Tags: A Unifying Primitive for the Storage Data Path

Weisu Wang, *Florida State University*
An-I Andy Wang, *Florida State University*
Christopher Meyers, *Ansible, Inc.*
Sarah Diesburg, *University of Northern Iowa*

Abstract

The legacy storage data path is largely structured in black-box layers and has four major limitations: (1) functional redundancies across layers, (2) poor cross-layer coordination and data tracking, (3) presupposition of high-latency storage devices, and (4) poor support for new storage data models.

We introduce *Tags*, a unifying primitive that can be used throughout the storage data path. This white-box approach enables all storage layers to coordinate and track data using shared data structures that are constructed through the Tags API. Our case studies show that by eliminating redundant services, our Tags-based key-value store can outperform LevelDB by 20-170% when inserting and deleting 100-byte key-value pairs. We also build a Tags-based file system (TagFS) to demonstrate the usability and robustness of Tags. In addition, we build per-file secure deletion via TagFS to show data-path-wide coordination and data tracking.

1. Introduction

The legacy storage data path is structured in layers and is largely disk-centric. Layering offers good abstraction with which hide underlying details, enabling each layer to evolve swiftly. The storage-wide disk-centric assumptions reflect storage devices' continuing standing as a system-wide bottleneck for decades.

However, disks are replaced by low-latency SSDs, which have very different traits. Applications also demand more coordination and control across storage layers (e.g., data tracking). These driving forces make us rethink how to preserve the advantages of layering, while granting more cross-layer control and how to design a data model to provide more support for different emerging storage media.

We propose *Tags*, a unifying primitive that enables various data path components to build cross-layer data structures, even across kernel and application boundaries. Tags enables cross-layer coordination and data tracking, supports both disks and SSDs, and eases the extension of new data path features.

1.1. Legacy Storage Data Path

The legacy storage data path is composed of layers (Figure 1.1.1). Under UNIX, the bottom layer consists of device-specific drivers. A higher-level device-driver layer provides services, including mapping (e.g., NAND flash translation layer) and to coordinate multiple devices. The logical, device-independent file-system layer provides file names for data, organization for files, and data layouts on storage media to minimize access overhead. The VFS layer enables multiple file systems to coexist and contains common file-system functions, including caching. Finally, applications issue storage requests via file-system system calls. (Since the Windows storage data path uses similar organization to that of UNIX, we use UNIX terminology in the remainder of this paper.)

The legacy storage data path has four major limitations. First, storage layers are large black boxes and introduce unnecessary functional redundancies and missed opportunities for optimizations. For example, both logical and physical layers try to manage data layouts. Thus, B-trees in databases can be remapped to extent-based trees at the file-system layer [13], and then remapped to linked lists at the flash-translation layer, rendering the original optimization ineffective. Second, layered abstraction hiding makes coordination and data tracking difficult. For example, a device driver cannot discern the file membership of a block [21]. The third limitation is that the legacy data path is not designed for the low-latency storage. Thus, for small IO requests, the storage-stack latency can no longer be masked by low-latency SSDs [23]. Finally, the legacy data path has limited support for new storage data models (e.g., key-value store), and they suffer fates similar to those in the B-tree example and are remapped to underlying storage layers.

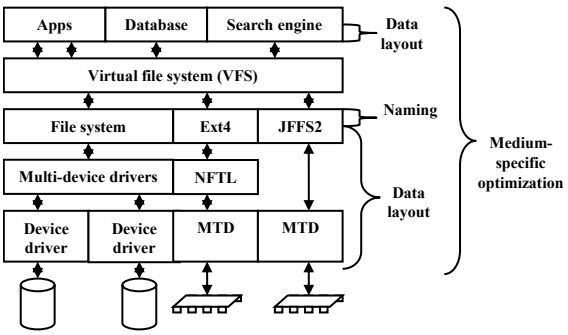

Figure 1.1.1: Conventional storage data path.

1.2. Some Alternatives

One approach to these limitations is to bypass the legacy storage data path by accessing the storage device directly (e.g., direct IOs, DAX [26]). The downside to this is that application programmers may need to duplicate existing services in the legacy storage stack. Some solutions insert layers to separate the management of

metadata and data (e.g., [8]) or deduce information across layers (e.g., [20]). However, these solutions do not address the issues of redundant services and medium-specific mechanisms. Imperfectly deduced information may lead to optimizations based on conservative decisions [2]. To streamline storage requests and avoid redundant services, integrated design across multiple layers is possible (e.g., [22]). However, either such solutions are tailored for specific workloads [17], or the black-box treatment of layers remains and hinders information flow.

1.3. In Search of a New Storage Data Path Design

Legacy limitations prompt the question of how to design and build a new storage data path that is modular, and supports data-path-wide coordination, tracking, and emerging storage media. A more fundamental question is what makes a storage system a storage system? In essence, a storage system provides storage and retrieval of data. At the minimum, a storage data path needs the ability to store and retrieve data from a storage medium, and to tag data to provide persistence and control. From these basic requirements, we can rethink and design a unifying framework that addresses various limitations of the legacy data path.

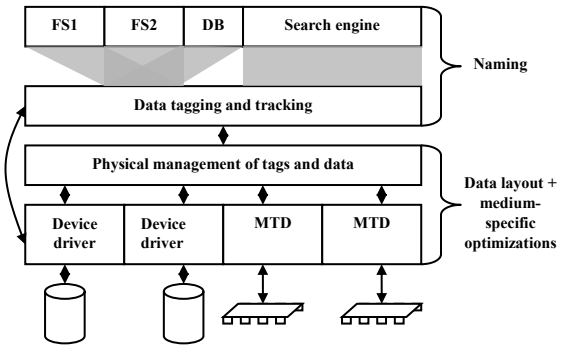

Figure 2.1: Tagging-based storage data path.

2. Tags

We introduce the *Tags* framework, which uses *tags*, a unifying primitive (Figure 2.1), to construct shared data structures throughout the storage data path. Conceptually, each piece of data is associated with one or more tags, indicating how data pieces are related and should be handled within the data path. The collection of data pieces and tags forms a single-level data tagging layer. To ease coordination, these tags provide global and logical communication throughout the data path. Tags also provide a common denominator for high-level storage layers and applications, providing enough flexibility to accommodate the direct construction of name spaces by file systems and of indices by databases and to bypass redundant services (e.g., data structure remapping). Below the tagging layer, a consolidated layer comprising the physical tags and data management makes informed decisions on how tags and data pieces are accessed and stored. As data traverse through the data path, they can be tracked using tagging.

Unlike in the traditional data path, Tags separates logical access from physical storage management, which enables medium-specific optimizations, easing the accommodation of emerging storage technologies.

3. Tags Design Space

Conceptually, each piece of data is associated with a globally unique ID (i.e., <data ID, "data">). Each data ID can be associated with one or more types of globally registered and extensible tags, each in the form of <tag type ID, data ID>. Figure 3.1.1 shows that the ID for "data" is 0. The access permission tag for "data" refers to the data ID of 1, which is "READ_ONLY". The size tag for "data" refers to the data ID of 2, which is "5".

Although the data model is simple, storage modules can use tags as a common denominator when building data structures for cross-layer coordination and tracking. For example, through data ID indirections, we can build hierarchical graphs commonly used in file systems.

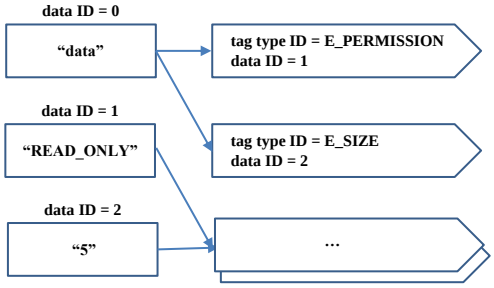

Figure 3.1: Tags primitive example.

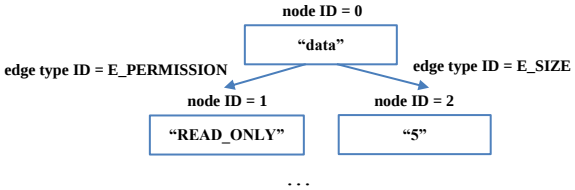

Figure 3.1.1: Graph-based representation of Tags.

3.1 Graph-based API

Because a tag expresses the relationship between two pieces of data, we can logically transform Tags in terms of nodes and edges, with the nodes holding data, and the tags types representing directional edges (Figure 3.1.1).

Figure 3.1.2 shows the core API for Tags. A node can be created to hold a dynamically allocated copy of data. A node can be destroyed given a node ID. An edge-type ID can be created with a given name. To create or delete an edge, we must specify the IDs of both the source and the destination nodes and of the edge type.

Because dangling edges (without end nodes) may lead to corrupted graphs, this API requires that the end nodes be created first, before the edge between the two. Before an edge can be deleted, the nodes must exist on both ends, and the user must delete the edge before deleting the end nodes. When an edge needs to point to NULL, an empty node can be used to assure that each edges is formed between two nodes. Certain edge types involve enumeration (e.g., block ID edge type); thus, when operating on edges, an additional optional info parameter is used to pass in the enumerated number.

A node can be accessed through its ID or through the incoming edge of another node. To disambiguate, although a node can potentially be reached from different nodes through the same in-bound edge type, a node can be associated only with unique out-bound edge types.

```
node_ID = tags_create_node(data, len, …);
tags_delete_node(node_ID);

edge_type_ID = tags_create_edge_type(name);
tags_delete_edge_type(edge_type_ID);

tags_create_edge(src_node_ID, dest_node_ID,
                 edge_type_ID, <edge_info>);
tags_delete_edge(src_node_ID, dest_node_ID,
                  edge_type_ID, <edge_info>);

tags_get_dest_node(src_node_ID, edge_type_ID,
                    <edge_info>);
tags_ID_to_node(node_ID);
```

Figure 3.1.2: Core API for Tags.

```
group_op_ID = tags_begin_group_ops();
tags_abort_ops(group_op_ID);
tags_commit_ops(group_op_ID);
```

Figure 3.2.1: Group operations for Tags.

3.2 Group Operations

One problem with using the graph-based API on fine-grained tags is achieving atomicity across many tags operations. Any failure along a sequence of graph operations would require lengthy cleanup code. To mitigate this problem, we added group operations (Figure 3.2.1). If an error occurs between the begin and commit calls, the abort call automatically performs the graph cleanup and the rollback to the graph states.

For instance, Tags periodically takes snapshots of edges (using semantics akin to the ordered journaling mode for ext3) and maintains a list of committed and pending group operations. A new snapshot can be created by applying committed group operations to the latest snapshot. As the periodic snapshot only concerns edges and is built incrementally, the overhead is light. In the case of aborting group operations, Tags tries to undo operations when it is possible. When it is not, Tags rolls back to the most recent snapshot.

This rollback exploits two Tags properties. First, a node must be created prior to establishing its edges. When aborting a group operation that involves creating a new node, all edge operations on the newly created node can be discarded, and the new node can be deleted. Second, all edges related to a node must be deleted before deleting the node. Thus, when a node deletion is aborted, the node should no longer be reachable by the remaining graph. Therefore, delayed node deletion via garbage collection suffices.

3.3 Physical Representation

In a nutshell, Tags is a single-level store with operations revolving around nodes and edges.

Nodes: Tags nodes are variable-sized, memory-mapped storage chunks governed by a memory allocator (e.g., slab [3] or buddy allocators [11]). A node's memory address (offset by the starting memory-mapped address) is used as a unique ID for that node, freeing us from implementing node-allocation management.

Edges: Tags edges are implicitly stored in an extensible hash table [5]. Basically, hash(source node ID, edge type ID, edge info) returns the destination node ID. The destination node can be tagged with a magic number to perform a dynamic type check prior to accessing the node's content.

Persistence: To survive reboots, the states of the memory allocator must be persistent, using techniques similar to [25]. The governed memory is divided into separate flushable regions for persistent states, ephemeral states (to optimize the Tags internal data structures), shared memory (for IPCs), snapshots and journals (for rollbacks), and reserved locations (for the states of the memory allocator itself).

Data layout: With the storage organization for Tags, data layouts are largely governed by the representation of the hash table and by the memory allocator. Thus, by exploiting the notion of temporal and spatial locality for hashing and for memory allocation, we can tune the system's performance by aligning the characteristics of the workload and with the characteristics of the underlying storage devices. Currently, we use a customized slab allocator [3] for sub-page requests and buddy allocators [11] for requests larger than one page. Alternatively, we can use hierarchical hashing or a log-structured memory allocator [15] to exploit spatial or temporal locality.

3.4 Access control

Since Tags aims to create primitives smaller than the granularity of common data structures, we anticipate many small tags, rendering high overhead for per-node permission checks. Allowing edges to be created between any two nodes is also an unwieldly way to enforce the permission to access restricted nodes. However, since many tags share the same permission, it would make sense to check and enforce permissions at fewer locations. Also, a certain degree of restrictions

on how edges can be formed can help manage the access control properties of the resulting graph topology.

Super nodes: The idea of super nodes (or ***s-nodes***) is to reduce the number of places where permissions are set and checked: only s-nodes have edges to permission nodes. All nodes belonging to the same s-node implicitly share the same permissions. In terms of the restrictions, edges can be created from an s-node to its nodes (Table 3.4.1). Also, edges can be created from any node to an s-node, since that destination s-node can enforce the access permissions. However, forming edges between nodes that are under different s-nodes is prohibited. Also, one source s-node cannot create an out-bound edge node to a node that is under another s-node.

One challenge to realizing s-nodes is finding a node's s-node without additional edges or lookup tables. Since our unique node IDs are based on 64-bit memory-mapped addresses, we borrowed unused *S* high-order bits. An s-node ID is a unique *S*-bit number, zero-extended to form a 64-bit ID. To access its nodes, we also must connect the s-node to at least one of its nodes. To locate the permission from a node under an s-node, we hash(zero extended upper *S* bit of the node ID, permission edge type ID).

In terms of the API, a programmer can use a special call to create s-node IDs and use them to create node IDs (Figure 3.4.1). The s-node tracks the number of nodes created beneath it. To delete an s-node, all its nodes must first be deleted. Otherwise, the permission of the undeleted node will be either undefined, or defined by a newly allocated s-node with a reused s-node ID.

Sessions: Since node IDs are capabilities, we need the ability to revoke privileges. Thus, other than for the root node ID, the user should interact with Tags through translated IDs. This mechanism is enabled using session open and close calls and a primitive translation table (Figure 3.4.1). An open session call is needed before accessing the translated root node of a Tags graph. All subsequent node IDs obtained from the root node's edges are translated via a translation table. At the end of a program, a close session call is needed to delete the translation table. Optionally, a timeout can be specified to close a session when the system registers no activity occurring within a timeout period.

Table 3.4.1: Rules for creating edges.

from \ to	s-node A/B	s-node A's nodes	s-node B's nodes
s-node A	Yes	Yes	No
s-node B	Yes	No	Yes
s-node A's nodes	Yes	Yes	No
s-node B's nodes	Yes	No	Yes

```
s_node_ID = tags_create_s_node(mode);
node_ID = tags_create_node(data, len,
                           s_node_ID);
```

Figure 3.4.1: Super node operations.

```
translated_root_node_ID
  = tags_open_session(root_node_ID,
                      <time_out_minutes>);
tags_close_session(translated_root_node);
```

Figure 3.4.2: Super node operations for sessions.

4. Implementation

Tags is prototyped in C as a user-level library. Tags applications link and load the library to use the Tags API to perform storage tasks. Figure 4.1 shows how a Tags-based key-value store (§5.1) uses the Tags library to interact with the kernel and communicates with the kernel via memory mapping and shared memory.

Figure 2.1 shows the two major components of Tags. The data-tagging and data-tracking component implements the graph API, the nodes and edges, the group operations, and the access control. The physical management component implements the persistent memory allocator, which also controls the physical data layout. Currently, the Tags library does not support sessions or multi-threaded and nested group operations.

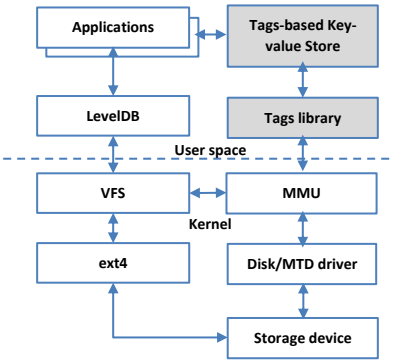

Figure 4.1: Storage data paths for Tags-based key-value store (shaded boxes) and LevelDB [6].

5. Tags Evaluation via Case Studies

While evaluating Tags, we wanted to show (1) its ability to avoid redundant layered features when supporting new data models, (2) its usability and robustness when building complex software, (3) its ability to coordinate and track data across layers, and (4) its ability to perform well with both disk and SSD storage media.

To show that Tags can perform well with HDDs and SSDs, in each experimental setting, we conducted benchmarks on both media. Each experiment was repeated 5 times and presented at the 95% confidence interval. Table 5.1 shows the system configuration.

5.1 Tags-based Key-value Store

To show the benefit of the direct support for new data models, we prototyped a key-value store using the Tags

library. The data path had no file system and associated redundant efforts to manage data layout (Figure 4.1).

Given that Tags is built on a hash table that stores edges to nodes, Tags operations can be directly mapped to support key-value store operations. We began by creating a root node. For the key-value `Put(key, data)` operation, we created a node to store the data and used the key as an edge type ID. For `Get(key)`, we called `tags_get_dest_node(root node ID, key)` to retrieve the data node. For `Delete(key)`, we called `tags_delete_edge(root node ID, node ID)`, followed by `tags_delete_node(node ID)`.

We compared the Tags-based key-value store with LevelDB 1.9.0 [6]. Figure 4.1 shows the differences between the two data paths. For the workload, we inserted 10 million, 100-byte key-value pairs, each with 16-byte keys. Figures 5.1.1 and 5.1.2 show the results.

For both storage media, Tags and LevelDB have similar read performance, since both systems use memory-mapped IOs to avoid copying. Both systems also use bulk updates (group operation for Tags) to speed up small updates. For disks, Tags can outperform LevelDB in terms of inserts by a factor of 1.8 and for deletes, by a factor of 2.7. For SSDs, Tags can outperform LevelDB in terms of inserts by a factor of 1.2, and for deletes, by a factor of 1.6.

5.2 Tags-based File System

To demonstrate usability, we prototyped TagFS to show that the interface and primitives provided by Tags are expressive enough to build meaningful and complex applications. TagFS was implemented at the user space via the FUSE framework [24]. Figure 5.2.1 illustrates the flow of data requests.

TagFS translates POSIX file system calls into Tags-based nodes and edges, and this task involves many node and edge operations, simplified by group operations. Basically, all i-nodes (permission holding nodes) are replaced with s-nodes, and all attributes are accessed through edges (Figure 5.2.2). Directory entries can be accessed via ID hashes. For traversals, a directory entry can locate the next and previous entries through hash(current ID, next edge type) or hash(current ID, previous edge type). Data blocks are accessed through enumerated edges to support indexing on top of the hashing data structure.

Although we could instead use a single node to contain all attributes of an i-node, we explored this pedantic scenario to show that even if Tags are naively applied, we can still configure the system to achieve reasonable performance. We compared our TagFS with ext4 stacked on FUSE. The times elapsed for TagFS and ext4 + FUSE to compile the openSSL (v1.1.0f) [2017] were statistically the same (87 $\pm$ 0.01 seconds).

When running LFS large-file and small-file benchmarks [14], TagFS performed reasonably well when its block size reaches 32KB, to amortize the cost of fine-grained access to attribute nodes and dynamic type checks (Tables 5.2.1 and 5.2.2). Future work will include optimizing the dynamic behavior of Tags.

Table 5.1: System configurations.

CPU	2.2Ghz Intel® Xeon® E5-2430, 15MB cache
Memory	32 GB RDIMM 1333 MT/s
HDD	Seagate® SAS 146GB 15K RPM
SSD	Intel® S3500 200 GB SATA Value MLC
Operating system	Linux Mint 3.19

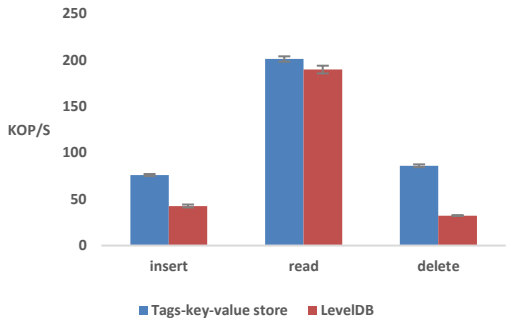

Figure 5.1.1: Key-value store performance for HDD.

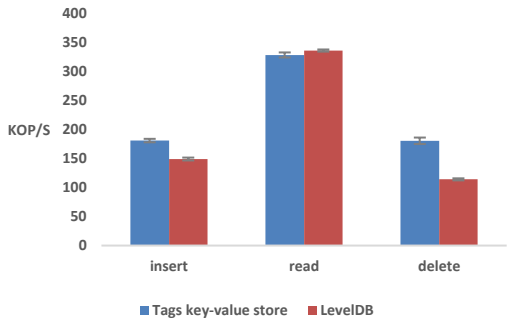

Figure 5.1.2: Key-value store performance for SSD.

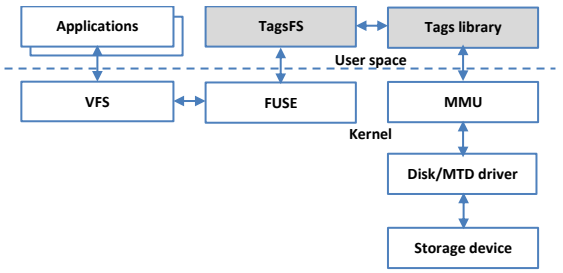

Figure 5.2.1: TagFS and the Tags library (shaded).

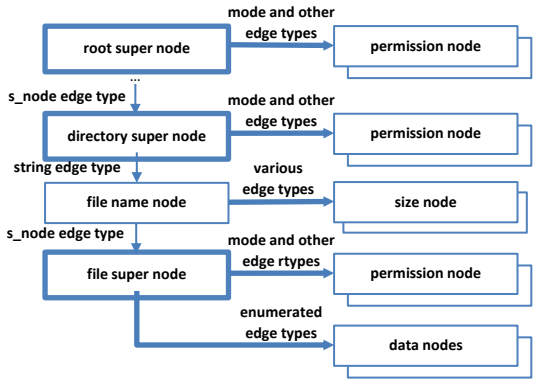

Figure 5.2.2: The Tags representation of file system.

Table 5.2.1: LFS large-file benchmark numbers (MB/s), with one 512MB file for HDD and one 2GB file for SSD.

		Seq w	Rand w	Seq r	Rand r
Tags + FUSE	**HDD**	**190** (±2.4)	**25** (±1.0)	**190** (±1.7)	**26** (±0.4)
	SSD	**240** (±4.8)	**120** (±1.1)	**340** (±7.9)	**140** (±1.9)
Ext4 + FUSE	**HDD**	**91** (±0.2)	**43** (±0.3)	**190** (±1.1)	**2.7** (±0.0)
	SSD	**150** (±3.2)	**100** (±3.2)	**350** (±4.6)	**39** (±0.4)

Table 5.2.2: LFS small-file benchmark numbers (ops/sec), with 20K 16KB file for HDD and 100K 16KB files for SSD.

		Create	Read	Delete
Tags + FUSE	**HDD**	**1.4K** (±21)	**5.5K** (±170)	**5.5K** (±260)
	SSD	**3.8K** (±66)	**12K** (±120)	**17K** (±290)
Ext4 + FUSE	**HDD**	**1.6K** (±27)	**2.6K** (±100)	**7.3K** (±290)
	SSD	**4.1K** (±80)	**5.1K** (±120)	**19K** (±1.5K)

5.3 Per-file secure deletion

To demonstrate cross-layer coordination and tracking, we augmented TagFS with a per-file secure-deletion feature akin to that of TrueErase [4]. First, a user can use `chattr +s` to set the secure-deletion bit of a file at the file-system layer. However, by the time a storage request arrives at the device driver layer, the layer can no longer tell the file membership of a block.

In TagFS, since each group of nodes is governed by an s-node to manage the permission, any node (e.g., a data block node) under an s-node can reach the s-node (see §3.4). Then, TagFS can access the permission. The secure-deletion bit indicates that the corresponding overwrite or truncate should be handled securely.

We handled the disk case by zeroing out data blocks that needed to be securely overwritten and truncated at the block layer. Without the open FTL and raw flash setup, we did not implement this feature. Note that the TRIM command is insufficient, since it only specifies what pages are obsolete, so that the garbage collection would not migrate them as live pages during the garbage collection process [19].

6. Related Work

Since the advent of SSDs, research systems have attempted to address some of the limitations posed by the legacy storage data path.

Cross-layer redundancy removal: Conquest [25] and TableFS [12] have dedicated data paths for large files and remaining small files and metadata. File-system journals can be turned off for databases [17].

Cross-layer coordination: The gray-box approach leverages inferred information across layers for coordination [1]. TrueErase [4] provides an auxiliary data path, so that a file system can propagate the information to the device layer to indicate whether a file needs to be securely deleted or overwritten. Willow [16] augments the data path with customizable API to coordinate across layers.

Support for low latency storage: JFFS [27] consolidates logging for the file-system and the device-driver layers. DAX [26] uses direct IOs and bypasses the memory caching designed for high-latency storage. Arrakis [10] removes the kernel from the data IO path. IO requests are routed to and from the applications' address spaces. To perform IOs, applications rely on a user-level IO stack that is provided as a library.

Support for new storage data models: [18] shows how mixed workloads from file systems and databases can be efficiently handled using separate KVFS and KVDB layers. Cassandra [7] uses a customized graph API to store and retrieve data objects.

7. Lessons Learned and Future Work

Tags takes a minimalist approach to design and building a storage data path. The idea seemed simple; however, Tags began as an analogue of sticky notes and was transformed into graph nodes and edges, implemented with the semantics of single-level stores and representations akin to those of key-value stores. Little did we know that this journey would lead us to revisit numerous legacy concepts and design decision, and help us develop a better appreciation of storage advances.

Low-level single-level store model is tricky to program: When building the core Tags, low-level single-level-store style programming was confusing at times. Since the memory allocator and all its allocated memory regions are persistent, all changes to memory data structures may result in unintended IOs. To overcome this hurdle, we separated persistent and transient data structures. Fortunately, users need only to handle node and edge IDs.

Locality is still important for hashing: We avoided hashing repeated path prefixes using the parent path ID as a seed to short-circuit the hash functions; however, effectively, this scheme made hashing hierarchical. Our future work will find additional ways to improve the locality of hashing.

Access control dictates the unit of access: Although Tags allows fine-grained data representation and organization beyond the granularity of legacy data structures, the access control dictates which groups of tags are accessed together and how they can form graphs.

Convoluted path forward: In some ways, the Tags design reintroduced certain aspects of the components of legacy storage data path (e.g., group operations). However, once we pierce through the legacy data structures, with fine-grained system calls, we can directly support data and metadata layouts not previously possible [28].

8. Conclusions

We have presented Tags, a white-box approach to addressing legacy storage data path constraints. Using a unifying primitive and an API of nodes and edges, we have shown how Tags can be used to build applications as complex as a file system and robust enough to compile the Linux kernel. The Tags-based key-value store shows how direct system support and bypassing redundant services can significantly improve performance for both disks and SSDs. Tags also eases data-path-wide tracking and coordination to support features such as per-file secure deletion.

Acknowledgement

This work is sponsored by FSU and NSF CNS-0845672. Opinions, findings, conclusions, or recommendations expressed in this document do not necessarily reflect the views of FSU, NSF, or the U.S. Government.

References

[1] Arpaci-Dusseau AC, Arpaci-Dusseau RH. Information and Control in Gray-box Systems. *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[2] Arpaci-Dusseau AC, Arpaci-Dusseau RH, Bairavasundaram LN, Denehy TE, Popovici FI, Prabhakaran V, Sivathanu M. Semantically-smart Disk Systems: Past, Present, and Future. *Proceedings of the 2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2006.

[3] Bonwick J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. *Proceedings of the USENIX Summer 1994 Technical Conference (ATC)*, 1994.

[4] Diesburg S, Meyers C, Stanovich M, Mitchell M, Marshall J, Gould J, Wang AIA, Kuenning G. TrueErase: Per-file Secure Deletion for the Storage Data Path. *Proceedings of the 2012 ACM Annual Computer Security Applications Conference (ACSAC)*, 2012.

[5] Fagin R, Nievergelt J, Pippenger N, Strong HR. Extensible Hashing—A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315-344, 1979.

[6] Ghemawat S, Dean J, LevelDB, https://github.com/google/leveldb, 2017.

[7] Lakshman A, Malik P 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2): 35-40, 2010.

[8] Lu L, Pillai TS, Arpaci-Dusseau AC, Arpaci-Dusseau RH, WiscKey: Separating Keys from Values in SSD-Conscious Storage. *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST),* 2016.

[9] OpenSSL Crytpgraphy and SSL/TLS Toolkit. https://www.openssl.org/news/openssl-1.1.0-notes.html, 2017.

[10] Peter S, Li J, Zhang I, Ports DRK, Woos D, Krishnamurthy A, Anderson T, Roscoe T. Arrakis: The Operating System is the Control Plane. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[11] Peterson JL, Norman TA. Buddy Systems. *Communications of the ACM* 20(6):421-431, 1997.

[12] Ren K, Gibson G. TABLEFS: Enhancing Metadata Efficiency in the Local File System. *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[13] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3), Article No. 9, 2013.

[14] Rosenblum M, Ousterhout JK. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26-52, 1992.

[15] Rumble SM, Kejriwal A, Ousterhout J. Log-structured Memory for DRAM-based Storage. *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[16] Seshadri S, Gahagan M, Bhaskaran S, Bunker T, De A, Jin Y, Liu Y, Swanson S, 2014. Willow: A User-programmable SSD. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[17] Shen K, Park S, Zhu M. Journling of Jounal is (Almost) Free. *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[18] Shetty PJ, Spillane RP, Malpani RR, Andrews B, Seyster J, Zadok E, Building workload-independent

storage with VT-trees. *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[19] Shu F, Obr N. Data set management commands proposal for ATA8-ACS2. http://www.t13.org/documents/UploadedDocuments/docs2007/e07154r2-Data_Set_Management_Proposal_for_ATA-ACS2.pdf, 2007.

[20] Sivathanu M, Prabhakaran V, Popovici FI, Denehy TE, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Semantically-Smart Disk Systems. *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST)*, March 2003.

[21] Sivathanu M, Bairavasundaram LN, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Life or Death at Block Level. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[22] Sun Microsystems. In a Class by Itself—the Solaris 10 Operating System, A Technical White Paper, November 2004.

[23] Swanson S, Caulfield A. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer*, 46(8):52-59, August 2013.

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

[25] Wang AI, Reiher PL, Popek GJ, Kuenning GH, Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. *Proceedings of the 2002 USENIX Annual Technical Conference (ATC)*, 2002.

[26] Wilcox M. DAX: Page Cache Bypass for Filesystems on Memory Storage. https://lwn.net/Articles/618064, 2014.

[27] Woodhouse D. JFFS: The Journaling Flash File System. *Proceedings of the Ottawa Linux Symposium*, 2001.

[28] Zhang S, Catanese H, Wang AIA. The Composite-file File System: Decoupling the One-to-one Mapping of Files and Metadata for Better Performance. *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.