 shows how the overhead changes over time during the normal file system operation, omitting the cost of database maintenance. The time overhead is usually between 8 and 9 μs, which is what we saw for the synthetically generated workload, and as we saw there, the overhead remains stable over time. Unlike the overhead observed with the synthetic workload, this workload exhibits occasional spikes and one period where the overhead dips (between hours 200 and 250).

The spikes align with periods of low system load, where the constant part of the CP overhead is amortized across a smaller number of block operations, making the per-block overhead greater. We do not consider this behavior to pose any problem, since the system is under low load during these spikes and thus can better absorb the temporarily increased overhead.

The period of lower time overhead aligns with periods of high system load with a large proportion of `setattr` commands, most of which are used for file truncation. During this period, we found that only a small fraction

Benchmark	Base	Original	Backlog	Overhead
Creation of a 4 KB file (2048 ops. per CP)	0.89 ms	0.91 ms	0.96 ms	7.9%
Creation of a 64 KB file (2048 ops. per CP)	2.10 ms	2.11 ms	2.11 ms	1.9%
Deletion of a 4 KB file (2048 ops. per CP)	0.57 ms	0.59 ms	0.63 ms	11.2%
Creation of a 4 KB file (8192 ops. per CP)	0.85 ms	0.87 ms	0.87 ms	2.0%
Creation of a 64 KB file (8192 ops. per CP)	1.91 ms	1.92 ms	1.92 ms	0.6%
Deletion of a 4 KB file (8192 ops. per CP)	0.45 ms	0.46 ms	0.48 ms	7.1%
DBench CIFS workload, 4 users	19.59 MB/s	19.20 MB/s	19.19 MB/s	2.1%
FileBench /var/mail, 16 threads	852.04 ops/s	835.80 ops/s	836.70 ops/s	1.8%
PostMark	2050 ops/s	2032 ops/s	2020 ops/s	1.5%

Table 1: **Btrfs Benchmarks**. The Base column refers to a customized version of btrfs, from which we removed its original implementation of back references. The Original column corresponds to the original btrfs back references, and the Backlog column refers to our implementation. The Overhead column is the overhead of Backlog relative to the Base.

of the block operations survive past a consistency point. Thus, the operations in this interval tend to cancel each other out, resulting in smaller time overheads, because we never materialize these references in the read store.

This workload exhibits I/O overhead of approximately 0.010 to 0.015 page writes per block operation with occasional spikes, most (but not all) of which align with the periods of low file system load.

Figure 8 shows how the space overhead evolves over time for the NFS workload. The general growth pattern follows that of the synthetically generated workload with the exception that database maintenance frees less space. This is expected, since unlike the synthetic workload, the NFS trace does not delete entire snapshot lines. The space overhead after maintenance is between 6.1% and 6.3%, and it does not increase over time. The exact magnitude of the space overhead depends on the actual workload, and it is in fact different from the synthetic workload presented in Section 6.2.1. Each maintenance operation completed in less than 25 seconds, which we consider acceptable, given the elapsed time between invocations (8 or 48 hours).

6.3 Performance in btrfs

We validated our simulation results by porting our implementation of Backlog to btrfs. Since btrfs natively supports back references, we had to remove the native implementation, replacing it with our own. We present results for three btrfs configurations—the *Base* configuration with no back reference support, the *Original* configuration with native btrfs back reference support, and the *Backlog* configuration with our implementation. Comparing Backlog to the Base configuration shows the absolute overhead for our back reference implementation. Comparing Backlog to the Original configuration shows the overhead of using a general purpose back reference implementation rather than a customized implementation that is more tightly coupled to the rest of the file system.

Table 1 summarizes the benchmarks we executed on btrfs and the overheads Backlog imposes, relative to baseline btrfs. We ran microbenchmarks of create, delete, and clone operations and three application benchmarks. The create microbenchmark creates a set of 4 KB or 64 KB files in the file system's root directory. After recording the performance of the create microbenchmark, we `sync` the files to disk. Then, the delete microbenchmark deletes the files just created. We run these microbenchmarks in two different configurations. In the first, we take CPs every 2048 operations, and in the second, we take CP after 8192 operations. The choice of 8192 operations per CP is still rather conservative, considering that WAFL batches up to 32,000 operations. We also report the case with 2048 operations per CP, which corresponds to periods of a light server load as a point for comparison (and we can thus tolerate higher overheads). We executed each benchmark five times and report the average execution time (including the time to perform `sync`) divided by the total number of operations.

The first three lines in the table present microbenchmark results of creating and deleting small 4 KB files, and creating 64 KB files, taking a CP (btrfs transaction) every 256 operations. The second three lines present results for the same microbenchmarks with an inter-CP interval of 1024 operations. We show results for the three btrfs configurations—Base, Original, and Backlog. In general, the Backlog performance for writes is comparable to that of the native btrfs implementation. For 8192 operations per CP, it is marginally slower on creates than the file system with no back references (Base), but comparable to the original btrfs. Backlog is unfortunately slower on deletes – 7% as compared to Base, but only 4.3% slower than the original btrfs. Most of this overhead comes from updating the write-store.

The choice of 4 KB (one file system page) as our file size targets the worst case scenario, in which only a small number of pages are written in any given operation. The overhead decreases to as little as 0.6% for the creation of

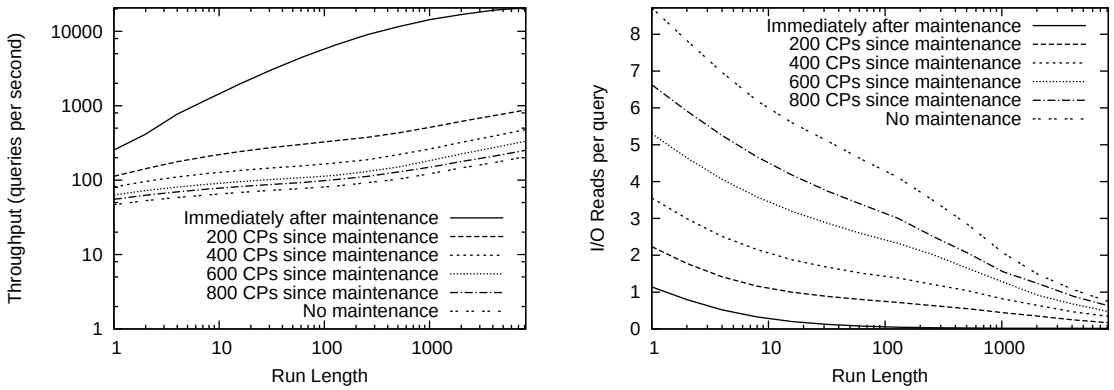


Figure 9: **Query Performance**. The query performance as a factor of run length and the number of CP's since the last maintenance on a 1000 CP-long workload. The plots show data collected from the execution of 8,192 queries with different run lengths.

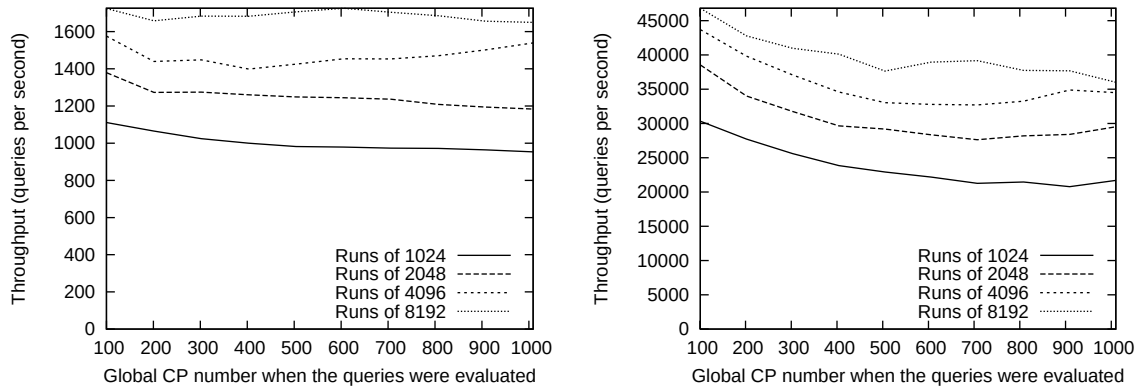


Figure 10: **Query Performance over Time**. The evolution of query performance over time on a database 100 CP's after maintenance (left) and immediately after maintenance (right). The horizontal axis is the global CP number at the time the query workload was executed. Each run of queries starts at a randomly chosen physical block.

a 64 KB file, because btrfs writes all of its data in one extent. This generates only a single back reference, and its cost is amortized over a larger number of block I/O operations.

The final three lines in Table 1 present application benchmark results: dbench [8], a CIFS file server workload; FileBench's /var/mail [11] multi-threaded mail server; and PostMark [14], a small file workload. We executed each benchmark on a clean, freshly formatted volume. The application overheads are generally lower (1.5% – 2.1%) than the worst-case microbenchmark overheads (operating on 4 KB files) and in two cases out of three comparable to the original btrfs.

Our btrfs implementation confirms the low overheads predicted via simulation and also demonstrates that Backlog achieves nearly the same performance as the btrfs native implementation. This is a powerful result as the btrfs implementation is tightly integrated with the btrfs data structures, while Backlog is a general-purpose solution that can be incorporated into any write-anywhere file system.

6.4 Query Performance

We ran an assortment of queries against the back reference database, varying two key parameters, the sequentiality of the requests (expressed as the length of a run) and the number of block operations applied to the database since the last maintenance run. We implement runs with length n by starting at a randomly selected allocated block, b, and returning back references for b and the next $n-1$ allocated blocks. This holds the amount of work in each test case constant; we always return n back references, regardless of whether the area of the file system we select is densely or sparsely allocated. It also gives us conservative results, since it always returns data for n back references. By returning the maximum possibly number of back references, we perform the maximum number of I/Os that could occur and thus report the lowest query throughput that would be observed.

We cleared both our internal caches and all file system caches before each set of queries, so the numbers we present illustrate worst-case performance. We found the

query performance in both the synthetic and NFS workloads to be similar, so we will present only the former for brevity. Figure 9 summarizes the results.

We saw the best performance, 36,000 queries per second, when performing highly sequential queries immediately after database maintenance. As the time since database maintenance increases, and as the queries become more random, performance quickly drops. We can process 290 single-back-reference queries per second immediately after maintenance, but this drops to 43 – 197 as the interval since maintenance increases. We expect queries for large sorted runs to be the norm for maintenance operations such as defragmentation, indicating that such utilities will experience the better throughput. Likewise, it is reasonable practice to run database maintenance prior to starting a query intensive task. For example, a tool that defragments a 100 MB region of a disk would issue a sorted run of at most 100 MB / 4 KB = 25,600 queries, which would execute in less than a second on a database immediately after maintenance. The query runs for smaller-scale applications, such as file defragmentation, would vary considerably – anywhere from a few blocks per run on fragmented files to thousands for the ones with a low degree of fragmentation.

Issuing queries in large sorted runs provides two benefits. It increases the probability that two consecutive queries can be satisfied from the same database page, and it reduces the total seek distance between operations. Queries on recently maintained database are more efficient for for two reasons: First, a compacted database occupies fewer RS files, so a query accesses fewer files. Second, the maintenance process shrinks the database size, producing better cache hit ratios.

Figure 10 shows the result of an experiment in which we evaluated 8192 queries every 100 CP's just before and after the database maintenance operation, also scheduled every 100 CP's. The figure shows the improvement in the query performance due to maintenance, but more importantly, it also shows that once the database size reaches a certain point, query throughput levels off, even as the database grows larger.

7 Related Work

Btrfs [2, 5] is the only file system of which we are aware that currently supports back references. Its implementation is efficient, because it is integrated with the entire file system's meta-data management. Btrfs maintains a single B-tree containing *all* meta-data objects.

A file extent back reference consists of the four fields: the subvolume, the inode, the offset, and the number of times the extent is referenced by the inode. Btrfs encapsulates all meta-data operations in transactions analogous to WAFL consistency points. Therefore a btrfs transaction ID is analogous to a WAFL CP number. Btrfs supports efficient cloning by omitting transaction ID's from back reference records, while Backlog uses ranges of snapshot versions (the `from` and `to` fields) and structural inheritance. A naïve copy-on-write of an inode in btrfs would create an exact copy of the inode (with the same inode ID), marked with a more recent transaction ID. If the back reference records contain transaction IDs (as in early btrfs designs), the file system would also have to duplicate the back references of all of the extents referenced by the inode. By omitting the transaction ID, a single back reference points to both the old and new versions of the inode simultaneously. Therefore, btrfs performs inode copy-on-write for free, in exchange for query performance degradation, since the file system has to perform additional I/O to determine transaction ID's. In contrast, Backlog enables free copy-on-write by operating on ranges of global CP numbers and by using structural inheritance, which do not sacrifice query performance.

Btrfs accumulates updates to back references in an in-memory balanced tree analogous to our write store. The system inserts all the entries from the in-memory tree to the on-disk tree during a transaction commit (a part of a checkpoint processing). Btrfs stores most back references directly inside the B-tree records that describe the allocated extents, but on some occasions, it stores them as separate items close to these extent allocation records. This is different from our approach in which we store all back references together, separately from block allocation bitmaps or records.

Perhaps the most significant difference between btrfs back references and Backlog is that the btrfs approach is deeply enmeshed in the file system design. The btrfs approach would not be possible without the existence of a global meta-store. In contrast, the only assumption necessary for our approach is the use of a write-anywhere or no-overwrite file system. Thus, our approach is easily portable to a broader class of file systems.

8 Future Work

The results presented in Section 6 provide compelling evidence that our LSM-Tree based implementation of back references is an efficient and viable approach. Our next step is to explore different options for further reducing the time overheads, the implications and effects of horizontal partitioning as described in Section 5.3, and experiment with compression. Our tables of back reference records appear to be highly compressible, especially if we to compress them by columns [1]. Compression will cost additional CPU cycles, which must be carefully balanced against the expected improvements in the space overhead.

We plan to explore the use of back references, implementing defragmentation and other functionality that uses back reference meta-data to efficiently maintain and improve the on-disk organization of data. Finally, we are currently experimenting with using Backlog in an update-in-place journaling file system.

9 Conclusion

As file systems are called upon to provide more sophisticated maintenance, back references represent an important enabling technology. They facilitate hard-to-implement features that involve block relocation, such as shrinking a partition or fast defragmentation, and enable us to do file system optimizations that involve reasoning about block ownership, such as defragmentation of files that share one or more blocks (Section 3).

We exploit several key aspects of this problem domain to provide an efficient database-style implementation of back references. By separately tracking when blocks come into use (via the `From` table) and when they are freed (via the `To` table) and exploiting the relationship between writable clones and their parents (via structural inheritance), we avoid the cost of updating per block meta-data on each snapshot or clone creation or deletion. LSM-trees provide an efficient mechanism for sequentially writing back-reference data to storage. Finally, periodic background maintenance operations amortize the cost of combining this data and removing stale entries.

In our prototype implementation we showed that we can track back-references with a low constant overhead of roughly 8-9 μs and 0.010 I/O writes per block operation and achieve query performance up to 36,000 queries per second.

10 Acknowledgments

We thank Hugo Patterson, our shepherd, and the anonymous reviewers for careful and thoughtful reviews of our paper. We also thank students of CS 261 (Fall 2009, Harvard University), many of whom reviewed our work and provided thoughtful feedback. We thank Alexei Colin for his insight and the experience of porting Backlog to other file systems. This work was made possible thanks to NetApp and its summer internship program.

References

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. Integrating compression and execution in column-oriented database systems. In *SIGMOD* (2006), pp. 671–682.

[2] AURORA, V. A short history of btrfs. *LWN.net* (2009).

[3] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.

[4] BRODER, A., AND MITZENMACHER, M. Network applications of bloom filters: A survey. *Internet Mathematics* (2005).

[5] Btrfs. `http://btrfs.wiki.kernel.org`.

[6] CHAPMAN, A. P., JAGADISH, H. V., AND RAMANAN, P. Efficient provenance storage. In *SIGMOD* (2008), pp. 993–1006.

[7] CLEMENTS, A. T., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized deduplication in SAN cluster file systems. In *USENIX Annual Technical Conference* (2009), pp. 101–114.

[8] DBench. `http://samba.org/ftp/tridge/dbench/`.

[9] EDWARDS, J. K., ELLARD, D., EVERHART, C., FAIR, R., HAMILTON, E., KAHN, A., KANEVSKY, A., LENTINI, J., PRAKASH, A., SMITH, K. A., AND ZAYAS, E. R. FlexVol: Flexible, efficient file volume virtualization in WAFL. In *USENIX ATC* (2008), pp. 129–142.

[10] ELLARD, D., AND SELTZER, M. New NFS Tracing Tools and Techniques for System Analysis. In *LISA* (Oct. 2003), pp. 73–85.

[11] FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench/`.

[12] HITZ, D., LAU, J., AND MALCOLM, M. A. File system design for an NFS file server appliance. In *USENIX Winter* (1994), pp. 235–246.

[13] JAGADISH, H. V., NARAYAN, P. P. S., SESHADRI, S., SUDARSHAN, S., AND KANNEGANTI, R. Incremental organization for data recording and warehousing. In *VLDB* (1997), pp. 16–25.

[14] KATCHER, J. PostMark: A new file system benchmark. *NetApp Technical Report TR3022* (1997).

[15] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley DB. In *USENIX ATC* (June 1999).

[16] O'NEIL, P. E., CHENG, E., GAWLICK, D., AND O'NEIL, E. J. The log-structured merge-tree (LSM-Tree). *Acta Informatica 33*, 4 (1996), 351–385.

[17] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *USENIX FAST* (2002), pp. 89–101.

[18] RODEH, O. B-trees, shadowing, and clones. *ACM Transactions on Storage 3*, 4 (2008).

[19] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[20] SCHLOSSER, S. W., SCHINDLER, J., PAPADOMANOLAKIS, S., SHAO, M., AILAMAKI, A., FALOUTSOS, C., AND GANGER, G. R. On multidimensional data and modern disks. In *USENIX FAST* (2005), pp. 225–238.

[21] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *USENIX Winter* (1993), pp. 307–326.

[22] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S. R., O'NEIL, E. J., O'NEIL, P. E., RASIN, A., TRAN, N., AND ZDONIK, S. B. C-Store: A column-oriented DBMS. In *VLDB* (2005), pp. 553–564.

[23] ZFS at OpenSolaris community. `http://opensolaris.org/os/community/zfs/`.