ARTICLE IN PRESS

Journal of Systems Architecture xxx (xxxx) xxx

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

ADAPT: An auxiliary storage data path toolkit

Weisu Wang [a], Christopher Meyers [b], Robert Roy [a], Sarah Diesburg [c], An-I Andy Wang [a,*]

[a] *Florida State University, USA*
[b] *Ansible, Inc, USA*
[c] *University of Northern Iowa, USA*

ARTICLE INFO

Keywords:
Storage data path and file systems

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.

While addressing all these limitations is a daunting challenge, we introduce ADAPT, an auxiliary storage data path toolkit that complements the legacy storage data path to help mitigate these limitations. This toolkit enables all storage layers to coordinate and track data using shared data structures constructed through the ADAPT API. Our case studies have shown that we can directly support applications such as a key-value store without going through the file system. We also built an ADAPT-based file system and prioritized caching to demonstrate the usability, extensibility, and robustness of ADAPT. In addition, we built per-file secure deletion via our ADAPT-based file system to demonstrate 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, which hides underlying details, enabling each layer to evolve swiftly. The storage-wide disk-centric assumptions reflect the decades-long standing of storage devices as a system-wide bottleneck.

However, hard disk drives (HDDs) are routinely replaced by low-latency solid-state storage devices (SSDs), which have very different traits. Applications also demand more coordination and control across storage layers (e.g., tracking and deleting remnants of sensitive data across storage layers). These driving forces caused us to rethink how to preserve the advantages of layering, grant more cross-layer control, and provide a data model with more support for different emerging storage media.

We propose ADAPT, an auxiliary storage data path toolkit, to complement the legacy storage data path. ADAPT enables various data path components to build cross-layer data structures, even across kernel and application boundaries. ADAPT also 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 (Fig. 1.1.1). Under UNIX, the bottom layer consists of device-specific drivers. A higher-level device-driver layer provides services for mapping. Examples include multi-device driver layer that can coordinate multiple devices, a flash translation layer (FTL), and a light non-volatile memory (NVM) subsystem [2]. 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 virtual file system (VFS) layer allows multiple file systems to coexist and contains common file-system functions, including caching. Applications issue storage requests via file-system system calls. (Since the Windows and UNIX storage data paths apply a similar organization, we use UNIX terminology in the remainder of this paper.)

The legacy storage data path has four major limitations. First, storage layers are black boxes and introduce unnecessary functional redundancies and missed opportunities for optimizations. For example, both logical and physical layers attempt to manage data layouts. Therefore, B-trees in database applications can be remapped to extent-based trees at the file-system layer [21] and then remapped to linked

CCS concepts**Information systems** → Information storage Systems
* Correposnidng author.
E-mail address: awang@cs.fsu.edu (A.-I.A. Wang).

https://doi.org/10.1016/j.sysarc.2020.101902
Received 10 June 2020; Received in revised form 8 September 2020; Accepted 2 October 2020
Available online 13 October 2020
1383-7621/© 2020 Elsevier B.V. All rights reserved.

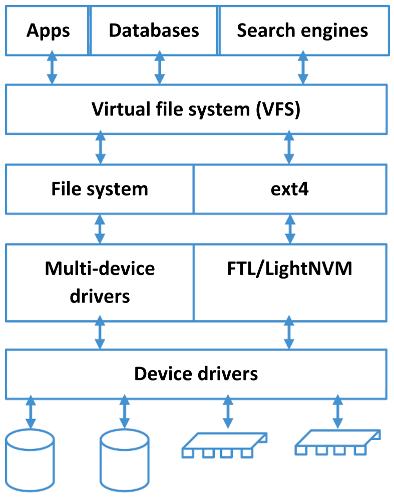

Fig. 1.1.1. Conventional storage data path [2].

lists at the flash-translation layer, rendering the original optimization ineffective. Logical pointers used in memory also need to be *serialized* into a storage format so that the stored pointers can be resurrected or *deserialized* into different memory addresses after a reboot. While not redundant, serialization and deserialization can be avoided with the right system architecture. Another example is logging. A database application has its own commit log; a journaling file system has its own recovery log and uses a copy-on-write log at the flash layer. As a result, several tailored solutions have been designed to consolidate redundant logging [36].

Second, layered abstraction makes coordination and data tracking difficult. For example, a file system interacts with a device driver via reading and writing blocks, and a device driver cannot discern the file membership of a block, whether a block is currently in use or free, or whether a block contains data or metadata [28]. These characteristics make it difficult to implement data-path-wide features, such as per-file secure deletion in which the device driver does not know whether a block belongs to a file to be securely deleted because information is only available at the file-system layer [4].

Third, the legacy data path is not designed for low-latency storage. Thus, for small IO requests, the storage-stack latency can no longer be masked by low-latency SSDs [32].

Finally, the legacy data path has limited support for new storage data models (e.g., key-value store); these models suffer fates similar to those in the B-tree example and are remapped to underlying storage layers.

1.2. 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 [35]). 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., [14]) or to deduce information across layers (e.g., [27]). However, these solutions do not address the issues of redundant services and medium-specific mechanisms. Imperfectly deduced information (e.g., whether a block belongs to a file to be securely deleted) may lead to optimizations based on conservative decisions [1] (when in doubt, delete a block securely). To streamline storage requests and avoid redundant services, integrated design across multiple layers is possible (e.g., [31]). However, some solutions are tailored for specific workloads [24], or the black-box treatment of layers (e.g., the device-driver layer) remains and hinders the information flow.

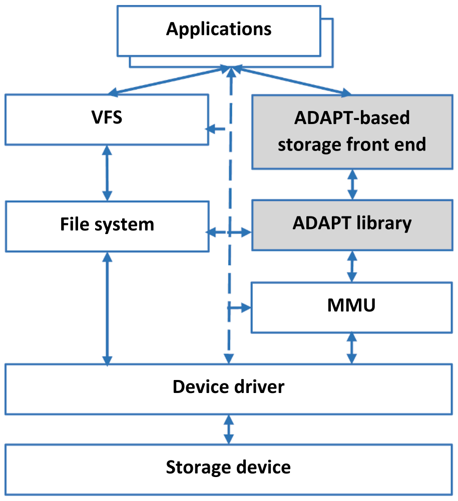

Fig. 2.1. The ADAPT library and its interactions with various storage data path components.

1.3. In search of a remedy

The semantic gap between storage services and the block interface is fundamentally large. While many services, such as [7], fill this gap with local file systems, this solution comes with a significant cost in terms of performance. Our approach consists of a shared data-path-wide library of useful storage primitives to mitigate the limits of the legacy data path. While many solutions address the limitations of the storage data path in specific problem domains, our approach aims to be broadly applicable. Unlike solutions that revamp the entire storage data path, our approach supplements the legacy storage data path, so that storage components that use our library can reap the benefits, while legacy components can still operate.

2. ADAPT conceptual overview

We introduce ADAPT, an auxiliary data path storage toolkit library that enables the coordination of legacy storage components and allows for the quick construction of new storage data path components (Fig. 2.1).

The ADAPT toolkit provides two different deployment models. Legacy applications that are not ADAPT-aware can use either the legacy storage data path or our ADAPT-based POSIX-compliant file system to take advantage of ADAPT-enabled capabilities. Coordination can be achieved through legacy data path components communicating with ADAPT components (via the dashed lines). Examples of ADAPT-enabled capabilities include prioritized caching and secure deletion, as detailed in Section 5. Alternatively, an ADAPT-aware application can go through an ADAPT storage front end (e.g., an ADAPT-based key-value store or a front end based on new data models) and rely on ADAPT to manage data storage. Data structures directly implemented via ADAPT can bypass the common overhead of serialization and deserialization. Having both deployment models simultaneously enables the parts of the system that need the performance to use the right interface while also allowing the legacy applications to continue working. The key to data-path-wide coordination is to allow storage components to use the same set of library primitives to build shared data structures. In designing these primitives, we attempted to explore the lowest common denominator for storage systems. In essence, a storage system minimally provides storage and retrieval of data with some ability to tag data for persistence and control. Based on these observations, we designed our primitives to resolve the notion of ***tags***.

Conceptually, each piece of data is associated with one or more tags, which indicate how the data pieces are related and should be handled

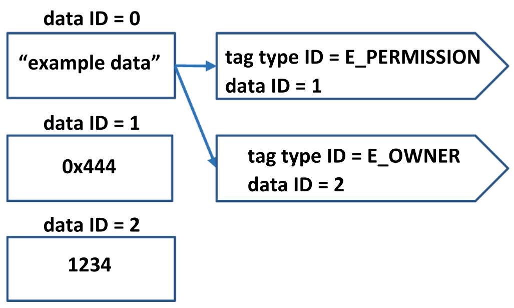

Fig. 3.1.1. ADAPT primitive example.

within the data path. The collection of data pieces and tags forms a single-level store. To ease coordination, operations on these tags provide global and logical communications throughout the data path. Tags can also serve as a common denominator for high-level storage layers and applications, enabling redundant services (e.g., data structure remapping) to be bypassed and allowing for the direct construction of namespaces via file systems and of indices by way of databases. New data models and access methods can also be constructed directly using tags and supported in the ADAPT-based front end. ADAPT can also be used to track data as they flow through the storage data path.

We demonstrated ADAPT by showing how easily we can build native support for key-value systems and file systems. Then, we measured the performance of these systems and confirmed that they have comparable performance to state-of-the-art systems.

To summarize our contribution, we (1) designed and prototyped the ADAPT storage toolkit framework to supplement (not replace) the legacy storage data path and (2) conducted case studies to show that ADAPT can support new data models, avoid redundant serialization and deserialization overheads, be used to build applications as complex as file systems, enable prioritized caching based on file system information made available via ADAPT, and provide data-path-wide per-file secure-deletion functionality.

3. ADAPT design

Before diving into design details, we will first describe the properties and guiding principles of our design.

Backward compatibility: While the legacy storage data path has limitations, many tailor-optimized feature-rich applications and storage data-path components cannot be easily replaced. Therefore, this design point allows ADAPT-aware components to coexist with and complement existing storage components, provide additional data-path-wide communication channels to mitigate existing limitations and build new features. Storage components built from the ground up using ADAPT can coexist with legacy components with and without ADAPT enhancements.

Fine-grained primitives: ADAPT provides fine-grained common denominator primitives for storage components to build shared data structures with arbitrary topologies.

Single-level store: Given that ADAPT aims to allow different storage components across applications and kernel to share data structures, ADAPT uses a single-level store backplane, in which storage components use the ADAPT allocation/deallocation interface to weave and share data structures using persistent pointers.

Atomicity of updates: While fine-grained primitives enable developers to weave complex data structures, ADAPT must provide atomicity of updates to ensure the consistency of the data structure in the event of a crash or failure.

Efficient permission model: Fine-grained primitives can come with high overhead to provide permission control. Therefore, we seek to design mechanisms to lower this overhead.

The following research challenges must be met to realize our storage

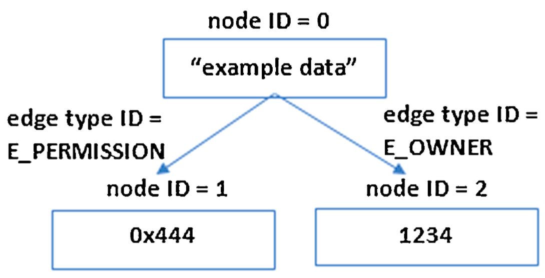

Fig. 3.1.2. Graph-based representation of ADAPT.

```
adapt_guid adapt_create_node(void *data, size_t len, …);
int adapt_delete_node(adapt_guid NID);

adapt_guid adapt_create_edge_type(unsigned char *name);
int adapt_delete_edge_type(adapt_guid etype_ID);

int adapt_create_edge(adapt_guid src_NID, adapt_guid dest_NID,
                      adapt_guid etype_ID, <edge_info>);
int adapt_delete_edge(adapt_guid src_NID, adapt_guid dest_NID,
                      adapt_guid etype_ID, <edge_info>);

adapt_guid adapt_get_dest_node_ID(adapt_guid src_NID,
                adapt_guid etype_ID, <edge_info>);
void *adapt_node_ID_to_data(adapt_guid NID);
```

Fig. 3.1.3. Core API for ADAPT.

library approach: (1) providing an API expressive enough to construct complex storage components, (2) streamlining program flow constructed by our API, (3) keeping various metadata and data updates consistent, (4) representing our internal states, and (5) providing fine-grained access control. The following subsections address these design challenges.

3.1. Graph-based API

Conceptually, each piece of data is associated with a globally unique ID (e.g., <0, "example 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>. Fig. 3.1.1 shows that the ID for "example data" is 0. The access permission tag for "example data" refers to the data ID of 1, which is 0x444. If another piece of data has the same permission, it can also be tagged with the same permission tag. The owner tag for "example data" refers to the data ID of 2, which is an owner ID of 1234. This contrived example demonstrates an extreme use of tags, in which each attribute is a tag. However, a more conglomerate use of in which a tag can represent a set of attributes, is more practical.

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

Because a tag expresses the relationship between two pieces of data (e.g., 0x444 is the permission of "data"), we can transform the representation in Fig. 3.1.1 logically in terms of nodes and edges, with the nodes holding data and the tag types representing directional edges (Fig. 3.1.2). While it is possible for permission node 1 to have its own permission node to specify who can change the permission of node 0, we can recursively define permission to infinity. Practically, after a few layers, the permission will fall back on the administrative root privilege, which is the end of the recursion. The owner ID of node 0 can also be inherited by its sub-nodes, so that each sub-node does not need to specify the owner. The size of a piece of data is tracked by the ADAPT allocator, so that a programmer does not need to worry about recursively

```
 1 char *head_node, *null_node;
 2 adapt_guid head_NID, null_NID, next_node_etype_ID;
 3 adapt_edge_info e_info;
 4
 5 void init() {
 6   head_node = null_node = “NULL”;
 7   next_node_etype_ID = adapt_create_edge_type(“next_node”);
 8   head_NID = adapt_create_node(head_node, strlen(head_node) + 1);
 9   null_NID = adapt_create_node(null_node, strlen(null_node) + 1);
10   adapt_create_edge(head_NID, null_NID, next_node_etype_ID, e_info);
11 }
12
13 char *get_node_value(adapt_guid, NID) {
14   return (char *) adapt_node_ID_to_data(NID);
15 }
16
17 adapt_guid insert_node(char *node) {
18   adapt_guid NID = adapt_create_node(node, strlen(node) + 1);
19   adapt_guid next_NID = adapt_get_dest_node_ID(head_NID, next_node_etype_ID, e_info);
20
21   adapt_create_edge(NID, next_NID, next_node_etype_ID, e_info);
22   adapt_delete_edge(head_NID, next_NID, next_node_etype_ID, e_info);
23   adapt_create_edge(head_NID, NID, next_node_etype_ID, e_info);
24   return NID:
25 }
26
27 int delete_node(adapt_guid NID) {
28   adapt_guid pre_NID = head_NID;
29   adapt_guid cur_NID = adapt_get_dest_node_ID(head_NID, next_node_etype_ID, e_info);
30   adapt_guid next_NID;
31
32   for (;cur_NID != null_NID; prev_NID = cur_NID, cur_NID = next_NID) {
33     next_NID = adapt_get_dest_node_ID(cur_NID, next_etype_ID, e_info);
34     if (cur_NID == NID) {
35        adapt_delete_edge(prev_NID, cur_NID, next_node_etype_ID, e_info);
36        adapt_create_edge(prev_NID, next_NID, next_node_etype_ID, e_info);
37        adapt_delete_edge(cur_NID, next_NID, next_node_etype_ID, e_info);
38        adapt_delete_node(cur_NID);
39        return 0;
40     }
41   }
42   return -1;
43 }
```

Fig. 3.1.4. An example use of the ADAPT API.

specifying the size of the size tag.

Fig. 3.1.3 shows the core API for ADAPT. A node can be created to hold a dynamically allocated piece of data. A node can be destroyed given a node ID. An edge-type global unique ID can be created with a given name. To create or delete an edge, we must specify the IDs of the source, destination nodes, and edge type. Because dangling edges (those 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. Since newly created nodes cannot be accessed until they are attached by at least an edge, nodes can be created in parallel. In the case of failure, nodes can be garbage collected without any effects visible to end users.

When an edge must point to NULL, an empty node can be used to ensure that each edge is formed between two nodes. Certain edge types involve enumeration (e.g., block ID edge type); thus, when operating on edges (`adapt_create_edge` and `adapt_delete_edge`) or accessing a node through an enumerated edge (`adapt_get_dest_node_ID`), an additional optional `edge_info` parameter is used to pass in the enumerated number. For example, suppose we want to connect a file node to data blocks 1 and 2.

To connect to block 1, we can call `adapt_create_edge`(file node ID, enumerated edge type ID, `edge_info` with the edge number field set to 1). To connect to block 2, we can call `adapt_create_edge`(file node ID, enumerated edge type ID, `edge_info` with the edge number field set to 2). To access the node ID for the second data block of a file, we can issue `adapt_get_dest_node_ID`(file node ID, enumerated edge type ID, `edge_info` with the edge number field set to 2).

A node's data can be accessed through its ID (`adapt_node_ID_to_data`), and a node's ID can be accessed through the incoming edge of another node (`adapt_get_dest_node_ID`). Although a node can potentially be reached from different nodes through the same in-bound edge type, a node can also only be associated with unique out-bound edge types. For example, for a node in a doubly linked list to be associated with the previous node and the next node, it must have two separate previous-node and next-node edge types. Although the underlying node pointer type is the same, we must define two separate edge types to distinguish the two.

Fig. 3.1.4 shows an example of how to use the ADAPT API to implement a singly linked list of character strings, with error checking omitted. Line 1 declares node pointers for the list head and the null, which are sentinel elements, indicating the start and end of the list. Line 2 declares variables to hold unique IDs for the list head, null node, and edge type ID representing the edge to access the next node. Line 3 declares the dummy variable `edge_info`, which does not hold any meaningful information for this example.

In line 6, the `init` procedure initializes the head and null nodes to the “NULL” string. Line 7 creates an edge type ID that is used to connect

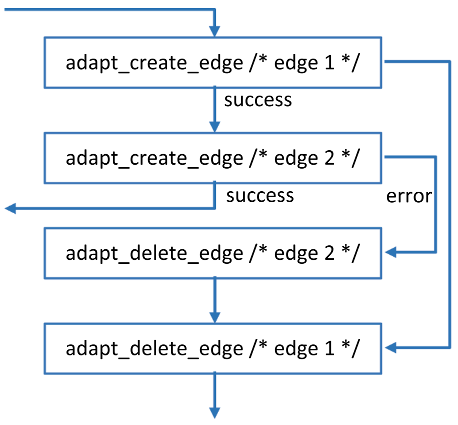
adapt_create_edge /* edge 1 */
success
adapt_create_edge /* edge 2 */
success
error
adapt_delete_edge /* edge 2 */
adapt_delete_edge /* edge 1 */

Fig. 3.2.1. ADAPT's error handling execution flow without transactions.

```
adapt_tx_id adapt_begin_tx();
adapt_abort_tx(adapt_tx_id tx_ID);
adapt_commit_tx(adapt_tx_id tx_ID);
```

Fig. 3.2.2. Transactions for ADAPT.

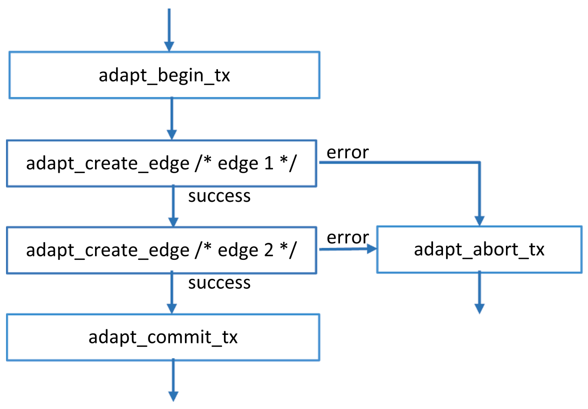

Fig. 3.2.3. ADAPT's error handling execution flow with the use of transactions.

and access the next node in a linked list. In lines 8-9, the list head and the null node are allocated with `adapt_create_node`, which returns global unique IDs. In line 10, the head node is connected to the null node via `adapt_create_edge` with a next node edge type ID.

Since ADAPT maintains direct mapping between node IDs and nodes, one can invoke `adapt_node_ID_to_data` with a node ID to access a node's data (line 14). There is no need to traverse the linked list to locate a node.

To insert a node, the `insert_node` function prepends a new node between the head node and the next node (lines 17–25). To delete a node, `delete_node` uses two node IDs; that is, similar to the use of pointers to traverse a linked list and identify the target node for removal. Then, the previous node's next node edge is set to bypass the target node to be deleted (lines 35-36), with the remaining used edge and node deleted (lines 37-38). The example code shows similar logic as the normal in-memory linked list data structure with pointers replaced by ADAPT primitives.

In this pointer-rich example, the use of node and edge routines is somewhat verbose; however, the non-pointer-rich part of programs is expected to be the same as the non-ADAPT code. The resulting linked list has persistent pointers (Section 3.4) that can survive reboots. Therefore, if the head of the linked list and the list interface are exported, we can have different storage components and applications to directly access this persistent linked list (or any other data structures such as key-value stores or b-trees). By doing so, the persistent linked list does not need to be transformed into the file system storage format.

3.2. Transactions

One problem with using the graph-based API on fine-grained tags is the difficulty of achieving atomicity across many ADAPT operations. Any failure (e.g., out of memory) along a sequence of operations would require lengthy cleanup code. Fig. 3.2.1 shows the execution flow of a typical fallback code structure. During the fallback process, the node and edge structures created before the failure are cleared. Multiple undo execution paths can obscure the program flow.

To mitigate this issue, we added transactions to roll back multiple operations (Fig. 3.2.2). If an error occurs between the begin and commit calls, the abort call automatically performs graph cleanup and rollback to the graph states. The execution control flow graph shown in Fig. 3.2.1 is transformed into the graph presented in Fig. 3.2.3.

3.3. Consistency and correctness properties

ADAPT maintains three copies of metadata: one in memory for metadata updates (MM), one on storage as (previously checkpointed) metadata (MS) for rolling back, and one on storage for committing in-transit updates (MS'). ADAPT maintains two copies of data: one in memory (DM) and one in storage (DS). Therefore, we must ensure that ADAPT can recover to a consistent state in the case of a system crash.

A group of operations is first journaled in memory. To process transactions of updating nodes and edges, ADAPT performs the various steps in three phases:

Phase 1 (commit data): Update data items from DM to DS. Deleted data items are marked to be deleted in MM until the metadata commit is complete. If a failure occurs before Phase 1 is committed, then no updated metadata will be written. In the case of a crash, it may be possible for old metadata to point to newly updated data items; however, the consistency semantics are akin to those of the ordered mode for ext4. Data nodes without edges can be detected and deleted via delayed garbage collection.

Phase 2 (commit metadata): Commit metadata creations, updates, and deletions from MM to MS'. If a failure occurs before Phase 2 is completed (e.g., some pages are `msync`ed while others are not), then MM and MS' will be restored to the previously checkpointed metadata (MS).

Phase 3 (deletion and metadata checkpoint): Delete data items that are marked to be deleted. Propagate metadata creations, updates, and deletions from MM to MS. If a failure occurs before this phase is complete, then the transaction can be replayed (rolled forward) from the commit.

We recognize that the current implementation of transactions involves writing metadata twice and can cause performance overhead similar to that of the popular file system ext4 (our baseline comparison). However, given that a small percentage of written blocks are metadata blocks, updated in batches to aggregate updates, the journaling overhead is approximately 6-13% (see Section 5.1).

Correctness: We applied the file system consistency properties defined in [29] to reason the correctness of our transactions.

The reuse-ordering property ensures that once a file's data block (an ADAPT node) is freed, the block will not be reused by another file before its free status becomes persistent. Otherwise, a crash may lead a file's metadata (an ADAPT edge) to point to the wrong file's content. In ADAPT, all edges are deleted and committed prior to deleting the connected node, assuring that the node is no longer reachable from the remaining graph before the node is deleted and reused.

The pointer-ordering property ensures that a reference data block (an ADAPT node) in memory will become persistent before the metadata (an ADAPT edge) in memory that references the data block. If this ordering were reversed, a crash could cause the persistent metadata to point to a persistent data block location that has not yet been written. In ADAPT, all nodes are flushed prior to the edges.

The non-rollback property ensures that older data or metadata versions will not persistently overwrite newer versions. ADAPT fulfills this property because journaled entries are ordered and applied to the global transactions chronologically.

3.4. Physical representation

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

Nodes: One way to manage nodes is to create a node manager, which is responsible for allocating storage, assigning unique IDs, and maintaining ID-to-storage mappings. However, as we observed similarities between node management and memory management, we overloaded the memory management functionality for node management. ADAPT nodes are variable-sized, memory-mapped storage chunks governed by a memory allocator. A node's memory address (offset by the starting memory-mapped address) is used as a unique ID for that node, so we do not need to worry about having two nodes mapped to the same ID. Additionally, a node's ID can give us direct access to the node's storage location. This approach implies that we must make the memory allocator persistent across reboots, and the storage device must map to the same memory address across reboots.

Edges: ADAPT edges are implicitly stored in an extensible hash table [6] with collisions handled by double hashing. The key of the edge hash table *edge_table* is generated from the source node ID, the edge type ID, and edge info, which may contain a number for enumerated edge types (see an example use in Section 3.1). edge_table[key] returns the destination node ID. The destination node can be tagged with a magic number to perform a dynamic type check before accessing the node's content. For example, the magic number can confirm whether the node content "123" will be accessed as a string type or an integer type. To update an edge, we just update the edge_table[key] to a new destination node ID. To delete an edge, the edge_table[key] entry is deleted.

Persistence: Since applications and various data path components can use ADAPT to build data structures that will survive across reboots, and we leverage memory-mapped addresses as unique IDs and persistent pointers, the memory allocator's states for ADAPT must be persistent. (We implemented a separate memory allocator for ADAPT.) The governed memory is divided into regions for metadata, data, ephemeral states (to optimize the ADAPT internal data structures), and storing critical start-up information (e.g., offset of active persistent state storage). The metadata region includes the states of the memory allocator and the edge table. Metadata and data regions (except for the ephemeral states) are flushed from memory and disk according to the snapshot protocols in Section 3.3.

Data layout: Within the storage organization for ADAPT, data layouts are largely governed by the memory allocator for nodes and the representation of the hash table for edges. We first allocate a memory pool and then perform customized allocation within it. To encourage locality, we use a customized slab allocator [3] for sizes below one page: this encourages objects of similar sizes to be collocated within a memory page. For memory allocation requests greater than one page, we use a buddy allocator [19]. Currently, we do not support resizing an existing allocation if a node needs to grow in size. Instead, the user or developer needs to reallocate a new larger node.

Since hashing has poor locality, we modified it to encourage destination node IDs from the same source node ID to be collocated. As a simplified 32-bit example, suppose that a from_node_ID has a hash key of 0x0011011F. We use the upper 20 bits of the source node to determine the storage of the key in a 4KB memory bucket 0x00110. If a to_node_ID has a hash key of 0x00001100, then it will use the upper bits of its source node 0x00110 to determine in which memory bucket to store the to_node_ID, and it will use the lower 12-bits to locate the to_node_ID within the bucket. As the hash table increases in size, fewer upper bits will be used to locate bigger memory buckets.

Table 3.5.1
Rules for creating edges.

From \ to	s-node A	s-node B	s-node A's nodes	s-node B's nodes
s-node A	Yes	Yes	Yes	No
s-node B	Yes	Yes	No	Yes
s-node A's nodes	Yes	Yes	Yes	No
s-node B's nodes	Yes	Yes	No	Yes

```
adapt_guid
  adapt_create_s_node(adapt_mode mode);
adapt_guid
  adapt_create_node(void *data, size_t len,
                   adapt_guid _s_node_ID);
```

Fig. 3.5.1. Super-node operations.

Note that it is possible to reach the same to_node_ID from different from_node_IDs, and each from_node_ID can have its own cluster of to_node_IDs (which are persistent pointers to the destination node)

3.5. Access control

Since ADAPT 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 an unwieldy way of enforcing permission to access nodes in a general graph. However, since many tags share the same permission, it is logical to check and enforce permissions at fewer locations. Also, a certain degree of restrictions on how edges can be formed can manage the access control properties of the resulting graph topology.

Super nodes: The idea of super nodes (***s-nodes***) is to use fewer places to check permissions. In other words, only s-nodes have edges to permission nodes (e.g., ownership and permissions). All nodes belonging to the same s-node share the same permissions. In terms of restrictions, edges can be created from an s-node to its nodes (Table 3.5.1). Edges can also be created from any node to an s-node, since the destination s-node can enforce access permissions. However, forming edges between nodes that are under different s-nodes is prohibited, and a source s-node cannot create an out-bound edge to a node under another s-node. For example, for a file system to support ".", an edge can be formed from a directory node *N1* to itself with the same s-node *S1*. For a file system to support "..", an edge can be formed from *N1* to its parent directory's s-node *S2*. Then, *N1* can follow the directory edge from *S2* to locate *N2*, which the parent directory node of is *N1*'.

One challenge of permission lookups is that of 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, the s-node must be connected to at least one of its nodes. To locate the permission from a node under an s-node, we use 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 produce s-node IDs and create node IDs alongside with them (Fig. 3.5.1). 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.

In this model, the s-nodes can form loops themselves without a conflict access rights problem. Developers can determine whether to use

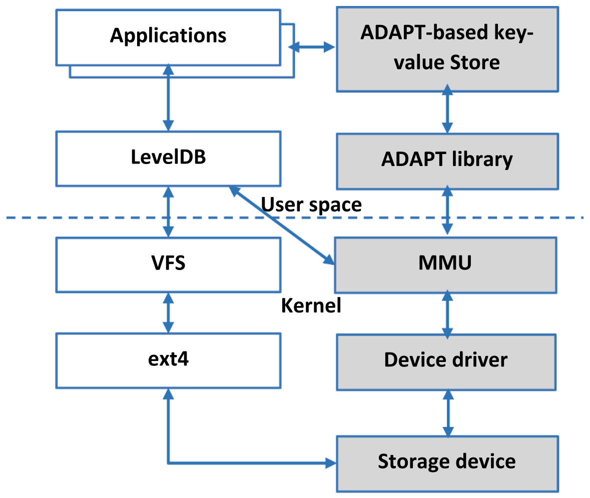

Fig. 4.1. Storage data paths for an ADAPT-based key-value store (shaded boxes) and LevelDB [7].

this feature, and they must consider the data model to divide normal nodes into different access rights groups.

Suppose that a developer decides to change the design of the graph and wants a node *N1* under s-node *S1* to have a different permission. This change will involve creating a new s-node *S2* so that the prefix of the s-node ID can be used to create a new node *N2*, which is a copy of *N1*. All existing relationships between *N2* and nodes under *S1* must be created or reestablished through *S2* as an intermediary node for permission checks, and *N1* can be removed from the graph. Given the complexity and potential associated overhead of this operation, a user should not change the access control points lightly.

4. Implementation

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

For the logical management of ADAPT (2,258 lines), we implemented the graph API for data-tagging and data tracking, nodes and edges, transactions, and access control. For the physical management component of ADAPT (1,980 lines), we implemented the persistent memory allocator, which also controls the layout of physical data. The ADAPT library currently does not support multi-threaded and nested transactions. Multi-threaded transactions could potentially speed up ADAPT further. However, this speedup is not automatic because transactions may have dependencies. For example, if we rename file A to file B in transaction 1, and rename file B to file C in transaction 2, then transaction 2 cannot proceed until transaction 1 is completed.

To demonstrate ADAPT, we implemented (1) an ADAPT-based key-value store (493 lines) to show how ADAPT can natively support a different access method, (2) a B+tree (581 lines) to show how ADAPT can natively support a slightly more complicated data model, (3) a file system (1,837 lines) to show how ADAPT API is rich enough to build a complex application, (4) a prioritized caching scheme (464 lines) to show how legacy storage components can benefit the enhancement of ADAPT, and (5) secure deletion (380 lines) to show how ADAPT facilitates storage-data-path-wide data tracking.

5. Evaluation via case studies

While evaluating ADAPT, we aimed to demonstrate its ability to (1) avoid redundant layered features, (2) achieve usability and robustness when building complex software, (3) extend and support new data models and features beyond those provided by the legacy data path, (4) coordinate and track data across layers, and (5) perform well with both HDD and SSD storage media.

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

Table 5.1
System configuration.

CPU	2.2Ghz Intel® Xeon® E5-2430, 15MB cache
Memory	8 GB RDIMM 1333 MT/s
HDD	Seagate® SAS 146GB 15K RPM
SSD	Intel® S3500 200 GB SATA Value MLC
OS	Linux Mint 4.4

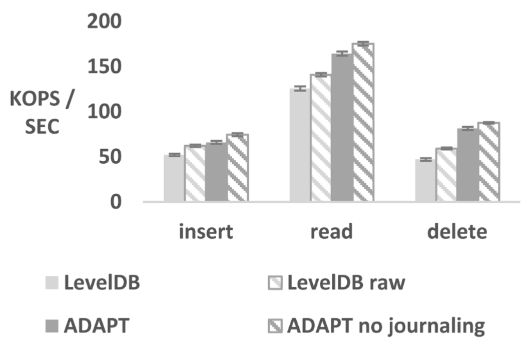

Fig. 5.1.1. Key-value store performance for HDD.

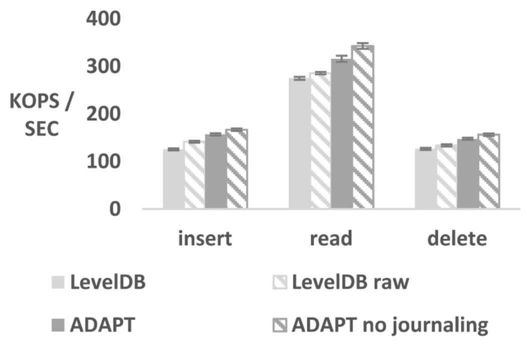

Fig. 5.1.2. Key-value store performance for SSD.

5.1. ADAPT-based key-value store

To show the benefit of direct support for new data models, we prototyped a key-value store using the ADAPT library. The ADAPT data path had no file system or associated redundant efforts to manage the data layout (Fig. 4.1).

Given that ADAPT is built on a hash table that stores edges to nodes, its 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 `adapt_node_ID_to_data (adapt_get_dest_node_ID(root node ID, key))` to retrieve the data. For `Delete(key)`, we called `adapt_delete_edge(root node ID, node ID)`, followed by `adapt_delete_node(node ID)`.

We compared the ADAPT-based key-value store with LevelDB 1.20 [7]. We also measured LevelDB using the raw mode, which bypasses the file system, to see the performance overhead imposed by the file system

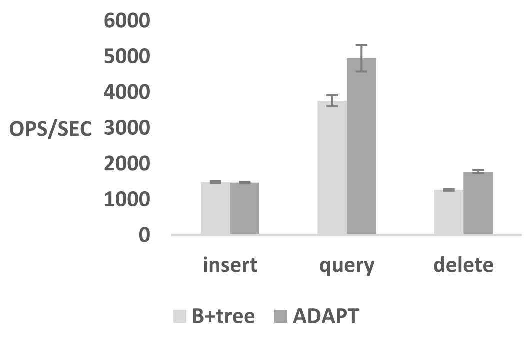

Fig. 5.2.1. B+tree performance for HDD.

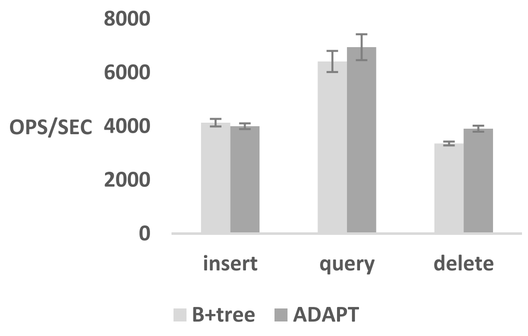

Fig. 5.2.2. B+tree performance for SSD.

layer. We further measured ADAPT without journaling to see the overhead of maintaining consistency.

Fig. 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. Figs. 5.1.1 and 5.1.2 show the results in kilo key-value store operations per second.

For both storage media, ADAPT and LevelDB have similar read performance because both systems use memory-mapped IOs to avoid copying. Both systems also use bulk updates (transactions for ADAPT) to speed up small updates. This also explains why the SSD performance (bandwidth-bound) is only twice as fast as the HDD performance numbers. For HDDs, ADAPT can outperform LevelDB by a factor of 1.3 for inserts and 1.7 for deletes. For SSDs, ADAPT can outperform LevelDB by a factor of 1.2 for inserts and deletes.

In comparing the numbers between LevelDB and LevelDB running with the raw mode, we can see that the file system layer imposes 12-26% overhead for HDD and 3-12% overhead for SSD. In comparing LevelDB running with the raw mode with ADAPT, we can see that the remaining overhead can be attributed to serialization and deserialization. We further compared ADAPT with and without the use of journaling to ensure consistency, revealing 7-13% overhead for HDD and 6-9% overhead for SSD.

5.2. Tags-based B+tree implementation

Another example of ADAPT's capability to support new data models is the implementation of B+tree using ADAPT. Similar to the key-value store, the B+tree is built on the ADAPT library and involves no file system.

The nodes in B+tree are represented by ADAPT nodes, and the connections between the nodes are represented by ADAPT edges.

We compared ADAPT-based B+tree implementation with normal B+tree implementation, which utilizes the serialization/deserialization method to load from/save to files on storage media. The serialization/deserialization method takes a breadth-first way to process the nodes in a B+tree into data in a file.

The dataset for our experiment contains five B+trees, each having a depth of 3 and a fan-out factor of 342. The key size is 4 bytes and the pointer size is 8 bytes; therefore, each node occupies approximately 4KB. The entire size of each tree is approximately 480 MB, and the total size of all the trees is approximately 2.4 GB.

We assessed performance in terms of query, deletion, and insertion to ADAPT-based and file-based implementation. For query, we performed 500K queries for random-key-mapped pointers to an 80% full B+tree. For insertion, we performed 500K insertion operations to a 40% full B+tree with random key-pointer pairs. For deletion, we performed 500K deletion operations to an 80% full B+tree. Figs. 5.2.1 and 5.2.2 show the results. For the B+tree insertions, we can see that the overhead of ADAPT persistent pointer creation can offset the benefit of bypassing serialization and deserialization. However, ADAPT still performs better for queries and deletions.

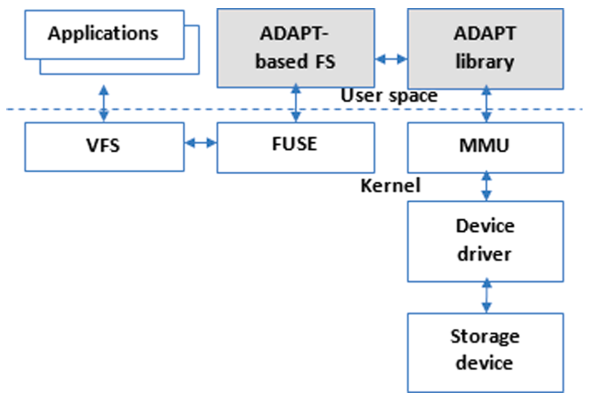

Fig. 5.3.1. The ADAPT-FS and the ADAPT library (shaded).

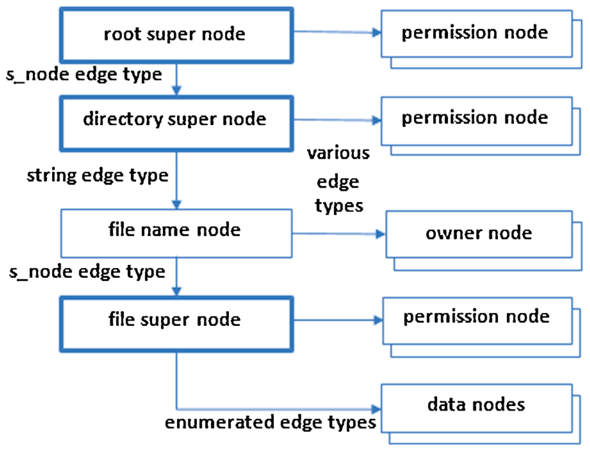

Fig. 5.3.2. The ADAPT representation of a file system.

5.3. ADAPT-based (POSIX-compliant) file system

To demonstrate usability, we prototyped an ADAPT-based file system (ADAPT-FS) to show that the interface and primitives provided by ADAPT are expressive enough to build meaningful complex applications. While users of ADAPT need to learn a new interface, ADAPT saves them from writing serialization and deserialization code. ADAPT-FS also functions as a POSIX-compatibility layer for legacy applications and users; it is not necessary to rewrite these applications. The ADAPT-FS was implemented at the user space via the FUSE framework [33]. Fig. 5.3.1 illustrates the flow of data requests.

The ADAPT-FS translates POSIX file-system calls into ADAPT nodes and edges; this task involves many node and edge operations that are simplified by transactions. Basically, all i-nodes (permission-holding

Table 5.3.1
Time to recursively list Linux 4.1 build.

	HDD	SSD
ext4 + FUSE	2.8 (±0.018) secs	0.57 (±0.0050) secs
ADAPT-FS + FUSE	3.0 (±0.029) secs	0.65 (±0.0080) secs

Table 5.3.2
LFS large-file benchmark numbers with one 4GB file for HDD and SSD.

HDD	ext4 + FUSE	ADAPT-FS + FUSE
sequential write	110 (± 1.3) MB/s	100 (±1.0) MB/s
random write	46 (±1.9) MB/s	12 (±0.73) MB/s
sequential read	190 (±2.1) MB/s	170 (±4.9) MB/s
random read	2.7 (±0.21) MB/s	13 (±0.65) MB/s
SSD		
sequential write	160 (±2.8) MB/s	180 (±3.4) MB/s
random write	100 (±2.4) MB/s	100 (±1.5) MB/s
sequential read	350 (±4.9) MB/s	300 (±6.6) MB/s
random read	63 (±0.59) MB/s	94 (±1.2) MB/s

Table 5.3.3
LFS small-file benchmark numbers with 20K 16KB files for HDD and 100K 16KB files for SSD.

	operation	ext4 + FUSE (files/s)	ADAPT-FS + FUSE (files/s)
HDD	create	1,700 (±66)	1,100 (±41)
	read	2,700 (±100)	3,300 (±85)
	delete	7,400 (±360)	4,000 (±120)
SSD	create	3,900 (±79)	3,400 (±60)
	read	5,300 (±80)	9,500 (±250)
	delete	20,000 (±1,500)	17,000 (±500)

nodes) are replaced with s-nodes, and all attributes are accessed through edges (Fig. 5.3.2). 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 use a single node to contain all the attributes of an i-node, we explored this scenario to show that, we can still configure the system to achieve reasonable performance even if tags are naively applied. We compared our ADAPT-FS stacked on FUSE with ext4 stacked on FUSE. The elapsed times for ADAPT-FS and ext4 + FUSE to compile the OpenSSL (v1.1.0f) [17] were statistically the same (87 ± 0.19 seconds). We were concerned that hashing-based data structures may perform poorly for directory traversals. Thus, we prepared a directory containing Linux 4.1 (after a complete build) and examined the elapsed time spent for recursively listing through the directory (Table 5.3.1). The hash-based doubly linked list of directory entries in ADAPT is marginally slower than ext4 + FUSE for both HDD and SSD.

For LFS large-file and small-file benchmarks [22], ADAPT-FS performed reasonably well when its block size was configured to 32KB to amortize the cost of fine-grained access to attribute nodes and dynamic type checks (Tables 5.3.2-5.3.3). We admit that using a larger data block is not a fair comparison (4KB for ext4 + FUSE); however, our objective was to show that ADAPT's API is sufficiently rich and that its implementation is robust enough to build software with comparable complexity to a file system with reasonable performance. As a side effect of the large block size of ADAPT-FS, it increases the performance of random reads (due to unintended cache warmup per read access). The random write bandwidth numbers for both ext4 + FUSE and ADAPT-FS + FUSE are higher than expected due to buffering. However, the overhead of fine-grained per-file-attribute ADAPT appears on the creation and deletion benchmarks, for which each operation involves creating and deleting many tags.

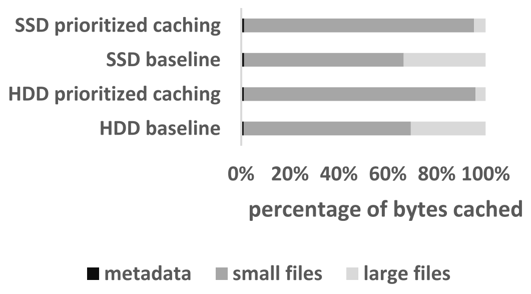

Fig. 5.4.1. Percentage of bytes cached for each IO class.

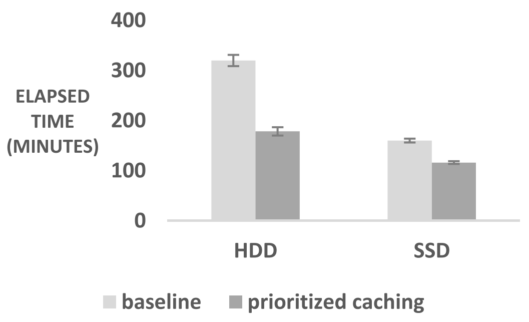

Fig. 5.4.2. Elapsed times for web trace replay.

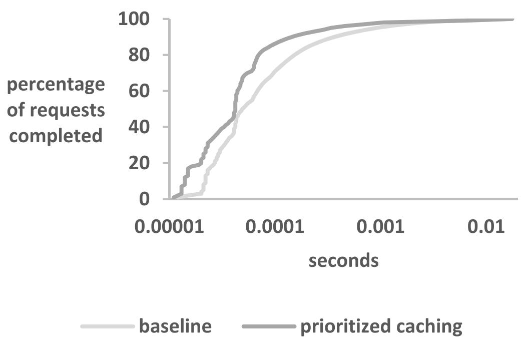

Fig. 5.4.3. Percentage of individual HDD requests completed within a given time frame in seconds (log scale).

5.4. Prioritized caching

The ability of ADAPT to provide data-path-wide annotation and communication enables the addition of new features into file systems. To demonstrate how ADAPT can be extended to support features beyond the legacy data path, we implemented a prioritized caching similar to [15], which allows different classes of data to be treated differently. In our example, we defined large files, small files, and metadata as different classes, and we modified the LRU cache to give preference to caching small files and metadata to achieve performance gains.

In terms of ADAPT, we leveraged the per-file permission lookup mechanism to directly access a file's s-node and reach its subsidiary node tagged with the data class. The caching mechanism can then prioritize caching for blocks tagged as small files and metadata.We compared the performance of ADAPT-FS with prioritized caching enabled to its performance with prioritized caching disabled. The workload involves a zero-think-time replay of a departmental web server trace, which contains 8.6M file references to 1.0 TB of data, among which 1.5M files are unique with 12 GB of unique data. We classified files under 18 KB (75% of files) as small.

Fig. 5.4.1 shows that by the end of the trace replay, up to 95% of the cache is populated with small files and metadata. Fig. 5.4.2 shows that

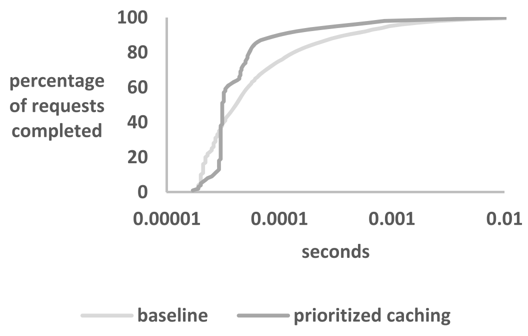

Fig. 5.4.4. Percentage of individual SSD requests completed within a given time frame in seconds (log scale).

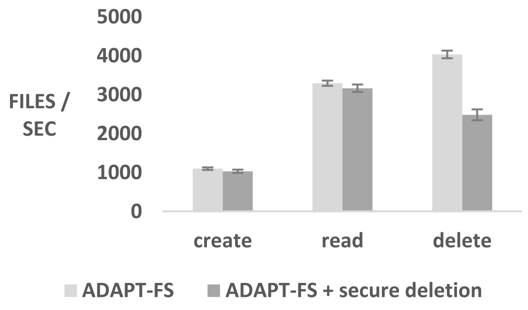

Fig. 5.5.1. LFS small-file benchmark numbers with 20K 16KB files for HDD.

the overall elapsed time for the trace replay is improved by 40% for both HDD and SSD. Figs. 5.4.3 and 5.4.4 show the cumulative distribution function (CDF) of request completion times. In Fig. 5.4.3, approximately 90% of prioritized HDD requests were completed within 0.0001 seconds, while only 60% of requests for the baseline system were completed within the same time frame. In Fig. 5.4.4, nearly, 90% of prioritized SSD requests were completed within 0.0001 seconds, while only 70% of requests for the baseline system were completed within the same time frame.

5.5. *Per-file secure deletion*

To demonstrate cross-layer coordination and tracking, we augmented the ADAPT-FS with a per-file secure-deletion feature akin to that of [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 identify the file membership of a block.

Since each group of nodes in the ADAPT-FS 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 Section 3.4). Consequently, the ADAPT-FS can access the permission. The secure-deletion bit indicates that the corresponding overwrite or truncate should be handled securely.

For disk, we borrowed the `ioctl` call mechanism from FUSE to allow the block layer to issue calls from the kernel space to ADAPT to query whether a block belongs to a file that has the secure-deletion bit set. If so, the block layer would perform triple writes of random bits to ensure secure deletion. Updates to blocks belonging to a file marked for secure deletion are prepended with triple writes of random bits to securely delete the previous content. We also modified the truncate mechanism, so that it would issue calls to erase data blocks.

Without the open FTL and raw flash setup, we did not implement this feature for SSD. Note that the TRIM command is insufficient; it only specifies which pages are obsolete to prevent migrating these as live pages during the garbage collection process [26].

To evaluate the performance overhead introduced by secure deletion, we used the LFS small-file benchmark with the same setting as that we used in Section 5.2, with the first 5% of files marked to be securely deleted to reflect locality. Fig. 5.5.1 shows that secure deletion does not slow down file creations and reads, but significantly degrades the performance of deletions as expected.

6. Limitations

ADAPT provides a framework for different layers to coordinate through a single unified interface, but this does not necessarily mean that such coordination is possible. For example, consider two databases that want to share the same dataset to optimize the data layout: one for row access, and the other for column access. Efforts to reconcile conflicting goals across storage components that want to coordinate is beyond the scope of this work.

The white-box approach will also interact with storage software components developed by people who wish to keep their internal representations proprietary. One possible solution is to have enriched permission semantics, so that only a limited subset of translated edge type IDs is made available for external use.

7. Related work

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

7.1. *Cross-layer redundancy removal*

JFFS [36] consolidates logging for the file-system and flash-device-driver layers. ADAPT complements JFFS in terms of extensibility to new features and data models.

DevFS [9] moves the file-system component into the storage device so that applications can directly access a storage device without being trapped into the OS for most operations while maintaining integrity, consistency, and security guarantees. DevFS also leverages device-level power-loss-protection capacitors to eliminate redundant writes caused by logging and associated garbage collection mechanisms. ADAPT, on the other hand, focuses on the legacy software storage stack rather than altering the behavior of hardware storage devices.

7.2. *Cross-layer coordination*

Existing methods to coordinate across layers are mostly designed to solve specific problems (e.g., secure deletion and performance), and are not suitable for arbitrary extension of new functionalities.

Shen et al. [24] showed that different file-system journaling modes can be changed dynamically for database files to achieve better performance. ADAPT can be used to support this feature and propagate per-file journaling status to the underlying file system.

The differentiated storage services [15] coordinate IO class information by expanding the block IO data structure to propagate this information. ADAPT uses a data-path-wide repository to store the additional coordination information, so that all storage components can have direct access to their information.

The gray-box approach leverages inferred information across layers for coordination [1]. Since inferences may occasionally be incorrect, mechanisms based on these inferences must make conservative decisions (e.g., when in doubt, delete a data block securely).

TrueErase [4] provides an auxiliary data path, so that a file system can propagate information to the device layer to indicate whether a file needs to be securely deleted or overwritten. The auxiliary data path is tailored for secure deletion, so it may not be readily applicable to the addition of new features such as prioritized caching.

Willow [23] expands the interface for SSDs to exploit built-in storage processor units within SSDs to run tailored applications. Different storage components can issue RPC calls to interact with SSDs. The applications that can run under Willows are limited by the amount of built-in memory and processing power on SSDs. ADAPT, on the other hand, focuses on the software stack instead of altering the behavior within storage devices.

Spiffy [30] allows file-system developers to annotate file-system-specific data structures and then compiles and generates a library so that application developers can write file-system-aware utilities using only generic library calls without knowing the specific file system's format. In a sense, Spiffy creates another abstraction layer that integrates the knowledge of file systems, while ADAPT aims to pierce through the abstraction of layers.

Strata [10] leverages the strengths of NVM, SSD, and HDD and provides an integrated cross-layer design (NVM at the user level and SSD and HDD at the kernel level) to achieve both high throughput and low latency. Strata coordinates layers for performance. However, ADAPT can be used to extend Strata for new system features.

7.3. Support for low-latency storage

DAX [35] uses direct IOs and bypasses memory caching designed for high-latency storage. However, applications may need to duplicate the functionalities (e.g., storage allocation, mapping, and serialization) originally provided by the bypassed legacy storage components.

The Persistent Memory Development Kit (PMDK) [20] provides user-space libraries and tools for allocating persistent memory objects, performing transactions, object typing, and so on. The PMDK exposes persistent pointers for developers to weave persistent data structures, while ADAPT uses the abstraction of nodes and edges to weave data structures. ADAPT also provides a POSIX-compliant deployment model.

PMFS [5] uses memory mapping and bypasses the block layer to achieve substantial performance gains for byte-addressable persistent memory. Although ADAPT can bypass the legacy VFS caching layer, the internal representation of ADAPT does not contain tailored optimizations for byte-addressable persistent memory. Data structures such as [16] will be incorporated in future work.

Arrakis [18] removes the kernel from the data IO path. Both network and storage 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. Unlike ADAPT, Arrakis does not provide a backward-compatible, POSIX-compliant interface for storage, and legacy applications must be modified to interact with the internal representation of Arrakis.

File systems as processes (FSP) [13] move the entire legacy kernel-level storage stack into userspace. Combined with lightweight IPC, FSP can provide sub-microsecond latency while accessing NVM. Similar to FSP, the ADAPT-FS is implemented as a user-level library. Unlike DashFS, a prototype using the concept of FSP, ADAPT supports crash consistency. With the use of a byte-addressable data structure [16] and direct communication with a user-level NVMe device driver [37], the performance of the ADAPT-FS can improve significantly.

In the high-performance computing domain, Distributed Asynchronous Object Storage (DAOS) [8] shares a similar architecture with ADAPT. DAOS operates in the user space and provides a layer of libraries to handle legacy semantics (e.g., POSIX). DAOS also uses the Apache Arrow format to avoid serialization and deserialization for data-analytics applications. Unlike DAOS, ADAPT supports communication and coordination across layers.

7.4. Support for new storage data models

Shetty et al. [25] showed that mixed workloads from file systems and databases can be efficiently handled using separate KVFS and KVDB layers. Similarly, ADAPT can attach different data models (e.g., B-tree and key-value store) into the same namespace and provide direct access to different data structures without translating them into the internal representation of the file system.

Table 7.1
Related work comparison.

	Support for low latency storage	Support for new data model	Avoid redundant functions across layers	Support for coordination across layers
[36]	√		√	√
[24]				√
[9]	√		√	√
[15,23]		√		√
[1,4, 30]				√
[10]	√			√
[5,13, 35]	√			
[18]	√		√	
[25]	√	√		√
[12]		√		
[34]	√	√	√	
[8]	√	√	√	
[20]	√	√	√	
[11]	√	√		
ADAPT	√	√	√	√

Cassandra [12] uses a customized table API to store and retrieve data objects. ADAPT uses the general notion of nodes and edges.

Aerie [34] provides a user-level library to give applications direct access to storage-class memory. Each application can define its own storage access interface built on a primitive called a memory file. Since Aerie user-level processes can directly access storage hardware, a rogue process can issue stylized write patterns to significantly shorten the lifespan of storage-class memory.

In the context of high-performance computing, JULEA [11] is a user-space layer that provides different interfaces (e.g., key-value store and file system) to different applications. On the other hand, ADAPT also provides an additional communication channel and coordination across the entire storage data path.

7.5. Overall

Various solutions have been devised to mitigate a subset of constraints in the legacy storage data path. Table 7.1 summarizes the comparisons between ADAPT and existing solutions.

8. Conclusions

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

Declaration of Competing Interest

None.

Acknowledgment

We would like to thank anonymous reviewers for providing us with invaluable feedback. We would also like to thank Geoff Kuenning and Leah Rumancik for reviewing early drafts of this paper. This work is sponsored by FSU and the NSF CNS-1125275. The opinions, findings, conclusions, and recommendations expressed in this document do not necessarily reflect the views of FSU, the NSF, or the U.S. Government.

References

[1] AC Arpaci-Dusseau, RH. Arpaci-Dusseau, Information and control in gray-box systems, in: Proceedings of the 18th Symposium on Operating Systems Principles (SOSP), 2001.
[2] M Bjorling, J Gonzalez, P Bonnet, LightNVM: the linux open-channel ssd subsystem, in: Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST), 2017.
[3] J. Bonwick, The Slab Allocator: an object-caching kernel memory allocator, in: Proceedings of the USENIX Summer 1994 Technical Conference (ATC), 1994.
[4] S Diesburg, C Meyers, M Stanovich, M Mitchell, J Marshall, J Gould, AIA Wang, G Kuenning, TrueErase: per-file secure deletion for the storage data path, in: Proceedings of the 2012 ACM Annual Computer Security Applications Conference (ACSAC), 2012.
[5] SR Dulloor, S Kumar, A Keshavamurthy, P Lantz, D Reddy, R Sankaran, J Jackson, System software for persistent memory, in: Proceedings of 2014 European Conference on Computer Systems (EuroSys), 2014.
[6] R Fagin, J Nievergelt, N Pippenger, HR Strong, Extensible hashing—a fast access method for dynamic files, in: ACM Transactions on Database Systems 4, 1979, pp. 315–344.
[7] Ghemawat S, Dean J, LevelDB, https://github.com/google/leveldb, 2018.
[8] Intel®. DAOS: revolutionizing high-performance storage with intel optane™ technology. https://www.intel.com/content/www/us/en/high-performance-computing/daos-high-performance-storage-brief.html, 2019.
[9] S Kannan, AC Arpaci-Dusseau, RH Arpaci-Dusseau, Y Wang, J Xu, G Palani, Designing a true direct-access file system with DevFS, in: Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST), 2018.
[10] Y Kwon, H Fingler, T Hunt, S Peter, E Witchel, T Anderson, Strata: a cross media file system, in: Proceedings of the 26th ACM Symposium on Operating Systems Principles, 2017.
[11] M. Kuhn, JULEA: A flexible storage framework for HPC, in: Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.
[12] A Lakshman, P Malik, Cassandra: a decentralized structured storage system, in: ACM SIGOPS Operating Systems Review 44, 2010, pp. 35–40, 2010.
[13] J Liu, AC Arpaci-Dusseau, RH Arpaci-Dusseau, S Kannan, File systems as processes, in: Proceedings of the 11th USNEIX Workshop on Hot Topics in Storage and File Systems, 2019.
[14] L Lu, TS Pillai, AC Arpaci-Dusseau, RH Arpaci-Dusseau, WiscKey: separating keys from values in SSD-conscious storage, in: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST), 2016.
[15] M Mesnier, F Chen, T Luo, JB Akers, Differentiated storage services, in: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), 2011.
[16] M Nam, H Cha, YR Choi, SH Noh, Write-optimized dynamic hashing for persistent memory, in: Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST), 2019.
[17] OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/news/openssl-1.1.0-notes.html, 2019.
[18] S Peter, J Li, I Zhang, DRK Ports, D Woos, A Krishnamurthy, T Anderson, T Roscoe, Arrakis: The operating system is the control plane, in: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014.
[19] JL Peterson, TA. Norman, Buddy systems, in: Communications of the ACM 20, 1997, pp. 421–431.
[20] pmem.io PMDK Introduction, https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-pmdk, 2020.
[21] O Rodeh, J Bacik, C Mason, BTRFS: the linux b-tree filesystem, in: ACM Transactions on Storage (TOS) 9, 2013. Article No. 9.
[22] M Rosenblum, JK. Ousterhout, The design and implementation of a log-structured file system, in: ACM Transactions on Computer Systems (TOCS) 10, 1992, pp. 26–52.
[23] S Seshadri, M Gahagan, S Bhaskaran, T Bunker, A De, Y Jin, Y Liu, S Swanson, 2014. Willow: a user-programmable SSD, in: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014.
[24] K Shen, S Park, M Zhu, Journaling of journal is (almost) free, in: Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST), 2014.
[25] PJ Shetty, RP Spillane, RR Malpani, B Andrews, J Seyster, E Zadok, Building workload-independent storage with VT-trees, in: Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST), 2013.
[26] 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.
[27] M Sivathanu, V Prabhakaran, FI Popovici, TE Denehy, AC Arpaci-Dusseau, RH Arpaci-Dusseau, Semantically-smart disk systems, in: Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST), March 2003.
[28] M Sivathanu, LN Bairavasundaram, AC Arpaci-Dusseau, RH Arpaci-Dusseau, Life or death at block level, in: Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), December 2004.
[29] M Sivathanu, AC Arpaci-Dusseau, RH Arpaci-Dusseau, S Jha, A logic of file systems, in: Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST), 2005.
[30] K Sun, D Fryer, J Chu, M Lakier, AD Brown, A Goel, in: Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST), 2016.
[31] Sun Microsystems. In a class by itself—the solaris 10 operating system, A Tech. White Paper, November 2004.
[32] S Swanson, A. Caulfield, Refactor, reduce, recycle: restructuring the I/O stack for the future of storage, in: Computer 46, August 2013, pp. 52–59.
[33] Szeredi M. Filesystem in Userspace. http://fuse.sourceforge.net, 2005.
[34] H Volos, S Nalli, S Panneerselvam, V Varadarajan, P Saxena, MM Swift, Aerie: flexible file-system interfaces to storage-class memory, in: Proceedings of the 2014 European Conference on Computer Systems (EuroSys), 2014.
[35] Wilcox M. DAX: page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064, 2014.
[36] D. Woodhouse, JFFS: the journaling flash file system, in: Proceedings of the Ottawa Linux Symposium, 2001.
[37] Z Yang, JR Harris, B Walker, D Verkamp, C Liu, C Chang, G Cao, J Stern, V Verma, LE Paul, SPDK: a development kit to build high performance storage applications, in: Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2017.