Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3772052.3772208

RESEARCH-ARTICLE

DFUSE: Strongly Consistent Write-Back Kernel Caching for Distributed Userspace File Systems

HAOYU LI, Columbia University, New York, NY, United States

JINGKAI FU, Columbia University, New York, NY, United States

QING LI, Alibaba Group, USA, San Mateo, CA, United States

WINDSOR HSU, Alibaba Group, USA, San Mateo, CA, United States

ASAF CIDON, Columbia University, New York, NY, United States

Open Access Support provided by:

Columbia University

Alibaba Group, USA

PDF Download
3772052.3772208.pdf
03 February 2026
Total Citations: 0
Total Downloads: 46

Published: 19 November 2025

Citation in BibTeX format

SoCC '25: ACM Symposium on Cloud Computing
November 19 - 21, 2025
Online, USA

Conference Sponsors:
SIGOPS
SIGMOD

DFUSE: Strongly Consistent Write-Back Kernel Caching for Distributed Userspace File Systems

Haoyu Li[†] Jingkai Fu[†] Qing Li[♮] Windsor Hsu[♮] Asaf Cidon[†]
Columbia University[†] Alibaba Cloud[♮]

Abstract

Cloud platforms host thousands of tenants that demand POSIX semantics, high throughput, and rapid evolution from their storage layer. Kernel-native distributed file systems supply raw speed, but their privileged code base couples every release to the kernel, widens the blast radius of crashes, and slows innovation. FUSE-based distributed file systems flip those trade-offs: they run in user space for fast deployment and strong fault isolation, yet the FUSE interface disables the kernel's write-back page cache whenever strong consistency is required. Practitioners must therefore choose between (i) weak consistency with fast write-back caching or (ii) strong consistency with slow write-through I/O, a limitation that has kept FUSE distributed file systems out of write-intensive cloud workloads.

To this end, we present DFUSE, the first distributed FUSE file system that delivers write-back kernel caching and strong consistency. DFUSE achieves this by offloading userspace consistency control to the kernel driver, allowing coordinated access to the kernel's page cache across nodes. This design eliminates blind local cache updates and ensures cluster-wide strong consistency without compromising performance. In our evaluation, DFUSE achieves up to 68.0% higher throughput and 40.4% lower latency than the existing write-through design of FUSE-based distributed file systems.

CCS Concepts

• **Software and its engineering → File systems management**; **Consistency**; • **Information systems → Distributed storage**; • **Computer systems organization →** *Cloud computing*; • **General and reference → Performance**.

Keywords

FUSE, cache, consistency, distributed file systems

ACM Reference Format:
Haoyu Li[†] Jingkai Fu[†] Qing Li[♮] Windsor Hsu[♮] Asaf Cidon[†], Columbia University[†] Alibaba Cloud[♮] . 2025. DFUSE: Strongly Consistent Write-Back Kernel Caching for Distributed Userspace File Systems. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772208

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '25, Online, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772208

1 Introduction

Large, multi-tenant clouds demand storage that can evolve quickly, survive component failures, and plug into unmodified POSIX applications [7, 8, 44, 55]. Kernel-native distributed file systems [48, 49] excel at raw throughput, but their code runs with full kernel privileges: a single bug can crash the entire node, upgrades require kernel rollouts, and every new feature must wait for the next kernel release cycle. These constraints slow innovation and increase the blast radius of faults (Table 1). To sidestep those hurdles, cloud operators are increasingly embracing FUSE-based DFSs—userspace file systems that preserve the kernel's VFS and page-cache interfaces while isolating file-system logic in separate processes. FUSE lets developers iterate rapidly, ship features without kernel changes, and contain crashes to the file-system process itself, making it a natural fit for fast-moving services such as object-store gateways, snapshot engines, and ML-training checkpoints [3, 8, 9, 13, 15, 16, 18, 25]. Recent innovations [22, 26, 33, 45, 59] further mitigate historical performance bottlenecks making FUSE increasingly viable for large-scale deployments.

However, a fundamental challenge persists: FUSE-based distributed file systems struggle to provide both high performance and strong consistency (i.e., allow distributed clients to read the most recent written value). The kernel's page cache—a key driver of efficiency in local file systems—remains largely incompatible with distributed environments. The write-back page cache enables applications to return from I/O operations immediately after buffering data in the kernel, removing the expensive userspace synchronization of FUSE from the critical path. Yet, in distributed settings, this risks data races and inconsistency, as nodes may operate on stale or uncoordinated cached copies. For example, if node A writes data to its local page cache (happens in the kernel) without immediately synchronizing with other nodes (happens in the userspace), node B might read stale data or make conflicting updates, leading to data corruption. Using the write-through mode of kernel page cache mitigates the consistency issue at the cost of low performance (Figure 2). Therefore, prior solutions either sacrifice strong consistency for performance (e.g., enabling a write-back cache without synchronization)[2, 12] or default to write-through caching[55], which synchronizes every write to userspace (Table 1). In write-through mode, each operation must complete synchronization before returning to the application, incurring high latency. This trade-off limits practicality in consistency-sensitive cloud workloads, such as databases or real-time analytics.

We present DFUSE, a FUSE-based distributed file system that bridges the gap between high performance and strong consistency. DFUSE enables write-back page caching while enforcing strong consistency across nodes, eliminating the need to compromise between performance and correctness. The core insight of DFUSE is offloading userspace consistency control (e.g., lease management)

Table 1: Comparison between DFUSE and existing solutions.

		Existing FUSE-based DFSes		Existing Kernel-native DFSes	
Dimension	**DFUSE**	**Ceph-fuse**	**GlusterFS**	**Lustre**	**GPFS**
Client location	User space	User space	User space	Kernel	Kernel
Page-cache mode	Write-back	Write-through (slow)	Write-back	Write-back	Write-back
Consistency	Strong	Strong	Weak	Strong	Strong
Fault isolation	Strong	Strong	Strong	Weak	Weak
Deployment	Unprivileged	Mostly unprivileged	Unprivileged	Privileged	Privileged
Development	Fast	Fast	Fast	Slow	Slow

to the kernel FUSE driver. By embedding distributed metadata directly into the FUSE driver, DFUSE coordinates access permissions within the kernel. This design resolves the kernel space deadlock issue in current systems (discussed in §3.2). This allows nodes to gain cluster-wide access within the kernel during I/O operations, ensuring that only authorized nodes read or modify cached data. For instance, when a write lease is revoked, the kernel proactively invalidates the local page cache and flushes dirty pages. This design retains FUSE's userspace flexibility while leveraging the kernel's efficiency, avoiding costly round trips for common-case operations.

Our evaluation demonstrates that DFUSE consistently improves I/O throughput and latencies compared to a baseline, which represents the architecture of systems such as GlusterFS[12], an open-source distributed file system based on FUSE. Results demonstrate a 1.75x throughput improvement at best, outperforming write-through approaches. DFUSE also scales linearly with node count, maintaining consistency even at 25% contention.

Our contributions include:

- An analysis showing that leveraging kernel's page cache fundamentally breaks strong consistency in FUSE-based distributed file systems.
- The kernel offloading mechanism that embeds distributed coordination logic into the FUSE driver.
- The design and implementation of DFUSE, the first FUSE-based system to support write-back caching with strong consistency.

The rest of this paper is organized as follows: §2 provides background on FUSE and FUSE-based distributed file systems; §3 details consistency and locking challenges; §4 and §5 present DFUSE's design and implementation, respectively; §6 evaluates its efficacy; §7 discusses related work; and §8 concludes the paper.

2 Background

2.1 FUSE in Cloud Computing

FUSE (Filesystem in Userspace) has become a very popular file system for modern cloud infrastructure [3, 8, 9, 13, 15, 16, 18, 25], enabling developers to build custom file systems without modifying the kernel. By running the file system in userspace, FUSE prevents kernel crashes due to bugs and accelerates development cycles. For example, systems like S3FS [16] (mounting cloud object storage as POSIX file systems) and JuiceFS [13] (optimizing cloud data access with caching) leverage FUSE to expose a familiar POSIX and file system to applications while abstracting different backend storage interfaces such as object stores.

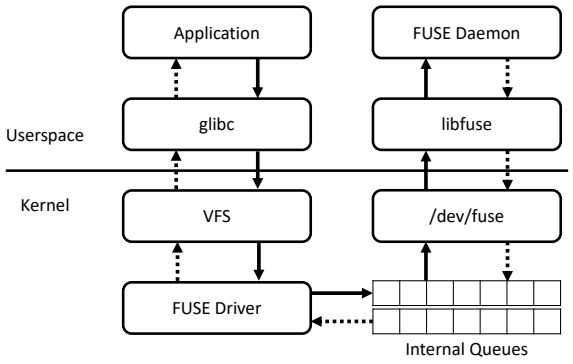


Figure 1: Overview of the FUSE Framework

Recent projects further enhance FUSE's viability for large-scale deployments. For example, XFUSE [33] reduces kernel-userspace context switches by batching requests and scalable communication patterns. Stackable file systems [45] layer caching or encryption on top of existing FUSE implementations, improving metadata operation efficiency. These enhancements improve FUSE performance when running a file system on a single node, but fundamental challenges remain when trying to run a FUSE-based distributed file system.

2.2 FUSE Architecture Overview

FUSE is a framework that allows developers to build userspace file systems exportable to the Linux kernel. As illustrated in Fig 1, it comprises three key components: the FUSE kernel driver, the userspace library (libfuse [14]), and a custom FUSE daemon. The kernel driver registers the file system with the VFS, allowing applications to interact with it via standard POSIX interfaces (e.g., open, read, write) exposed by the GNU C Library (glibc). When an application issues a syscall, glibc invokes the corresponding VFS function, which routes the request to the FUSE driver. The driver translates the syscall into a FUSE protocol message (e.g., `FUSE_READ`) and forwards it to userspace via a special device called /dev/fuse. To summarize, while FUSE is a userspace-based file system, when applications interact with it, the I/O request gets routed first to the kernel then back to the file system in userspace.

The userspace library libfuse abstracts low-level protocol details, providing developers with higher-level APIs to implement file system operations (e.g., read, write) in the custom FUSE daemon. This daemon—a standalone process—executes developer-defined logic, such as fetching data from remote storage or enforcing access policies. Once the daemon processes a request, it returns a response through /dev/fuse to the kernel driver, which propagates the result back to the application via the VFS.

Communication between the kernel driver and userspace daemon occurs through /dev/fuse, which maintains prioritized queues for different request types. The daemon polls these queues, blocking when empty until the driver signals new requests. Moreover, FUSE supports a notification mechanism: the daemon can proactively notify the kernel driver of certain events. For example, in distributed scenarios, upon receiving a revocation request from a lock manager, the daemon can notify the kernel driver to flush buffered data from the page cache.

The above FUSE's architectural design introduces performance overheads, primarily due to frequent kernel-userspace mode switches and extra memory copies [26, 33, 36, 53, 54, 59], where every file system operation traverses the kernel-to-userspace boundary via the /dev/fuse device. For instance, a simple `write()` syscall incurs multiple context switches (first to the kernel driver, then to the userspace daemon, and back) along with data copies between kernel and userspace buffers.

2.3 Leveraging the Page Cache

Even though the file system implementation resides in userspace, the FUSE kernel driver utilizes the kernel's page cache to accelerate I/O operations by indexing cached data using inode numbers and page offsets. FUSE leverages the kernel's page cache in one of two modes. In write-back mode, writes return immediately after buffering data in the kernel cache, deferring synchronization to the userspace daemon. This minimizes both mode switches (by avoiding synchronous userspace round trips) and memory copies (by retaining data in the kernel's page cache). Consequently, write-back mode significantly reduces latency and improves throughput, making it attractive for performance-sensitive workloads.

In contrast, write-through mode synchronously updates both the kernel cache and userspace daemon. Then, the userspace daemon can propagate the update to other nodes, ensuring strong consistency in a distributed file system setting. However, this approach increases the write latency as each write has to go to userspace (the daemon typically maintains its own userspace in-memory cache to reduce communication overhead with remote storage services in distributed FUSE-based file systems).

2.4 FUSE Performance Breakdown

We quantify the performance benefits of kernel's write-back page cache in Fig 2. To break down the overhead of the FUSE protocol itself, we implement a `NullFS` which simply returns zero in any file system operations in the userspace. Note that none of the requests actually go to the disk. The stacked bars (from the bottom up) present the life cycles of write requests chronologically.[1] First, the application issues a write system call, which triggers a mode switch

[1]The workload is 1M random 4KB writes over 1K files of 16MB after warm-up.

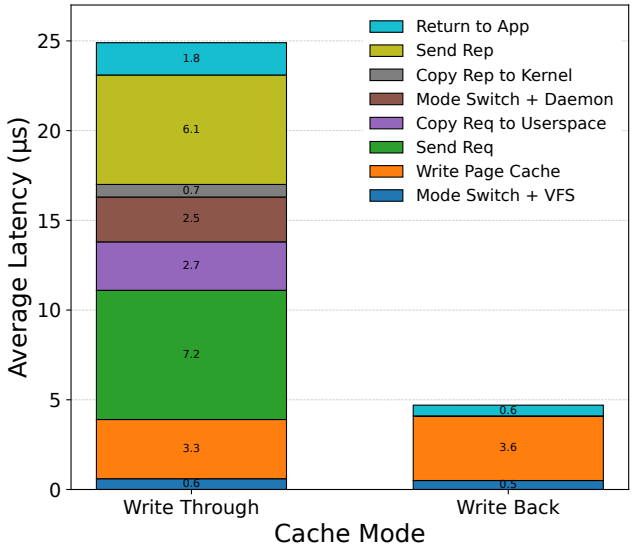


Figure 2: Latency breakdown of write requests in FUSE

into the kernel, and the VFS layer dispatches the request to the FUSE driver. In both cache modes, the content within the write request will be copied to the kernel's page cache. If the write-back cache is enabled, FUSE then returns the number of written bytes or an error to the application, with an average total latency of 4.7 μs (the sum of the left bar).

In contrast, in the write-through mode, after writing the page cache, FUSE continues to pass the write request to the userspace daemon. In particular, the FUSE driver as in Fig 1 first places the request in one of the internal queues and next wakes up the daemon thread that waits on the queues (7.2 μs). Subsequently, the daemon thread dequeues the request and copies it to the userspace buffer (2.7 μs). After switching to the userspace, the daemon invokes the userspace write function to handle the requests (2.5 μs). Finally, the response returns to the kernel driver in the same way, i.e., the daemon copies the reply to the kernel buffer (0.7 μs) and notifies the FUSE driver thread of the incoming result (6.1 μs).

From Fig 2, the overhead of the extra request round trip in FUSE write-through cache is on average 19.2 μs, while write-back cache mode incurs only 4.7 μs.

2.5 Achieving Strong Consistency

Strong consistency is a critical property for distributed file systems, ensuring that any update made to data is immediately visible to subsequent read operations across all nodes. Implementing strong consistency typically involves synchronization mechanisms that ensure updates propagate immediately and atomically across all nodes.

A common approach used in distributed file systems is the lease mechanism [31, 37, 42, 48, 49]. Compared to other synchronization mechanisms, e.g., distributed locks, leases achieve lower synchronization overhead as the lease-holding node can safely perform operations locally during a lease period without communication with the lease coordinator. Once the lease expires or is revoked, the

node must either renew it or cease operations on the leased data until it obtains a new lease.

In FUSE-based distributed file systems, implementing leases is particularly challenging due to the separation between kernel-resident page caches and userspace consistency logic. For instance, a node holding a write lease may buffer updates in its kernel page cache, leaving other nodes unaware of pending changes until a flush occurs. Prior systems like Ceph-FUSE resort to write-through caching to synchronize every update with userspace, while Gluster sacrifices consistency for performance. We elaborate on these challenges in §3 and explain how DFUSE addresses these limitations by embedding lease management directly into the kernel driver, enabling write-back caching without compromising consistency in §4.

3 Challenges

This section explores why it is challenging to leverage the kernel's page cache under either write-back or write-through mode effectively in distributed FUSE settings.

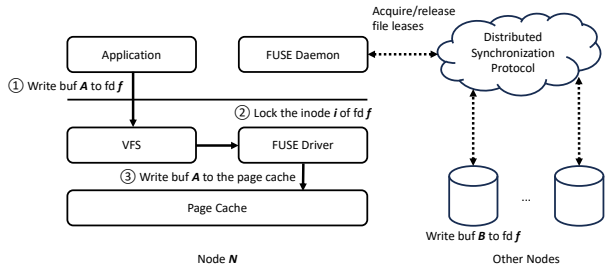
3.1 Kernel-Userspace Coordination Failure in Write-Back Mode

Write-back mode in the kernel page cache fundamentally breaks strong consistency in distributed FUSE systems. The primary reason is the architectural separation between where data operations occur (kernel) and where distributed coordination happens (userspace).

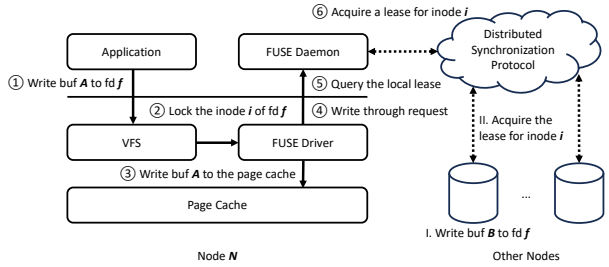
We illustrate the data inconsistency issue caused by enabling write-back cache in Figure 3a, where multiple nodes form a distributed file system and follow a synchronization protocol. To save space, we only plot the internal details of node N, since all nodes are symmetric. Suppose that node N writes some content `buf A` to file `f` (step ①). Under the write-back mode, the FUSE driver will only acquire the inode lock for `f` in the kernel (step ②) and then return a completion to the application after writing the data to the page cache (step ③). This data path bypasses the userspace daemon which implements the synchronization protocol between nodes. Consequently, not only are other nodes unable to read the updated content of `buf A` in file `f`, but also they themselves might write different values to the same file, leading to data inconsistency.

3.2 Deadlock Caused by Reversed Lock Order With Write Through

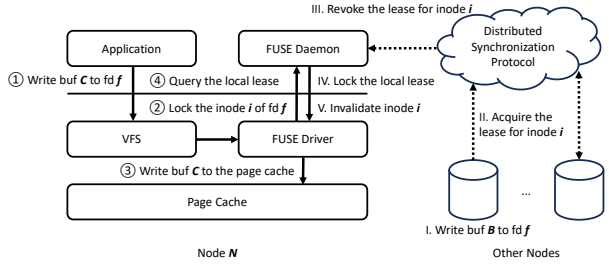
As using the write-back mode bypasses any userspace synchronization, FUSE-based distributed file systems can resort to the write-through mode to utilize the kernel's page cache. As Figure 3b shows, after writing the data `buf A` to the page cache with the inode lock, to obtain knowledge of the distributed access permission, the FUSE driver issues a write-through operation to the userspace daemon (step ④). Then, the FUSE daemon is able to check its access permission for the file `f` (step ⑤) and send requests to acquire the write permission if necessary (step ⑥). So, write operations will only return to the application when the current node has the permission to do so, excluding data races shown in Figure 3a.



(a) Data inconsistency caused by missing cross-layer coordination: kernel inode locks vs. userspace distributed protocols



(b) The workflow in a FUSE-based distributed file system. Each read and write request need to go through the userspace to obey the distributed synchronization protocol.



(c) The deadlock issue caused by the reversed lock order during lease revocation in the write-through mode.

Figure 3: Challenges of leveraging the write-back page cache in FUSE-based distributed file systems

Nevertheless, there is an issue of locking order in the above write-through solution. Because different threads may acquire and revoke a lease concurrently, the file system usually needs to protect it with an additional read-write lock.

Figure 3c presents an example. Suppose that node N has held a write lease for file `f` from the beginning. When writing `f`, it first needs to grab the inode lock in the kernel (step ②). Once the write request arrives in the userspace, it has to obtain another lock of the userspace lease to read its content (step ④). This kernel-userspace locking order would conflict with the sequence required during a lease revocation request. In particular, if another node wants to write the same file `f` (step I) and tries to acquire the distributed lease of it (step II), a revoke request for file `f`'s lease may be sent to node *N* (step III). At this time, the revoking request will grab locks in the reversed order: first locking the lease (step IV) and then the inode in the kernel (step V), which leads to deadlocks.

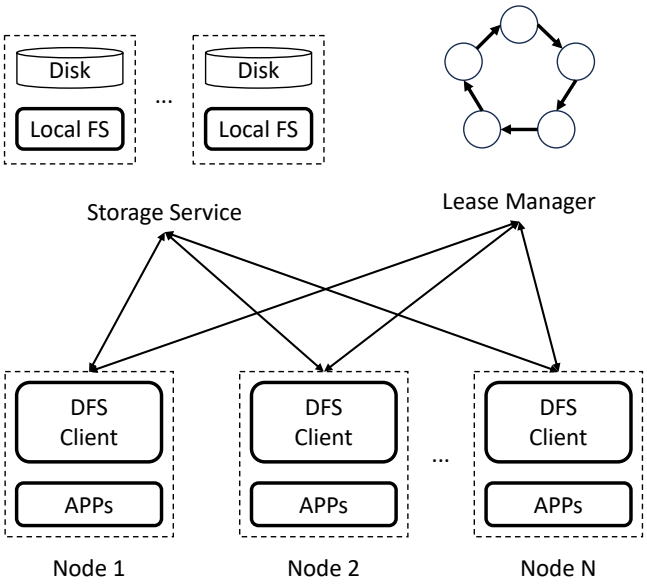


Figure 4: The overview of DFUSE

One workaround would be to use optimistic concurrency control (OCC). Specifically, at step IV of Figure 3c, the revoker does not hold the lock of the lease when invalidating the inode, assuming that there is no concurrent writer that will update file f's page cache. After the invalidation finishes, the revoker repeats the invalidation if any concurrent writers are detected. So, the deadlock issue is resolved. However, under high contention, the revoking performance would be low, leading to many aborts[34, 57, 58]. Even worse, OCC would lead to unfair lease distribution, as the node that already has the lease can keep writing to the file even if it is being explicitly revoked[21, 57].

In summary, it is challenging to leverage the kernel's page cache under the write-back or the write-through mode in a FUSE-based distributed file system. The fundamental reason is the lack of userspace information when writing to the page cache. Consequently, the file system either sacrifices the strong consistency (write back) or experiences high latencies of writes and revocations (write through). Systems like GlusterFS prioritize performance with write-back caching at consistency's expense, while others like Ceph-FUSE choose consistency through write-through implementations, accepting the performance penalty.

4 Design

DFUSE is a FUSE-based distributed file system that enables write-back page caching while maintaining strong consistency. The system bridges the gap between high performance and strong consistency by offloading distributed concurrency control to the kernel driver, allowing the kernel's page cache to be safely utilized across distributed nodes. Table 1 highlights DFUSE's differences from existing FUSE-based and kernel-native solutions. Figure 4 illustrates DFUSE's overall architecture, which consists of three major components:

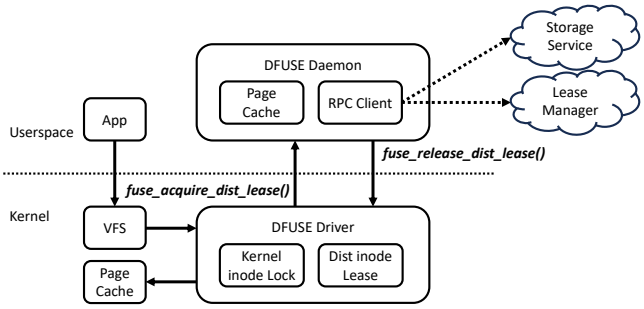


Figure 5: DFS client components.

- DFS client: Multiple userspace file system instances mounted on different nodes, providing POSIX-compliant file access to applications.
- Lease manager: A service that coordinates distributed leases to ensure consistency across nodes.
- Storage service: A centralized backend responsible for durable data storage.

These components run on different physical servers and communicate with each other via RPCs. To use DFUSE, an application simply mounts a DFS client to a local directory (e.g., /mnt/fuse) and accesses files as if using a local file system. Applications across all nodes see a unified and consistent view of the file system through the standard POSIX interface.

4.1 DFS Client

Each DFS client functions as a FUSE-based file system mounted to a local directory, collectively providing a unified namespace across all nodes. Applications interact with DFUSE using standard POSIX interfaces. The key innovation in DFUSE is offloading userspace distributed lease management to the kernel driver, enabling the system to validate lease status directly within the kernel during I/O operations.

This lease offloading mechanism is what enables DFUSE to use write-back page caching while maintaining strong consistency across the distributed system. Figure 5 illustrates the internal components of a DFS client. Importantly, DFUSE confines all kernel modifications to the FUSE kernel driver, leaving other kernel components (e.g., VFS and page cache) unchanged. Users simply need to load the DFUSE kernel driver and install our customized libfuse.

4.1.1 Why DFUSE chooses leases. Distributed file systems typically enforce cache consistency with callback locks [32], leases [31], version vectors [50], or optimistic validation [38]. We adopt leases because they already balance low-latency reads with bounded staleness and are widely deployed in production DFSs [23, 27, 30, 47, 51]. The lease protocol itself is therefore not a contribution of this work; its role is to provide a familiar yard-stick against which to evaluate DFUSE's kernel-offload mechanism. DFUSE's design also has the potential to utilize other consistency protocols mentioned above.

4.1.2 Lease Offloading. DFUSE's core innovation is embedding distributed lease information directly in the kernel's FUSE driver. In this section, we show how this design allows the system

to coordinate page cache access across nodes without sacrificing performance.

DFUSE adds a distributed lease field to the FUSE inode data structure, representing the current node's access permission for that inode. This lease has three possible states: shared read, exclusive write, or null. The lease initializes to null when an application opens a file, indicating no initial read or write access.

For each inode operation requiring shared read or exclusive write, the DFUSE driver checks the lease type after acquiring the kernel inode lock. If it finds that the node is not holding the appropriate lease (null or weaker than required), the driver requests the userspace daemon to update the lease. Conversely, when the DFUSE daemon receives a revocation from the lease manager, it instructs the kernel driver to release the lease.

To manage these offloaded leases, DFUSE introduces two new FUSE operations:

`fuse_acquire_dist_lease()`. Called when the driver determines its local lease does not provide sufficient permission. The driver packages the inode number and desired permission (read or write) into a request sent to the userspace daemon through /dev/fuse. The daemon then acquires the distributed lease from the lease manager via RPC. Upon success, the driver updates the local lease and proceeds with the application's operation. The application blocks on a read or write until the FUSE driver finishes the operation.

While lease acquisition still requires an extra kernel-userspace round trip, subsequent accesses with the same lease can operate directly on the kernel page cache without userspace involvement, significantly reducing I/O latency.

`fuse_release_dist_lease()`. Handles lease revocation by: 1) blocking new reads and writes, 2) waiting for ongoing operations to complete, 3) flushing dirty pages to storage, and 4) marking the local lease as null. This notification ensures the driver follows the same locking order as during lease acquisition, avoiding the deadlock issue described in §3. If a revocation happens concurrently with application read or write requests (to the same file), the application will have to wait until the FUSE driver re-acquires the lease.

Example end-to-end workflow. To better demonstrate the lease offloading mechanism in DFUSE, we show the workflow of a write request from an application running on DFS client `A`. First, the application issues a write syscall with glibc, which will be routed to the FUSE kernel driver by VFS. Next, the FUSE driver locks the lease of the inode (serialize concurrent revocation) and checks the lease status. If the lease has the write permission, then the FUSE driver proceeds with the normal write-back operation, i.e., acquiring the inode lock and updating the page cache. However, if the lease status is either null or shared read, the FUSE driver will send a `fuse_acquire_dist_lease()` operation to the userspace daemon through /dev/fuse. The kernel thread falls into interruptible sleep while waiting for the result. Once the userspace daemon acquires a write lease from the manager, it wakes up the FUSE driver thread. Lastly, the driver updates the lease type to write, updates the kernel page cache, and returns the control to the application. This solves the challenge mentioned in §3.1.

We next discuss how a DFS client handles lease revocations. Suppose that during a write request, the userspace daemon of client `A` receives a lease revocation request for the same file. To properly revoke the lease, client `A` must avoid new I/Os to this file and the deadlock issue mentioned in §3.2. So, the daemon triggers a `fuse_release_dist_lease()` notification to the FUSE driver, which will first lock the lease (to prevent new I/Os from reading the status) and then lock the inode (to flush its cached pages). By following the same locking order as normal read and write requests, DFUSE resolves the deadlock challenge in §3.2. Subsequent I/Os to the file will find the lease in the null status and trigger the above lease acquisition.

4.1.3 Userspace Buffer Cache. To further optimize performance and reduce communication with the remote storage service, DFUSE implements a secondary in-memory buffer cache in the userspace daemon. This inclusive userspace cache prevents network bottlenecks by batching operations that would otherwise require immediate network round trips when data is flushed from kernel, and it enables customized caching policies in the userspace. Unlike the kernel's page cache, which can grow dynamically based on system memory pressure, the userspace buffer cache maintains a fixed memory reservation to provide predictable performance and resource utilization.

This dual-caching strategy creates a tiered approach to data access. When the kernel's background threads asynchronously flush dirty pages from the kernel page cache, the data is not immediately sent to the remote storage service. Instead, these pages are first buffered in the userspace buffer cache, which serves as an intermediary layer between the kernel and remote storage.

Similarly, when a read operation encounters a cache miss in the kernel page cache, the DFUSE driver forwards the request to the userspace daemon. The daemon first attempts to locate the requested data in its userspace buffer cache before making any RPC calls to the remote storage service. This approach effectively reduces the frequency of expensive remote storage operations, especially for workloads with temporal locality beyond what the kernel cache can capture.

To further minimize RPC overhead, DFUSE implements batch read and write operations that consolidate multiple page accesses into a single RPC request. When the userspace daemon needs to read or write multiple pages, it aggregates these operations and sends them as a batch to the remote storage service, amortizing the cost of network latency and protocol overhead across multiple operations. This batching mechanism is particularly beneficial for sequential I/O patterns and large file transfers where multiple adjacent pages are likely to be accessed together.

As we will show in the evaluation (§6), the combination of kernel page caching, userspace page caching, and batch operations significantly reduce the performance impact of remote storage access while maintaining strong consistency guarantees across the distributed system.

4.1.4 Global File Identifier. Since FUSE reuses existing data structures from VFS, its inode number generation is only based on local assignment. Therefore, every DFS client may have different inode numbers for the same file. Fortunately, FUSE provides a tag for each file to save custom fields and passes the tag in FUSE operations.

DFUSE stores a global file identifier (GFI) in the tag to identify the same files among different DFS clients. Specifically, the GFI consists of two parts: 1) the ID of the node in the storage service that actually stores the file and 2) the local inode number from that storage node. Both DFS clients and the lease manager use GFIs to identify shared files. For example, any messages of acquiring and revoking leases contain the GFI of the target file, and a DFS client can determine the target storage node when flushing dirty pages of a file.

4.2 Lease Manager

DFUSE employs a lease-based design to coordinate distributed file access. The system maintains a conceptually global lease manager that enforces strong consistency through per-file distributed read-write leases. This lease mechanism follows the classic distributed read-write lock pattern, with algorithms detailed in Algorithm 1 and Algorithm 2 for DFS clients and the lease manager, respectively.

To avoid a single point of failure, the lease manager can be replicated either by running a consensus protocol such as Paxos [39] or Raft [43], or by outsourcing the state to an off-the-shelf coordination service such as ZooKeeper [35]. DFS clients communicate with the lease manager via RPCs. If the lease manager becomes unreachable due to network partition, clients retain the lease until its soft timeout and fall back to write-through on lease expiry, preserving correctness. At any time, a file may have at most one exclusive writer or multiple readers. Before each read or write operation, a DFS client calls AcquireLease (Algo 1) to ensure it holds the appropriate lease for the target inode.

If the local lease is invalid (e.g., expired) or insufficient for the intended operation (e.g., attempting to write with only a read lease), the DFS client requests an updated lease from the lease manager. A DFS client may also receive revocation requests from the manager explicitly requesting release of a lease. In this case, the instance clears cached data for the inode and flushes any dirty pages back to the storage service, ensuring other nodes can access the latest content.

The lease manager maintains information about current lease types and owners for each file. When receiving lease applications, it determines the appropriate action based on the global lease state and the requesting node's intent. For compatible requests (e.g., multiple read leases), the manager simply adds the requesting node to the owner list. For conflicting requests (e.g., when either the current owner or requester needs write access), the manager revokes existing leases before granting the new lease.

Algorithm 1 Lease Algorithm on DFS clients

```
1:  procedure AcquireLease(inode, intent)
2:      lease ← GetLocalLease(inode)
3:      if lease.type satisfies intent then
4:          return
5:      end if
6:      if lease.type = READ ∧ intent = WRITE then
7:          ReleaseLease(inode)
8:          manager.RemoveOwner(inode, self)
9:      end if
10:     manager.GrantLease(inode, intent, self)
11:     lease.type ← intent
12: end procedure
13: procedure ReleaseLease(inode)
14:     Clear the page cache of inode
15:     Sync dirty pages to the storage service
16:     inode ← None
17: end procedure
```

Algorithm 2 Lease Algorithm on Lease Manager

```
1:  procedure GrantLease(inode, intent, node)
2:      lease ← GetLocalLease(inode)
3:      if lease.owner is empty then
4:          lease.type ← intent
5:          lease.owner ← node
6:      else if lease.type = READ ∧ intent = READ then
7:          Add node to lease.owner
8:      else if lease.type = READ ∧ intent = WRITE then
9:          for holder ∈ lease.owner do
10:             holder.ReleaseLease(inode)
11:         end for
12:         lease.type ← WRITE
13:         lease.owner ← node
14:     else if lease.type = WRITE then
15:         holder ← lease.owner
16:         holder.ReleaseLease(inode)
17:         lease.type ← intent
18:         lease.owner ← node
19:     end if
20: end procedure
```

4.3 Storage Service

DFUSE separates data storage from file system processing by implementing a dedicated storage service. This design aligns with modern cloud computing's disaggregation of compute and storage resources. Rather than reinventing storage management, DFUSE leverages existing mature storage services (such as S3[6], GCS[7], or Pangu[40]) that already provide high reliability and availability. These services offer block or object storage interfaces sufficient to support POSIX file system semantics.

This separation simplifies DFS clients, as they no longer need to manage storage resource allocation or data replication. The design also acknowledges that local storage attached to cloud VMs (e.g., Amazon's ECS) is typically ephemeral—storing data locally on client VMs would risk data loss upon VM failures.

In environments where cloud storage services are unavailable or undesirable, such as on-premise clusters, the storage service can alternatively be implemented as a replicated storage cluster. This approach still offers benefits like more efficient resource allocation. The number and implementation of storage nodes remain decoupled from DFS clients, allowing flexibility to select storage backends (e.g., SPDK[56], RAID[24], Ext4[41], XFS[52]) based on workload characteristics.

Table 2: Filebench workload characteristics

Workload	# Files	Dir Width	File Size	R/W Ratio
Fileserver	10,000	20	1.25MB	1:2
Webserver	80,000	20	160KB	10:1
Netsfs	74,000	20	267KB	5:2

4.4 Adoption by Existing Systems

DFUSE's architecture is designed with portability and extensibility in mind, making it readily adaptable for existing FUSE-based distributed file systems. The core innovation of lease offloading is implemented as a modular extension to the standard FUSE kernel driver. Existing systems like GlusterFS, Ceph-FUSE, or JuiceFS can adopt DFUSE's lease offloading mechanism with three straightforward integration steps: 1) replace their standard FUSE driver with DFUSE's modified kernel driver, 2) adapt their userspace daemon to implement the two new FUSE operations (`fuse_acquire_dist_lease()` and `fuse_release_dist_lease()`), and 3) integrate with a lease manager service or extend their existing metadata coordination services to handle lease management. This approach allows these systems to maintain their established storage backends, authentication mechanisms, and specialized optimizations while gaining DFUSE's ability to leverage write-back caching with strong consistency guarantees.

5 Implementation

We implemented DFUSE on libfuse 3.16 and Linux 5.10. We selected gRPC as the inter-node communication protocol. For this prototype implementation, the storage service is comprised of multiple ext4 file systems, where each distributed file system instance translates low-level FUSE operations to their corresponding ext4 operations. To manage the userspace buffer cache efficiently, we integrated Facebook's CacheLib framework[20], which enables flexible admission and eviction policies with minimal overhead. The implementation required approximately 4,000 lines of code for the userspace file system component, with relatively minimal modifications to the existing codebase: approximately 100 lines of code for libfuse modifications and 300 lines for the FUSE kernel driver enhancements.

6 Evaluation

We seek to answer three evaluation questions:

Q1: How does DFUSE perform end-to-end under different workloads (§6.2)?

Q2: How do various contention levels affect DFUSE's performance (§6.3)?

Q3: How does DFUSE scale with increasing number of nodes (§6.4)?

6.1 Experiment Setup

Setup. We conduct all experiments on CloudLab's [28, 46] c220g1 servers. Each c220g1 server has two 8-core Intel E5-2630 CPUs at 2.4 GHz, 128 GB DDR4 memory, a 480 GB Intel DC S3500 SSD, and a dual-port Intel X520-DA2 10Gb NIC [2]. The OS is Ubuntu 20.04 LTS with a Linux version of 5.10.

In our evaluation, unless otherwise specified, we run DFUSE with four DFS clients, one storage node with ext4 file system, and one lease manager node. For each DFS client, we mount the FUSE daemon with default permissions and multi-threading. Both the baseline and DFUSE use a userspace cache of 1 GB per instance.

Workload. We use fio [19] and filebench [10] as micro and macro benchmarks, respectively, to evaluate DFUSE. In fio, the patterns cover both random and sequential I/O with five read-write ratios. There are four threads per node, each operating on a working set of 100 16MB files. We choose such a working set size to emulate SSTable objects (typically sized in the tens of MB) [5, 29], generate enough hot inodes to exercise leases/invalidations, and avoid saturating storage/network–our conclusions target per-inode coherence costs rather than absolute dataset size. We run three representative applications from filebench in this evaluation: fileserver, webserver, and netsfs (their characteristics are shown in Table 2).

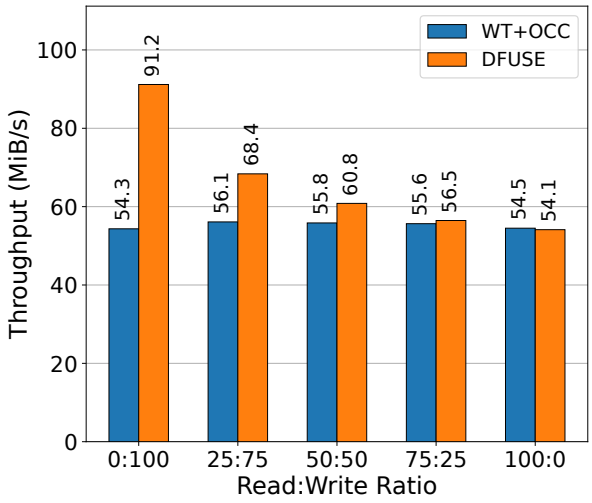
Metric. We compare the aggregated throughput and average latency to reflect DFUSE's performance. The throughput reported in this section is the cluster-wide result, summing each node's throughput. The latency is the arithmetic mean of all operations' latency.

Baseline. Our goal is to isolate the effect of kernel write-back vs. write-through under identical file system logic, so we compare against a same-code baseline that differs only in cache/write policy (WT+OCC). As we discussed in §3, this is the best that current systems like Ceph-FUSE[55] and GlusterFS[12] can do if they wish to leverage kernel's page cache while ensure strong consistency. Head-to-head with Ceph/Gluster would conflate client, metadata, striping, recovery, and placement policies with our lease offloading mechanism, making causal attribution unclear.

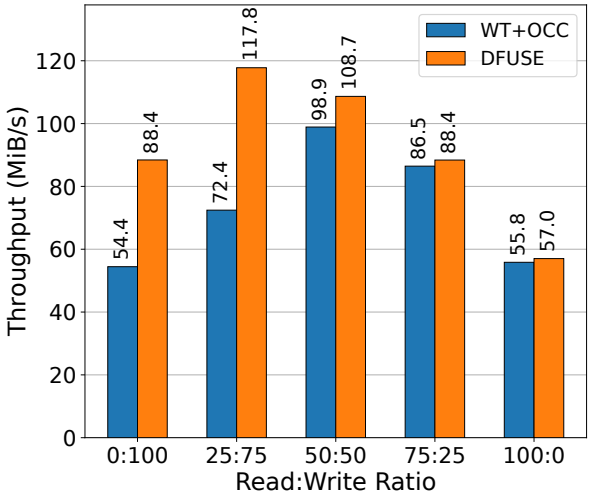
6.2 Different Workloads (Q1)

To evaluate DFUSE's end-to-end performance across various workloads, we conducted experiments using both random and sequential I/O patterns with varying read-write ratios. Figures 6 present the total throughput and average latency comparison between DFUSE and the baseline (write-through with OCC) implementation. For random I/O workloads (Figure 6a and 6c), DFUSE demonstrates consistent performance advantages across all workload with writes. The improvement is particularly pronounced in write-heavy scenarios (i.e., 0:100 and 25:75 read-write ratios), where DFUSE achieves up to 68.0% higher throughput and 40.4% lower latency. This significant improvement stems from DFUSE's ability to leverage write-back caching without synchronizing every write to the userspace. As the workload becomes more read-intensive, the performance gap narrows but remains favorable for DFUSE, with improvements of 9.0% for balanced workloads (50:50) and 1.6% for read-heavy workloads (75:25). For pure read workloads (100:0), both systems perform similarly since read operations benefit equally from the kernel's page cache under both write-back and write-through mode.

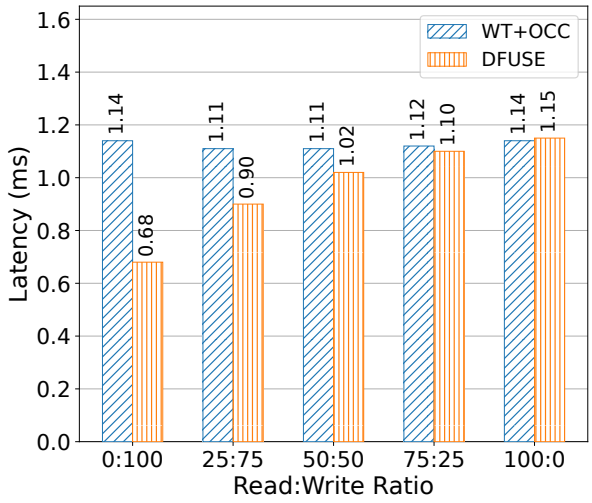
[2]The second port was unused in the experiment.



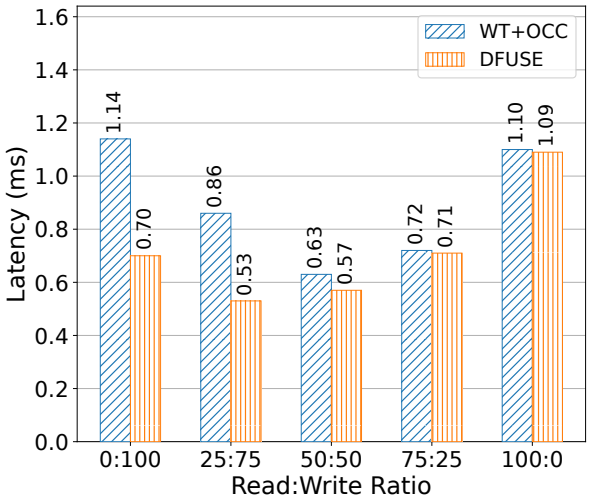
(a) fio Random I/O (Throughput)



(b) fio Sequential I/O (Throughput)



(c) fio Sequential I/O (Latency)



(d) fio Sequential I/O (Latency)

Figure 6: DFUSE's throughput and latency under different fio workloads without contention

Sequential I/O results (Figure 6b and 6d) show even more substantial improvements. DFUSE outperforms the baseline by 62.2% for write-only workloads (0:100) and 62.7% for mostly-write workloads (25:75). This larger improvement in sequential patterns can be attributed to DFUSE's efficient batching of operations and the tiered caching approach that reduces kernel-userspace transitions. Even for balanced (50:50) and read-dominant workloads (75:25), DFUSE maintains improvements of 9.9% and 2.2% respectively. The lack of improvement for pure read workloads (100:0) is expected, as both systems can effectively utilize the kernel's page cache for reads. The latency results in Figure 6d follow the same trend.

The application-level benchmarks (Figure 9) further validate DFUSE's performance advantages across realistic workloads. In the fileserver benchmark, which simulates a write-heavy file server workload, DFUSE achieves 26.2% higher throughput and 20.0% lower latency under no contention. The Netsfs benchmark shows substantial throughput gains of 18.5% (under no contention), too. The webserver benchmark, which is predominantly read-oriented, shows a decrease of 11.0% under no contention, suggesting some optimization opportunities for specific read-dominated patterns.

6.3 Contention Levels (Q2)

Definition. A critical aspect of distributed file systems is performance under varying levels of contention. We define the contention level as the percentage of shared files with other nodes in one's working set. We treat any concurrent accesses (read or write) on the same file as contention, because even multiple readers will also trigger the distributed consistency mechanism, i.e., acquiring a shared read lease.

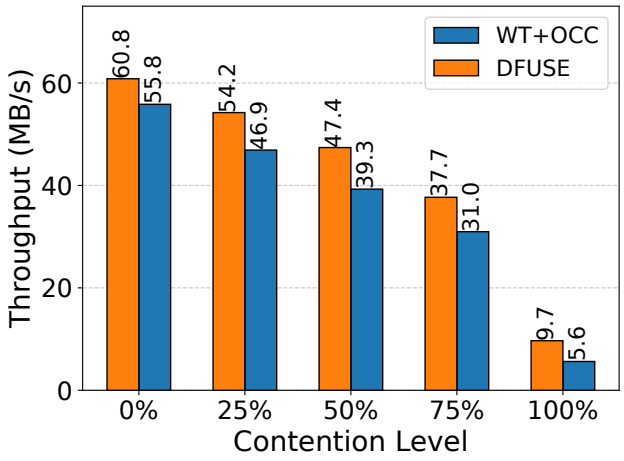


Figure 7: DFUSE under various contention levels (with a 50:50 random read-write fio workload)

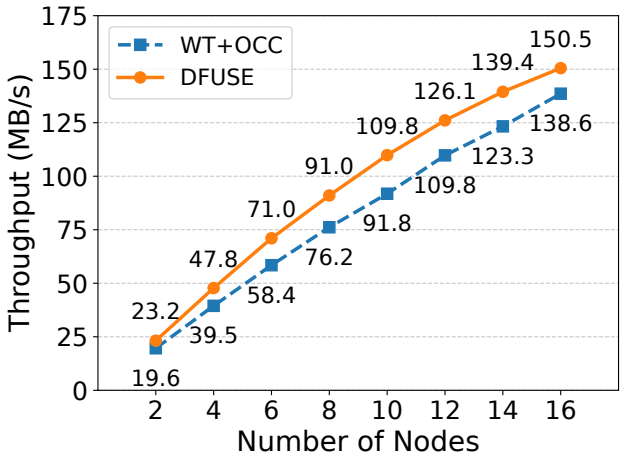


Figure 8: DFUSE scales with increasing number of nodes (with a contention level of 50% and a 50%-50% random read-write workload)

Setup. Following the above definition, each node divides its working set into shared and private files. All nodes in the cluster will operate on the shared files concurrently, whereas the private files are only accessed by one node. For example, a contention level of 25% in Figure 7 means that each node reads/writes 25 globally shared files and operates on 75 private files without other contenders (a working set of 100 files). We control the contention level by first creating all files on the storage nodes before the experiment starts, then specifying the working set of each node by file names during runtime.

Figure 7 illustrates that DFUSE constantly outperforms the baseline as contention increases from 0% to 100%. At low contention (0%), DFUSE delivers 60.8 MB/s throughput compared to the baseline's 55.8 MB/s, representing an 9.0% performance advantage. This benefit stems from DFUSE's efficient use of write-back caching without the overhead of frequent lease revocations. As contention increases to moderate levels (25-50%), both systems experience performance degradation, though DFUSE maintains its edge with 54.2 MB/s at 25% contention (15.6% higher than baseline's 46.9 MB/s) and 47.4 MB/s at 50% contention (20.6% higher than baseline's 39.3 MB/s). This trend demonstrates DFUSE's ability to effectively manage lease revocations while minimizing their impact on performance.

Under high contention (75-100%), the absolute performance of both systems decreases substantially, but DFUSE's relative advantage grows. At 75% contention, DFUSE delivers 37.7 MB/s versus the baseline's 31.0 MB/s (21.6% higher). At extreme contention (100%), DFUSE still achieves 9.7 MB/s compared to the baseline's 5.6 MB/s, representing a 73.2% advantage. This demonstrates that DFUSE's kernel-level lease management approach significantly reduces the overhead of maintaining strong consistency even under challenging workload conditions.

The application benchmarks in Figure 9 provide further insight into real-world performance under contention. For the Fileserver workload, DFUSE achieves 221.2 MB/s with no contention and maintains 171.4 MB/s under contention (a 22.5% reduction), whereas the baseline drops from 175.3 MB/s to 125.8 MB/s (a 28.2% reduction). For the read-heavy Netsfs and Webserver workloads, changing contention from 0% to 100% hardly affects either design because both applications are dominated by reads and thus trigger few lease revocations.

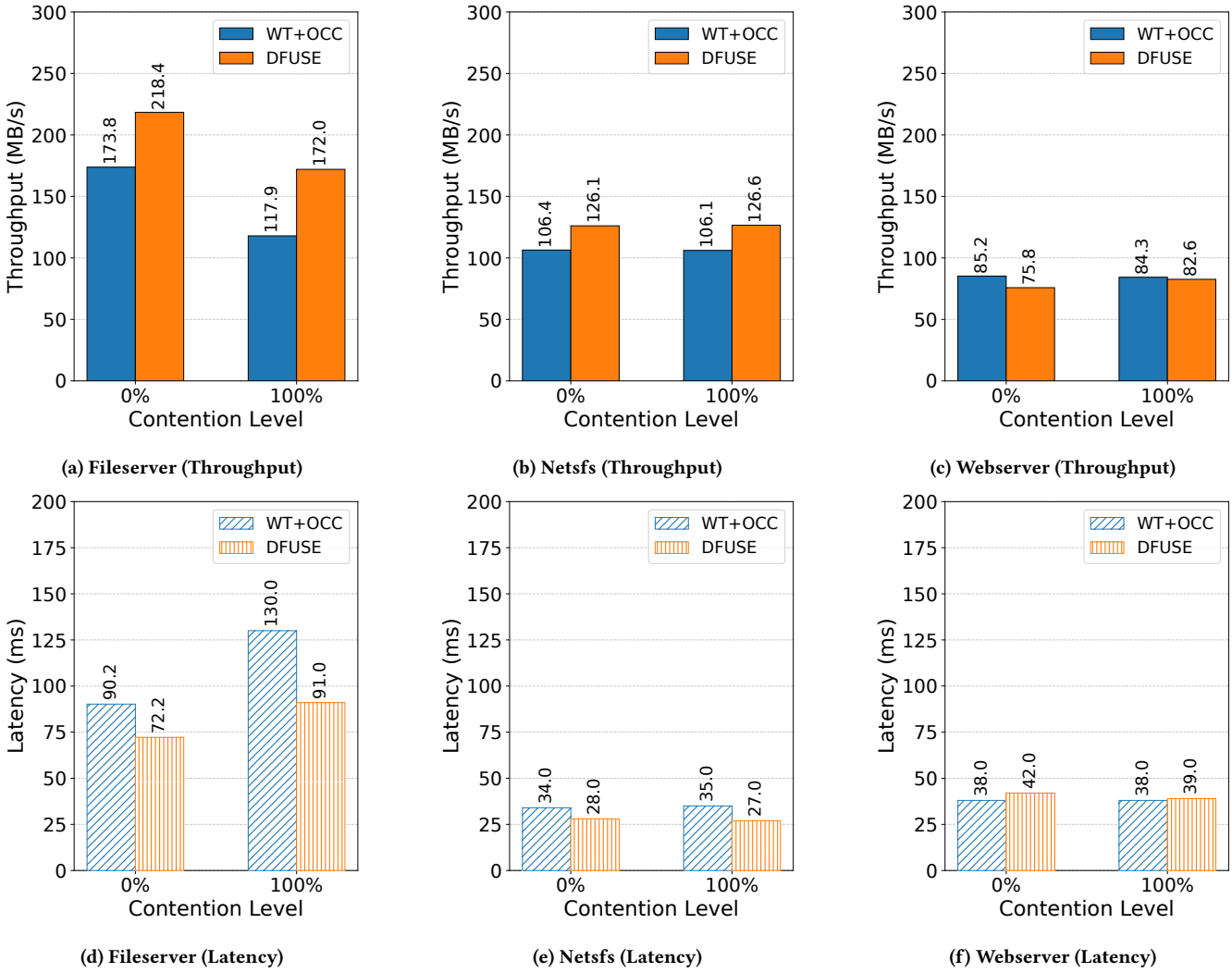
These results confirm that DFUSE's kernel offloading mechanism effectively balances strong consistency with high performance across varying contention levels. By embedding lease information directly in the kernel driver, DFUSE minimizes the performance impact of lease management and enables efficient write-back caching even in high-contention scenarios.

6.4 Scalability (Q3)

Figure 8 illustrates DFUSE's scalability as the number of nodes (i.e., the count of client machines) increases from 2 to 16. The workload is 50%-50% random read-write with fio, and we set the contention level to 50%. Both DFUSE and the baseline demonstrate near-linear scaling, with throughput increasing proportionally to the number of nodes, suggesting neither system suffers from significant coordination bottlenecks.

At smaller scales (2-4 nodes), DFUSE demonstrates a significant performance advantage over the baseline, achieving approximately 18-21% higher throughput. This early advantage highlights that DFUSE's kernel-level lease management approach delivers benefits even in modest deployments common in departmental or small enterprise environments.

As the deployment scales to medium size (6-10 nodes), DFUSE maintains a consistent performance lead of approximately 19-22% over the baseline. This steady advantage demonstrates that DFUSE's distributed coordination mechanisms scale efficiently without introducing significant overhead as the system grows. For instance, at 8 nodes, DFUSE achieves 91 MB/s versus the baseline's 76 MB/s, a performance difference that would be meaningful in production environments.



(a) Fileserver (Throughput) **(b) Netsfs (Throughput)** **(c) Webserver (Throughput)**

(d) Fileserver (Latency) **(e) Netsfs (Latency)** **(f) Webserver (Latency)**

Figure 9: DFUSE under various contention levels (filebench applications)

At larger scales (12-16 nodes), DFUSE continues to outperform the baseline, though the relative advantage gradually diminishes from 14.8% at 12 nodes to 8.6% at 16 nodes. This trend suggests that while DFUSE remains more efficient throughout the tested range, its coordination mechanisms may benefit from further optimization for very large deployments.

Importantly, DFUSE achieves this superior performance while maintaining strong consistency guarantees across all nodes and scales. The linear scaling pattern observed up to 16 nodes indicates that DFUSE's architecture successfully addresses the scalability requirements of modern cloud environments without compromising on consistency semantics.

These results validate DFUSE's design approach of offloading lease management to the kernel driver, demonstrating that it effectively balances performance, consistency, and scalability for distributed file system deployments of varying sizes.

7 Related Work

DFUSE builds upon and extends prior work in distributed file systems, FUSE optimizations, and consistency mechanisms in distributed file systems. This section contextualizes our contributions within the broader research landscape.

7.1 FUSE-based Distributed File Systems

Several distributed file systems leverage FUSE for userspace implementation flexibility. GlusterFS[12] employs a FUSE-based architecture that prioritizes scalability and performance, but sacrifices strong consistency when using write-back caching. CephFUSE[55] provides strong consistency guarantees but relies on write-through caching, resulting in higher latency for write operations. JuiceFS[13] optimizes cloud data access with caching mechanisms while maintaining POSIX compliance, but faces similar consistency-performance tradeoffs. S3FS[16] enables mounting

cloud object storage as POSIX file systems, though with limited performance due to the impedance mismatch between object and file semantics. DeepSeek's 3FS[9] focuses on AI workloads, prioritizing random read performance by disabling read caching and using Direct I/O with Linux-based AIO and io_uring interfaces. GCS FUSE[8] enables mounting cloud storage buckets as local directories with performance benefits through caching of frequently accessed files.

Unlike these systems, DFUSE uniquely bridges the consistency-performance gap by enabling write-back caching with strong consistency through kernel-level lease management, avoiding the traditional tradeoff between performance and correctness.

7.2 FUSE Performance Optimizations

Significant research has focused on mitigating FUSE's performance overheads. RFUSE[26] reduces kernel-userspace context switches by deploying ring buffer-based, per-core communication channels, substantially improving metadata operation efficiency. XFUSE[33] modernizes the userspace file system framework through scalable kernel-userspace communication patterns. Direct-FUSE[59] eliminates middleware components to provide high-performance FUSE file system support, particularly for HPC environments.

Several other optimization techniques have emerged to address FUSE's performance limitations. ExtFUSE[22] allows registering thin eBPF extensions for handling low-level file system requests inside the kernel, enabling fast-path execution without userspace transitions. Passthrough file system[11] functionality permits direct access to the underlying file system for specific operations, bypassing the userspace daemon entirely. Stackable file systems[17, 45] provide a layered approach that can cache or filter operations, reducing unnecessary kernel-userspace crossings.

The Linux kernel has incorporated various FUSE optimizations, including the ability to splice read and write operations that move data between kernel buffers without copying to userspace[1], significantly reducing memory copy overhead. Recent io_uring-based FUSE implementations[4] further improve performance by avoiding core switching and using shared memory for data transfers. These approaches collectively address the fundamental bottlenecks in FUSE: frequent context switches, memory copies, and lock contention.

These approaches primarily optimize local FUSE performance, whereas DFUSE addresses the fundamental distributed consistency challenge while leveraging these performance improvements. By offloading lease management to the kernel driver, DFUSE reduces the performance impact of maintaining strong consistency in distributed environments.

7.3 Consistency Mechanisms in Distributed File System

Distributed file systems employ various mechanisms to maintain consistency. Lease-based approaches, as pioneered by Gray and Cheriton[31], provide time-bound permissions that balance performance with consistency. GPFS (now Spectrum Scale)[48] implements distributed locking for strong consistency but requires specialized hardware for optimal performance. Lustre[49] employs a distributed lock manager to coordinate file access across nodes, though with different architectural tradeoffs than DFUSE.

DFUSE's innovation lies in embedding lease management directly in the kernel's FUSE driver while maintaining userspace flexibility. This approach differs fundamentally from previous systems that either implement consistency mechanisms entirely in userspace (sacrificing performance) or require extensive kernel modifications (sacrificing flexibility and isolation).

8 Conclusions

This paper presents DFUSE, a FUSE-based distributed file system that reconciles write-back kernel page caching with strong consistency for cloud environments. DFUSE eliminates the traditional trade-off between performance and correctness by offloading distributed lease management to the kernel driver, enabling coordinated access to cached data across nodes while retaining FUSE's userspace flexibility. Our experiments demonstrate DFUSE's efficacy: it achieves up to 68.0% higher throughput and 40.4% lower latency than write-through baselines for write-heavy workloads and near-linear scalability to 16 nodes while maintaining consistency under contention. Application benchmarks show 36.2% throughput improvements and 30.0% latency reduction in contended Fileserver workloads, validating its practicality for real-world deployments. By embedding lease coordination in the kernel, DFUSE demonstrates that distributed file systems can deliver both high performance and strong consistency, advancing the viability of FUSE-based solutions for cloud-scale workloads.

9 Acknowledgments

We thank the anonymous reviewers for their insightful feedback and helpful suggestions. This research was supported by NSF awards #2106530 and #2143868.

References

[1] 2010. [RFC PATCH] fuse: support splice() reading from fuse device - Miklos Szeredi. https://lore.kernel.org/lkml/E1OF3kc-00084X-Hi@pomaz-ex.szeredi.hu/
[2] 2011. Performance in a Gluster System. (2011).
[3] 2023. Use HDFS FUSE to connect to and use LindormDFS. https://www.alibabacloud.com/help/en/lindorm/use-hdfs-fuse-to-connect-to-and-use-lindormdfs
[4] 2024. fuse: fuse-over-io-uring. https://lwn.net/Articles/997400/
[5] 2024. Tune TiKV Memory Parameter Performance. https://docs.pingcap.com/tidb/stable/tune-tikv-memory-performance/
[6] 2025. Amazon S3 - Cloud Object Storage - AWS. https://aws.amazon.com/s3/
[7] 2025. Cloud Storage. https://cloud.google.com/storage
[8] 2025. Cloud Storage FUSE. https://cloud.google.com/storage/docs/cloud-storage-fuse/overview
[9] 2025. deepseek-ai/3FS. https://github.com/deepseek-ai/3FS
[10] 2025. filebench/filebench. https://github.com/filebench/filebench
[11] 2025. FUSE passthrough. https://source.android.com/docs/core/storage/fuse-passthrough
[12] 2025. Gluster. https://www.gluster.org/
[13] 2025. JuiceFS - Open Source Distributed POSIX File System for Cloud. https://juicefs.com/en/
[14] 2025. libfuse/libfuse. https://github.com/libfuse/libfuse
[15] 2025. minio/minfs. https://github.com/minio/minfs
[16] 2025. s3fs-fuse/s3fs-fuse. https://github.com/s3fs-fuse/s3fs-fuse
[17] 2025. sbu-fsl/fuse-stackfs. https://github.com/sbu-fsl/fuse-stackfs
[18] 2025. Shared S3 File System for EC2 instances and Servers | ObjectiveFS. https://objectivefs.com/
[19] Jens Axboe. 2022. Flexible I/O Tester. https://github.com/axboe/fio
[20] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter,

et al. 2020. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[21] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[22] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 121–134.

[23] Brad Calder, Ju Wang, Aaron Ogus, and et al. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 143–157. doi:10.1145/2043556.2043571

[24] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. 1994. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)* 26, 2 (1994), 145–185.

[25] Ka-Hing Cheung. 2025. kahing/goofys. https://github.com/kahing/goofys original-date: 2015-09-14T20:30:22Z.

[26] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. 2024. {RFUSE}: Modernizing Userspace Filesystem Framework through Scalable {Kernel-Userspace} Communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 141–157.

[27] James C. Corbett, Jeffrey Dean, Michael Epstein, and et al. 2012. Spanner: Google's Globally Distributed Database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. 251–264.

[28] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[29] facebook. 2023. Compaction. https://github.com/facebook/rocksdb/wiki/Compaction

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. 29–43. doi:10.1145/945445.945450

[31] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 202–210.

[32] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems* 6, 1 (1988), 51–81. doi:10.1145/35037.35059

[33] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. 2021. {XFUSE}: An infrastructure for running filesystem services in user space. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 863–875.

[34] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. (2020).

[35] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[36] Giorgos Kappes and Stergios V Anastasiadis. 2024. Faster FUSE Filesystems with Efficient Data Transfers. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*. 122–129.

[37] Nancy P Kronenberg, Henry M Levy, and William D Strecker. 1986. VAXcluster: A closely-coupled distributed system. *ACM Transactions on Computer Systems (TOCS)* 4, 2 (1986), 130–146.

[38] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226. doi:10.1145/319566.319567

[39] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[40] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. 2023. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 331–346.

[41] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. Citeseer, 21–33.

[42] Ajay Mohindra and Murthy Devarakonda. 1994. Distributed token management in calypso file system. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*. IEEE, 290–297.

[43] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[44] Aditya Rajgarhia and Ashish Gehani. 2010. Performance and Extension of User Space File Systems. In *Proc. ACM SAC*.

[45] Kai Ren and Garth Gibson. 2013. {TABLEFS}: Enhancing Metadata {Efficiency} in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 145–156.

[46] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[47] Stephen M. Rumble, Navneeth R. Ganesan, John Ousterhout, and et al. 2012. RAMCloud: Scalable High-Performance Storage Using Random Access Memory. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. 99–112.

[48] Frank Schmuck and Roger Haskin. 2002. {GPFS}: A {Shared-Disk} file system for large computing clusters. In *Conference on file and storage technologies (FAST 02)*.

[49] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. 380–386.

[50] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. 2003. Network File System (NFS) version 4 Protocol. Request for Comments 3530. doi:10.17487/RFC3530

[51] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. doi:10.1109/MSST.2010.5496972

[52] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system.. In *USENIX Annual Technical Conference*, Vol. 15. 2.

[53] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. 2019. Performance and resource utilization of fuse user-space file systems. *ACM Transactions on Storage (TOS)* 15, 2 (2019), 1–49.

[54] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 59–72.

[55] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*. 307–320.

[56] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[57] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling transaction priority in optimistic concurrency control. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

[58] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515.

[59] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. 2018. Direct-fuse: Removing the middleman for high-performance fuse file system support. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*. 1–8.