The Composite-file File System: Decoupling One-to-one Mapping of Files and Metadata for Better Performance

SHUANGLONG ZHANG, Google
ROBERT ROY, Florida State University
LEAH RUMANCIK, Florida State University
AN-I ANDY WANG, Florida State University

The design and implementation of traditional file systems typically use the one-to-one mapping of logical files to their physical metadata representations. File system optimizations generally follow this rigid mapping and miss opportunities for an entire class of optimizations.

We designed, implemented, and evaluated a composite-file file system, which allows many-to-one mappings of files to metadata. Through exploring different mapping strategies, our empirical evaluation shows up to a 27% performance improvement under webserver and software development workloads, for both disks and SSDs. This result demonstrates that our approach of relaxing file-to-metadata mapping is promising.

CCS Concepts: • **Information systems~Hierarchical storage management** • *Information systems~Inodes* • Information systems~Clustering • **Software and its engineering~File systems management**

KEYWORDS

General Terms: Design, Performance

Additional Keywords and Phrases: File systems, metadata

ACM Reference format:

Shuanglong Zhang, Robert Roy, Leah Rumancik, and An-I Andy Wang. 2018. *ACM Transactions on Storage,* x, x, Article X (Month year), x pages.
https://doi.org/0000001.0000001

XX

1 INTRODUCTION

File system performance optimization is a well-researched area, and most optimization techniques (e.g., caching, better data layout) retain the one-to-one mapping of logical files to their physical metadata representations (i.e., each file is associated with a unique i-node on UNIX platforms). Such mapping is intuitive and straightforward from the software design viewpoint, and is compatible with deep-rooted

This work is sponsored by FSU and National Science Foundation, under grant CNS-144387. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the FSU, NSF, Google, or the U.S. government.

Author's addresses: S. Zhang, 1600 Amphitheatre Parkway, Mountain View, CA 94043; R. Roy, 253 Love Building, Florida State University, FL 32306-4530; L. Rumancik, 253 Love Building, Florida State University, FL 32306-4530; A.I. A. Wang, 269 Love Building, Florida State University, Tallahassee, FL 32306-4530.

http://dx.doi.org/10.1145/336684

metadata constructs. Many storage components and mechanisms—such as VFS API [19], prefetching (e.g., readahead in Linux), and metadata caching—rely on such constructs. However, this rigid mapping also limits the opportunity for a class of performance optimizations.

Taking prefetching and caching under Linux VFS as an example—generally, file blocks are prefetched when a sequential access pattern is detected, and prefetching stops at the file boundary. Prefetching for metadata is mostly a side effect of the access granularity, where i-nodes (regardless of whether they are accessed together) are stored in the same storage block. Thus, prefetching related files that are stored nearby on storage involves accessing metadata items that can be stored elsewhere whenever file boundaries are crossed.

Although the overhead of such interruptions varies depending on the extent to which related files and their metadata are cached, the logical notion of files and associated metadata access can impose a high overhead. For example, accessing 32 small files can incur a 50% higher latency than accessing a single file with a size equal to the sum of the 32 files, even with warm caches [3].

With examples like these in mind, we have designed and prototyped a system called the Composite-file File System (CFFS), where many logical files can be associated with a single i-node (with extra information stored as extended attributes). Such an arrangement is possible because many files accessed together share similar metadata subfields (e.g., owner and permission fields of related web pages) [10] and these can be consolidated and deduplicated. Thus, accessing files under the CFFS requires fewer metadata storage accesses, which represent a nontrivial source of performance overhead for the small files that still represent many file references for modern workloads [13, 23].

Based on web server and software development workloads, the CFFS can outperform Ext4 by up to 27%. This suggests that the approach of relaxing the file-to-metadata mapping is promising.

2 OBSERVATIONS

In addition to the limitations of caching and prefetching mentioned above, the observations mentioned below led to the CFFS design:

Frequent access to small files: Studies [13, 24] have shown that small files receive most file references. Our in-house analysis of a desktop file system confirmed that more than 80% of accesses are to files smaller than 32 KB. Furthermore, nearly 40% of the access time required to access a small file on a disk can be attributed to metadata access. Thus, reducing this access overhead may lead to a large performance gain.

Redundant metadata information: A traditional file is associated with its own physical metadata, which tracks information, such as the locations of file blocks, access permissions, and timestamps. However, many files share similar file attributes, as the number of file owners, permission patterns, and so on are limited. A prior paper [10] showed up to a 75% metadata compression ratio for a typical workstation. Thus, we foresee many opportunities to reduce redundant metadata information. While the reduction of the storage of metadata needed on a storage device is rather small, reducing the number of metadata IOs needed to access and cache the same amount of information can make a significant performance difference.

Files accessed in groups: As many prior studies on disk-layout optimizations and prefetching [8, 16, 17, 18] have suggested, files tend to be accessed together. For example, webpage access typically involves accessing many referenced files. However, naïve optimizations that exploit file grouping may not yield automatic performance gains, as the process of identifying and grouping files incurs overhead. Thus, grouping needs to be prioritized for files that are accessed together frequently.

The overhead of per-file access is high. Figure 2.1 shows that, for a constant number of bytes (e.g., 10MB) being read from an SSD device, the read access time increases linearly as 10MB is divided into more files. This performance characteristic is shared across a variety of file systems. This simple experiment shows that the per-file overhead is high enough to mask the effect of varying small file sizes, where the small files often represent the majority of file references [13, 24].

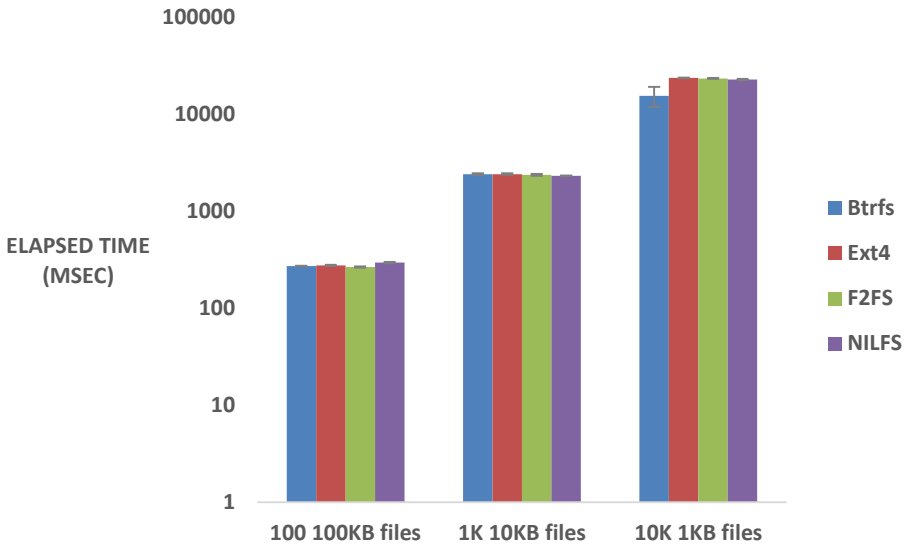

Figure 2.1: Time taken to read 30MB divided into a different number of files for Btrfs, Ext4, F2FS, and NILFS.

Furthermore, we conducted a 2^2x5 factorial design experiment [15] for the same file systems. The first factor is the file size, with two levels of 1KB and 100KB. The second factor is the number of files, with two levels of 10 and 1,000. The experiments were repeated five times. For various file systems, the results show that up to 98% of read access time is attributable to the number of files. This experiment again confirms the high overhead of logical file abstraction.

These observations raise the question of whether we can improve performance by consolidating small files that are accessed together. This is achieved through our approach of decoupling the one-to-one mapping of logical files to their physical representation of data and metadata.

3 COMPOSITE-FILE FILE SYSTEM

We introduce the CFFS, which allows multiple small files to be combined to share a single i-node. The consolidation of separate small files and their i-nodes reduces the number of disk accesses and improves performance.

3.1 Design Overview

The CFFS introduces an internal physical representation called a ***composite file***, which holds the contents of small files, or ***subfiles***, that are frequently accessed together. A composite file is oblivious to end users and is associated with a single composite i-node shared among small files. The original metadata stored in small files' i-nodes are deduplicated, consolidated, and stored as extended attributes of a composite file. The metadata attributes of individual small files can still be reconstructed, checked, and updated, so that the legacy access semantics (e.g., types, permissions, timestamps) are unchanged. The extended attributes also record the locations within the composite file for individual small files. In addition, since both the composite file's i-node and the extended attributes of the composite file are journaled, metadata and data updates to subfiles will be protected and ordered to the storage with the same reliability semantics. With this representation, the CFFS can translate a physical composite file into multiple logical files. By ordering the subfiles within a composite file according to access patterns, a composite file can transparently exploit the legacy VFS prefetching mechanisms to potentially prefetch the entire composite file (across subfiles) as a unit.

The CFFS can be configured in three ways to identify file candidates for forming composite files. One scheme is ***directory-based consolidation***, where all files within a directory (excluding subdirectories) form a composite file. Another scheme is ***embedded-reference consolidation***, where file references within the file contents are extracted to identify files that can form composite files. The third is ***frequency-mining-based consolidation***, where file references are analyzed through set frequency mining [2], so that files that are often accessed together form composite files.

3.2 Data Representation

The content of a composite file is formed by concatenating related small files. All subfiles within a composite file share the same i-node, as well as indirect blocks, doubly indirect blocks, and so on. The maximum size limit of a composite file is not a concern, as composite files are designed to group small files. If the total size of subfiles exceeds the maximum file size allowed by the file system, we split the composite file into two.

Often, the first subfile in a composite file is the entry point, and its access will trigger the prefetching of the remaining composite file. For example, when a browser accesses an `html` file, it loads a `css` file and flash scripts automatically. The `html` file can serve as the entry point and prefetching trigger of this three-subfile composite file. For frequency-based consolidation, the ordering of subfiles reflects how the subfiles are accessed. Although the same group of files may have different access patterns with different entry points, the data layout is based on the most prevalent access pattern.

3.3 Metadata Representations and Operations

Composite file creation: When a composite file is created, the CFFS allocates an i-node, and concatenates copies of the contents of the subfiles as its data. The concatenation is block-aligned to support `mmap`. The composite file records the composite file block offsets and sizes of individual subfiles as well as their deduplicated i-node information in its extended attributes. The original subfiles are then truncated, with their directory entries remapped to the i-node of the composite file and their original i-nodes deallocated. Thus, end users still perceive individual logical files in the name space, while individual subfiles can still be located (Figure 3.3.1).

I-node content reconstruction: Deduplicated subfile i-nodes can be reconstructed on the fly. By default, a subfile's i-node field inherits the value of the composite file's i-node field, unless otherwise specified in the extended attributes. Thus, if the i-node of a composite file belongs to owner A, all subfiles are assumed to be owned by owner A, unless otherwise specified.

Permissions: At file open, the permission is first checked based on the composite i-node. If this fails, no further check is needed. Otherwise, if a subfile has a different permission stored as an extended attribute, the permission will be checked again. For this scheme to work, the composite i-node will have the broadest permissions across all subfiles. For example, within a composite file, suppose we have a read-only subfile A and a writable subfile B; then, the permission for the composite i-node will be read-write. However, when opening subfile A with a write permission, the read-only permission in the extended attribute will catch the violation.

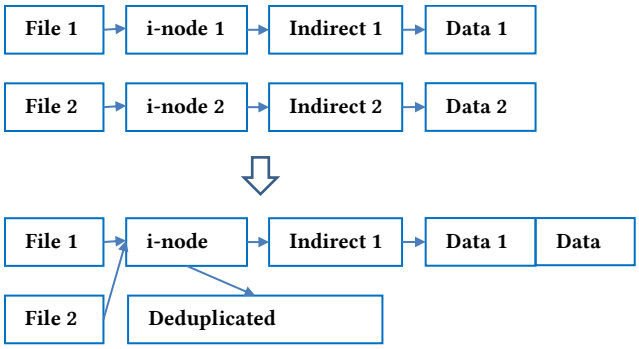

Figure 3.3.1: Creation of the internal composite file (bottom) from the two original files (top).

Timestamps: The timestamps of individual subfiles and the composite file are updated with each file operation. However, during checks (e.g., `stat` system calls), we return the timestamps of the subfiles. If the timestamp information is not essential for the proper function of a system (e.g., `html` files on a webserver), the CFFS can also be configured to use the timestamp of the composite file as opposed to tracking the timestamps of the individual subfiles.

Sizes: For data accesses (e.g., reads and writes), the offsets are translated and bound checked via subfile offsets and sizes encoded in the extended attributes. The size of a composite file is its length; this can be greater than the total size of the subfiles. For example, if a subfile in the middle of a composite file is deleted, the region is marked free within the CFFS without changing the size of the composite file. The composite file's size allows the legacy VFS to prefetch across subfile boundaries within a composite file.

I-node namespace: One naïve scheme to form the i-node namespace involves using upper zero-extended bits as i-node numbers, and lower M bits reserved for subfile IDs. This scheme will reduce the available i-node numbers significantly for files that are not members of composite files. A refined scheme involves partitioning the i-node namespace only when the i-node number exceeds a certain threshold. We refer to this range of i-node numbers as CFFS unique IDs (CUIDs). For example, we can set the threshold to ***0b 0111 1111***, which means an i-node number below ***0b 1000 0000*** is an ordinary i-node number, equivalent to the legacy notion of an i-node. If i-node number is greater than or equal to ***0b 1000 0000*** (e.g., ***0b 1000 0001***), the zero-extended upper bits are viewed as the i-node number to the composite file (e.g., ***0b 1000 0000***) and the lower bits are viewed as the subfile numbers. So, in this case, ***0b 1000 0001*** means the second subfile of the composite file with an i-node number of ***0b 1000 0000***.

Figure 3.3.2 illustrates how i-node numbers are mapped in a directory, where the i-node number exceeds the threshold of ***0b 0111 1111*** indicating a composite file. Since File 1's i-node number is below the threshold, it is a regular file and the file name is mapped to a regular i-node. Files 2 and 3 illustrate how traditional hardlinks are supported, where multiple file names are mapped to the same i-node number. In this case, both i-node numbers are below the threshold. Files 4, 5, and 6 are members of a composite file, and all i-node numbers exceed the threshold. Since the zero-extended upper bits for these files are ***0b 1000 0000***, the names of Files 4, 5, and 6 are mapped to the same composite i-node number, ***0b 1000 0000***. The last seven bits of the i-node numbers (***0b 000 0000*** and ***0b 000 0001***) denote the subfile numbers within the composite file. Since Files 5 and 6 have the same subfile number, they are hardlinks.

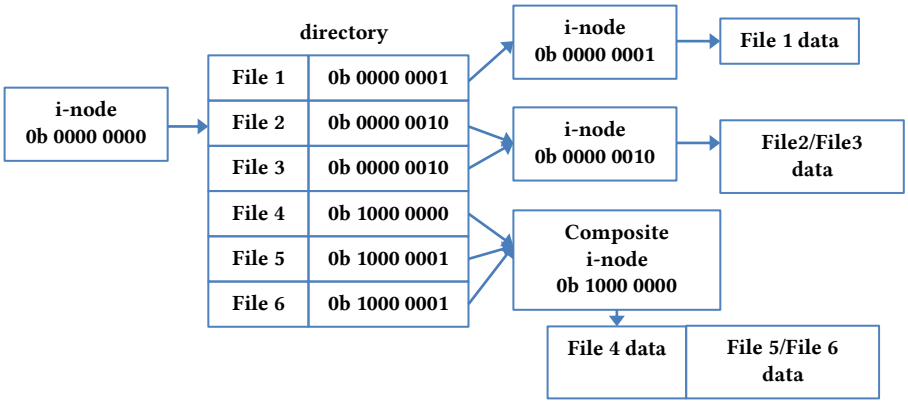

Figure 3.3.2: Example of CFFS directory mapping, where an i-node number greater than 0b 0111 1111 indicates a composite file.

Subfile lookups and renames: If a name in a directory is mapped to a CUID, a subfile's attributes can be looked up via the subfile ID. If the rename operation does not involve moving a subfile from one composite file to another, renaming proceeds as if a CUID were an i-node number in a non-CFFS system (no changes to the CUID). However, if the rename operation involves moving a subfile such that there is a conversion of a composite subfile to a regular file or of a regular file to a composite subfile, the CFFS changes the CUID of the file to be moved, and we need to store backpointers [7] to update all names mapped to the CUID.

The changes in CUID may break applications (e.g., backups) that uniquely identify a file by its i-node number. However, today's file systems can also lead to different files sharing the same i-node number at different times; the CFFS design amplifies the reasons that applications should not assume that an i-node number is a unique property of a file.

Subfile and subfile membership updates: When a subfile is added to a composite file, it is appended to the composite file. When a subfile is deleted from a composite file, the corresponding allocation mapping information within the CFFS extended attributes is removed, so the regions occupied by the subfile can be remapped or allocated to newly created subfiles.

Subfile open/close operations: An open/close call to a subfile is the same as an open/close call to the composite file, with the file-position pointer translated accordingly.

Subfile write operations: In-place updates are handled the same way as those in a traditional file system. However, if an update involves growing a subfile in the middle of a composite file and no free space is available at the end of the subfile, we move the updated subfile to the end of the composite file. This scheme exploits the potential temporal locality wherein a growing subfile is likely to grow in the near future.

Space compaction: The composite file compacts its space when half of its allotted size is freed.

Hardlinks: Different names in directories can be mapped to the same i-node number or CUID.

Locks: To avoid lock contentions, we built a two-level locking mechanism. The composite file lock can be shared for concurrent accesses to subfiles, and each subfile has its own lock. The composite-file lock becomes exclusive when updates can affect the consistency among multiple subfiles. (e.g., updating the subfile memberships, subfile offsets within the composite file, etc.).

For example, if subfiles A and B of a composite file are being updated, each subfile is locked for writing (one writer per subfile). The lock of the composite i-node is initially shared. If subfile A is updated in place to an already mapped block, it can update its file size and timestamps in the composite i-node without upgrading the composite i-node's lock, since different single writer subfiles update disjoint fields of composite-file metadata. However, if subfile B is updated and involves remapping the subfile's location when B grows beyond the current allocated range, the lock of the composite i-node becomes exclusive while making this update, then becomes shared after the update. This temporary lock upgrade is necessary, since

when different subfiles are remapped, they need to agree on the end location of the composite file, among other things. Note that subfile A can retain its exclusive lock, since the remapping operations by subfile B are not in conflict with in-place updates of subfile A. Should subfile A make changes that involve growing the subfile and remapping during subfile B's remapping operation, it must wait until the completion of subfile B's operation. If subfile A is deleted, the composite i-node's lock becomes exclusive again for the duration of this update and becomes shared after the update. The compaction operation requires exclusive locks on all subfiles and the composite i-node, since the reorganization involves making changes for the entire composite file.

3.4 Identifying Composite File Membership

3.4.1 Directory-based consolidation. Given that legacy file systems have deep-rooted spatial locality optimizations revolving around directories, a directory is a good approximation of file access patterns and for forming composite files. Currently, this consolidation scheme excludes subdirectories.

Directory-based consolidation can be performed on all directories without prior analyzing or tracking file references. After the deployment, subsequent file references are tracked to update the composite file membership (e.g., adding a file into a directory, moving a file across directories, etc.). The directory-based scheme will not capture file relationships across directories. In addition, many composite files may be formed even when many directories are rarely accessed.

3.4.2 Embedded-reference-based consolidation. Embedded-reference-based consolidation identifies composite file memberships based on embedded file references in files. For example, more than one hyperlink may be embedded in an `html` file, and a web crawler is likely to access each web page via these links. In this case, we consolidate the original `html` file and the referenced files. Similar ideas can be applied to the source code compilation. We can extract the dependency rules from the `Makefile` and consolidate source files that lead to the generation of the same binary target. Since file updates may break dependency, we need to continuously or periodically sift through modified files to update composite file memberships.

The embedded-reference-based scheme can identify related files accessed across directories, but it may not be easy to extract embedded file references beyond text-based file formats (e.g., `html`, source code). In addition, knowledge of specific and evolving file formats is required.

3.4.3 Frequency-mining-based consolidation. Frequency-mining-based consolation directly identifies files that are frequently accessed together. We applied a variant of the Apriori algorithm [2]. The key observation is that if a set of files is frequently accessed together (its number of occurrences exceeds a threshold), its subsets must be accessed frequently as well (the Apriori property). Figure 3.4.3.1 illustrates the algorithm with an access stream to files A, B, C, D, and E.

Initial pass: First, we count the number of accesses for each file, and then we remove files with counts less than a threshold (say two) for further analysis.

Second pass: After the first pass, for the files that exceed the specified threshold, we permute, build all possible two-file reference sets, and count their occurrences. Whenever file A is accessed right after B, or vice versa, we increment the count for file set {A, B}. Sets with counts lower than the threshold are removed (e.g., {B, D}).

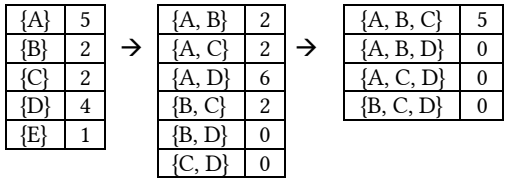

Figure 3.4.3.1: Steps for the Apriori algorithm to identify frequently accessed file sets for a file reference stream E, D, A, D, A, D, A, B, C, A, B, C, A, D.

Third pass: For the files that have not been eliminated, we can generate all three-file reference sets based on the remaining two-file reference sets. However, if a three-file reference set occurs frequently, all its two-file reference sets also need to occur frequently. Thus, file sets like {A, B, D} are pruned, since {B, D} is eliminated in the second pass.

Termination: After the third pass, as we can no longer generate four-file reference sets, the algorithm terminates. Now, if a file can belong to multiple file sets, we return sets {A, B, C} and {A, D} as two frequently accessed sets. Sets like {A, B}, {A, C}, and {B, C} are removed, as they are already subsets of {A, B, C}.

With a larger input file stream, the process may run for more than three passes to generate larger reference sets. To cap the algorithm running time, we may also choose to terminate the algorithm by either setting a maximum number of passes or a maximum running time. In practice, the average number of passes is from 4 to 5.

Variations: An alternative is to use a normalized threshold, or ***support***, which is the percentage of set occurrences (the number of the occurrences of a set divided by the total occurrences, ranging between 0 and 1).

Instead of tracking file sets, we can also track file reference sequences to determine the entry point and the content layout of the composite file.

We currently disallow overlapping of file sets to avoid the complexity of replication and maintaining consistency. To choose a subfile's membership between two composite files, the decision depends on whether a composite file has (1) more subfiles, (2) higher support, and (3) more recent creation timestamps. These three rules are applied in that order.

Optimizations: One practical problem of using the Apriori algorithm is that as the number of passes increases to find longer frequent sequences, the memory required can become prohibitive to holding the number of potential permuted candidate sequences. We trimmed the memory requirement via (1) using higher thresholds to reduce the number of frequently occurring sequences at each pass, (2) using only the file reference sequences encountered instead of building the complete set of permuted sequences, and (3) using a fixed-size sliding window to limit the number of file references to be analyzed in a batch.

Another problem is that our gathered traces contain file references from concurrent reference streams. The overlapping of concurrent reference streams can reduce the chance of identifying frequently referenced file sequences when they are performed independently. Thus, we separated the traces by PID or IP address and concatenated the separated traces before analyzing them via the Apriori algorithm.

Our algorithm offers the choice of grouping files either by how they are referenced in sequence or by how they are referenced as a set. If we look up exact sequences, multiple Apriori table entries may contain the same set of files referenced in a different order; thus, the size of the Apriori tables will be larger. Instead, we can look up by frequently referenced file sets to reduce the memory requirement. The design space of set lookups may involve sorting files within a file set into canonical order prior to matching or may involve constructing Bloom filter bitmaps [5] based on the file set and then matching the bitmaps. Either approach can be too heavy in terms of computation or implementation complexity. Instead, we used commutative hash functions (i.e., hash(A, B) = hash(B, A)) to speed up set lookups.

Consolidation based on frequency mining can identify composite file candidates based on dynamic file references. However, the computation and memory cost limits its application to more popular file reference sequences, and it requires a learning period before the system can reap the performance benefit. For future work, we seek to explore more advanced, lighter-weight data-mining techniques to reduce the cost and learning period and improve the benefits of the CFFS.

4 IMPLEMENTATION

The two major components of the CFFS are the composite file membership generator tool and the CFFS. We prototyped the CFFS in the user space via the FUSE (v2.9.3) framework [26] (Figure 4.1) running on Linux 3.16.7. The CFFS was stacked on Ext4, to leverage legacy tools and features such as persistence bootstrapping (e.g., file-system creation utilities), extended attributes, and journaling.

The CFFS periodically takes recommendations from the composite file membership generator tool to create composite files. We leveraged hard-link-like mechanisms to enable mapping multiple file names to the same composite i-node. Basically, if multiple names are mapped to the same i-node number, they are interpreted as hardlinks. If multiple names are mapped to the same i-node number with a zero-extended prefix that exceeds the specified threshold, these files are mapped to a composite file (see Section 3.2). Currently, we use an i-node number threshold of 15, so that the last 4 bits of the i-node number denote the subfiles' IDs. This also means we currently support 16 subfiles per composite file. Should a composite file require additional subfiles, we break the composite file into multiple composite files.

We intercepted all file-system-related calls due to the need to update the timestamps of individual subfiles. We also needed to ensure that various accesses used the correct permissions (e.g., `open` and `readdir`), translated subfile offsets and sizes (e.g., `read` and `write`), and timestamps (e.g., `getattr` and `setattr`). The actual composite file, its i-node, and its extended attributes were stored by the underlying Ext4 file system as a regular file with a rich extended attributes set. We did not implement the backpointers for the rename operations, which account for less than 0.01% of the replayed file system operations. The CFFS was implemented in C++ with ~1,900 lines of code.

For the directory-based consolidation, we used a Perl script to list all the files in a directory as composite file members. For the embedded-reference-based scheme, we focused on two scenarios. For the web server workload, we consolidated the `html` files and their immediately referenced files. In the case of conflicting composite file memberships, the preference was given to `index.html`, and then the `html` that first included the file. The other scenario was the source code compilation. We used the `Makefile` file as a guide to consolidate source code files. For the frequency-mining-based scheme, the membership generator tool took either the Apache `http` access log or the `strace` output. The generator implemented the Apriori algorithm, with the support parameter set to 5%. The analysis batch size was set to 50K references. The parameters were chosen based on empirical experience to limit the amount of memory and processing overhead. The generator code contained ~1,200 semicolons.

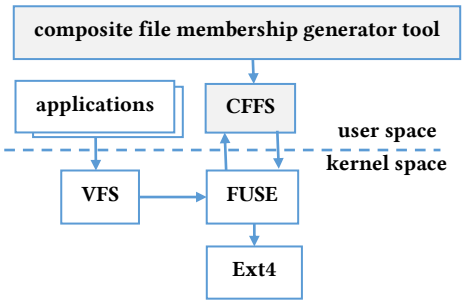

Figure 4.1: CFFS components (shaded) and data path from applications to the underlying Ext4.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

We compared the performance of the CFFS stacked on Ext4 via FUSE with the baseline Ext4 file system. In the case of Ext4, the IO requests were routed through an empty FUSE module, so that both settings contained the effects of FUSE. The experiments were conducted on a Dell T3600 workstation (Table 5.1). The microbenchmark included workloads that stressed various aspects of the CFFS (e.g., metadata updates and migrating updated subfiles to the end of a composite file). The macrobenchmark replayed two real-world traces to compare various methods for forming composite files. Each experiment was repeated five times; the results are presented at 90% confidence intervals.

Table 5.1. Experimental Platform.

Processor	2.8GHz Intel® Xeon® E5-1603, L1 cache 64KB, L2 cache 256KB, L3 cache 10MB
Memory	2GBx4, Hyundai, 1067MHz, DDR3
Disk	250GB, 7200 RPM, WD2500AAKX with 16MB cache
Flash	200GB, Intel SSD DC S3700

5.2 Microbenchmark

To show the performance of the CFFS in various dimensions, eight workloads were constructed: sequential read, random read, sequential rewrite, random rewrite, general write with extension and compaction, pure metadata read, and create and remove operations. A working set was generated by creating 80K composite files, each with eight 16KB subfiles (10GB total). For Ext4, 640K 16KB files (10GB) were generated. For sequential read operations, each subfile was read sequentially; there was no fixed order for random reads. The same ordering applied to the rewriting workloads. For general writing operations, subfiles were chosen randomly. The size of the written data was uniformly distributed from 0KB to 64KB, resulting in in-place rewriting or subfile migration to the end of the composite file. This enabled further updates by leaving a "hole" in the middle of the composite file. For the metadata-only operation, the `stat` system call was applied to randomly chosen subfiles. For creation and removal, the underlying Ext4 file system was tested to compare creation and deletion of large numbers of files with the same total size. Each benchmark was exercised 10,000 times on HDD and 100,000 times on SSD to ensure that the benchmark would run for a sufficient duration (e.g., > 30 seconds); the completion times were then reported.

Figures 5.2.1 and 5.2.2 show the completion time comparison between the CFFS and Ext4 on HDD and SSD, respectively.

On HDD, the CFFS completed the sequential and random read operations 31% and 21% faster than Ext4. The CFFS provided faster sequential reads due to cross-boundary prefetching; for random read operations, the 21% benefit was attributed to metadata consolidation. The CFFS consolidated metadata for small files and was more likely to avoid reading metadata from the disk. By concatenating the subfile data, the CFFS enabled prefetching across subfiles, which further improved the throughput. These results validate the benefits of CFFS for both cross-boundary prefetching and metadata consolidation. However, the actual contribution of each component was dependent on the actual testing setup.

A similar trend was found for rewrite operations. The CFFS completed the sequential and random rewrite operations 36% and 40% faster compared to Ext4, respectively. Since it was restricted to in-place updates, the CFFS handled the in-place write well; these were similar to the read operations but included metadata updates. However, for the same amount of written data, the CFFS handled the metadata updates more efficiently by consolidating common metadata attributes, as illustrated by the `stat` microbenchmark: the CFFS was nearly 92% faster than Ext4. This shows that the CFFS excels at metadata- and read-only operations.

The CFFS did not perform better than Ext4 for writing with space compaction operations because it potentially needed to migrate a subfile to the end of a composite file. Allocating more space can be a costly operation in a near-full file system that may further increase the seek time for future read or write operations. In addition, the CFFS periodically compacts a composite file if it is too sparsely populated. This operation creates further overhead when exercising the benchmark; thus, for HDD write operations that involve migrations and compactions, the CFFS is 20% slower than Ext4.

Because the CFFS was built on top of Ext4, creating a composite file is translated to creating an Ext4 file followed by appending the contents of subfiles, and deleting a composite file is translated to deleting an Ext4 file containing the contents of subfiles. Thus, creating and removing composite files translates to creating and removing Ext4 files of different sizes. In comparison, Ext4 must create and delete eight times as many files without the use of composite files. Composite file creation and deletion operations were triggered during the composite file generation and regeneration processes, which appeared either as part of the initial setup or as incremental overhead. These tasks required 21% and 13%, respectively, of the time spent on the same operations using Ext4. This implied that the additional overhead was much smaller compared to that of Ext4 when the system was deployed for generating or removing composite files.

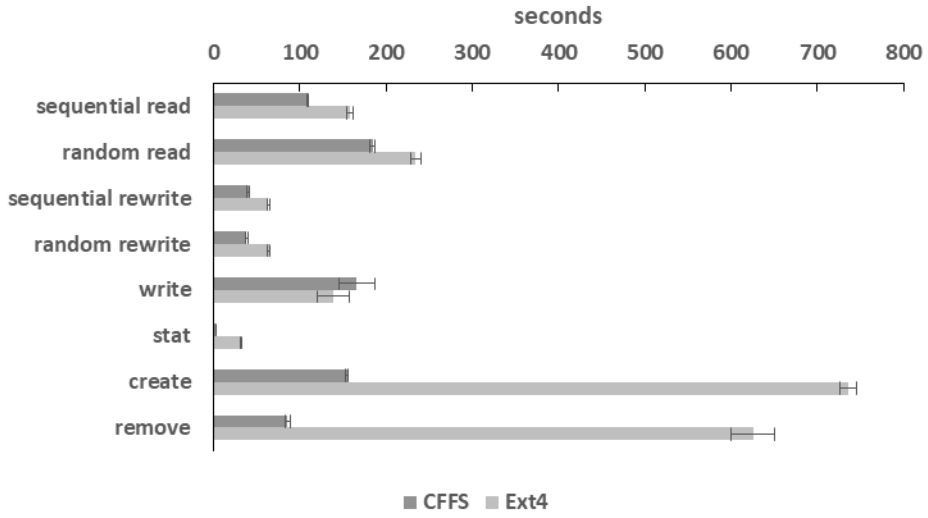

Figure 5.2.1: Microbenchmark results for HDD.

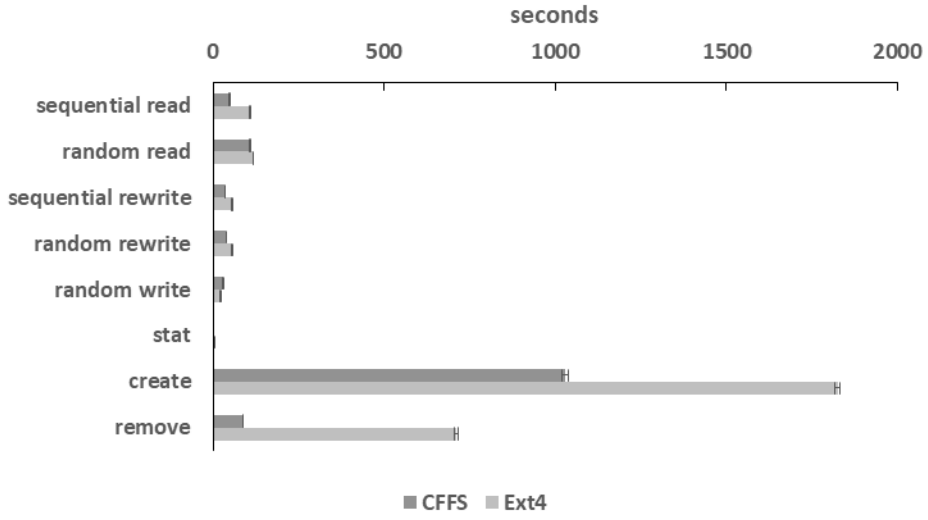

Figure 5.2.2: Microbenchmark results for SSD.

In the case of SSD, a similar trend was observed. The CFFS performed better for all operations except for writes that involved migrations and compactions. However, the performance gain for random operations was lower than that of HDD because the SSD did not include seek time delay. For example, SSD random reads and random rewrites on the CFFS were only 8% and 32% faster, respectively, compared to SSD operations on Ext4. The reasons are similar to those in the case of HDD, where the metadata updates are

carried out more efficiently by consolidating common metadata attributes. This trend also applied to `stat` operations, where the gap between the CFFS and Ext4 was less than that for HDD, at about 58%. This is reasonable because the latency of reading data from the SSD device is much lower than that of HDD. For sequential operations, the SSD performance gain remained comparable to that of HDD.

5.3 Web Server Trace Replay

The first trace replay was based on an `http` log gathered from our departmental web server (01/01/2015 - 03/18/2015). The trace contained 14M file references to 1.0TB of data. Among the references to files, 3.1M files were unique, holding 76GB of data.

We conducted multi-threaded, zero-think-time trace replays on a storage device. We also skipped trace intervals with no activities. The replays were performed on the Dell workstation.

Prior to each experiment, we rebuilt the file system with dummy content. For directory and embedded-reference-based schemes, composite file memberships were updated continuously. For the frequency-mining-based consolidation, the analysis was performed in batches, but the composite files were updated daily.

HDD performance: Figure 5.3.1 shows the CDF of web server request latency for a disk, measured from the time a request is sent to the time a request is completed.

The original intent of our work was to reduce the number of metadata IOs and improve the layout for small files that are frequently accessed together. However, the benefit of fewer accesses to consolidated metadata displays itself as metadata prefetching for all subfiles, and the composite-file semantics enables cross-file prefetching, resulting in much higher cache-hit rates.

The embedded-reference-based consolidation performed best, with 62% of requests serviced from the cache; this was 20% higher than the result for Ext4. Thus, composite files created based on embedded references captured the access pattern more accurately. The overall replay time was also reduced by ~20%.

The directory-based composite files could also improve the cache hit rate by 15%, reflecting the effectiveness of directories for capturing spatial localities.

The frequency-mining-based consolidation performed worse than the directory-based consolidation. We examined the trace and found that 48% of references were made by crawlers, and the rest by users. Thus, the bifurcated traffic patterns for the mining algorithm formed less aggressive file groupings, yielding reduced benefits.

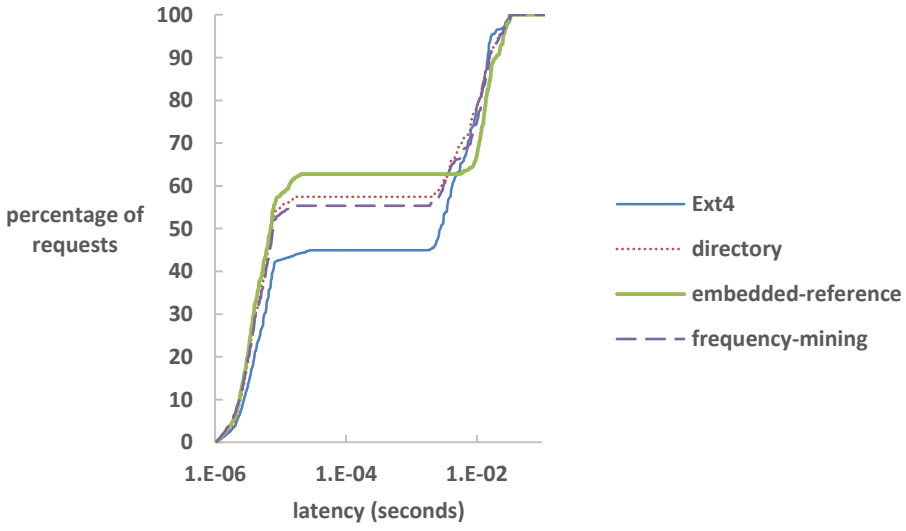

Figure 5.3.1. Webserver request latency for HDD.

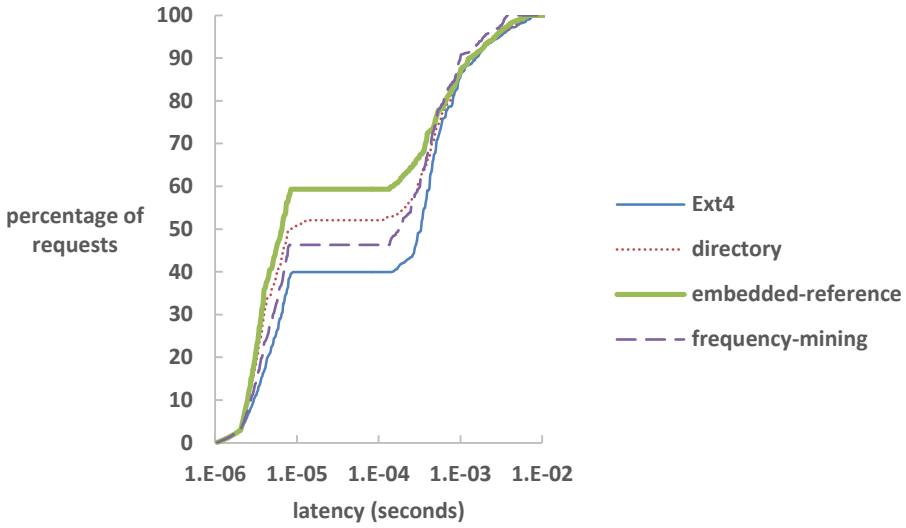

Figure 5.3.2. Webserver request latency for SSD.

SSD performance: Figure 5.3.2 shows the CDF of web server request latency for an SSD. Compared to a disk, the relative trends are similar, with request latency times for cache misses reduced by two orders of magnitude due to the speed of the SSD. As the main performance gains were caused by referencing data and metadata that were already prefetched and cached through the composite file granularity, this 20% performance benefit can be seen for both HDD and SSD devices.

5.4 Software Development File-system Trace Replay

The second trace was gathered via `strace` from a software development workstation (11/20/2014 – 11/30/2014). The trace contained over 240M file-system-related system calls to 24GB of data. Among the references to files, 291,133 files were unique, holding 2.9GB. Between the read and write operations, 59% were reads, and 41% were writes.

For this replay, it is more difficult to capture the latency of individual file-system call requests, as many are asynchronous (e.g., writes), and calls like `mmap` omit the details of the number of requests sent to the underlying storage. Thus, we summarize our results with overall elapsed times, which include all overheads of composite file operations, excluding the initial setup cost for the directory- and embedded-reference-based schemes (Figure 5.4.1).

HDD performance: The embedded-reference-based scheme had poor coverage, as many references are unrelated to compilation. Therefore, the elapsed time was closer to that of Ext4. Directory-based consolidation achieved a 17% elapsed time reduction, but the frequency-mining-based scheme could achieve 27% because the composite files included files across directories.

SSD performance: The relative performance trend for different consolidation settings was similar to that of HDD. Similar to the web traces, the gain was up to 20%.

When comparing the performance improvement gaps between the HDD and SSD experiments, up to an 11% performance gain under HDD could not be realized by SSD, as an SSD does not incur disk seek overheads.

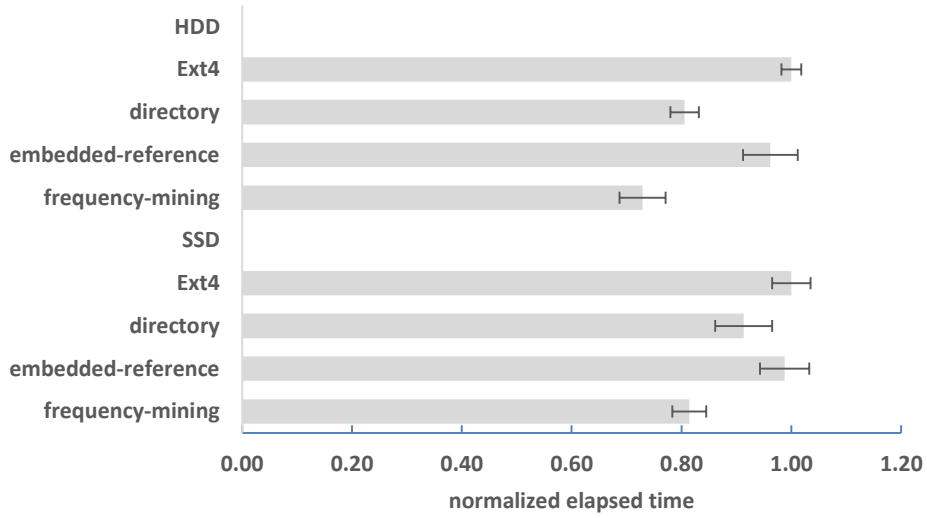

Figure 5.4.1: Elapsed times for the software development file system trace replay.

5.5 Overheads

Directory-based and embedded-reference-based schemes: Directory-based and embedded-reference-based schemes incur an initial deployment cost to create composite files based on directories and embedded file references. The initial cost of the embedded-reference scheme depends on the number of file types from which file references can be extracted. For our workloads, this cost was anywhere from 1 to 14 minutes.

As for the incremental cost of updating composite file memberships, adding members involves appending subfiles to the composite files. Removing members mostly involves metadata updates. A composite file is not compacted until half its allotted space is freed. As the trace replay numbers already include this overhead, this cost seems negligible as the frequency of compaction operation is fairly low and is offset by the benefits.

The frequency-mining-based scheme: The trace-gathering overhead was below 0.6%, and the memory overhead for trace analyses was within 200MB, for an average of 15M lines of daily logs.

The frequency-mining-based scheme involves learning from recent file references. It took a few weeks of replay days to reap the full benefit of this scheme (Figure 5.5.1). Near the beginning, the benefit was close to zero or even negative due to the overhead of analyzing and forming composite files. However, the benefit steadily increased and stabilized after six weeks.

In terms of daily overhead, since the frequency-mining-based scheme does not involve an initial deployment, the cost is incremental during the process of analyzing trace logs and forming composite-file memberships. The composite-file membership churn rate slowed and stabilized after a few days, further reducing the incremental cost. At the steady state, reorganizing composite file memberships involves copying on average 148 files (44MB) each day. Figure 5.5.2 shows the daily incremental cost for analyzing the web log. During the first few days, the log processing could take up to a few hundred seconds. However, after this initial period, the daily overhead tended to be less than 100 seconds, unless new, frequently occurring sequences were found.

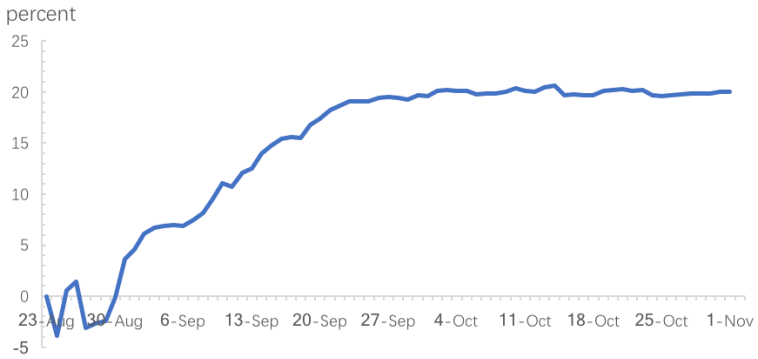

Figure 5.5.1: Performance gain of the frequency-mining-based scheme over time.

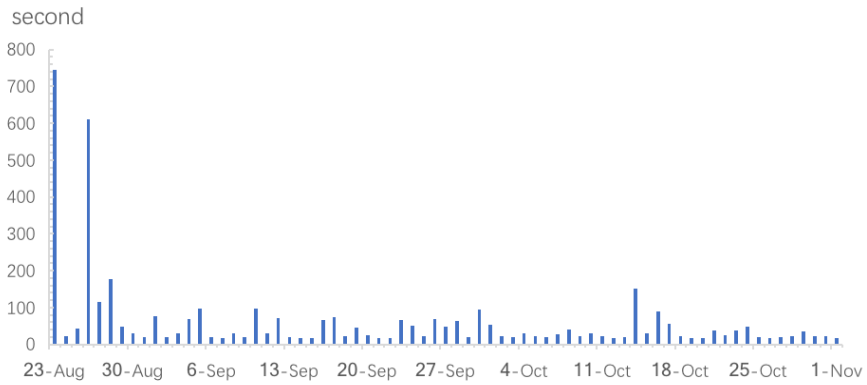

Figure 5.5.2: Daily overhead for analyzing logs.

5.6 Discussion and Future Work

Composite files can benefit both read-dominant and read-write workloads using different storage media, suggesting that the performance gains are mostly due to the reduction in the number of IOs (~20%). The performance improvement gaps between the SSD and HDD suggest that the performance gains due to reduced disk seeks and modified data layouts may reach ~10%.

Based on this experience, we are intrigued by the relationship among ways to form composite files, the performance effects of consolidating metadata and the prefetching enabled by the composite files. Future work will explore additional ways to form composite files and quantify their interplay with different components of performance contributions. In addition, future research will explore the full ramifications of metadata compression, concurrency, and security.

At the time this work was conceived, the study on the FUSE overhead was not yet published [28]. However, we acknowledge that the measured performance can be distorted due to the presence of the FUSE layer. As future work, we will port the CFFS implementation to the kernel and measure the raw performance.

Currently, the decision on the membership of composite files does not take advantage of knowledge of file types. For example, files that are updated once and never updated again would be ideal candidates to form composite files. Conversely, log files would be poor candidates, and small files are likely to be better candidates than large files. Although we currently have tools to allow users to specify composite files directly, in the future, we will explore the automated use of heuristics to determine composite file membership.

The current prototype of the CFFS does not take advantage of the fallocate function, nor does it support sparse files, which can provide further performance improvements. We will incorporate these features in our future versions.

6 RELATED WORK

Small file optimizations: While our research focused on the many-to-one mapping of logical files and physical metadata, this work is closely related to ways to optimize small-file accesses by reducing the number of metadata accesses. Some early works involved collocating a file's i-node with its first data block [20] and embedding i-nodes in directories [11]. Later, hFS [30] uses separate storage areas to optimize small file and metadata accesses. UmbrellaFS [12] tries to match the diversity of file access characteristics via a stackable file system. Btrfs [23] packs metadata and small files into copy-on-write b-trees. TableFS [22] packs metadata and small files into a table and flushes 2MB logs of table entries, which are organized via log-structured merge trees. The CFFS complements many existing approaches by consolidating i-nodes for files that are often accessed together.

The idea of accessing subfile regions and consolidating metadata has also been explored in the parallel and distributed computing domain, where CPUs on multiple computers need to access the same large data file [6, 9, 29]. Facebook's photo storage [4] leverages the observation that the permissions of images are largely the same and can be consolidated. However, these mechanisms are tailored for mostly homogeneous data types. With different ways to form composite files [1], the CFFS can work with subfiles with more diverse content and access semantics.

File packing: Archival files created by utilities such as TAR and ZIP share similar flavors of composite files, where the bulk accesses of the tarballs are more efficient than accessing the individual files within the tarballs. Due to legacy reasons, some archival file formats such as TAR, lack a centralized content indexing structure to support random access [27]. A ZIP file does contain a centralized directory and supports optional data compression [20]. On the other hand, the CFFS composite files are much simpler and lighter in weight compared to feature-rich archival files and are integrated into the file system.

Prefetching: While a large body of work can be found to improve prefetching, perhaps C-Miner [18] is closest to our work. In particular, C-Miner applies frequent sequence mining at the block level to optimize the layout of the file and metadata blocks and improve prefetching. However, the CFFS reduces the number of frequently accessed metadata blocks and avoids the need for a large table to map logical to physical blocks. In addition, our file-system-level mining deals with significantly fewer objects and associated overheads. DiskSeen [8] incorporates the knowledge of disk layout to improve prefetching across file and metadata boundaries. The CFFS proactively reduces the number of physical metadata items and alters the storage layout to promote sequential prefetching. [25] observed that by passing high-level execution contexts (e.g., thread, application ID) to the block layer, the resulting data mining can generate prefetching rules with longer runs under concurrent workloads. Since the CFFS performs data mining at the file-system level, we can use PIDs and IP addresses to detangle concurrent file references. Nevertheless, CFFS's focus is on altering the mapping of logical files to their physical representations, and it can adopt various mining algorithms to consolidate metadata and improve storage layouts.

7 CONCLUSIONS

We presented the design, implementation, and evaluation of CFFS to explore the many-to-one mapping of logical files and metadata. The CFFS can be configured differently to identify files that are frequently accessed together, and it can consolidate their metadata. The results showed up to a 27% performance improvement under two real-world workloads. The CFFS experience demonstrates that the approach of decoupling the one-to-one mapping of files and metadata is promising and can lead to many new optimization opportunities.

ACKNOWLEDGMENTS

We thank Garth Gibson and anonymous reviewers for their invaluable feedback. We also thank Erika Dennis for contributing some of the preliminary numbers and Weisu Wang for his comments. This work was sponsored by FSU and NSF CNS-144387. The opinions, findings, conclusions, or recommendations expressed in this document do not necessarily reflect the views of FSU, NSF, or the U.S. Government.

REFERENCES

[1] M. Abd-El-Malek, W. V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. (2005). "Ursa Minor: Versatile Cluster-based Storage". Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05). San Francisco, CA, 59-72.

[2] R. Agrawal and R. Srikant. (1994). "Fast Algorithms for Mining Association Rules". Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94). Santiago de Chile, Chile, 487-499.

[3] R. Albrecht. (2017). "Web Performance: Cache Efficiency Exercise". Retrieved from https://code.facebook.com/posts/964122680272229/web-performance-cache-efficiency-exercise/.

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. (2010). "Finding a Needle in Haystack: Facebook's Photo Storage". Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 10). Vancouver, BC, Canada, 47- 60.

[5] B. Bloom. (1970). "Space/Time Tradeoffs in Hash Coding with Allowable Errors". Communications of the ACM, 13(7):422-426,.

[6] S. Chandrasekar, R. Dakshinamurthy, P.G. Seshakumar, B. Prabavathy, and B. Chitra. (2013). "A Novel Indexing Scheme for Efficient Handling of Small Files in Hadoop Distributed File System". Proceedings of the 2013 International Conference on Computer Communication and Informatics (ICCCI'2013). Coimbatore, India, 1-8. DOI:http://dx.doi.org/10.1109/iccci.2013.6466147

[7] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. (2012). "Consistency Without Ordering". Proceedings of the 11st USENIX Conference on File and Storage Technologies (FAST '12). San Jose, CA, 101-116.

[8] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. (2007). "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch". Proceedings of the 2007 USENIX Annual Technical Conference (ATC' 07). Santa Clara, CA, 261-274

[9] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li. (2010). "A Novel Approach to Improving the Efficiency of Storing and Accessing Smaller Files on Hadoop: a Case Study by PowerPoint Files". Proceedings of the 2010 IEEE International Conference on Services Computing. 65-72

[10] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt. (2004). "MRAMFS: A Compressing File System for Non-volatile RAM". Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS' 2004). 596-603.

[11] G. R. Ganger and M. F. Kaashoek. (1997). "Embedded Inode and Explicit Grouping: Exploiting Disk Bandwidth for Small Files". Proceedings of the USENIX 1997 Annual Technical Conference (ATC). 1-17.

[12] J. A. Garrison and A. L. N. Reddy. (2009). "Umbrella File System: Storage Management across Heterogeneous Devices". ACM Transactions on Storage 5(1) Article 3.

[13] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. (2012). "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications". ACM Transactions on Computer Systems 30(3) Article 10 (August 2012).

[14] J. S. Heidemann and G. J. Popek. (1994). "File-System Development with Stackable Layers". ACM Transactions on Computer Systems – Special issue on operating systems principles 12(1):58-89.

[15] R. Jain. (1991). The Art of Computer Systems Performance Analysis. Wiley.

[16] S. Jiang, X. Ding, Y. Xu, and K. Davis. (2013). "A Prefetching Scheme Exploiting Both Data Layout and Access History on Disk". ACM Transactions on Storage 9(3) Article 10. DOI:http://dx.doi.org/10.1145/2508010.

[17] T. M. Kroeger and D. E. Long. (2001). "Design and Implementation of a Predictive File Prefetching". Proceedings of the USENIX 2001 Annual Technical Conference (ATC '01). Boston, Massachusetts.

[18] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Y. Zhou. (2004). "C-Miner: Mining Block Correlations in Storage Systems". Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04). San Francisco, CA.

[19] M. K. McKusick, M. J. Karels, and K. Bostic. (1990). "A Pageable Memory Based Filesystem". Proceeding of USENIX Summer Conference.

[20] S. J. Mullender and A. S. Tanenbaum. (1984). "Immediate Files", Software Practice and Experience (SPE), 14(4):365-368.

[21] PKWARE. (2018). ".ZIP File Format Specification". https://pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.5.TXT.

[22] K. Ren and G. Gibson. (2013). "TABLEFS: Enhancing Metadata Efficiency in the Local File System". Proceedings of the 2013 USENIX Annual Technical Conference (ATC '13). San Jose, CA, 145-156.

[23] O. Rodeh, J. Bacik, and C.Mason. (2013). "BTRFS: The Linux B-tree filesystem". ACM Transactions on Storage 9(3) Article 9 DOI:http://dx.doi.org/10.1145/2501620.2501623

[24] D. Roselli, J. R. Lorch, and T. E. Anderson. (2000). "A Comparison of File System Workloads". Proceeding of 2000 USENIX Annual Technical Conference (ATC '00).

[25] G. Soundararajan, M. Mihailescu, and C. Amza. (2008). "Context-aware Prefetching at the Storage Server". Proceedings of the 2008 USENIX Annual Technical Conference (ATC '08), Boston, Massachusetts, 377-390.

[26] M. Szeredi. (2017). "Filesystem in Userspace". Retrieved from https://github.com/libfuse/libfuse.

[27] Terry M. (2017). duplicity, http://duplicity.nongnu.org/index.html, 2017.

[28] B. K. R. Vangoor, V. Tarasov, and E. Zadok. (2017). "To FUSE or Not to FUSE: Performance of User-Space File System". Proceedings of the 15th USENIX Conference on File and Technologies (FAST).

[29] W. Yu, J. Vetter, R. S. Canon, S. Jiang. (2007). "Exploiting Lustre File Joining for Effective Collective IO". Proceedings of the 7th International Symposium on Cluster Computing and the Grid(CCGRID '07). Rio De Janeiro, Brazil.

[30] Z. Zhang and K. Ghose. (2007). "hFS: A Hybrid File System Prototype for Improving Small File and metadata Performance", Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys '2007), Lisbon, Portugal, 175-187. DOI:https://doi.org/10.1145/1272996.1273016

Received February 2018; revised July 2019; accepted October 2019