

# FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories

Miao Cai, Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University; School of Computer and Information, Hohai University; State Key Laboratory for Novel Software Technology, Nanjing University; Junru Shen, School of Computer and Information, Hohai University; Bin Tang, School of Computer and Information, Hohai University; Hao Huang, State Key Laboratory for Novel Software Technology, Nanjing University; Baoliu Ye, State Key Laboratory for Novel Software Technology, Nanjing University; Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University; School of Computer and Information, Hohai University

https://www.usenix.org/conference/atc22/presentation/cai

# This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11-13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

**■** NetApp<sup>®</sup>







# FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories

Miao Cai<sup>†‡§</sup>, Junru Shen<sup>‡</sup>, Bin Tang<sup>†‡</sup>, Hao Huang<sup>§</sup>, Baoliu Ye<sup>§†‡</sup>
Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University<sup>†</sup>
School of Computer and Information, Hohai University<sup>‡</sup>
State Key Laboratory for Novel Software Technology, Nanjing University<sup>§</sup>

## **Abstract**

The conventional file system provides a hierarchical namespace by structuring it as a directory tree. Tree-based namespace structure leads to inefficient file path walk and expensive namespace tree traversal, underutilizing ultra-low access latency and good sequential performance provided by non-volatile memory systems. This paper proposes FlatFS, a NVM file system that features a flat namespace architecture while provides a compatible hierarchical namespace view. FlatFS incorporates three novel techniques: coordinated file path walk model, range-optimized NVM-friendly  $B^r$  tree, and write-optimized compressed index key layout, to fully exploit flat namespace structure to improve file system namespace performance on high-performance NVMs. Evaluation results demonstrate that FlatFS achieves significant performance improvements for metadata-intensive benchmarks and real-world applications compared to other file systems.

#### 1 Introduction

Large, deep directory tree stresses file system namespace performance. For example, Hive, a famous data warehousing software system, uses a column-based partition technique to provide efficient query for big database tables [39]. Every column value represents a path component, and all columns constitute a file path to index the data. Such a partition technique causes a large deep directory tree for a wide table, making data query prohibitively expensive.

The recent advent of ultra-fast non-volatile memories revolutionized file system design, shifting performance bottlenecks from hardware I/O to the software stack. However, the current file system namespace structure, which is designed for slow storage devices, encounters severe performance issues with high-performance non-volatile memory systems [1, 20, 30, 36].

The file system provides a hierarchical namespace that is structured as a directory tree [5, 37], as shown in Figure 1a. The virtual file system (VFS) unifies multiple volatile namespace hierarchies to provide a single global namespace

view [40]. For any metadata system calls, the VFS first performs a pathname lookup (i.e., path walk) by walking the directory tree to find the associated directory entry (dentry), and then performing metadata operations, e.g., changing file permissions. Hierarchical namespace structure causes two major performance problems: *inefficient file path walk* and *expensive namespace tree traversal*.

First, current file path walk is slow and non-scalable. Resolving a path component involves costly dentry search and other coupled system operations like security module enforcement [10, 23, 24, 40]. Moreover, suppose a file path contains n components, resolving the last component needs to locate previous n-1 components. The total path walk task is linear to the number of components in the file path. The path walk efficiency has a slight performance impact on system calls for slow storage devices since hardware I/O dominates. In contrast, for NVMs offering ultra-low access latency [17, 42, 46], such high critical-path latency is unacceptable (see §2.1).

Second, traversing the namespace tree recursively is expensive with the hierarchical namespace structure. Entries of different directories are physically scattered over the storage device. It results in poor data access locality and indirect memory addressing during tree traversal. Namespace tree traversal is prevalent in common system usages such as *find*, rm - r and ls - R. Moreover, commercial NVMs like Intel Optane memory provide > 5GB/s memory bandwidth and prefer sequential memory access behaviors [17, 46]. Unfortunately, hierarchical namespace structure fails to utilize this important system characteristic (see §2.2).

Research efforts have been devoted to addressing these two problems. Tsai et al. [40] propose full-path caching to reduce path walk latency in the VFS layer. Solaris also incorporates a similar path-to-vnode cache [23]. However, caching is heavily dependent on file access locality. ByVFS [41] bypasses the VFS directory tree and directly manipulates dentry from the file system. As NVM devices deliver unbalanced read/write performance, the dentry write performance is suboptimal. A file system level optimization is utilizing efficient data structures (e.g., radix tree [44], hash table [15, 16],  $B^+$ 

tree [5,34,37], skip list [13]) or key-value stores (e.g., LevelDB [2,32] and TokuDB [6,18]) to manage and index persistent directory entries. Though these data structures offer good indexing performance, the data locality problem is not well-addressed. Moreover, all existing in-kernel file systems are developed underneath the VFS framework [14,18,26,31,44,48]. Therefore, pathname lookup performance in these file systems is constrained by VFS path walk efficiency.

This paper revisits namespace structure for ultra-fast, byte-addressable NVMs and proposes a novel file system named FlatFS. FlatFS exhibits a flat namespace architecture but still provides a compatible hierarchical namespace view. File metadata are directly indexed by their unique hierarchical paths. Hierarchical paths are organized without any further structure.

The flat namespace structure brings two performance advantages for file system design for NVMs. First, file path walk is fast and scalable as it only requires a single pathname lookup to flat namespace irrespective of path length. It effectively shortens the path walk for metadata system calls. Second, data locality is improved because contiguous directory entries in the namespace are also stored consecutively on the storage device. It accelerates file system namespace traversal and benefits directory range operations.

To exploit flat structure to improve namespace operation performance on high-performance NVMs, FlatFS incorporates three core techniques. First, a coordinated path walk is proposed to orchestrate two distinct path walk models in a global unified VFS namespace §4.1. FlatFS achieves fast, scalable path walk by separating pathname lookup from other intricate system operations, yet preserves the system semantics as the conventional path walk model. Instead of reconstructing the existing path walk module, coordinated path walk offers a flexible and backward-compatible solution to integrate a distinctive path walk model into the global namespace.

Second, a range-optimized NVM-friendly  $B^r$  tree is devised to manage variable-sized index keys §4.2.  $B^r$  tree provides efficient data-structure-level range operations (e.g.,  $range\ insert$ ) in logarithmic time. It effectively remedies the directory move shortcoming for the flat namespace, as well as facilitates designing other fast directory range operations §4.3.

Third, a write-optimized compressed (WoC) index key design is proposed by leveraging NVM byte-addressability to improve variable-sized key storage efficiency as well as reduce expensive small memory writes and data persistence overheads in key management on NVM systems §4.4.

Finally, FlatFS achieves low-cost metadata crash-consistency for the flat namespace \$4.5. The WoC key design effectively reduces the performance costs for data consistency in index key insert and removal. For directory range operations that involve complex tree structure manipulation, FlatFS also simplifies namespace tree crash-safety design based on an insight derived from  $B^r$  tree structure primitives.

We evaluate FlatFS with extensive experiments using both benchmarks and three applications (a version control system

Git, a parallel file indexer Psearchy [11], and a data warehouse software Hive [39]) on Intel Optane DC Persistent Memory §5. Evaluation results demonstrate that our flat namespace fully unleashes the high-performance NVM system. FlatFS outperforms other file systems by a factor of 4.02× for micro- and macro-benchmarks and improves metadata-intensive application performance by up to 37.5%.

In summary, this paper makes the following contributions:

- We analyze two performance issues in hierarchical namespace structure with high-performance NVMs.
- We propose a metadata-optimized file system FlatFS with three core techniques to flatten the hierarchical namespace for fast, byte-addressable NVM systems.
- We conduct extensive experiments to demonstrate the performance benefits of FlatFS to both metadata-intensive benchmarks and applications.

# 2 Background and Motivation

This section takes Linux kernel as an example to describe performance issues in the current file system namespace.

# 2.1 Inefficient File Path Walk

**Costly component resolution.** We conduct an experiment to demonstrate path component resolution inefficiency. We measure the execution latencies of six typical metadata system calls (*creat*, *open*, *stat*, *chmod*, *unlink*, *mkdir*) in NOVA file system on our testbed machine. Every system call operates on a file in a six-depth directory. Figure 1b shows the system call execution time and performance breakdown.

When the dentry cache (dcache) is hot, profiling results show that an average 15.87% execution time is spent on the dcache lookup. The dcache lookup includes filename hashing and hash table lookup. Currently, there is only a centralized dcache in the VFS layer. Increasing dentries stresses the dcache indexing performance. The permission checking takes 8.83% of the execution time on average. The path walk also spends 11.65% execution time on other system operations (denoted as *Other Walk* in Figure 1b), such as mount point checking and reference counter updating. In summary, when the dcache is hot, resolving a six-component file path occupies 14.17%-67.19% execution time for five system calls. The file operations take an average of 31.49% execution time.

On the other hand, when the dcache is cold, the underlying file system has to search the missing dentry. Dentry lookup of NOVA file system takes 69.26% of execution time for cold dcache. The dentry miss penalty is high. Besides the I/O transfer for the missing dentry, it also includes searching the dentry in the storage device and inserting this dentry into various VFS management data structures. Table 1 compares the block reading and dentry lookup latencies on NVM and SSD. Relatively slow SSD delivers long I/O latency, which dominates the overall execution latency. However, the dentry lookup time percentage increases around 20% when the storage device shifts from SSD to NVM. It indicates that

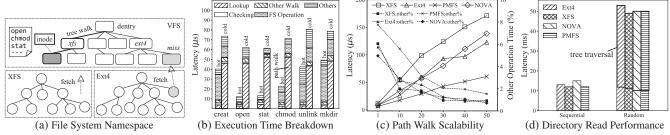


Figure 1: File System Namespace Structure in Linux and Performance Issues

Table 1: Comparing Block Read and Dentry Lookup Latency on NVM and SSD

	SSD: block	SSD: dentry	NVM: block	NVM: dentry
	reading (%)	lookup (%)	reading (%)	lookup (%)
Ext4	58.74%	4.12%	27.11%	25.36%
XFS	60.98%	3.34%	29.45%	28.10%

the performance bottleneck transfers from the hardware I/O latency to software path walk design with ultra-fast NVMs.

Non-scalable path walk. The current component-at-a-time path walk design is non-scalable with the number of path components. Such path walk design is widely adopted in current operating systems as it is convenient to implement the component resolution. Moreover, many other system functionalities are implemented upon component resolution [10,24]. They further increase the latency and exacerbate the path walk scalability issue. In addition, it is difficult to decouple them from the path walk.

We perform an experiment to understand the path walk scalability problem. We measure the *stat* syscall latency with different path component numbers on four NVM file systems. The VFS dcache is cold in the experiment. As shown in Figure 1c, the *stat* execution latency of four file systems increases dramatically as path length increases. The operation latency of the 50-component file path is nearly 14× higher than the 1-component file path. The VFS invokes file system-specific *getattr* to retrieve the file attributes. However, *getattr* only takes 1.01% on average of four file systems for a 50-component path configuration (denoted as *FS:other%* in Figure 1c). The rest of the execution time is spent on resolving the lengthy file path.

**Summary.** Current slow, non-scalable file path walk design has a large impact on metadata system call performance. This is a common problem for all file systems running on different devices like SSDs or HDDs. However, as commercial NVM devices offer near-DRAM access latency, the major bottleneck shifts from long hardware I/O to software path walk design, motivating us to reduce such software latency.

#### 2.2 Expensive Namespace Tree Traversal

Recent Intel Optane memory studies report that there is an asymmetry between sequential and random access performance [17, 28, 43, 46]. The performance gap between sequential and random memory bandwidth ranges from 2.3× to 3.5×. Moreover, the memory access latency is also sensitive to sequential and random access patterns [17, 42, 46].

File systems directly persist their directory tree in the NVM device. Traversing such a hierarchical namespace tree recursively is expensive. The reason is twofold. First, as directories logically form a tree structure, traversing the hierarchical tree causes indirect memory accesses. Second, persistent tree traversal introduces random memory access as dentries of different directories are distributed across the device. Previous studies show that both these memory access behaviors are suboptimal [17, 46]. Recursive tree traversal is an important namespace operation and used by many real-world applications heavily (e.g., *cp -r*, *git status*). Moreover, as page cache is removed from the NVM storage stack [26, 31, 44, 48], directory reading operations are directly performed on the NVM device. Existing persistent namespace tree traversal degrades directory reading performance.

We perform an experiment to understand the performance costs of hierarchical namespace tree traversal. We create two sets of 1024 files. Files in the first set are stored under an eleven-depth directory hierarchy. All files of the second set are stored in a one-depth directory. We use *ls -R* to read these two directories recursively. The first listing operation walks a hierarchical namespace and the second listing operation simulates sequential access in a flat namespace. Figure 1d shows that sequential directory reading performs 4.08× better than random directory reading. Traversing the hierarchical directory tree occupies nearly 80% total execution time.

**Summary.** The hierarchical namespace structure leads to low directory reading performance due to expensive tree traversal. As the page cache is removed from the modern NVM storage stack, directory reading operations access data stored on NVM devices directly. How to architect the file system namespace structure to exploit the device characteristic to improve the namespace operation performance still remains unsolved.

#### 3 Overview

# 3.1 Design Goals

Through flattening the hierarchical namespace, we build a high-performance, POSIX-compliant NVM file system. In particular, we have four design goals.

 Short path walk. File path walk is essential for most of the metadata system calls. FlatFS aims to reduce such software latency to minimize performance impacts on system call execution.

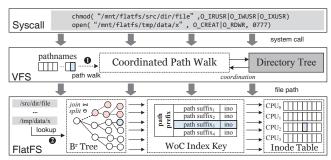


Figure 2: FlatFS System Architecture

- **Optimizing range operation.** As sequential accesses are optimal in NVMs, FlatFS aims to re-structure namespace architecture to fully exploit this system characteristic.
- Reducing persistence costs. FlatFS applies writeoptimization techniques to key layout design to reduce data persistence costs in its index key management.
- Ensuring namespace crash-safety. FlatFS achieves namespace crash-safety guarantees for conventional metadata syscalls and compound directory range operations.

## 3.2 FlatFS Architecture

FlatFS system architecture is presented in Figure 2. There is no cached namespace in the VFS. The namespace operation is directly performed on the device. There are four types of files in FlatFS: regular file, directory, symbolic link, and hard link. The file metadata is indexed by its full pathname relative to the FlatFS mount point.

In the system call layer, FlatFS provides a compatible hierarchical namespace view. Applications still use hierarchical file paths to access files and directories in FlatFS namespace.

In the virtual file system layer, we design a coordinated path walk (§4.1). The FlatFS namespace also appears in the VFS namespace to preserve a global unified namespace view. The coordinated path walk incorporates two distinct path walk models. It cooperates with the VFS directory tree to perform path walk across distinctive file system namespaces and dispatches requests to the corresponding file system instance.

In the file system layer, index keys are managed by a rangeoptimized persistent  $B^r$  tree (§4.2). All index keys are sorted in an lexicographical order.  $B^r$  tree is carefully designed with NVM properties [46]. Besides,  $B^r$  tree supports data structure level range operations based on two proposed tree structure primitives: *join* and *split*. It facilitates designing low-cost directory range operations (e.g., *rename*) and simplifies their implementation (§4.3).

In the key storage layer, the idiosyncratic NVM systems pose severe challenges to index key management. Frequent index key updates incur a large number of small random memory writes and cache line flushes, which is especially harmful to system performance [46]. FlatFS adopts a write-optimized compressed key layout to avoid most memory writes and cache line flushes during key inserts and removes (§4.4).

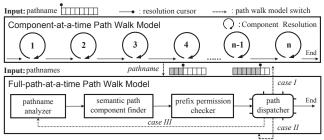


Figure 3: Two Path Walk Models

# 4 Design and Implementation

# 4.1 Coordinated File Path Walk

Coordinated file path walk applies two path walk models to resolve a pathname. A path component could be in five different forms: "." (dot), ".." (dot-dot), normal file, directory, and symbolic link (symlink). Figure 3 illustrates these two path walk models. The component-at-a-time path walk model resolves path component one by one while the full-path-at-a-time path walk processes the whole pathname at a time. This section describes how our path walk model correctly handle different kinds of path components and permission checking as well as coordination between these two models.

Pathname analyzer. The pathname analyzer generates a canonical pathname without any dots and redundant slashes by performing lexical processing on the file path. If FlatFS adopts the Plan 9 lexical file pathname [29], the analyzer resolves a dot-dot component by removing the path component before it. Otherwise, the dot-dot component is handled by the semantic path component finder, which will be described later. After analyzing, FlatFS passes the canonical pathname to the semantic path component finder.

**Semantic path component finder.** The pathname analyzer can only handle non-semantic path components. The semantic components (i.e., symbolic links and mount points) are identified by the semantic path component finder using a key indexing approach. Specifically, these two kinds of semantic components have associated entries  $\langle path, ino \rangle$  in the  $B^r$ tree. Besides that, there is another special *finder* entry with the key path/\xFE for each of these components in the  $B^r$ tree <sup>1</sup>. The *finder* entry value denotes component type. According to  $B^r$  tree lookup policy, if there is no equal entry for the requested key, the first entry whose key is greater than the requested key is returned. For example, if the file path is /a/./b/link/c where link is a symlink, the associated *finder* entry key is /a/b/link/\xFE. The pathname analyzer output is /a/b/link/c. Because /a/b/link/\xFE is the first key that is greater than /a/b/link/c, this finder entry is returned during  $B^r$  tree lookup.

Further, there is a remaining problem in finder design: a symlink following a dot-dot component. Suppose a path is a/symlink/.. in which symlink points to b/c, and the symlink element is removed after pathname analyzing. In

<sup>&</sup>lt;sup>1</sup>The ASCII character \xFF is reserved for shadow entry, see §4.3

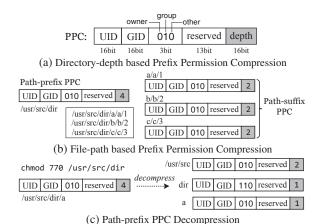


Figure 4: Two-dimensional Prefix Permission Compression

contrast, the component-at-a-time path walk will follow the symlink and generate a resolution result a/b. One solution is using the finder to check whether the current resolved path contains a symlink whenever meeting a dot-dot component during pathname analyzing. This approach degrades path walk performance if there are many dot-dot components in a file path. Another solution is using Plan 9 lexical file pathname [29]. It has better performance but causes compatibility issues. To address this dilemma, FlatFS lets users decide which file path type during file system mount.

If there are no semantic components in a path, FlatFS directly uses the pathname to search the namespace  $B^r$  tree to find the associated file inode.

**Prefix permission checker.** When the user accesses a file, the file system verifies whether the calling process has the execute permission for each directory listed in the file path. We call it prefix permission checking. FlatFS separates the prefix permission checking from the pathname lookup and adopts two-dimensional prefix permission compression to reduce checking performance costs.

First, directories in a file path often have same permission fields [15]: UID, GID, execute permission bit. FlatFS compresses these directory permission fields into a structure called *prefix permission compressor* (PPC for abbreviation), as shown in Figure 4a. The depth field denotes how many levels of directories are compressed in this PPC. Second, PPCs of those directories whose index keys are in the same  $B^r$  tree leaf node also can be compressed based on the file path (depicted in Figure 4b). The permission fields of the file path prefix are compressed into the path-prefix PPC. The permission fields of the remaining file path are compressed into the path-suffix PPC. File-path-based prefix permission compression is helpful in batching PPC updates.

A directory permission change may cause PPC decompression. Figure 4c illustrates a path-prefix PPC is decompressed into three PPCs caused by a *chmod*. Every new PPC records compressed permission fields of a sub-file path. Moreover, this decompression is propagated to PPC of all sub-directories and associated files under /usr/src/dir directory. Fortu-

nately, the file-path-based compression reduces update costs. If keys in a leaf node share a prefix /usr/src/dir, only the path-prefix PPC in tree leaf node is updated. Further, as directory permission changing are rare operations reported in a recent research paper [15], the PPC decompression incurs a slight performance impact on realistic applications.

The PPC allows batching permission checking. When performing a prefix permission checking, the task uses its credential to verify the UID and GID in both path-prefix and corresponding path-suffix PPCs. Then, it compares the directory depth of the file path and total execution bits compressed in path-prefix and path-suffix PPCs. If they are equal, file access permission is granted. For a relative file path, FlatFS extracts associated permission bits in path-prefix and path-suffix PPCs according to the path to be checked, and then performs permission checking.

The symlink and dot-dot components require careful consideration to preserve the semantic. If a symlink points to an absolute path, it may cause a namespace switch. FlatFS performs prefix permission checking on file path parts which belong to its namespace. Similarly, when adopting non-lexical file paths, FlatFS also performs prefix permission checking on path components before the dot-dot component. Overall, current POSIX prefix permission checking specification is more beneficial to the traditional path walk model.

**Path dispatcher.** Path components like dot-dot, symbolic link, and mount point could cause namespace switches during path walk. The path dispatcher performs a namespace switch and sends the pathname to the path walk model. Although the destination namespace may be hierarchical or flat (case I & II in Figure 3), the switch procedure is the same, mainly including mount point switch, pathname cursor adjustment, and other environment setups. In addition, If a symlink points to a relative file path (case III in Figure 3), the path dispatcher generates a new file path by resolving the symbolic link and restarts the whole path walk.

Coordination. A file path walk may involve different path walk models. These different path walk models are coordinated to resolve a pathname correctly. A path walk model can be viewed as a black box. Feeding a pathname input, it generates the resolved pathname output. The key to coordination is ensuring correct path walk model switch. Specifically, the VFS uses a pair <mnt point, rootdir> to specify a mounted file system. We also create one for FlatFS instance, which is used to switch in or out its namespace. Furthermore, we add a pathname cursor for each path walk model to indicate the current resolved pathname position. This cursor feeds a correct pathname input to the path walk model.

## 4.2 Range-optimized Index Tree

FlatFS uses a persistent range-optimized  $B^r$  tree as its indexing data structure. The index keys of  $B^r$  tree are full pathname byte strings. Figure 7 shows the  $B^r$  tree leaf node layout. The tree node is aligned to 256 bytes, which is the optimal Optane

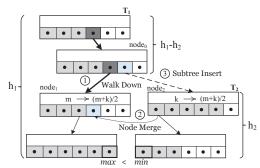


Figure 5: Tree Join Example:  $T_1 \bowtie T_2 = T$ 

memory access granularity [46]. The keys in the  $B^r$  tree nodes are unsorted [12]. Two small pieces of metadata (a bitmap and an offset array) are added into the leaf node.  $B^r$  tree adopts a hand-over-hand locking scheme with a top-down locking order. Every tree node owns a readers-writer lock.

Directory range operations in flat namespace are more expensive than the hierarchical namespace tree [47].  $B^r$  tree provides range operations at data structure level to overcome this challenge. The range operations are realized based on two novel tree structure primitives: join and split. We use symbol  $\bowtie$  and  $\ominus$  to denote tree join and split primitive, respectively. Also, we define tree node fanout f and total tree items N. The whole tree is locked during range operations.

**Tree join.** The primitive concatenates two smaller trees  $T_1$  and  $T_2$  and generates a larger tree  $T: T_1 \bowtie T_2 = T$ . The maximum key in  $T_1$  must be smaller than the minimum key in  $T_2$ . Figure 5 illustrates an example of joining  $T_1$  and  $T_2$ . These two tree heights are  $h_1$  and  $h_2$  respectively, where  $h_1 > h_2$ . The detailed tree join steps are described as follows. First, we walk downside the  $T_1$  from top level to the level  $h_1 - h_2$  by always walking the rightmost tree node in each level (Step 1). Then, we concatenate these two trees by merging  $node_1$ and  $node_2$  (Step (2)). The key numbers of these two nodes are m and k. If  $m+k \le f$ , we only need to copy all keys and children from  $node_2$  to  $node_1$ . Otherwise, the key number of two nodes is re-balanced as (m+k)/2. These associated keys and children are moved from  $node_2$  to  $node_1$ . Finally, a new subtree  $tree_2$  is inserted into the parent node  $node_0$  (Step (3)). The time complexity of tree join  $\bowtie$  is  $O(|h_1 - h_2|)$ .

**Tree split.** The  $\ominus$  primitive splits the tree T into two smaller trees LTree and RTree for split point x:  $T \ominus x = \{LTree, RTree\}$ . The maximum key of LTree is less than x and the minimum key value of RTree is greater than or equal to x. Figure 6 depicts tree split steps with the split point 9.

At the root node level, we divide all keys in the  $node_1$  into two parts. The maximum key of the left part is smaller than 9. The minimum key of the right part is greater than or equal to 9. This key division also partitions the tree into two subtrees  $L_1$  and  $R_1$ . Then, we walk down to the  $node_2$  pointed by child  $C_1$ . The key in  $node_2$  is in the range of [5,20]. Similarly, keys in  $node_2$  also can be divided into two parts. As a result, two subtrees  $L_2$  and  $R_2$  are generated. This operation is repeated until reaching the leaf level. Finally,

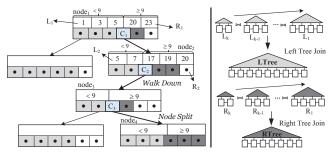


Figure 6: Tree Split Example:  $T \ominus 9 = \{LTree, RTree\}$ 

this tree split generates two small tree sets:  $\{L_1, L_2, ..., L_k\}$  and  $\{R_1, R_2, ..., R_k\}$ . Then, we perform tree join operations on all trees in each tree set in an ascending order of tree height. For example, the LTree is generated by joining k subtrees:  $L_1 \bowtie L_2 ... \bowtie L_k$ . Finally, the tree T is split into two trees: LTree and RTree. The time complexity of  $\ominus$  operation is  $O(log_f N + \Sigma(h_i - h_j)) = O(log_f N)$ .

**Range operation.**  $B^r$  tree provides four range operations: range query, range slice, range insert, range update. The range query operation workflow is the same as a textbook  $B^+$  tree. The range slice operation detaches a subtree t from the original tree T with a given key range  $[k_l, k_r]$ . The detailed range slice steps formulate as follows:

$$T \ominus k_l = \{p, q\} \Rightarrow q \ominus k_r = \{t, r\} \Rightarrow p \bowtie r = T'$$
 (1)

First, we split the tree T into two parts  $\{p,q\}$  for  $k_l$  using a  $\ominus$  operation. All keys in tree q are greater than  $k_l$ . Thus, the tree q is split again with the key  $k_r$ . Keys of tree t locate in the range  $[k_l,k_r]$ . Consequently, the tree t is the range slice result. However, the original tree t structure is destroyed by two split operations. We heal the tree t by using a t operation to concatenate the t and t. The tree range slice operation is useful in the directory remove and move operation because it can remove a bunch of indexes for targeted files and directories from the t

The tree range insert operation inserts a small tree into a large tree at a time. There are two cases for a tree range insert. To insert a tree  $T_2$  into another tree  $T_1$ , if all keys in  $T_2$  are smaller than keys in  $T_1$ , we can directly join these two trees:  $T_1 \bowtie T_2 = T_1'$ . Otherwise, the tree range insert operation formulates as follows:

$$T_1 \ominus key_{min} = \{T_1', T_r\} \Rightarrow T_1' \bowtie T_2 = T_1'' \Rightarrow T_1'' \bowtie T_r = T_1'''$$
 (2)

The tree  $T_1$  is split into  $T_1'$  and  $T_r$  with the minimum key  $key_{min}$  of tree  $T_2$ . The keys in  $T_1'$ ,  $T_2$ , and  $T_r$  are in ascending order:  $key_{T_1'} < key_{T_2} < key_{T_r}$ . This range insert operation is achieved by performing two join operations on three trees.

The tree range update modifies the index keys of a specific range. A naive solution to range update is walking the tree and updating all index keys in tree nodes. Our  $B^r$  tree uses the WoC key design to reduce the key update costs. The WoC key fetches the common part of all index keys as a key prefix. Therefore, the range update operation only modifies the key prefix without updating these keys one by one.

**Leaf node caching.** We design a leaf node cache to reduce tree walk performance costs. Every CPU owns a volatile in-DRAM node cache structured as a LRU list. The cache entry stores the leaf node memory address instead of the real leaf node. Hence, no data synchronization is required among multiple CPU caches. The  $B^r$  tree lookup consists of a fast path and a slow path. The fast path traverses the LRU list, locks and accesses these real leaf nodes, and searches for the item with the requested key. If there is a hit, the slow path that walks the whole tree can be avoided. The leaf node cache is effective as namespace operations in the real world often exhibit good locality. For example, creating files in a directory needs to search the same directory file inode multiple times.

Every leaf node has a reference counter for safe memory reclamation. Creating a node initializes the counter as one, and the cache insertion increases the counter. Because the cache is volatile, no crash safety is needed for counters. They will be reset as one during remount. A cache entry may refer to a removed tree node. The removed node with a non-zero counter is kept in a persistent list for lazy reclamation. Accessing the removed node is safe since it contains no valid keys due to the empty bitmap. This node will be recycled when the associated cache entry is evicted.

# 4.3 Directory Range System Call

Besides *rename*, FlatFS designs three new system calls for directory range operations.

**Directory read.** FlatFS offers a *getdents\_recur* system call. This system call lists all files and subdirectories in a directory recursively (similar to *ls -R*). To support recursive range query for a directory, FlatFS introduces two shadow entries for each directory. Their filenames are the first and last ASCII character. Keys of all entries in the directory are delimited by these two shadow entry keys. To perform a recursive directory reading, FlatFS uses the small shadow entry key to lookup the tree to find the leaf node. Then, it repeats this with the other large shadow entry key and fetches a range of keys at a time. For non-recursive directory reading, FlatFS uses a directory skip approach. When FlatFS meets the first shadow entry of a subdirectory during directory scanning, it skips entries in the subdirectory by performing a tree lookup with the other shadow entry key.

**Directory remove.** FlatFS introduces a new system call  $rmdir\_recur$ , which is used to remove all files and subdirectories in a directory at a time. In the data structure layer, we use a range slice to obtain the subtree that contains entries to be removed from the  $B^r$  tree. Then, we perform a range query to this subtree, find inodes of these files and directories, remove files and directories at the file system layer. To further reduce the  $rmdir\_recur$  latency, we also delay freeing the sliced subtree structure.

**Directory copy.** FlatFS provides a new system call *cpdir\_recur(src, dst)*, which copies a directory recursively to another directory. First, the subtree of *src* directory is sliced

and duplicated. The index key updates of the duplicated tree are batched. Then, both sliced and duplicated subtrees are inserted into the  $B^r$  tree. Then, new metadata are created for copied files and directories. We follow the default semantic of cp - r for hard links, i.e., the linking becomes invalid. A new inode is allocated for the copied hard link and its file content is the same as the original linked file. Besides, FlatFS also corrects the inode number of dot and dot-dot of every directory, making them point to the newly created directory and its parent directory. Another performance optimization adopted by  $cpdir\_recur$  is file data copy-on-write mechanism. We only duplicate the file mapping of every copied file. The last level pointers in the file mapping are set as copy-on-write.

**Directory move.** The directory rename mainly involves data structure level operation. Similar to directory remove, the subtree of the src directory is sliced. Then, all index keys of the subtree are updated with the dst directory. Depicted in Figure 7, every tree node contains an index key prefix. A key update traverses the  $B^r$  tree and iterates over every node. If the src directory pathname is a substring of the key prefix in this node, this key update just modifies the key prefix with the dst directory pathname. Finally, the updated subtree is inserted into the  $B^r$  tree.

# 4.4 Write-optimized Compressed Key

FlatFS uses key compression to reduce storage consumption and key batch update costs. Every variable-size index key is divided into a prefix and a suffix. All index keys in the same tree node share a prefix. However, key compression incurs performance overheads due to prefix and suffix adjustments. For example, inserting a key would shrink the prefix and expand all suffixes, leading to many small data writes, which causes expensive cache line flushes and write amplification.

**Basic idea.** We propose WoC key to address this challenge. WoC key prefetches a number of characters during suffix expansion. These data are cached in the suffix to avoid future data movements. Moreover, we also record the prefix size and the total size of each key. Suppose another key insert event causes all suffixes to expand and suffix size increasing. Fortunately, we only need to update the prefix size to represent all suffix data and size changes.

Furthermore, removing a key may cause all suffixes to shrink and the prefix to expand. There are no data movements for suffix shrinking. We append data to the prefix and increase the prefix size. There is no data prefetch optimization for the prefix because a tree node only has one prefix, and its adjustment cost is low. In contrast, every node entry owns a suffix. Suffix changes lead to high memory write costs.

**Detailed Design.** Figure 7 shows the key suffix is partitioned into inuse area and cache area. The *key suffix addrs* stores suffix memory addresses. The inode number is stored along with the suffix. The *woc-length array* contains a set of eight-bytes *woc-length* structures. Every index key owns a suffix and a woc-length structure. The total key size *key total sz* 

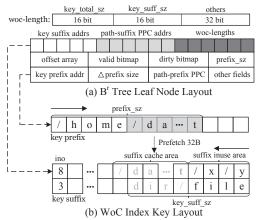


Figure 7:  $B^r$  Tree Leaf Node and Index Key Layout

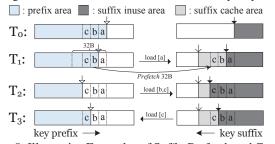


Figure 8: Illustrative Examples of Suffix Prefetch and Caching and the suffix size *key\_suff\_sz* are stored in the woc-length. The *key\_suff\_sz* is the sum of suffix inuse and cache area size. The inuse area size is *key\_total\_sz - prefix\_sz*. If the cache area size is smaller than the data to be appended, it indicates the suffix cache is insufficient and requires another prefetch.

Figure 8 illustrates three examples of how to reduce data movements with WoC key design. At time  $T_1$ , one-byte data a is moved to the suffix area. Instead of moving one character, we prefetch 32-bytes of data from the prefix memory area to the suffix memory area. Then we modify  $prefix\_sz$  and  $key\_suff\_sz$  and persist these two fields. 32 bytes is the optimal data prefetch granularity according to evaluation results. Prefetching 64-bytes data may introduce two cache line movements if the first byte of prefetched data is not cacheline aligned. On the other side, a smaller prefetch granularity introduces more frequent data movements.

The prefetched data are stored in the cache area to avoid future data movements. For instance, at time  $T_2$ , two bytes of data [b,c] are moved to the suffix area. Fortunately, these two-bytes data are already stored in the suffix cache area. We only need to update the  $prefix\_sz$  field to indicate this key suffix expansion. When the key suffix shrinks at time  $T_3$ , the associated data c is appended to the key prefix. There are also no data movements for this suffix shrinking event.

The key batch update introduces two issues to WoC key design: (1) if a key batch update modifies the key prefix, the  $key\_total\_sz$  fields of all key suffixes should be updated; (2) a key batch update may cause cached data in the suffix to become stale due to key prefix change. To solve the first problem, we introduce a  $\Delta prefix\_size$  field, which denotes

the key size changes to all index keys in a tree node caused by a batch update. Instead of updating all  $key\_total\_sz$  fields, the key batch update modifies the  $\Delta prefix\_size$ . The key suffix inuse area size is  $key\_total\_sz-prefix\_sz+\Delta prefix\_size$ .

The second problem is addressed by introducing a dirty suffix bitmap. The bitmap records those key suffixes whose cached data become stale in the index key batch update. We query the dirty bitmap before adjust the key suffix. If the associated suffix cache area is dirty, we clean the suffix cache area by reseting *key\_suff\_sz* as the suffix inuse area size.

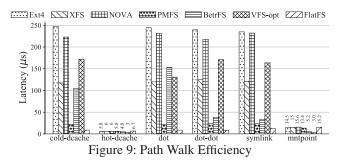
# 4.5 Namespace Crash Consistency

Existing metadata system calls except for *rename* only introduce tree point operation. The general workflow of these system calls consists of three steps: (1) pathname lookup; (2) inode manipulation; (3) tree point operation. Achieving crash consistency for these system calls is similar. We take *creat* system call as an example to describe the approach.

First, the newly allocated inode is logged at the file system layer. Then, FlatFS inserts a new entry into  $B^r$  tree. This entry insert may be a simple leaf node insert or requires a node split. In the worst case, a leaf node insert causes a key prefix adjustment, a new key suffix write, a cascade of updates to other key suffixes, and a valid bitmap update. Thanks to the WoC key design, we only needs to log 18-bytes data (2-bytes prefix size, 8-bytes prefix address, and 8-bytes valid bitmap). Logging key prefix size and address ensures data consistency for both key prefix and suffix. To be specific, all old suffix sizes can be restored with logged old prefix size. Then we reset  $key\_suff\_sz$  as key suffix inuse area size. The key prefix is also restored using logged old prefix address. In addition, if there is a node split for this entry insert, the valid bitmaps of the split node and parent node are also logged.

Ensuring metadata consistency is challenging for directory range operations as they introduce complicated tree structure manipulation. We first explain how to ensure crash-safety for  $B^r$  tree range operations. Depicted in Figure 5, a tree join involves three nodes: two merged nodes and a parent node. The valid bitmaps of these nodes are logged. Ensuring crash consistency for tree splits is more difficult. Supposing the tree T height is h, the  $\ominus$  operation splits h nodes at each tree layer with split point x. The valid bitmap of h nodes is logged. Next, two new trees L and R are generated by joining these split trees in two sets separately. It seems difficult to recover the tree state if a crash occurs during tree joins. A useful property between ⋈ and ⊖ simplifies the crash-consistency implementation, i.e.,  $\ominus$  is the reverse operation of  $\bowtie$ . The original tree T can be recovered by joining L and R:  $L \bowtie R =$ T. Thus, root node memory addresses of joined trees (i.e.,  $\{L_1,...,L_k\}$  and  $\{R_1,...,R_k\}$ ) are logged. During recovery, we replay tree join operations with logged root node addresses and re-generate the namespace tree T with L and R.

The tree *range insert* and *range slice* are implemented based on tree join and split. We already guarantee crash-



safety for tree  $\bowtie$  and  $\ominus$ . Crash consistency of these two tree operations is achieved based on an important insight. Even the original namespace tree is temporarily decomposed into many pieces due to split/join, however, it still can be rebuilt by joining these subtrees during recovery. Finally, FlatFS also logs these affected inodes for directory range operations.

#### 5 Evaluation

Our testbed machine has two Intel Xeon Gold 5220R processors. Each processor has 24 physical cores running at 2.2 GHz. Every physical core has a private 32 KB L1 cache and a 1 MB L2 cache. All cores that resided on the same socket also share a 35.75 MiB last level cache. The hyper-threading is disabled. Each processor has two integrated memory controllers. The machine has 192 GB (12×16 GB) DRAM, 1.5 TB (12×128GB) Intel Optane DC persistent memory, and a 512 GB Solid State Driver.

We implement FlatFS in Linux kernel 4.15 based on PMFS file system [31]. FlatFS reuses PMFS data path and journal mechanism. We compare FlatFS with four NVM file systems (NOVA [44], PMFS [31], Ext4 [5] and XFS [37]), a full-path-indexing file system BetrFS [47], a VFS dcache optimized system (VFS-opt) [40]. BetrFS and VFS-opt work on Linux 3.11.10 and 3.14, respectively. These old kernels do not support Intel Optane memory. We use Linux *brd* module to create a RAM disk for BetrFS and VFS-opt. The RAM disk uses the fast DRAM to emulate block devices. The device performance is better than the Optane memory. We also mount an Ext4 file system on the RAM disk for VFS-opt.

#### **5.1** Microbenchmark Performance

Path walk efficiency. We evaluate the path walk performance with different kinds of path components. We *stat* a file whose path comprises nine components in the Linux kernel 4.4 source code directory. When dcache is cold, file systems fetch data from the device, which causes long latency. This problem is especially serious for BetrFS. BetrFS adopts a stacked system architecture. A dentry missing event is handled by multiple software layers. Its path walk latency is  $11.93 \times$  higher than FlatFS. The sequential file access results in a hot dcache, which greatly reduces path walk latency. Fortunately, FlatFS also achieves low latency thanks to its node cache design. The VFS-opt delivers the lowest latency ( $3\mu$ s) for its hash-based full-path-indexing optimization.

VFS and FlatFS handle dot components lexically. Thus, the

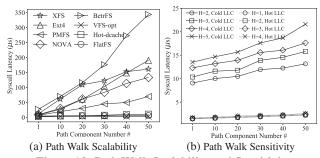


Figure 10: Path Walk Scalability and Sensitivity
Table 2: Path Sensitivity to Sequential and Random Access.
H: *B*<sup>r</sup> Tree Height; L: File Path Length; N: File Number

Setting	H=3, N=10 <sup>4</sup> , L=20		H=4, N=10 <sup>5</sup> , L=20		H=5, N=10 <sup>6</sup> , L=20	
System	VFS	FlatFS	VFS	FlatFS	VFS	FlatFS
Seq.	2.05μs	1.63µs	2.04µs	1.63µs	2.05µs	1.63µs
Rnd.	2.11µs	2.19µs	2.24µs	2.54µs	2.26µs	3.66µs

path walk latencies are similar to the cold dcache case. FlatFS performs semantic path component checking for dot-dot and symlink components, which incurs an extra namespace tree query. Even so, FlatFS performs 1.87-28.52× better than others for its non-caching namespace design. Finally, we mount an Ext4 file system under each tested file system. The mount operation causes a hot dcache for path walk. The performance results are similar to the hot dcache case. Overall, for a nine-component path, FlatFS outperforms other file systems significantly for the cold dcache and delivers similar latencies for the hot dcache.

**Path walk scalability.** We vary path component number from 1 to 50 to evaluate path walk scalability using *stat* syscall. As shown in Figure 10a, when the file path becomes lengthy, the execution latencies of five file systems (i.e., BetrFS, XFS, Ext4, PMFS, NOVA) also increase significantly. The FlatFS path walk latency  $(5.1\mu s)$  is stable against different path lengths. The VFS-opt system achieves a low constant path walk latency  $(3\mu s)$ . We also measure the path walk scalability of hot dcache. When the path component is less than 10, its latency is close to ours. However, its latency increases  $5 \times (10.8\mu s)$  varying from 1- to 50- component.

Path walk sensitivity. We evaluate FlatFS path walk sensitivity by changing four variables:  $B^r$  tree height, Last-Level-Cache (LLC), path component number, and access pattern. We create four file sets of different sizes  $(10^3, 10^4, 10^5, 10^6)$  with tree height H ranging from 2 to 5. We vary the path length and measure the syscall latency. We make three observations from Figure 10b. First, hot LLC significantly boosts FlatFS path walk performance. Second, path walk performance is sensitive to tree height, especially for cold LLC. Resolving a 50-component path in a 5-level  $B^r$  tree takes  $21.61\mu$ s, which is  $1.64\times$  longer than 1-level  $B^r$  tree. Third, path length has a slight impact on FlatFS path walk performance. A 50-component path walk is  $1.5\times$  longer than 1-component. In contrast, other file systems deliver  $9.8\times$ - $16.3\times$  latency increment in Figure 10a.

Table 3: Directory Range Operation Latency (s)

	Ext4	XFS	NOVA	PMFS	BetrFS	VFS-opt	FlatFS
readdir	0.152	0.349	0.208	0.097	0.388	0.157	0.031
rmdir	0.61	1.262	1.131	0.548	3.680	0.736	0.190
cpdir	2.398	2.907	2.334	1.949	4.652	1.829	0.450
mvdir	0.004	0.004	0.004	0.004	0.019	0.004	0.007

Table 2 shows path walk latency of sequential and random file access. The LLC and VFS dcache is warmed before experiments. FlatFS outperforms VFS for sequential access in three settings. The reason is batched operations in FlatFS path walk. To confirm it, we use the perf tool to collect instruction numbers during syscall execution in setting  $S_3$ . The profiling results report that VFS executes  $1.43\times$  more instructions than FlatFS. Random access has a larger impact on FlatFS than VFS. VFS gains much more benefits from its cached dentries of the prefix path. Random access causes  $3.24\times$  more cache misses for FlatFS than VFS. As the file size decreases, FlatFS also achieves better performance for higher LLC hit radio. Finally, sequential access outperforms random access due to higher node cache hit radio.

**Directory range operation.** Table 3 shows four common directory range operation performance. FlatFS uses four directory range system calls. For other file systems, we use *ls -R, rm -r, cp -r*, and *rename* instead. All directory range operations manipulate a Linux kernel 4.4 source code tree.

Reading a directory recursively is well-optimized in FlatFS. FlatFS reduces directory read latency by up to 12.52× compared to others. These file systems except BetrFS scatter the dentries in data blocks across the storage device, which greatly hurts the data locality. For directory removing, FlatFS obtains performance gains from 2.88× to 19.37× relative to other file systems. Since namespace traversal is fast in FlatFS, it benefits directory remove as FlatFS only needs to perform a simple scan to obtain the entries for de-allocation. Other file systems require expensive tree traversal to retrieve target inodes. Also, FlatFS delays freeing the directory subtree to further reduce the latency.

FlatFS outperforms other file systems by  $4.06 \times -10.34 \times$  for directory copy. Because the directory copy is data I/O dominated. The file data copy-on-write optimization in FlatFS effectively avoid such performance costs. BetrFS is well-optimized for HDDs. For ultra-fast NVMs, its stacked architecture is suboptimal. We find interactions between BetrFS and the underlying storage device involve many software layers, which causes high software latencies due to both additional and duplicated works. Directory move is the only range operation provided by conventional file systems. Directory move is cheap for the hierarchical namespace structure, which only takes  $4\mu$ s for moving a Linux directory tree. FlatFS also achieves a low rename latency  $7\mu$ s using tree-level range operation and batching key update.

#### **5.2** Macrobenchmark Performance

**Filebench.** We choose two application-level workloads: fileserver and varmail from Filebench [38]. We create a file set

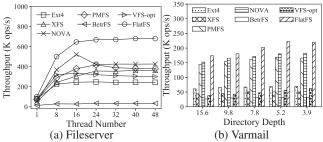


Figure 11: Performance of Filebench Benchmark

with 100 thousand 4KB files for the fileserver. We stress the namespace performance by setting the average directory width as two. Figure 11a shows the fileserver throughput varying thread number from 1 to 48. FlatFS achieves a 1.79 performance speedup over PMFS. FlatFS and PMFS share the same file data path but differ in metadata management. When thread number increases, PMFS suffers from poor scalability due to its centralized inode table design. NOVA and FlatFS address this issue with per-CPU inode table design. NOVA performs worse than FlatFS for its low namespace insert/remove performance and fault tolerance costs [45]. NOVA uses a Radix tree to manage entries in the directory. A *createlunlink* inserts/removes the associated entry in the parent directory. Because a directory only has two files, file creation/removal incurs high tree management costs.

The centralized journal in Ext4 also causes severe scalability issues [19,35]. VFS-opt requires Ext4 to create files, which accounts for its low scalability. VFS-opt achieves 23% higher throughput than Ext4 for its dcache optimization. Finally, BetrFS exhibits the lowest performance. Since the cache is hot during execution, we explain it as NVM-unware  $B^{\varepsilon}$  tree design.  $B^{\varepsilon}$  tree adopts big internal (4MB) and leaf nodes (4-11 MB). Moreover, it uses an order maintenance tree (OMT) to organize entries inside the leaf node. The entry insert or remove is expensive because it introduces many entry movements and OMT adjustment costs during file create and remove. In contrast, FlatFS reduces key management costs during tree insert/remove with its write-optimized key design.

In varmail experiment, we create a file set that contains 50 thousand 64-byte files to simulate a large set of small e-mails in the mailbox. We vary the *meandirwidth* parameter from 2 to 16. It affects the directory depth of the tested file set. When the directory depth increases, the path walk time increases. Therefore, the throughput of all file systems decreases. FlatFS obtains 14.10%-3.97× higher throughputs than other file systems due to its fast path walk design. Moreover, varmail workload frequently creates and deletes files. Besides fast path walk, FlatFS also gains performance benefits from its high namespace insert/remove performance.

**FxMark.** We pick MWCM and MWUM benchmarks from FxMark [25] to measure the file system multicore scalability. Figure 12a and Figure 12b shows experimental results. We draw three conclusions from these experiments. First, FlatFS performs much better than other file systems in file creation

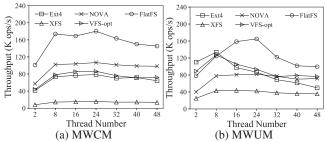


Figure 12: Scalability of FxMark Benchmark

and removal, which achieves a maximum 2× performance speedup. Other file systems exhibit low namespace insert/remove performance due to inefficient dentry index structure design. For instance, PMFS uses a linear array to organize entries in a directory [31]. When there are a large number of files, searching an entry in the linear array takes long time. We even fail to run these two benchmarks on PMFS and BetrFS due to their extremely low dentry search performance.

Second, the inode lock is the major scalability bottleneck for file creation and removal. All in-kernel file systems including FlatFS adopt the same inode locking scheme. Thus, their performance trends with different thread numbers are similar. However, FlatFS still outperforms others for its higher namespace performance. Third, NUMA impacts greatly affect file system scalability. Both Figure 12a and Figure 12b show that all file systems experience a degradation when there exists remote memory node access beyond 24-threads.

# 5.3 Factor Analysis of Each Optimization

We analyze the performance benefits of three optimization techniques in FlatFS.

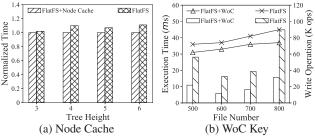


Figure 13: Performance Benefit of Node Cache and WoC Key

**Node caching optimization.** The node cache design avoids tree traversal for entries in the same tree node. It is especially useful for sequential file access. We create four large directories containing  $10^4$ ,  $10^5$ ,  $10^6$ ,  $10^7$  files, respectively. The corresponding  $B^r$  tree height ranges from three to six. We create a microbenchmark to access all files in the directory. Figure 13a shows FlatFS achieves 2.01%-11.07% performance improvements with node caching optimization. In addition, when tree height increases, FlatFS obtains more benefits from tree node caching.

**Rename optimization.** The rename operation is cheap and fast in directory tree-based namespace. It takes 0.004s to move a Linux-4.4 source code repository. We measure two

rename implementations in FlatFS. The slow rename implementation removes all associated entries in the flat namespace, updates pathname keys, and then inserts all entries. Its latency is ten times slower than conventional file systems. Fortunately, FlatFS improves rename performance with tree range operation. Its optimized rename takes 0.007s.

WoC key design. We use a microbenchmark to measure the performance benefits of WoC key design. The microbenchmark creates a large number of files in a directory. File creation causes index key prefix and suffix adjustments, which incurs high data persistence costs. Figure 13b demonstrates that FlatFS+WoC delivers average 15.94% lower latency than FlatFS without WoC key optimization. We also use IPMWatch tool [9] to collect the number of writes received by the memory controller. Figure 13b shows WoC key design achieves a nearly 2.64× write reduction.

# 5.4 Version Control System: Git

We demonstrate FlatFS performance benefits with git application. We create a large deep directory tree containing ten Apache Hadoop 2.10.1 source code repositories [3]. The average directory depth of all files is 17.71. We use two frequently used git commands git status and git commit to evaluate file system directory reading and path walk performance. In the original git status implementation, it traverses the directory tree of the target repository recursively, reads every directory entry, validates its state, and puts it into the associated list according to its state (e.g., *untracked*). Figure 14a shows that FlatFS outperforms others by up to 2.21×.

Besides, we also modify the git status implementation using the *getdents\_recur* syscall in Section 4.3 (denoted as Git-opt+FlatFS in Figure 14a). The *getdents\_recur* system call performs a range query to the target directory and sub-directories and returns all entries at a time. It greatly improves git status performance by eliminating expensive hierarchy tree traversal. The optimized git status only takes 0.6s, which reduces the latency by 4.12× compared to the unmodified *git status* command.

The git commit first uses *lstat* to check the file existence, then it opens every file and reads its content using a *mmap* system call. The massive number of *lstat* and *open* syscall in git commit greatly stresses the path walk efficiency. The experimental results show that FlatFS reduces the git commit latency by 13.79%-37.50% compared to others.

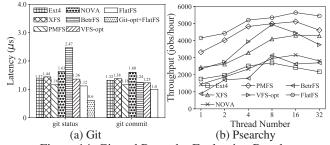


Figure 14: Git and Psearchy Evaluation Results

#### 5.5 File Indexer: Psearchy

Psearchy is a parallel version of searchy [11]. The Psearchy creates multiple worker threads. Each thread processes a number of files. Specifically, the thread opens every file, reads file content, records word positions in a local hash table, sorts word positions, and persists results in a Berkeley DB file. We set the local hash table size 16 MB. We also create a dataset using the taxonomy of known animals from the Catalogue of Life website [4]. The dataset directory hierarchy is generated according to the hierarchy of biological classification. It contains 276,616 files and directories. Because the file size is small, the psearchy performance is largely dependent on file open performance. Figure 14b shows that applications throughput increases as the thread number increases. Among them, FlatFS always achieves the highest throughputs. However, scalability issues caused by the glibc *qsort* function and VFS inode hash table lock prevent throughput improvements as thread number increases.

Table 4: Total and stat Syscall Execution Time of Hive

	Ext4	XFS	NOVA	PMFS	BetrFS	VFS-opt	FlatFS
Total	8.44 <i>s</i>	8.71 <i>s</i>	8.55 <i>s</i>	8.50s	8.39s	8.34 <i>s</i>	8.28 <i>s</i>
Stat	1.46s	1.40s	1.52s	1.34s	-	-	1.18s
μs/Ca	11 3.07	2.94	3.20	2.81	-	-	2.50
Gain	18.4%	15.1%	21.8%	11.0%	-	-	

# 5.6 Data Warehousing System: Hive

Apache Hive is a data warehousing system that uses database table partition to improve data query performance [39]. We use TPC-H benchmark [8] to create a database table for evaluation. Then we use table partition technique to generate a dataset containing 74,090 files and directories. We create a Hive SQL benchmark to perform queries to all files. The major performance bottleneck of Hive is the Java virtual machine. Therefore, Table 4 shows FlatFS slightly outperforms other file systems in total execution time.

During Hive execution, We find that stat is the most frequently invoked metadata syscall. To measure the performance benefits of our namespace design, we uses the *strace* tool [7] to collect the stat syscall execution time. We fail to profile BetrFS and VFS-opt due to unknown bugs. Table 4 reports that FlatFS achieves 16.58% better performance than others in stat syscall for its optimized file path walk.

#### 6 Related Works

File Metadata Indexing Optimization. TableFS [32] and BetrFS [18] share similar idea that utilize existing key-value stores (LevelDB and TokuDB) to improve metadata indexing performance. TableFS [32] aggregates file metadata in a LevelDB table indexed by a combination of parent directory inode number and filename string. BetrFS utilizes the TokuDB to optimize both metadata and data block indexing performance with their full-path indexing schema [18]. These two file systems achieve good performance running on hard disks. However, they introduce new software layers into the

existing storage stack. Our experimental results show that such stacked system architecture is inefficient for file systems built for fast NVMs. FlatFS exhibits a non-caching system architecture to avoid data copy costs for ultra-fast NVMs.

File Path Walk Optimization. ByVFS [41] bypasses the VFS layer during path component resolution and directly fetches dentries from the device. Tsai et al. [40] propose two techniques to improve file path walk performance in the VFS layer. First, they reduce the pathname lookup latency using full-path indexing via an in-memory hash table. Second, they reduce the dcache hit latency by caching the permission results of file paths. DLFS [21] re-organizes the on-disk metadata and proposes a hashing-based metadata indexing solution to improve pathname lookup performance. Directory range operations (e.g., rename) are expensive in their system. FlatFS incorporates a full-path-at-a-time path walk model to accelerate pathname lookup performance.

Namespace Structure Optimization. ReconFS [22] decouples the file system namespace as a volatile and a persistent directory tree. It improves system performance and device endurance by reducing metadata writes to flash memory. Partition is an efficient solution to improve namespace scalability. IndexFS [33] proposes a scalable directory service based on GIGA+ [27] to dynamically partition a large directory tree across multiple nodes in the distributed environment. SpanFS [19] is a scalable local file system that distributes the global namespace into multiple disjoint domains to avoid contention caused by centralized namespace management. FlatFS exploits the flat namespace structure to optimize the namespace lookup and reading operation performance.

#### 7 Conclusion

This paper demonstrates a novel NVM file system FlatFS. FlatFS exploits flat namespace architecture to improve metadata operation performance by proposing three techniques: a coordinated path walk, a range-optimized  $B^r$  tree, and a write-optimized index key layout. Extensive experiments demonstrate the performance benefits of FlatFS to metadataintensive benchmarks and applications. FlatFS is open source at https://github.com/miaogecm/FlatFS.git.

# Acknowledgments

We thank the reviewers for their helpful feedback. We especially thank our shepherd Haris Volos for the careful, thorough reading of our paper and valuable suggestions to improve this paper substantially. This paper is supported by Fundamental Research Funds for the Central Universities (No. B220202073, B210201053), National Natural Science Foundation of China (No. 61832005, 61872171), CCF-Huawei Innovation Research Plan (No. CCF2021-admin-270-202101), Natural Science Foundation of Jiangsu Province (No. BK20190058), Future Network Scientific Research Fund Project (No. FNSRFP-2021-ZD-7), Jiangsu Planned Projects for Postdoctoral Research Funds (No. 2021K635C). Baoliu Ye is the corresponding author.

## References

- [1] Intel 3D XPoint. https://newsroom.intel.com/n ews-releases/intel-and-micron-produce-brea kthrough-memory-technology/, 2015.
- [2] Google LevelDB. https://github.com/google/le veldb, 2018.
- [3] Apache Hadoop Downloads. https://hadoop.apach e.org/releases.html, 2021.
- [4] Catalogue of Life. https://www.catalogueoflife. org/data/download, 2021.
- [5] Ext4 Disk Layout. https://ext4.wiki.kernel.org /index.php/, 2021.
- [6] Percona TokuDB. https://www.percona.com/soft ware/mysql-database/percona-tokudb, 2021.
- [7] Strace: Trace System Calls and Signals. https://ma n7.org/linux/man-pages/man1/strace.1.html, 2021.
- [8] TPC-H. http://www.tpc.org/tpch/, 2021.
- [9] Intel Persistent Memory Watch. https://github.com /intel/intel-pmwatch, 2022.
- [10] Daniel P. Bovet and Marco Cesati. Understand the Linux Kernel. O'Reilly Media, 2006.
- [11] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Tappan Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In USENIX Symposium on Operating Systems Design and Implementation, pages 1–16, 2010.
- [12] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. Proceeding of VLDB Endowment, 8(7):786-797, 2015.
- [13] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In USENIX Conference on File and Storage Technologies, pages 81-95, 2021.
- [14] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In ACM Symposium on Operating Systems Principles, pages 133-146, 2009.

- [15] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-space NVM File System. In ACM Symposium on Operating Systems Principles, pages 478-493, 2019.
- [16] Mingkai Dong and Haibo Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In USENIX Annual Technical Conference, pages 719–731, 2017.
- [17] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. Proceeding of VLDB Endowment, 14(4):626-639, 2020.
- [18] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In USENIX Conference on File and Storage Technologies, pages 301–315, 2015.
- [19] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In USENIX Annual Technical Conference, pages 249–261, 2015.
- [20] Takayuki Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. IEEE Design & Test of Computers, 28(1):52-63, 2011.
- [21] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct Lookup and Hash-based Metadata Placement for Local File Systems. In International Systems and Storage Conference, pages 1-11, 2013.
- [22] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: a Reconstructable File System on Flash Storage. In USENIX conference on File and Storage Technologies, pages 75-88, 2014.
- [23] Jim Mauro and Richard McDougall. Solaris Internals: Core Kernel Components, volume 1. Prentice Hall Professional, 2001.
- [24] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. The Design and Implementation of the FreeBSD Operating System. Pearson Education, 2015.
- [25] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In USENIX Annual Technical Conference, pages 71–85, 2016.
- [26] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-Volatile Main Memory. In

- European Conference on Computer Systems, pages 1-16, 2016.
- [27] Swapnil Patil and Garth A. Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In USENIX Conference on File and Storage Technologies, pages 177-190, 2011.
- [28] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. System Evaluation of the Intel Optane Byte-addressable NVM. In International Symposium on Memory Systems, pages 304–315, 2019.
- [29] Rob Pike. Lexical File Names in Plan 9, or, Getting Dot-Dot Right. In USENIX Annual Technical Conference, pages 85–92, 2000.
- [30] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System using Phase-Change Memory Technology. In International Symposium on Computer Architecture, pages 24-33, 2009.
- [31] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In European Conference on Computer Systems, pages 1–15, 2014.
- [32] Kai Ren and Garth A. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In USENIX Annual Technical Conference, pages 145–156, 2013.
- [33] Kai Ren, Qing Zheng, Swapnil Patil, and Garth A. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In International Conference for High Performance Computing, Networking, Storage and Analysis, pages 237-248, 2014.
- [34] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. ACM Transactions on Storage, 9(3):1–32, 2013.
- [35] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. High-Performance Transaction Processing in Journaling File Systems. In USENIX Conference on File and Storage Technologies, pages 227–240, 2018.
- [36] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. Nature, 453(7191):80, 2008.
- [37] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In USENIX Annual Technical Conference, pages 1-14, 1996.

- [38] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *Usenix Magazine*, 41(1), 2016.
- [39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. Proceeding of VLDB Endowment, 2(2):1626-1629, 2009.
- [40] Chia-che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value from Your File System Directory Cache. In ACM Symposium on Operating Systems Principles, pages 441–456, 2015.
- [41] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or Not: Rethinking Virtual File System for Non-Volatile Main Memory. In USENIX Workshop on Hot Topics in Storage and File Systems, 2018.
- [42] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In International Symposium on Microarchitecture, pages 496-508, 2020.
- [43] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon D. Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *International* Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-19, 2019.
- [44] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In USENIX Conference on File and Storage Technologies, pages 323–338, 2016.
- [45] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Silva, Steven Swanson, and Andy Rudoff. NOVA-fortis: a Faulttolerant Non-Volatile Main Memory File System. In ACM Symposium on Operating Systems Principles, pages 478-496, 2017.
- [46] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In USENIX Conference on File and Storage Technologies, pages 169-182, 2020.
- [47] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and

- Jun Yuan. The Full Path to Full-Path Indexing. In USENIX Conference on File and Storage Technologies, pages 123-138, 2018.
- [48] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a Tiered File System for Non-Volatile Main Memories and Disks. In USENIX Conference on File and Storage Technologies, pages 207-219, 2019.

# **Artifact Appendix**

#### A.1 Abstract

We provide FlatFS artifact description in this section. FlatFS is a metadata-optimized NVM file system that features a flat namespace. This section describes (1) how to build FlatFS on NVM systems §A.2; (2) how to reproduce the main experimental results of our paper §A.3.

#### How to Build FlatFS **A.2**

This section describes software requirements for FlatFS and how to build FlatFS.

• OS version: Ubuntu 18.04 or Ubuntu 14.04.6

• **Kernel version:** Linux 4.15.0

#### 1. Download FlatFS

```
$ git clone https://github.com/miaogecm/FlatFS.git
```

This repository contains FlatFS source code which locates in linux-4.15/fs/flatfs and a modified virtual file system that supports coordinated path walk.

#### 2. Compile and Install FlatFS

```
$ make localmodconfig
$ make menuconfig
```

Modify the Linux kernel configuration file, and make sure these two configurations are enabled:

```
File systems/DAX support
File systems/FlatFS
```

#### and disable these four configurations:

```
Security options/AppArmor support
Security options/Yama support
Security options/TOMOYO Linux support
Security options/Security hooks for pathname based
     access control
```

#### Compile the kernel

\$ make -j

#### Install the new kernel:

```
$ make install
$ make modules_install
$ update-grub
```

#### Reboot the system:

\$ reboot

#### 3. Mount FlatFS

FlatFS can be mounted on a real or an emulated NVM device. Create a mount directory and mount FlatFS:

```
$ mkdir /mnt/flatfs
$ mount -t flatfs -o init /dev/pmem0 /mnt/flatfs
```

#### 4. Umount FlatFS

\$ umount /mnt/flatfs

# **Experiment Reproducibility**

We provide a number of helpful scripts to reproduce the main experimental results automatically. Specifically, readers could reproduce Figure 9, Figure 10a, Figure 10b, Figure 11a, Figure 11b, Figure 14a, Figure 14b, Table 2, Table 3, and Table 4.

Reproducing an experiment takes three steps except for the Hive experiment.

#### **Step 1.** Clean old data:

If you are reproducing Table 4, generate data for the experiment first:

```
$ export TBL_PATH=~/hive/table
$ ./mktable
```

**Step 2.** Collect data for each tested file system, where FS=ext4, xfs, pmfs, nova, flatfs, betrfs, vfs\_opt. The generated data is saved in a .data file in the current directory.

\$ ./run \$FS

## **Step 3.** Draw the figure with collected data:

\$ ./plot.py

More details can be found in evaluation/README.md in the repository.