ENGINEERING Information Technology & Electronic Engineering
https://www.academax.com/EITEE; https://www.engineering.org.cn
https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=11344065
E-mail: eng.itee@zju.edu.cn

Engineering Journals
Available on IEEE Xplore®

Research Article
https://doi.org/10.1631/ENG.ITEE.2026.0080

SplitFUSE: a highly compatible framework for I/O acceleration of user-level file systems

Yushuqing ZHANG, Kai LU, Wenzhe ZHANG, Ruibo WANG, Huijun WU, Zhenwei WU✉

College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China

Abstract: User-level file systems are widely adopted in research and production environments due to their flexibility and reduced risk of kernel crashes. Filesystem in Userspace (FUSE) is a general-purpose framework for developing user-level file systems in Linux. Compared with library-based file systems, FUSE adopts a cooperative architecture between kernel modules and userspace libraries, ensuring metadata security and compliance with standard portable operating system interface (POSIX) semantics. However, this architecture introduces substantial context-switching overhead. Existing optimization approaches for high-performance computing environments often improve input/output (I/O) performance at the expense of FUSE's cross-environment compatibility and kernel-level security guarantees. To address this limitation, this study proposes SplitFUSE, an I/O-acceleration framework for user-level file systems that preserves high compatibility and strong security. SplitFUSE introduces a split architecture that decouples metadata and data-request processing. Specifically, the kernel maintains full metadata consistency, while the userspace retains only a minimal metadata subset required to validate and process data requests that bypass the kernel securely. Implemented as a self-contained mechanism, SplitFUSE preserves the same cross-environment compatibility as conventional FUSE. Experimental results demonstrate that, under I/O-intensive small-write workloads, SplitFUSE achieves up to 4–6 times higher write bandwidth than native FUSE and outperforms state-of-the-art alternatives. For common file system workloads, it delivers substantial performance improvements with minimal migration overhead.

Key words: Filesystem in Userspace (FUSE); User-level file system; Input/Output (I/O) acceleration; Kernel bypass; Split architecture

1 Introduction

User-level file systems effectively address the limitations of kernel-level implementations by providing enhanced security, development flexibility, and ease of deployment, which has led to their widespread adoption in academic and industrial environments (Cornell et al., 2004; Ungureanu et al., 2010; Mashtizadeh et al., 2013; Ren and Gibson, 2013; Trapexit, 2014; Lawrence Livermore National Laboratory, 2015; Sigurbjarnarson et al., 2016; Pontes et al., 2017). However, existing user-level file systems still face substantial challenges in balancing data access efficiency, cross-platform compatibility, and kernel-grade metadata security.

Library-based file systems (Weil et al., 2006; Red Hat, 2019; Zhang et al., 2022; Jia et al., 2025) are often optimized for specific input/output (I/O) workloads and can achieve high-performance data access. However, they typically sacrifice the metadata security guarantees and generality provided by kernel-level mechanisms. In contrast, Filesystem in Userspace (FUSE), a predominant user-level file system framework in Linux, adopts a cooperative architecture between a kernel module and a userspace library to support diverse platform requirements. The kernel module interfaces with the virtual file system (VFS) to ensure portable operating system interface (POSIX) semantics while enforcing security validation and metadata integrity. In contrast, userspace components implement backend file system logic via a simple and flexible application programming interface (API).

Although this architecture ensures strong

✉ Zhenwei WU, zhenweiwu@nudt.edu.cn
Yushuqing ZHANG, https://orcid.org/0009-0006-9751-0497
Kai LU, https://orcid.org/0000-0003-2284-7897
Wenzhe ZHANG, https://orcid.org/0009-0003-0065-5269
Ruibo WANG, https://orcid.org/0000-0001-7952-3784
Huijun WU, https://orcid.org/0000-0002-9513-5359
Zhenwei WU, https://orcid.org/0000-0002-2005-2831
CLC number: TP316

Received: Mar. 29, 2026; Revision accepted: Apr. 29, 2026;
Crosschecked: May 13, 2026; Published online: June 4, 2026

cross-environment compatibility, it introduces substantial performance overhead due to frequent context switching and redundant data copying. Existing optimization efforts can be broadly classified into three categories. The first category improves kernel–user communication efficiency using techniques such as shared memory or extended Berkeley packet filter (eBPF) (Bijlani and Ramachandran, 2019; Yan et al., 2019; Huai et al., 2021; Cho et al., 2024). While these approaches reduce communication overhead, they either retain context-switching costs or depend on specific kernel features. The second category bypasses the kernel module entirely through userspace interception or API redirection (Zhu et al., 2018; Deng et al., 2022). However, this shifts responsibility for security enforcement onto developers and often tightly couples the implementation to specific backend APIs. The third category separates data and metadata processing paths (Lembke et al., 2022; An et al., 2024) to balance efficiency and security. Nevertheless, these approaches typically require kernel modifications or tight integration with specific backend systems, leading to high migration costs and limited cross-scenario applicability. Despite these advances, no existing solution simultaneously achieves high I/O efficiency, kernel-level security, or cross-scenario compatibility.

We observe that the correct execution of data requests depends only on a small subset of critical metadata, e.g., node identifiers (nodeids) and file descriptors (fds), rather than the full kernel-managed metadata context. This observation leads to two key insights. First, this essential metadata is already transferred to the userspace during metadata operations. By caching and reusing it for subsequent data requests, security validation and I/O processing can be performed entirely in the userspace without kernel modifications or additional drivers. Second, because only a minimal metadata slice is required, rather than a full metadata replica, the associated management overhead remains negligible and does not become a performance bottleneck.

Motivated by these insights, this study presents SplitFUSE, a highly compatible I/O-acceleration framework for FUSE-based file systems. By caching the security-relevant fields embedded in metadata requests, SplitFUSE enables subsequent data requests to bypass the kernel while still performing secure validation and I/O operations in the userspace. This design preserves compatibility with the traditional FUSE framework while maintaining minimal dependence on environmental configurations, such as the kernel version or privileged execution modes. Extensive evaluation demonstrates that SplitFUSE achieves up to 4–6 times higher write bandwidth than native FUSE under I/O-intensive small-write workloads, outperforming state-of-the-art alternatives. For typical file system workloads, it improves performance up to 3.10× when applied to existing FUSE-based backend systems.

The main contributions of this paper are summarized as follows:

1. Comprehensive analysis of existing FUSE optimization strategies. We systematically analyze the trade-offs among deployment convenience, backend generality, and performance overhead in existing FUSE optimization approaches, and identify the critical gap in simultaneously achieving high I/O efficiency, kernel-level security, and broad cross-environment compatibility.

2. Design of the SplitFUSE framework. We propose SplitFUSE, a self-contained I/O-acceleration framework that decouples metadata and data-request processing. By caching minimal metadata slices, SplitFUSE enables secure validation and efficient I/O execution entirely in the userspace, while incurring minimal migration overhead.

3. Implementation and extensive evaluation. We implement a fully functional SplitFUSE prototype and conduct comprehensive experiments to validate its compatibility and performance benefits. Additionally, we provide practical guidelines for configuring parameters to optimize performance across diverse workload scenarios.

2 Background and motivation

Fig. 1 illustrates three primary categories of optimization approaches, where the shaded regions indicate the system components that require modification for each method.

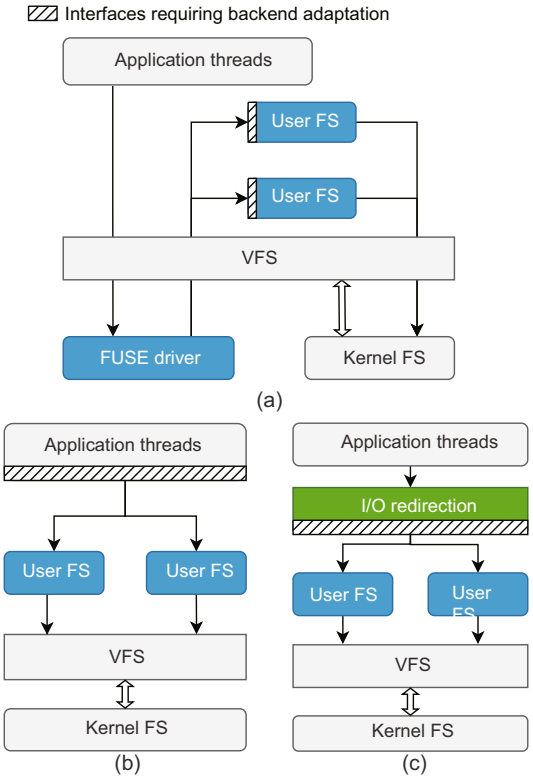

Fig. 1 Three request-path optimization approaches: (a) type I; (b) type II; (c) type III. FS: file system

Type I approaches retain the FUSE kernel module and reduce overhead by improving communication efficiency between kernel space and userspace. For example, RFUSE (Cho et al., 2024) replaces traditional pipe-based request forwarding with shared memory to minimize data copying. However, the allocation and management of shared memory introduce additional resource overhead, and the kernel module remains involved in request forwarding, preventing the complete elimination of context switches. ExtFUSE (Bijlani and Ramachandran, 2019) leverages eBPF to offload simple metadata operations (e.g., getattr) to the kernel, thereby avoiding userspace

wake-ups. Nevertheless, eBPF programs must be customized for each backend file system and require kernel version 4.4 or higher with BPF support, which limits portability and deployment flexibility. DEFUSE (Lembke et al., 2022) adopts a different strategy by caching mappings between fds and inodes in the userspace, enabling subsequent data requests to bypass the kernel module and access the backend directly using cached information. Although DEFUSE reduces kernel involvement by catching fd–inode mappings in our space, it relies on a kernel driver to intercept and redirect requests. This requirement introduces the need for kernel module loading and privileged permissions, which limits its applicability in unprivileged containerized or cloud-native environments.

Type II approaches implement file system functionality as a userspace library, with applications directly linking against it to invoke backend operations. Representative examples include the high-level FUSE API (Zhang et al., 2022) and library-based file systems such as TABLEFS (Ren and Gibson, 2013), which are designed for high-performance computing environments. The primary advantage of this approach is the complete elimination of kernel involvement, enabling near-optimal I/O performance. However, this comes at a substantial cost: Applications must be modified to use non-POSIX APIs, existing binaries cannot be executed transparently, and full responsibility for enforcing security checks is shifted to developers, thereby introducing substantial risks.

Type III approaches employ interception mechanisms, such as LD_PRELOAD or libsysio (Zhu et al., 2018; An et al., 2024), to intercept POSIX system calls in the userspace and redirect them to the backend file system. While this method is transparent to applications, it transfers all security responsibilities, such as permission enforcement and metadata consistency, from the kernel to userspace implementations, increasing the risk of vulnerabilities. Furthermore, FUSE provides low-level and high-level APIs, and different backend file systems may rely on different interfaces. Consequently, a single interception layer cannot generalize across all implementations and typically requires backend-specific adaptations. libplfs (Deng et al., 2022) introduces a partial kernel-bypass approach that maintains mappings from file paths to fds in the userspace. For I/O requests that match predefined paths, it uses cached fds directly to access the backend. However, this design is tightly coupled to the parallel log-structured file system (PLFS) architecture and assumes prior knowledge of backend APIs, limiting its applicability to other FUSE-based systems.

To date, no existing approach simultaneously satisfies the four critical requirements of compatibility (no modifications to applications or backend systems), security (preservation of kernel-level enforcement), deployability (no need for kernel modifications or privileged access), and performance improvement. SplitFUSE addresses this gap by introducing a general-purpose acceleration framework that substantially enhances I/O performance without modifying the kernel, altering existing FUSE-based file systems, or compromising security guarantees. The remainder of this paper presents the design and implementation of SplitFUSE, followed by comprehensive experimental evaluations against state-of-the-art approaches, including ExtFUSE and RFUSE, to demonstrate its effectiveness.

3 Design

3.1 Architecture

As illustrated in Fig. 2, the SplitFUSE architecture consists of three core components: an I/O redirection layer, a metadata cache, and a data-request queue.

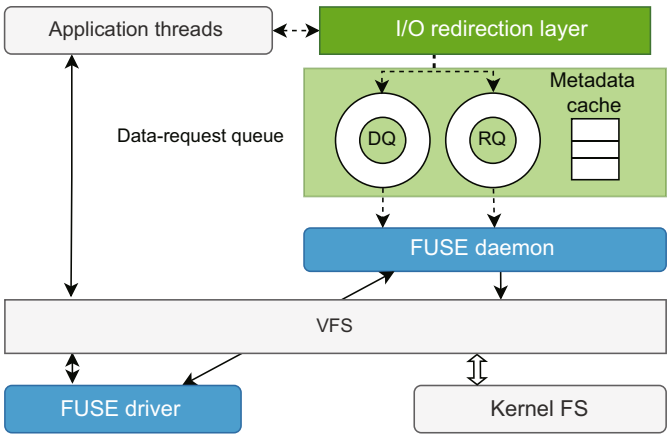

Fig. 2 SplitFUSE architecture

The I/O redirection layer operates within the private address space of each application and is responsible for intercepting and routing file operation requests, as well as maintaining metadata mappings. In Fig. 2, dashed lines indicate the execution paths introduced by SplitFUSE for redirecting data requests and performing metadata mapping, whereas solid lines represent the original execution flow of FUSE. The metadata mapping represents a reduced metadata slice derived from the complete FUSE metadata space. It contains only the minimal set of metadata attributes and security-related fields required to validate and execute I/O operations safely. By limiting the scope of maintained metadata, SplitFUSE minimizes management overhead while preserving correctness and security guarantees.

The metadata cache is implemented as a key-value store in shared memory, accessible to the application and the FUSE daemon. During metadata request processing, SplitFUSE extracts and caches critical security-related fields, such as thread ID (tid), nodeid, and fd, from kernel responses. This mechanism enables the establishment and reuse of application-to-daemon metadata mappings entirely within the userspace, thereby eliminating repeated kernel interactions for subsequent data requests.

The data-request queue consists of paired request queues (RQs) and data queues (DQs), which have a one-to-one correspondence. The RQ stores control information for each request, while the DQ holds the associated data payload. SplitFUSE provides configurable parameters, including the number of queue pairs (RING_NUM), the capacity of each queue (REQ_NUM), and the data size per request (PAGE_SIZE). These parameters allow developers to tune the degree of parallelism and optimize request processing performance in the userspace according to workload characteristics.

3.2 Metadata mapping

3.2.1 Challenge

The metadata mapping is implemented as a key-value table within the application's private address space, where the fd obtained from create and open operations serves as the key. Each entry stores the corresponding values returned by the FUSE daemon, including fh, nodeid, and file offset (pos). Establishing this mapping presents a fundamental challenge. The fd allocation and management are handled entirely within the VFS layer and are not transmitted to the userspace via FUSE requests. As a result, constructing a consistent and reliable mapping in the userspace is inherently difficult without introducing kernel modifications.

Fig. 3 illustrates this challenge using the open operation as an example. Using open() as an example, when an application invokes open(), the VFS first allocates an fd and then forwards the request to the FUSE daemon via the kernel module. The FUSE kernel module establishes metadata mapping only after receiving the fh from the daemon. Subsequent data requests rely on fd-to-fh translation performed within the kernel. Crucially, the application receives the fd only after the request has completed, at which point the original request context has already been discarded. As a result, the application cannot access the metadata information carried during the request, making it difficult to construct a consistent metadata mapping in the userspace.

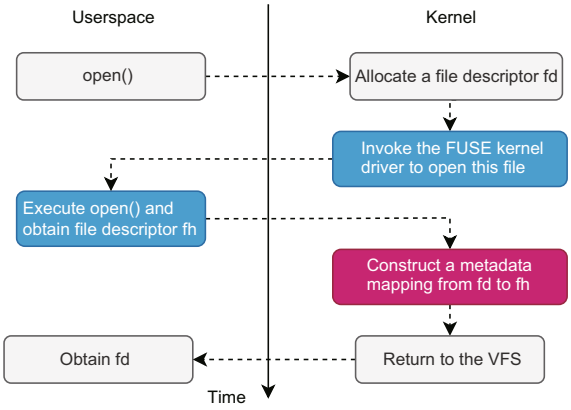

Fig. 3 fd allocation workflow

In existing approaches, DEFUSE employs a custom kernel driver to manage fds and exposes them to the userspace through specialized APIs. libplfs adopts a different strategy: It creates temporary virtual files to obtain fds, which are then mapped to PLFS structures. Although this approach is kernel-agnostic, it requires explicit exposure of backend APIs and overrides 69 standard libc I/O functions, introducing substantial implementation complexity and maintenance overhead. To avoid modifying either the kernel or the backend file system, SplitFUSE adopts a different design philosophy. Instead of forcing the FUSE daemon to establish metadata mappings during the execution of create and open operations, SplitFUSE defers mapping until the metadata operation completes, leveraging the metadata cache to reconstruct the required associations.

3.2.2 Establishing the mapping

As illustrated in Fig. 4, the application tid is propagated along with metadata requests to the userspace. This tid naturally serves as a unique key for the metadata cache and is accessible to the application and the FUSE daemon.

The I/O redirection layer intercepts create and open operations via LD_PRELOAD. After obtaining the application's tid using the gettid system call, the layer invokes openat to forward the request to the VFS. When the request reaches the FUSE daemon, SplitFUSE extracts and stores the tid, nodeid, and backend-returned fh (fh denotes an fd for the daemon) in a metadata cache entry. Once the openat returns with the allocated fd, the I/O redirection layer queries the metadata cache using the tid to retrieve the corresponding values. It then constructs a new metadata mapping entry keyed by fd, containing fh, nodeid, and an initialized pos (set to 0).

The returned fd allows applications to perform standard file operations transparently, without requiring additional library modifications. SplitFUSE tracks redirected operations by recording these descriptors in an internal fd table with 1024 entries, consistent with the typical per-process fd limit. Metadata mapping entries are removed upon close operations to ensure consistency and prevent stale mappings.

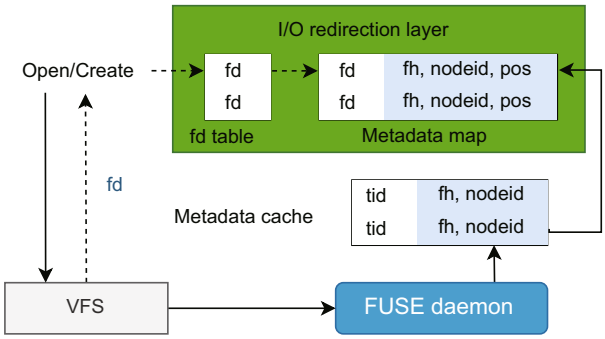

Fig. 4 Metadata mapping

3.3 Data request

SplitFUSE adopts a self-contained design that confines data-request validation and redirection within the FUSE framework, enabling seamless integration with applications and backend file systems without requiring code modifications.

As illustrated in Fig. 5, the I/O redirection layer first queries the metadata mapping table using the fd provided by the application. If a valid mapping is found, the required parameters and data associated with the request are retrieved from the mapping entry and placed into the RQ and DQ in shared memory. Otherwise, the request is redirected to the native FUSE execution path.

SplitFUSE manages queue slots using a global state table—a SLOT_EMPTY indicates that it is available for allocation by incoming application requests. SLOT_PENDING denotes a slot containing a request that is awaiting processing, while SLOT_GET indicates that a worker thread has already acquired the request in that slot, prompting other worker threads to skip it during traversal to avoid duplicate processing in concurrent environments.

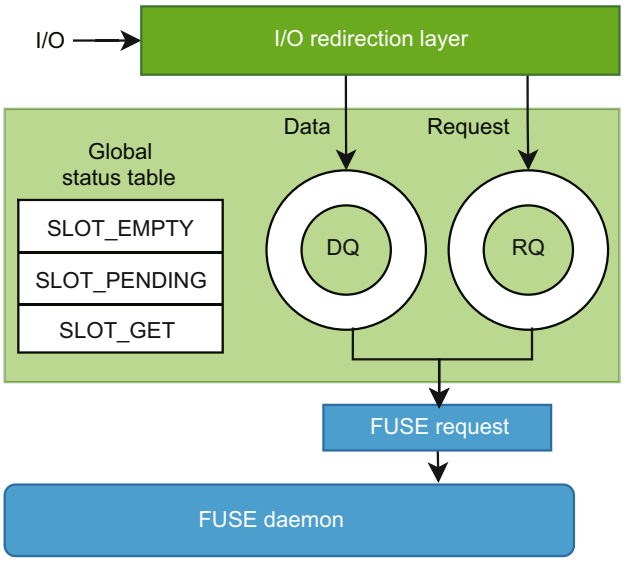

Fig. 5 Data-request execution

3.3.1 Security

Offloading data requests processing to the userspace improves performance but introduces potential security risks. This subsection analyzes three primary attack surfaces in SplitFUSE and presents the corresponding defense mechanisms and security boundaries.

1. Bypassing file permission enforcement. SplitFUSE delegates all metadata operations to the kernel, thereby inheriting the permission verification mechanisms of the VFS. Specifically, the VFS validates file access modes during open operations, and the fh returned by the underlying file system implicitly encodes the permissions associated with that access mode. The userspace metadata mapping is used solely for fast-path redirection of data requests and does not maintain independent permission states. When an application issues a write request, the backend file system enforces access control based on the open mode stored in the kernel's struct file data structure associated with the fh. Consequently, SplitFUSE provides file permission enforcement equivalent to that of native FUSE.

2. The fd reuse. The kernel allocates fds using a lowest available number policy, meaning that an fd may be reassigned after a file is closed. To prevent stale metadata mappings from redirecting I/O requests to incorrect targets, SplitFUSE strictly bounds the lifetime of cache entries. Each mapping exists only while the corresponding file remains open and is immediately removed upon a close operation. For a given thread, close and subsequent open operations occur sequentially. Therefore, before a new (tid, fh, nodeid) tuple can be inserted into the metadata cache, any previous entry associated with the same fd has already been removed. This design effectively eliminates incorrect redirection caused by fd reuse.

3. Malicious tampering of the userspace memory. If an attacker gains the ability to modify a process's userspace memory through techniques such as code injection, malicious shared libraries, or exploiting in-process vulnerabilities, the metadata cache or DQ may be corrupted. If tampering is limited to metadata mapping entries, SplitFUSE can safely fall back to the kernel execution path. In such cases, the kernel revalidates permissions and ensures correct data access when inconsistencies are detected (e.g., mismatches between fd and mapping entries). However, if an attacker constructs a syntactically valid but malicious request, the fast path in SplitFUSE may forward the request to the FUSE daemon without additional validation.

It is important to note that native FUSE also cannot fully defend against userspace memory tampering. A malicious process can, for example, modify the fd argument passed to system calls such as write. The key difference lies in the scope of impact. In native FUSE, each system call is independently validated by the kernel, limiting the effect of a malicious request to that specific call without affecting other processes or the daemon. In contrast, SplitFUSE introduces shared-memory communication between the application and daemon; thus, corrupted requests may propagate to the daemon, potentially causing state inconsistencies, improper resource handling, or even process crashes, and indirectly affecting other clients. Therefore, SplitFUSE exhibits weaker robustness than native FUSE under this specific attack model.

However, even in the event of a successful attack, the impact remains confined to the userspace file system service process. It does not compromise kernel stability or cause permanent data corruption. The FUSE daemon can be safely restarted to restore service functionality. This security boundary is consistent with the design philosophy of FUSE: Failures or malicious behavior in userspace components, whether in the FUSE daemon or SplitFUSE redirection layer, must not compromise kernel integrity.

3.3.2 Compatibility

FUSE provides high-level and low-level APIs for developers. However, all requests are structured uniformly as they pass through the kernel module. SplitFUSE exploits this abstraction to achieve broad compatibility. Specifically, the daemon retrieves request parameters from the shared-memory queue, reconstructs them into standard FUSE request structures, and invokes the backend file system through the unified FUSE interface. As a result, any backend file system that adheres to FUSE specifications can be seamlessly integrated with SplitFUSE without requiring code modifications.

This compatibility-oriented design, however, constrains the types of requests that can be efficiently accelerated in the userspace. In native FUSE, write operations involve two data copies (application→kernel→daemon). SplitFUSE reduces this overhead to a single copy (application→shared-memory queue), thereby achieving substantial performance improvements. In contrast, fully handling read operations in the userspace would still require two data copies (daemon→shared-memory queue→application) and would bypass the benefits of the kernel page cache and prefetching mechanisms. Given that the kernel read path is already highly optimized, SplitFUSE preserves the traditional kernel-based execution path for read operations and focuses on accelerating write requests. This design choice enables SplitFUSE to leverage kernel caching for read-intensive workloads while delivering substantial performance gains in scenarios dominated by small-write operations.

4 Implementation

SplitFUSE is implemented on libfuse2 and libfuse3 to ensure compatibility with a wide range of existing FUSE-based file systems. The framework is packaged as a dynamic library loaded at runtime via LD_PRELOAD, enabling transparent integration without requiring modifications to applications.

4.1 Data structures

SplitFUSE defines the request structure shm_req and the RQ structure ring_queue to support efficient shared-memory communication (Fig. 6). The queue implementation places three control parameters in the header of the shared-memory segment, followed by a preallocated array of request slots. This design minimizes allocation overhead and enables efficient, lock-free, or low-contention access patterns for high-throughput request processing.

```
struct shm_req {
 int fh;          // file descriptors kept by daemon
 u8 opcode;
 u32 flag;        // REQ_READY / REPLY_READY
 size_t size;
 size_t offset;
 size_t res;
 u64 nodeid;
 int index;       // index of the first slot
 int num;         // the total number of slots
}

struct ring_queue {
  int size;       // queue length
  u32 idx;        // index of the last allocated slot
  volatile u32 slots [REQ_NUM];  // global status table
}
```

Fig. 6 Data structure

4.2 Request transmission

SplitFUSE implements lock-free slot allocation by atomically updating slot states from SLOT_EMPTY to SLOT_PENDING using compare-and-swap (CAS) operations. This mechanism guarantees exclusive ownership of slots in concurrent environments. However, under high contention, repeated CAS retries may introduce non-negligible overhead. To mitigate this issue, SplitFUSE employs a batch allocation strategy that allocates contiguous slots (Algorithm 1).

The system first determines the number of required slots (num) based on the write size and PAGE_SIZE. Starting from the last allocated slot index, it computes the allocation boundary and wraps to the beginning of the queue if necessary (lines 2–4). The algorithm then scans the candidate range to locate the first available slot using CAS operations. Once a slot is successfully acquired, its index is designated as the starting point, and the first slot is reserved (lines 5–15). If no available slot is found, the process retries (lines 16–18). For multi-slot requests, subsequent CAS operations acquire adjacent slots (lines 19–23), incrementally increasing the number of allocated slots until the required num slots are obtained or

Algorithm 1 Contiguous allocation of shared-memory requests

Require: num
Ensure: req ← the first slot successfully allocated
1: restart:
2: start ← ring->idx atomic increase num // -> denotes accessing a // member of a structure (or object) via a pointer
3: idx_start ←start mod REQ_NUM
4: idx_end ← idx_start + num < REQ_NUM ? idx_start + num − 1: REQ_NUM − 1
5: **for** i from idx_start to idx_end **do**
6: set ring->slots[i] to SLOT_PENDING
7: **if** failed **then**
8: continue
9: **else**
10: req ← the pointer of slots[i] in the queue
11: req->index ← i
12: req->num ← 1
13: break
14: **end if**
15: **end for**
16: **if** $i >$ idx_end **then**
17: goto restart
18: **end if**
19: **if** idx_start ! = i **then**
20: idx_start ← i
21: **end if**
22: **for** i from idx_start + 1 to idx_end **do**
23: set ring->slots[i] to SLOT_PENDING
24: **if** success **then**
25: req->num ← req->num + 1
26: **else**
27: **return** req
28: **end if**
29: **end for**
30: **return** req

contention is encountered (lines 24–28). This batch allocation approach amortizes synchronization overhead while preserving lock-free progress guarantees.

SplitFUSE further adopts a relaxed allocation strategy that allows immediate acquisition of any available contiguous slots, even if <num slots are available, so that processing can begin without delay. When the system cannot allocate all required slots in a single attempt, it first writes the portion of data corresponding to the N acquired slots. After releasing these slots, it continues allocating the remaining (num−N) slots until the entire data transfer is completed.

For result retrieval, SplitFUSE adopts a hybrid strategy that balances responsiveness and efficiency. While polling is efficient for high-frequency and small I/O operations, production-grade user-level file systems often experience higher latency due to data processing or network delays. Prolonged polling under such conditions can lead to substantial central processing unit (CPU) contention. To address this issue, after placing a request into the shared-memory queue, application threads' activity polls for a short duration (10 μs, which is sufficient for lightweight write operations based on our empirical evaluation). If no response is received within this interval, the thread invokes a futex system call to enter a sleep state, thereby reducing CPU overhead while preserving responsiveness.

4.3 Request receiving and processing

In native FUSE, the daemon reads from the kernel request buffer in a blocking manner: Requests are processed immediately upon arrival; otherwise, the thread remains

blocked. In contrast, SplitFUSE adopts a conditional blocking scheduling policy to balance responsiveness and throughput. During initialization, the daemon configures the kernel request buffer in non-blocking mode. Worker threads first attempt to retrieve requests from the kernel buffer; if none are available, they immediately switch to processing data requests from the shared-memory queue.

Furthermore, each worker thread is assigned a monotonically increasing identifier upon creation. When this identifier exceeds a predefined threshold (default: 8), the thread switches the kernel buffer to blocking mode after scanning the shared-memory queue, waiting for incoming kernel requests. Once any worker thread successfully retrieves a request from the kernel buffer, it restores the buffer to non-blocking mode, allowing all threads to resume active polling. This adaptive scheduling mechanism ensures timely handling of metadata requests, prevents request accumulation in the kernel buffer, and maximizes parallel processing of data requests, thereby improving overall system throughput.

To prevent starvation of data requests under high concurrency, particularly when multiple threads are blocked on kernel requests, SplitFUSE introduces a dedicated I/O worker thread. This thread exclusively processes requests from the shared-memory queue, ensuring continuous progress along the data path and maintaining system responsiveness.

5 Evaluation

5.1 Experimental setup and methodology

All experiments are conducted on a system running Linux kernel version 5.2.2, equipped with 32 GB of memory, 200 GB of solid-state drive (SSD) storage, and a 16-core processor. The CPU is an Intel® Core™ i7 (13th generation), operating at a base frequency of 2.2 GHz with a maximum turbo frequency of 5.0 GHz. We evaluate the performance of the SplitFUSE against several representative FUSE optimization approaches under diverse workload scenarios. Table 1 summarizes the FUSE optimization techniques compared in Sections 5.3 and 5.4.

The evaluated systems include SplitFUSE, native FUSE, ExtFUSE, and RFUSE. ExtFUSE leverages eBPF to register extension programs within the kernel, enabling selected metadata operations to be executed directly in the kernel space.

Table 1 FUSE optimization options

Option	Description
direct_io	Bypasses the kernel page cache and forwards all I/O requests directly to the userspace daemon
writeback_cache	Enables writeback caching to coalesce small writes into larger requests
big_writes	Increases the maximum I/O size to 128 KB for more efficient data transfer
splice	Enables data copying between two kernel buffers, avoiding copying to the userspace
kernel_cache	Disables kernel caching to prevent flushing of file content each time a file is opened

RFUSE adopts a shared-memory design that binds a dedicated request queue to each CPU core, allowing request transmission without context switching.

5.2 Parameter sensitivity

We evaluate the sensitivity of SplitFUSE to key configuration parameters through stress testing. The workload consists of creating 1000 files of 1 MB each, with the following operation sequence: open, append (16 KB), and close. Throughput is measured in operations per second (operation/s) over a 20-s interval. For PAGE_SIZE evaluation, the thread count is fixed at 16, and the I/O size varies from 1 to 1024 KB. For RING_NUM and REQ_NUM evaluation, the I/O size is fixed at 16 KB, and the thread count ranges from 1 to 128.

Fig. 7a shows that for a fixed I/O size, throughput varies minimally across different PAGE_SIZE values. As the I/O size increases, an excessively small PAGE_SIZE may lead to RQ saturation under high concurrency, affecting execution stability. Overall, PAGE_SIZE primarily influences request buffering capacity and has a limited impact on aggregate throughput.

Fig. 7b shows that at low concurrency, fewer shared-memory queues yield higher throughput due to reduced scheduling overhead. With 16 threads, RING_NUM = 1 achieves the best performance. As concurrency increases, additional queues enable more effective parallelism. At 32 threads, RING_NUM = 2 outperforms RING_NUM = 1 by 7.0%. Throughput begins to converge at approximately 90 threads, and at 128 threads, RING_NUM = 8 achieves the highest throughput.

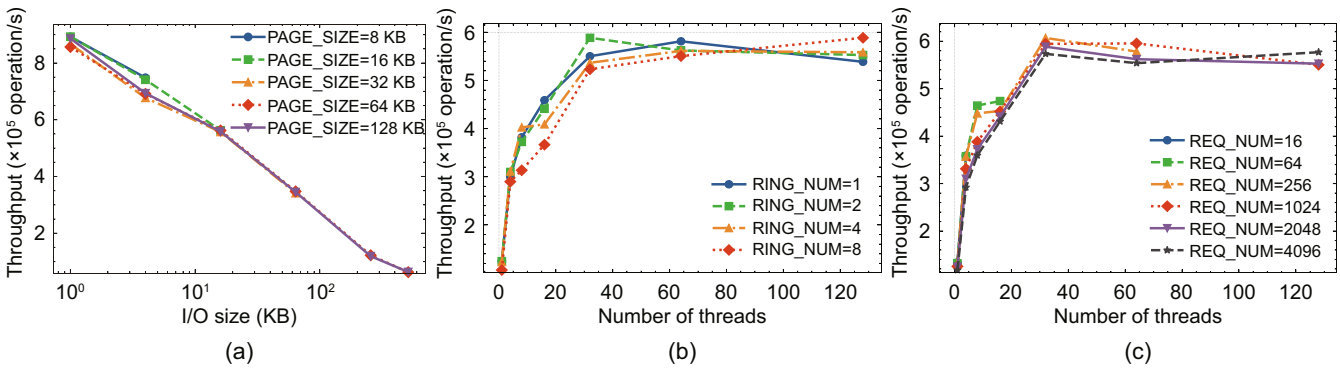

Fig. 7 Throughput under different parameter settings and workload configurations: (a) page size; (b) ring count; (c) queue depth. Note that the x-axis values in (a) have been log-scaled

Fig. 7c presents throughput variations under different REQ_NUM configurations as the thread count increases. Queue depth directly affects buffering capacity: Shallow queues are prone to saturation, whereas excessively deep queues introduce memory overhead and increased traversal latency. At low concurrency, smaller REQ_NUM values yield better performance. As concurrency increases, larger REQ_NUM values become advantageous. Specifically, when the thread count exceeds 16, configurations with REQ_NUM less than 256 exhibit instability; beyond 64 threads, configurations with REQ_NUM less than 1024 fail to sustain stable execution. Throughput stabilizes around 100 threads, beyond which REQ_NUM = 4096 achieves the highest performance.

Based on these observations, we adopt a fixed configuration for subsequent experiments: RING_NUM = 2 and REQ_NUM =2048. This configuration provides robust performance across a wide range of thread counts while remaining close to the optimal. PAGE_SIZE is retained at the default FUSE value, as its impact on performance is minimal, thereby simplifying configuration and improving experimental generalizability.

5.3 I/O bandwidth

This subsection evaluates the sequential and random write performance of SplitFUSE, native FUSE (including the baseline implementation, FUSE_directio, and FUSE_wb with writeback_cache), ExtFUSE, and RFUSE using the IOzone benchmark. To examine the influence of underlying storage performance, the evaluated FUSE file systems are deployed on ext4 and tmpfs backends.

ExtFUSE enables the splice and writeback_cache optimizations in its publicly available implementation. Accordingly, the results reported in this subsection incorporate both optimizations to reflect their achievable bandwidth under typical configurations. In contrast, SplitFUSE and RFUSE are evaluated without enabling these options, as their designs already overlap with or replace these optimization mechanisms.

5.3.1 Small-block test

This experiment issues write requests ranging from 1 to 16 KB to generate a total file size of 2 GB. SplitFUSE uses a PAGE_SIZE of 4 KB. The results are presented in Fig. 8.

Small I/O operations are highly sensitive to request processing overhead. SplitFUSE achieves the highest random and sequential write bandwidth among all evaluated systems when deployed on ext4, reaching up to 5.46× and 4.47× the performance of native FUSE, respectively. Compared with the best-performing FUSE_directio configuration, SplitFUSE delivers more than 2× higher bandwidth. When deployed on tmpfs, SplitFUSE exhibits even greater improvements, achieving up to 6.14× higher random write bandwidth than native FUSE. For sequential writes, it maintains the best performance, reaching up to 5.60× the bandwidth of native FUSE.

ExtFUSE is primarily optimized for metadata-intensive workloads. In I/O-intensive scenarios, its random write performance slightly exceeds that of FUSE_wb, while its sequential performance benefits from the splice optimization but remains below SplitFUSE's. RFUSE demonstrates performance trends consistent with those in Cho et al. (2024); however, under our experimental conditions, its bandwidth remains below that of native FUSE. In resource-rich environments (e.g., 80 logical cores and 256-GB double data rate 4 (DDR4) memory), RFUSE can achieve higher performance by allocating large shared-memory regions for request queues. In contrast, under more constrained system configurations, SplitFUSE provides superior performance gains.

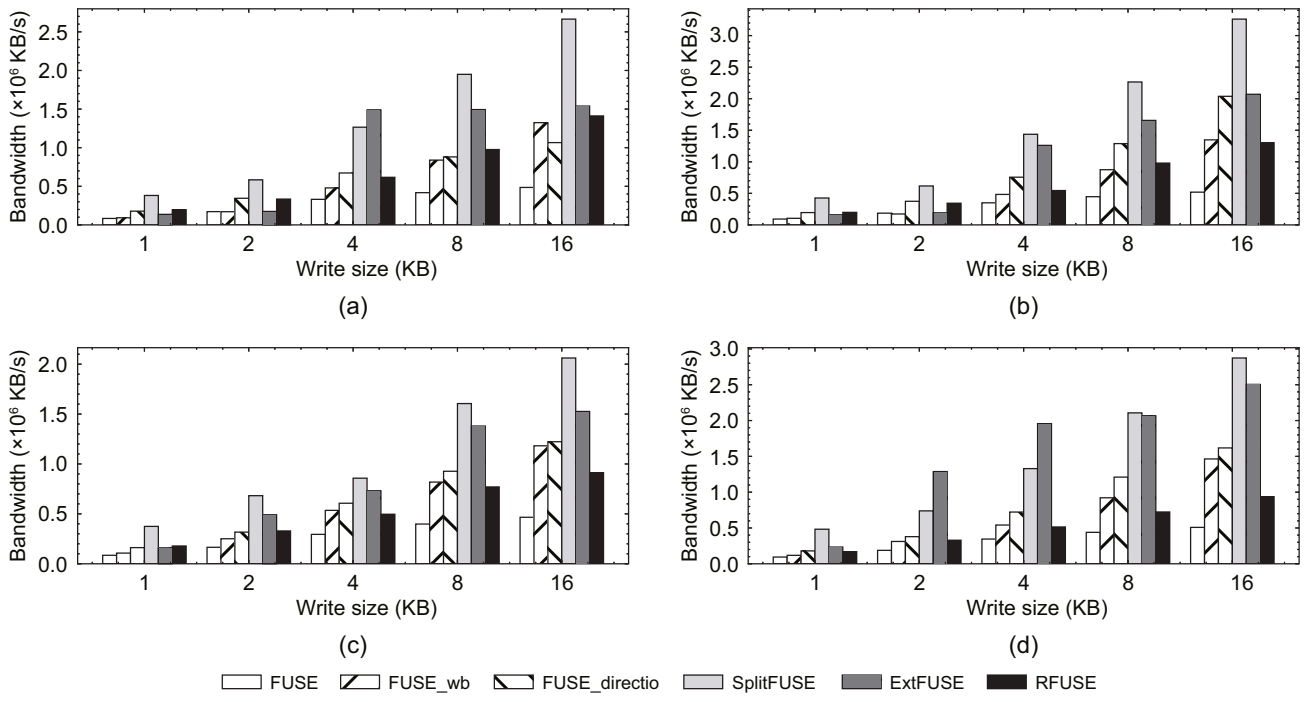

Fig. 8 Small-block I/O bandwidth: (a) ext4-random; (b) tmpfs-random; (c) ext4-sequential; (d) tmpfs-sequential

5.3.2 Large-block test

This experiment issues write requests ranging from 1 to 16 MB to generate a total file size of 2 GB. The corresponding bandwidth results are shown in Fig. 9. All FUSE variants are configured with the big_writes optimization. SplitFUSE uses a PAGE_SIZE of 128 KB.

For large-block workloads, the relative impact of context-switching overhead is reduced. As a result, the performance advantage of SplitFUSE over native FUSE is less pronounced than in the small-block scenario. Nevertheless, SplitFUSE consistently outperforms RFUSE and ExFUSE. On ext4, it achieves up to 1.82× and 2.11× higher random and sequential write bandwidth than native FUSE, respectively; on tmpfs, the improvements reach 2.12× and 1.96×, respectively.

Among all configurations, FUSE with direct_io achieves the best performance for large-block writes, as it bypasses the kernel page cache and eliminates associated overhead. The lower bandwidth of SplitFUSE, compared to direct_io, can be attributed to two primary factors.

First, shared-memory mapping introduces page faults when loading into previously unused slots. In small-block tests, the page fault ratio remains below 1% (2048×2×4/(2×1024×1024)), resulting in negligible overhead. However, in large-block tests, this ratio increases to approximately 25% (2048×2×128/(2×1024×1024)), making the overhead significant. Although pre-mapping strategies were explored to mitigate this issue, they proved unsuitable for general deployment scenarios. To preserve compatibility and maintain simplicity, SplitFUSE retains the current design.

Second, under intensive large-block write workloads, longer request execution time combined with a fixed queue length increases queue traversal latency. When worker threads cannot promptly consume queued requests, the daemon dynamically spawns additional worker threads. This leads to frequent switching between blocking and non-blocking buffer modes, introducing additional overhead. While this design effectively balances processing across data and metadata requests, it may reduce peak bandwidth in I/O-intensive benchmarks. Nevertheless, it performs efficiently under typical mixed workloads.

ExtFUSE exhibits performance comparable to native FUSE with writeback_cache enabled. As the number of requests decreases in large-block scenarios, the benefits of metadata offloading diminish, leading to lower relative performance than that in small-block tests.

RFUSE demonstrates good scalability, with stable write bandwidth as the number of application threads increases. At 16 threads, its random write performance is comparable to that of SplitFUSE. However, in most other configurations, its bandwidth remains below that of native FUSE. Notably, when deployed on tmpfs, RFUSE exhibits more pronounced performance degradation due to increased CPU and memory consumption, as discussed in Section 5.3.1.

5.4 Real-world use cases

This subsection evaluates the compatibility and performance of SplitFUSE on representative real-world FUSE-based backend file systems. We select three widely used systems (StackFS, LoggedFS, and BindFS), to cover diverse implementation strategies, optimization mechanisms, and API usage patterns (Table 2).

Table 2 Configuration of backend file systems

Usecase	Optimization	API	PAGE_SIZE (KB)
StackFS-1	direct_io, big_writes	Low level	128
StackFS-2	splice, writeback_cache	Low level	4
LoggedFS	splice, kernel_cache	High level	4
BindFS	direct_io	High level	4

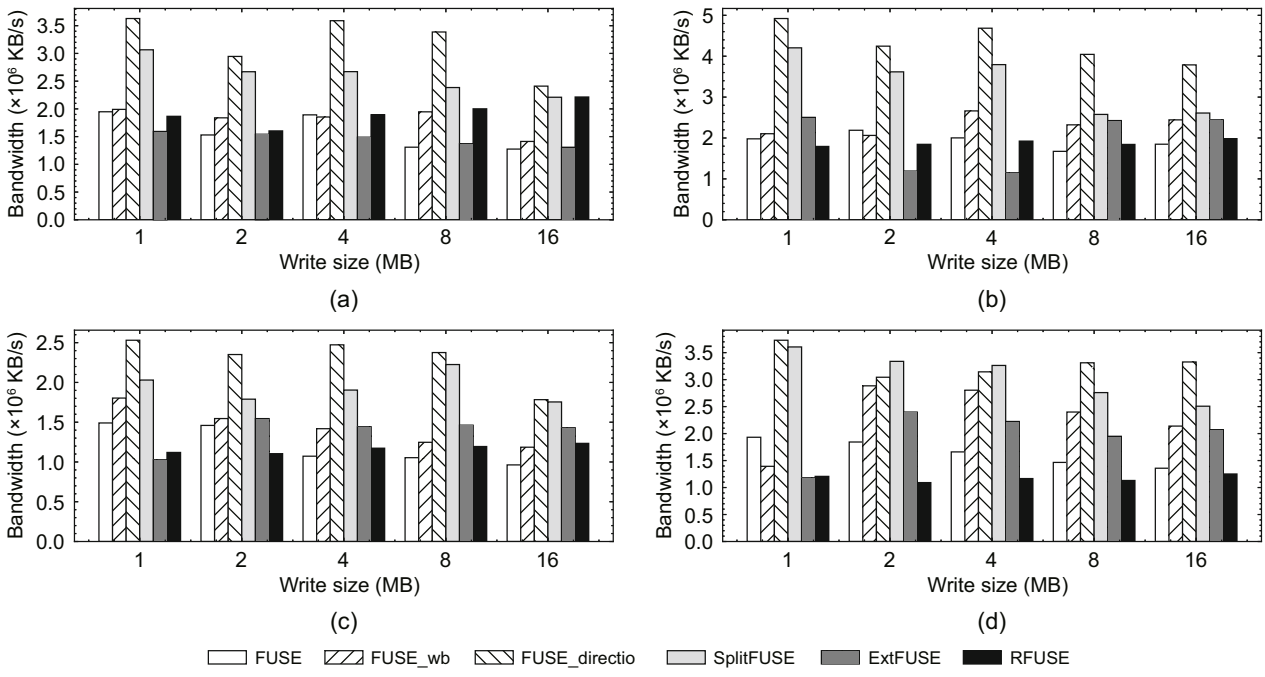

Fig. 9 Large-block I/O bandwidth: (a) ext4-random; (b) tmpfs-random; (c) ext4-sequential; (d) tmpfs-sequential

StackFS is deployed on ext4 and forwards all requests without modification, serving as a baseline for evaluating all approaches. LoggedFS optionally records file system operations; in this study, it is configured to log write operations, with kernel_cache and splice enabled as built-in optimizations. BindFS supports directory remapping and enables flexible modification of mount-point behavior, including permission control, encryption, and file hiding.

We use the file server, web server, and varmail workloads from Filebench to emulate write-intensive, read-intensive, and write-synchronous scenarios, respectively. The file server workload spawns 50 application threads that perform large writes (1 MB), appends (16 KB), and sequential reads across 10 000 files, reopening each file before every I/O operation. The web server workload creates 100 application threads that execute 10 consecutive read operations, followed by a single log write (16 KB) to 1000 files. The varmail workload consists of 16 application threads that perform file creation, append operations (16 KB), synchronization, and sequential reads on 1000 files. All experiments run for 60 s. Fig. 10 presents the throughput of the evaluated systems under these workloads.

5.4.1 Compatibility

This subsection evaluates the deployability and integration characteristics of each approach across different real-world backend file systems. The observed compatibility ranking is: FUSE≈SplitFUSE>RFUSE>ExtFUSE.

SplitFUSE is dynamically loaded via LD_PRELOAD. Similar to native FUSE, it supports all evaluated backend file systems and is compatible with libfuse2 and libfuse3, enabling zero-modification integration.

RFUSE is tightly coupled to specific versions of libfuse. In our experimental setup, it integrates seamlessly only with BindFS (which is based on libfuse3) and additionally requires a specific Linux kernel version (5.15.0), limiting its portability.

ExtFUSE relies on eBPF-based kernel extensions. By design, it requires backend file systems to modify their source code or reimplement certain interfaces to align with its optimization framework. As a result, it is incompatible with LoggedFS and BindFS in our evaluation.

5.4.2 Throughput

1. File server. Under the file server workload, SplitFUSE consistently outperforms native FUSE across all backend file systems. On StackFS-1 and StackFS-2, it achieves 1.43× and 1.32× higher throughput than native FUSE, respectively. On LoggedFS, where write operations incur additional logging overhead, the per-I/O cost increases, amplifying the benefits of SplitFUSE; throughput improves by up to 3.10× compared to native FUSE. On BindFS, SplitFUSE achieves 94.2% of the throughput of native FUSE. This is attributed to the high computational overhead of BindFS's advanced functionality, which reduces the relative impact of context-switching overhead. In this scenario, SplitFUSE worker threads handle both metadata and data operations. When metadata operations become the dominant bottleneck, the number of active worker threads quickly reaches the threshold, causing data requests to be delegated to the dedicated I/O worker thread, thereby reducing parallelism.

The file server workload is highly metadata-intensive, involving frequent file open and close operations. ExtFUSE achieves notable improvements under this workload, but it remains 12.9% to 31.0% below the performance of SplitFUSE. This workload also increases substantial memory and CPU demands, which negatively affect RFUSE stability. Under the original configuration, RFUSE is unable to complete file set initialization; it completes the benchmark only when the thread count is reduced to 10, which is not representative and is therefore omitted.

2. Web server. The web server workload is read-intensive, and direct_io cannot leverage kernel page cache read-ahead, while writeback_cache provides limited benefit due to the sparsity of write operations. SplitFUSE uses the kernel page cache to accelerate read performance while efficiently handling intermittent small writes. On StackFS-1, StackFS-2, and LoggedFS, SplitFUSE improves throughput over native FUSE by 49.3%, 27.2%, and 27.4%, respectively. On BindFS, the average I/O latency is lower than that in the file server workload, increasing the relative proportion of context-switching overhead; under these conditions, SplitFUSE achieves a modest improvement of 5.4%. Due to the sparsity of write operations, even when worker threads reach the scheduling threshold, the dedicated I/O worker thread experiences substantially less contention than in the file server workload.

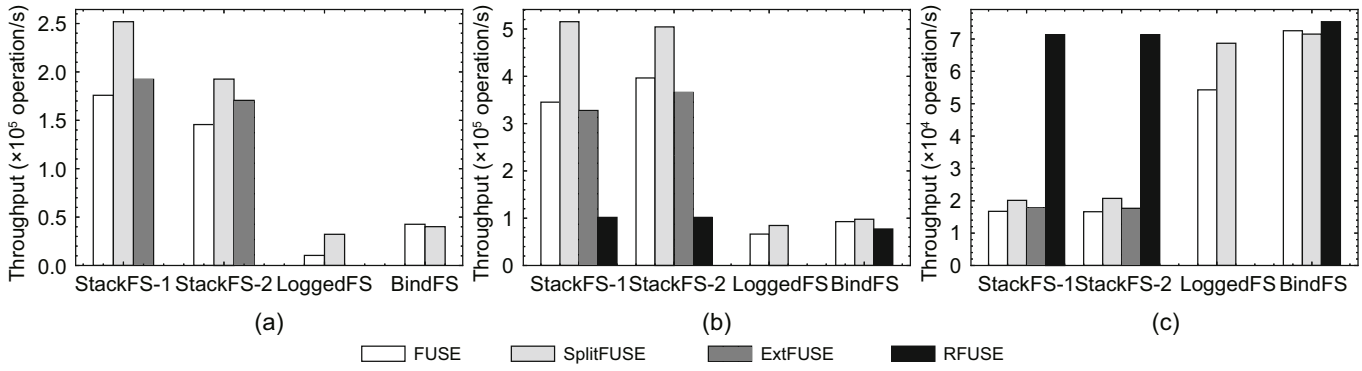

Fig. 10 Throughput comparison of backend file systems using FUSE and SplitFUSE under different workloads: (a) file server; (b) web server; (c) varmail

In the web server workload, files are pre-created and the total file count is relatively small. As a result, ExtFUSE does not benefit from metadata optimization to the same extent as in the file server scenario. Moreover, under high concurrency, the overhead of eBPF execution becomes more pronounced, reducing ExtFUSE throughput by 5.1%–7.8% on StackFS compared to native FUSE. Although memory usage is lower than that in the file server workload, CPU utilization remains high. RFUSE completes the full benchmark only on StackFS; on BindFS, execution terminates after approximately 20 s. Additional overhead from RQ management and load redistribution leads to resource contention, resulting in throughput substantially lower than that of native FUSE.

3. Varmail. In the varmail workload, metadata operations, particularly those involving synchronization, introduce serialization points that limit the effectiveness of metadata offloading. Consequently, ExtFUSE achieves only a 6.3% improvement over native FUSE. In contrast, SplitFUSE processes data requests in parallel via the shared-memory queue while metadata operations are pending, achieving throughput gains of 20.3% and 24.8% on StackFS-1 and StackFS-2, respectively. LoggedFS, which logs operations instead of performing synchronous writes, exhibits higher baseline throughput than StackFS; SplitFUSE further improves its performance by 26.5%. Similar to the file server workload, SplitFUSE provides limited performance gains on BindFS when metadata operations are computationally intensive and write operations constitute a large portion of the workload.

The varmail workload involves a relatively small number of files and application threads, allowing RFUSE to demonstrate significant performance, particularly on StackFS, where individual requests are processed quickly. This observation highlights that RFUSE performance is highly dependent on system resources and workload characteristics, and its compatibility remains substantially more limited than that of SplitFUSE.

5.5 Summary and discussion

The experimental results demonstrate the core advantages of SplitFUSE in local storage-backed environments. By eliminating frequent context switches, SplitFUSE delivers substantial performance improvements in write-intensive workloads with small I/O operations, such as metadata updates, log appends, and artificial intelligence (AI) training pipelines. Although SplitFUSE's large-block write bandwidth is slightly lower than that of direct_io, it still outperforms traditional FUSE by approximately 2×. Under mixed workloads that combine read-intensive operations with synchronous writes, SplitFUSE maintains strong cross-scenario compatibility. Developers can integrate existing FUSE-based file systems with minimal effort, without requiring privileged execution or kernel modifications, while preserving full transparency to applications.

In real-world deployments, userspace file systems are often layered on top of networked or distributed storage backends (e.g., network file system (NFS) or Ceph). These systems involve complex network communication, data partitioning, and consistency protocols, where concurrency control and network latency dominate overall performance. Consequently, the relative contribution of the context-switching overhead becomes less significant. SplitFUSE's shared-memory queue design assumes that backend systems can process requests with relatively low latency. In distributed environments, however, long-tail latency caused by network variability or load imbalance may lead to rapid queue accumulation. This can trigger adaptive mechanisms such as spawning additional worker threads or redirecting requests to the dedicated I/O worker thread, potentially reducing effective parallelism or increasing the risk of RQ saturation.

As a result, the performance benefits of SplitFUSE may be less pronounced in distributed storage scenarios than in local environments, similar to observations with BindFS. Nevertheless, its zero-modification integration remains a significant practical advantage, enabling straightforward deployment on existing distributed FUSE-based systems. To improve performance in such environments, developers can tune parameters such as queue depth and polling intervals to better accommodate high concurrency and variable latency. Future work will focus on developing a latency-aware request scheduling mechanism that dynamically adjusts queue depth, polling thresholds, and thread pool size based on backend response characteristics. This enhancement aims to improve the adaptability of SplitFUSE in high-latency and heterogeneous storage environments.

6 Conclusions

This study presents SplitFUSE, a lightweight and highly compatible I/O-acceleration framework for FUSE that preserves metadata security. Its self-contained split architecture maintains kernel-level metadata integrity while enabling an efficient userspace fast path for data operations, achieving significant performance improvements without compromising compatibility. SplitFUSE supports transparent integration with unmodified applications and facilitates seamless deployment across existing FUSE-based backends developed with diverse APIs. As a fully non-intrusive solution that requires neither kernel modifications nor privileged execution, it is particularly well-suited for security-sensitive and containerized environments. Future research will explore secure userspace metadata-handling mechanisms to further enhance performance, particularly for workloads involving frequent small-file updates, as well as adaptive scheduling strategies for distributed and high-latency storage systems.

Acknowledgments

This work was supported by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM202).

Author contributions

Yushuqing ZHANG developed the software and drafted the paper. Kai LU supervised the research. Wenzhe ZHANG conceptualized the study. Huijun WU performed the investigation. Ruibo WANG administered the project. Zhenwei WU reviewed and edited the paper.

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declaration on the use of generative AI tools

During the preparation of this work, the authors used DeepSeek to improve language. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

An W, Bi X, Chen GT, et al., 2024. Fire-flyer AI-HPC: a cost-effective software-hardware co-design for deep learning. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, Article 83. https://doi.org/10.1109/SC41406.2024.00089

Bijlani A, Ramachandran U, 2019. Extension framework for file systems in user space. Proc USENIX Annual Technical Conf, p.121-134.

Cho KJ, Choi J, Kwon H, et al., 2024. RFUSE: modernizing userspace filesystem framework through scalable kernel-userspace communication. Proc 22nd USENIX Conf on File and Storage Technologies, Article 9.

Cornell B, Dinda P, Bustamante F, 2004. Wayback: a user-level versioning file system for Linux. Proc USENIX Annual Technical Conf, p.19-28.

Deng TL, Chen C, Yin SY, 2022. On the efficiency of user-level parallel file systems. *J Univ Chin Acad Sci*, 39(2):275-282. https://doi.org/10.7523/j.ucas.2020.0027

Huai QB, Hsu W, Lu JW, et al., 2021. XFUSE: an infrastructure for running filesystem services in user space. Proc USENIX Annual Technical Conf, p.863-875.

Jia WQ, Jiang DJ, Xiong J, 2025. A high-performance and scalable userspace log-structured file system for modern SSDs. *ACM Trans Storage*, 21(4):30. https://doi.org/10.1145/3728645

Lawrence Livermore National Laboratory, 2015. ZFS. https://zfsonlinux.org [Accessed on Mar. 2, 2026].

Lembke J, Roman PL, Eugster P, 2022. DEFUSE: an interface for fast and correct user space file system access. *ACM Trans Storage*, 18(3):22. https://doi.org/10.1145/3494556

Mashtizadeh A, Bittau A, Huang Y, et al., 2013. Replication, history, and grafting in the Ori file system. Proc 24th Symp on Operating Systems Principles, p.151-166. https://doi.org/10.1145/2517349.2522721

Pontes R, Burihabwa D, Maia F, et al., 2017. SafeFS: a modular architecture for secure user-space file systems: one FUSE to rule them all. Proc 10th ACM Int Systems and Storage Conf, Article 9. https://doi.org/10.1145/3078468.3078480

Red Hat, 2019. Gluster. https://gluster.org [Accessed on Mar. 2, 2026].

Ren K, Gibson G, 2013. TABLEFS: enhancing metadata efficiency in the local file system. Proc USENIX Conf on Annual Technical Conf, p.145-156.

Sigurbjarnarson H, Ragnarsson P, Yang JC, et al., 2016. Enabling space elasticity in storage systems. Proc 9th ACM Int Systems and Storage Conf, Article 6. https://doi.org/10.1145/2928275.2928291

Trapexit, 2014. Mergerfs. https://github.com/trapexit/mergerfs [Accessed on Mar. 2, 2026].

Ungureanu C, Atkin B, Aranya A, et al., 2010. HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system. Proc 8th USENIX Conf on File and Storage Technologies, p.225-239.

Weil S, Brandt S, Miller E, et al., 2006. Ceph: a scalable, high-performance distributed file system. Proc 7th USENIX Symp on Operating Systems Design and Implementation, p.307-320.

Yan WR, Yao J, Cao Q, 2019. DeFUSE: decoupling metadata and data processing in FUSE framework for performance improvement. *IEEE Access*, 7:138473-138484. https://doi.org/10.1109/ACCESS.2019.2942954

Zhang J, Ren Y, Kannan S, 2022. FusionFS: fusing I/O operations using $CISC_{Ops}$ in firmware file systems. Proc 20th USENIX Conf on File and Storage Technologies, p.297-312.

Zhu Y, Wang T, Mohror K, et al., 2018. Direct-FUSE: removing the middleman for high-performance FUSE file system support. Proc 8th Int Workshop on Runtime and Operating Systems for Supercomputers, Article 6. https://doi.org/10.1145/3217189.3217195