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

Shuanglong Zhang, Helen Catanese, and An-I Andy Wang
Computer Science Department, Florida State University

Abstract

Traditional file system optimizations typically retain the one-to-one mapping of logical files to their physical metadata representations. This rigid mapping results in missed opportunities for an entire class of optimizations in which such coupling is removed.

We have designed, implemented, and evaluated a composite-file file system, which allows many-to-one mappings of files to metadata, and we have explored the design space of different mapping strategies. Under webserver and software development workloads, our empirical evaluation shows up to a 27% performance improvement. This result demonstrates the promise of decoupling files and their metadata.

1. Introduction

File system performance optimization is a well-researched area. However, 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 its own i-node on UNIX platforms). Such mapping is desirable because metadata constructs are deep-rooted data structures, and many storage components and mechanisms—such as VFS API [McKusick et al. 1990], prefetching, and metadata caching—rely on such constructs. However, this rigid mapping also presents a blind spot for a class of performance optimizations.

We have designed, implemented, and evaluated the composite-file file system (CFFS), where many logical files can be associated with a single i-node (plus extra information stored as extended attributes). Such an arrangement is possible because many files accessed together share very similar metadata subfields [Edel et al. 2004], which can be deduplicated. Thus, the CFFS can yield fewer metadata accesses to storage, a source of significant overhead for accessing small files, which still dominates the majority of file references for modern workloads [Roselli et al. 2000; Harter et al. 2011].

Based on web server and software development workloads, the CFFS can outperform ext4 by up to 27%, suggesting that the approach of relaxing the file-to-metadata mapping is promising.

2. Observations

The following observations led to the CFFS design:

Frequent access to small files: Studies [Roselli et al. 2000; Harter et al. 2011] show that small files receive the majority of file references. Our in-house analyses of a desktop file system confirmed that 82% of accesses are to files smaller than 32 bytes. Further, 41% of the access time to access a small file on a disk can be attributable 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, etc. However, many files share similar file attributes, as the number of file owners, permission patterns, etc. are limited. Edel et al. [2004] showed up to a 75% metadata compression ratio for a typical workstation. Thus, we see many opportunities to reduce redundant metadata information.

Files accessed in groups: Files tend to be accessed together, as shown by Kroeger and Long [2001], Li et al. [2004], Ding et al. [2007], and Jiang et al. [2013]. For example, web access typically involves accessing many associated files. However, optimizations that exploit file grouping may not yield automatic performance gains, as the process of identifying and grouping files incurs overhead.

These observations beg the question of whether we can improve performance by removing redundant metadata information and accesses among 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 metadata.

3. Composite-file File System

We introduce the CFFS, which allows multiple small files to share a single i-node.

3.1. Design Overview

The CFFS introduces an internal physical representation called a ***composite file***, which holds the content of files that are frequently accessed together, and such files tend to be small. A composite file is invisible to end users and is associated with a single composite i-node shared among small files. The original information stored in small files' inodes are deduplicated 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. With this representation, the CFFS can translate a physical composite file into logical files.

The CFFS can be configured three ways to identify file candidates for forming composite files and consolidating their metadata. The first scheme is ***directory-based consolidation***, where all files within a directory (excluding subdirectories) form a composite file. The second scheme is ***embedded-reference consolidation***, where embedded file references within 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 [Agrawal and Srikant 1994], so that files that are accessed together frequently form composite files.

A composite file is compatible to legacy VFS caching mechanisms and semantics, and the entire composite file can be prefetched as a unit. At the same time, it is also possible to access and update individual files within the composite file.

3.2. Data Representation

The content of a composite file is formed by concatenating small files, referred to as ***subfiles***. All subfiles within a composite file share the same i-node, as well as indirect blocks, doubly indirect blocks, etc. The maximum size limit of a composite file should not be a concern, as composite files are designed to encapsulate small files. Should the sum of subfile sizes exceed the maximum file size limit, we resort to the use of multiple composite files.

Often, the first subfile in a composite file is the ***entry point***, whose access will trigger the prefetching mechanisms to load the remaining composite file or its subfiles into the memory. For example, when a browser accesses an `html` file, it loads a `css` file and flash script. The `html` file can serve as the entry point and prefetching trigger of this three-subfile composite file. For the frequency-based consolidation, the ordering of subfiles reflects how they 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 copies and concatenates the contents of the subfiles as its data. The composite file records the composite file offsets and sizes of individual subfiles as well as their deduplicated i-node information into its extended attributes. The original subfiles then are 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: Deduped subfile i-nodes are 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.

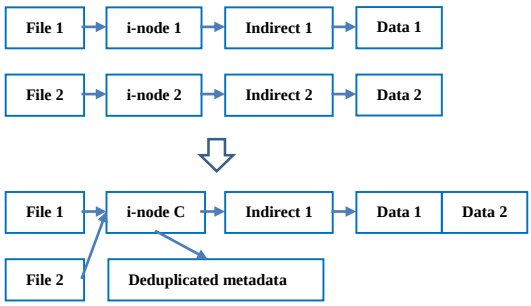

Figure 3.3.1: Creation of the internal composite file (bottom) from the two original files (top).

Permissions: At file open, the permission is first checked based on the composite i-node. If it failed, no further check is needed. Otherwise, if a subfile has a different permission stored as an extended attribute, the permission will be checked again. Therefore, the composite i-node will have the broadest permissions across all subfiles. For example, within a composite file, we have a read-only subfile A, and a writable subfile B, while 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.

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.

Sizes: For data accesses, the offsets are translated and bound-checked via subfile offsets and sizes encoded in the extended attributes. The size field of a composite file's i-node is the length of the composite file, which can be greater than the total size of its subfiles. For example, if a subfile in the middle of a composite file is deleted, the region is freed, but the size of the composite file remains unchanged. In addition, once the entry-point file is accessed, the legacy VFS would prefetch at the level of the composite file.

Subfile and subfile membership updates: When a subfile is added to a composite file, it is concatenated to the end of the composite file. When a subfile is deleted from a composite file, the corresponding data region within the composite file is marked freed. The composite file compacts its space when half of its allotted size is freed.

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 current file-position pointer translated. While lock contentions can be a concern, files detected to be involved in such reference streams can be opted out from forming composite files.

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 that a growing subfile is likely to grow in the near future.

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.

The directory-based consolidation can be performed on all directories without tracking and analyzing file references. However, it will not capture file relationships across directories.

3.4.2 Embedded-reference-based Consolidation

Embedded-reference-based consolidation identifies composite file memberships based on embedded file references in files. For example, hyperlinks 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, while the original `html` subfile works as an entry point. Similar ideas apply to compilation. We can extract the dependency rules from `Makefiles` and consolidate source files that lead to the generation of the same binary. As file updates may break dependency, we need to sift continuously or periodically through updated files to update composite file memberships.

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

3.4.3 Frequency-mining-based Consolidation

The frequency-mining-based consolidation applies a variant of the Apriori algorithm [Agrawal and Srikant 1994]. The key observation is that if a set of files is accessed frequently, its subset must be as well.

Initial pass: Figure 3.4.1 illustrates the algorithm with an access stream to files A, B, C, D, and E. During the initial pass, we count the number of accesses for each file. Files are first eliminated if they are not accessed at least twice. In addition, we can use a threshold (say two), to remove files with fewer accesses from further analyses.

Second pass: For the files that exceed the threshold, we can permute and build all possible two-file reference sets and count their occurrences. Thus, whenever file A is accessed right after B, or vice versa, we increment the count for file set {A, B}. If the access count for a two-file set is less than the threshold, the set is eliminated (e.g., the file set {B, D}).

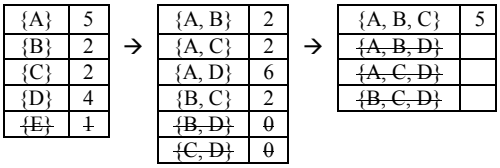

Figure 3.4.1: Steps for the Apriori algorithm to identify frequently accessed file sets for a file reference stream E, D, A, D, A, D, A, B, C, A, B, C, A, D.

Third pass: For the remaining files, we can generate all three-file reference sets. However, we apply the constraint that if a three-file reference set occurs frequently, all its two-file reference sets also need to occur frequently (the Apriori property). Thus, file sets such as {A, B, D} are pruned, since file set {B, D} is eliminated in the second pass.

Termination: As we can no longer generate four-file reference sets, the algorithm ends. 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 {A, B}, {B, C}, and {A, C} are eliminated as they are already subsets of {A, B, C}.

Variations: An alternative is to use a normalized threshold, or ***support***, which is the percentage of set occurrences (number of the occurrences of a set divided by the total occurrences, ranged 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 file sets. To choose a subfile's membership between two composite files, the decision depends on whether a composite file has more subfiles, higher support, and more recent creation timestamps.

Optimizations: One way to prune the number of file sets is to use a higher threshold or support. In addition, the tracking tables can be incrementally updated as new file references arrive. For Figure 3.4.1, suppose another access to file A arrives, we can update the counters for {A} in the first table and {A, D} in the

second. Further, we can analyze the data in batches of n recent file references, and process them by PIDs or web IP addresses to detangle interleaved concurrent file references. As reference patterns change, subfiles within composite files may be reassigned to other composite files.

Another optimization concerns how to locate the file set containing a file sequence quickly. One way is to sort and hash the file sequence, but sorting is expensive in the critical path. Another way is to hash files in the sequence into a signature vector (similar to Bloom filter [Bloom 1970]), and hash the signature, but the size of the vector is proportional to the number of unique items in the trace. Instead, we used a commutative hash function, where hash(A, B) is the same has hash(B, A) to speed up this lockup process.

The frequency-mining-based consolidation can identify composite file candidates based on the dynamic file references. However, the computational overhead limits its practical application to file sequences accessed above a certain frequency.

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 [Szeredi 2005] (Figure 4.1) running atop Linux 3.16.7. The CFFS is stacked atop of ext4, so that we can leverage legacy tools and features such as persistence bootstrapping (e.g., file-system creation utilities), extended attributes, and journaling.

The CFFS periodically takes the recommendations from the generator tool to create composite files. We leveraged mechanisms similar to hardlinks to allow multiple file names to be mapped to the same composite i-node. (Not to be confused with hardlinks to individual subfiles, which are tracked by per-subfile reference counters, as needed) We intercepted all file-system-related calls due to the need to update the timestamps of individual subfiles. We also need to ensure that various accesses use 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 are stored by the underlying ext4 file system. The CFFS is implemented in C++ with ~1,600 semicolons.

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 focus on two scenarios. For the web server workload, we consolidate the `html` files and their immediately referenced files. In the case of conflicting composite file memberships, the preference is given to `index.html`, and then the `html` that first includes the file. The other is the source code compilation. We used `Makefiles` as a guide to consolidate source code files. For the frequency-mining-based scheme, the membership generator tool takes either the output from the `http` access log or the `strace` output. The generator implements the Apriori algorithm, with the support parameter set to 5%. The analysis batch size is set to 50K references. The parameters were chosen based on empirical experience to limit the amount of memory and processing overhead. The generator code contains ~1,200 semicolons.

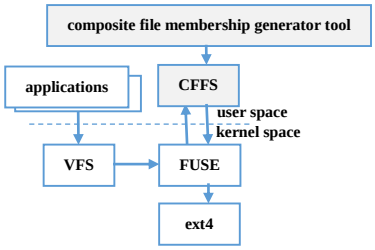

Figure 4.1: CFFS components (shaded) and data path from applications to the underlying ext4.

5. Performance Evaluation

We compared the performance of the CFFS stacked atop of ext4 via FUSE with the baseline ext4 file system (with the requests routed through an empty FUSE module). The replays were performed on a Dell T3600 workstation (Table 5.1). Each experiment was repeated 5 times, and results are presented at 90% confidence intervals.

We evaluated our system via replaying two traces. The first is a three month-long `http` log gathered from our departmental web server (01/01/2015-03/18/2015). The trace contains 14M file references to 1.0TB of data. Among which, 3.1M files are unique, holding 76GB of data. The second trace was 11 days long, 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 which, 291,133 files are unique with 2.9GB bytes. Between read and write operations, 59% are reads, and 41% are writes.

Table 5.1: Experimental platform.

	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
OS	Linux 3.16.7

We conducted multi-threaded, zero-think-time trace replays on a storage device. We also skipped trace

intervals with no activities. Prior to each experiment, we rebuilt the file system with dummy content. For directory-based and embedded-reference-based schemes, composite file memberships are updated continuously. For the frequency-mining-based consolidation, the analysis is performed in batches, but the composite files are updated daily.

5.2. Web Server Trace Replay

HDD performance: Figure 5.2.1 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 is 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 enable cross-file prefetching, resulting in much higher cache-hit rates.

The embedded-reference-based consolidation performed the best, with 62% of requests serviced from the cache, which is 20% higher than ext4. Thus, composite files created based on embedded references capture the access pattern more accurately. The overall replay time was also reduced by ~20%.

The directory-based composite files can also improve the cache-hit rate by 15%, reflecting the effectiveness of directories to capture spatial localities.

The frequency-mining-based consolidation performed worse than the directory-based scheme. We examined the trace and found that 48% of references are made by crawlers, and the rest by users. Thus, the bifurcate traffic patterns for the mining algorithm form more conservative file groupings, leading to reduced benefits.

SSD Performance: Figure 5.2.2 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 are caused by higher cache-hit rates and IO avoidance, this 20% benefit is rather independent of the underlying storage media.

5.3. Software Development File-system Trace Replay

For the software development workload 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 one-time setup cost for the directory- and embedded-reference-based schemes.

HDD performance: The embedded-reference-based scheme has poor coverage, as many references are unrelated to compilation. Therefore, the elapsed time is closer to that of ext4. Directory-based consolidation achieves a 17% elapsed time reduction, but the frequency-mining-based scheme can achieve 27% because composite files also include files across directories.

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

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

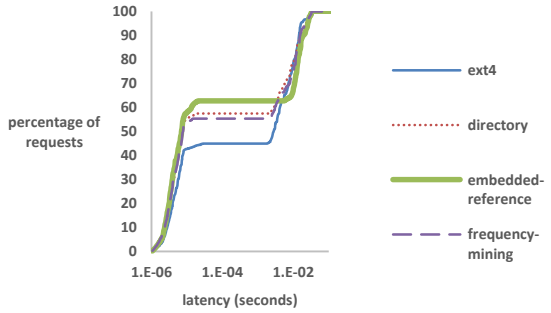

Figure 5.2.1: Web server request latency for HDD.

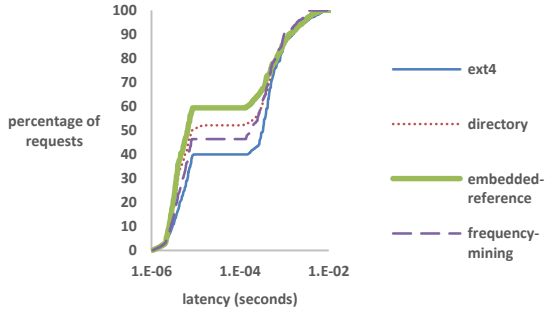

Figure 5.2.2: Web server request latency for SSD.

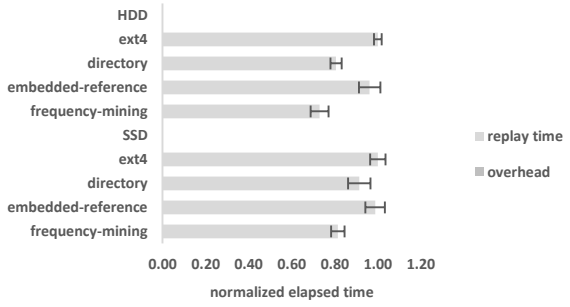

Figure 5.3.1: Elapse times for the software development file system trace replay.

5.5. Overheads

Directory- and embedded-reference-based schemes: Directory- and embedded-reference-based schemes incur a one-time cost at the deployment time to create composite files based on directories and embedded file references. The one-time 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 is anywhere from 1 to 14 minutes.

As for the incremental cost of updating composite file memberships, adding members involves appending to the composite files. Removing members involves mostly 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, and it is offset by the benefits.

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

The frequency-mining-based scheme involves learning from recent file references, and it took a few replay days to reach a steady state to reap the full benefit of this scheme.

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 the performance gains due to reduced disk seeks and modified data layouts are up to ~10%.

Overall, we are intrigued by the relationship among ways to form composite files, the performance effects of consolidating metadata and prefetching enabled by the composite files. Future work will explore additional ways to form composite files and quantify their qualities and their interplay with different components of performance contributions.

6. Related Work

Small file optimizations: While our research focuses 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 involve collocating a file's i-node with its first data block [Mullender and Tanenbaum 1984] and embedding i-nodes in directories [Ganger and Kaashoek 1997]. The CFFS consolidates i-nodes for files that are often accessed together.

The idea of accessing subfile regions and consolidating metadata is later explored in the parallel and distributed computing domain, where CPUs on multiple computers need to access the same large data file [Yu et al. 2007]. Facebook's photo storage [Beaver et al. 2010] leverages the observation that the permissions of images are largely the same and can be consolidated. However, these mechanisms are tailored for very homogeneous data types. With different ways to form composite files, the CFFS can work with subfiles with more diverse content and access semantics.

Prefetching: While a large body of work can be found to improve prefetching, perhaps C-Miner [Li et al. 2004] is closest to our work. In particular, C-Miner applied 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 [Ding et al. 2007] incorporates the knowledge of disk layout to improve prefetching, and the prefetching can cross file and metadata boundaries. The CFFS proactively reduces the number of physical metadata items and alters the storage layout to promote sequential prefetching. Soundararajan et al. [2008] 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, the CFFS's focus 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 have presented the design, implementation, and evaluation of a composite-file file system, which explores 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 show up to a 27% performance improvement under two real-world workloads. The CFFS experience shows that the approach of decoupling the one-to-one mapping of files and metadata is promising and can lead to many new optimization opportunities.

References

[Abd-El-Malek et al. 2005] Abd-El-Malek M, Courtright WV, Cranor C, Ganger GR, Hendricks J, CKlosterman AJ, Mesnier M, Prasad M, Salmon B, Sambasivan RR, Sinnamohideen S, Strunk JD,

Thereska E, Wachs M, Wylie JJ. Ursa Minor: Versatile Cluster-based Storage. *Proceedings of the 4th USENIX Conference on File and Storage Technology*, 2005.

[Agrawal and Srikant 1994] Agrawal R, Srikant R. Fast Algorithms for Mining Association Rules, *Proceedings of the 20th VLDB Conference*, 1994.

[Albrecht 2015] Albrecht R. Web Performance: Cache Efficiency Exercise. https://code.facebook.com/posts/964122680272229/web-performance-cache-efficiency-exercise/, 2015.

[Beaver et al. 2010] Beaver D, Kumar S, Li HC, Vajgel P. Finding a Needle in Haystack: Facebook's Photo Storage. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[Bloom 1970] Bloom B. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13, July 1970

[Chandrasekar et al. 2013] Chandrasekar S, Dakshinamurthy R, Seshakumar PG, Prabavathy B, Chitra B. A Novel Indexing Scheme for Efficient Handling of Small Files in Hadoop Distributed File System. *Proceedings of 2013 International Conference on Computer Communication and Information*, 2013.

[Ding et al. 2007] Ding X, Jiang S, Chen F, Davis K, Zhang X. DiskSeen: Exploiting Disk Layout and Access History to Enhance Prefetch. *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[Dong et al. 2010] Dong B, Qiu J. Zheng Q, Zhong X, Li J, Li Y. 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*, 2010.

[Edel et al. 2004] Edel NK, Tuteja D, Miller EL, Brandt SA. *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 2004.

[Ganger and Kaashoek 1997] Ganger GR, Kaashoek MF. Embedded Inode and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. *Proceedings of the USENIX 1997 Annual Technical Conference*, 1997.

[Garrison and Reddy 2009] Garrison JA, Reddy ALN. Umbrella File System: Storage Management across Heterogeneous Devices. *ACM Transactions on Storage*, 5(1), Article 3, 2009.

[Harter et al. 2011] Harter T, Dragga C, Vaughn M, Arpaci-Dusseau AC, Arpaci-Dusseau RH. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *Proceedings of 23rd Symposium on Operating Systems Principles*, 2011.

[Heidemann and Popek 1994] Heidemann JS, Popek GJ. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 22(1), pp. 58-89.

[Jiang et al. 2013] Jiang S, Ding X, Xu Y, Davis K. A Prefetching Scheme Exploiting Both Data Layout and Access History on Disk. *ACM Transactions on Storage*, 9(3), Article No. 10.

[Kroeger and Long 2001] Kroeger TM, Long DE. Design and Implementation of a Predictive File Prefetching. *Proceedings of the USENIX 2001 Annual Technical Conference*, 2001.

[Li et al. 2004] Li Z, Chen Z, Srinivasan SM, Zhou YY. C-Miner: Mining Block Correlations in Storage Systems. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[McKusick et al. 1990] McKusick MK, Karels MJ, Bostic K. A Pageable Memory Based Filesystem. *Proceeding of USENIX Conference*, June 1990.

[Mullender and Tanenbaum 1984] Mullender S, Tanenbaum. Immediate Files, *Software Practice and Experience*, 14(4):365-368, 1984.

[Roselli et al. 2000] Roselli D, Lorch JR, Anderson TE. A Comparison of File System Workloads. *Proceeding of 2000 USENIX Annual Technical Conference*, 2000.

[Soundararajan et al. 2008] Soundararajan G, Mihailescu M, Amza C. Context-aware Prefetching at the Storage Server. *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008.

[Szeredi 2005] Szeredi M. Filesystem in Userspace. http://userspace.fuse.sourceforge.net, 2005.

[Yu et al. 2007] Yu W, Vetter J, Canon RS, Jian S. Exploiting Lustre File Joining for Effective Collective IO, *Proceedings of the 7th International Symposium on Cluster Computing and the Grid*, 2007.