RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication

Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim,
Seoul National University

https://www.usenix.org/conference/fast24/presentation/cho

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication

Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim

Seoul National University

Abstract

With the advancement of storage devices and the increasing scale of data, filesystem design has transformed in response to this progress. However, implementing new features within an in-kernel filesystem is a challenging task due to development complexity and code security concerns. As an alternative, userspace filesystems are gaining attention, owing to their ease of development and reliability. FUSE is a renowned framework that allows users to develop custom filesystems in userspace. However, the complex internal stack of FUSE leads to notable performance overhead, which becomes even more prominent in modern hardware environments with high-performance storage devices and a large number of cores.

In this paper, we present RFUSE, a novel userspace filesystem framework that utilizes scalable message communication between the kernel and userspace. RFUSE employs a per-core ring buffer structure as a communication channel and effectively minimizes transmission overhead caused by context switches and request copying. Furthermore, RFUSE enables users to utilize existing FUSE-based filesystems without making any modifications. Our evaluation results indicate that RFUSE demonstrates comparable throughput to in-kernel filesystems on high-performance devices while exhibiting high scalability in both data and metadata operations.

1 Introduction

Traditionally, filesystems have been implemented within the OS kernel, primarily for direct-attached block devices, such as Hard Disk Drives (HDDs) or Solid State Disks (SSDs). With the advent of next-generation storage devices, there have been significant shifts in filesystem design. Since these emerging storage devices offer high performance and unique data access interfaces, there have been proposals for new filesystems specifically tailored to those innovative hardware advancements. For Non-Volatile Memory (NVM) [6], which offers low-latency performance comparable to main memory, many filesystems are designed to support Direct-Access (DAX) mode. This mode eliminates redundant memory copying and facilitates direct access to NVM [24, 26, 38, 39]. Filesystems optimized for Zoned-Namespace (ZNS) SSDs [11] actively control data placement, ensuring alignment with the device's interface that mandates sequential data writes [16, 31].

Furthermore, the explosive growth in data scale has led to the development of various distributed storage solutions. These storage platforms offer finely tuned APIs that are optimized for their internal architectures. Consequently, the customization of filesystems to enhance performance for specific workloads and platforms has become a prevalent practice [5, 8, 10, 17, 37, 41].

Yet, developing and modifying an in-kernel filesystem is challenging. Developers must possess a deep understanding of intricate kernel subsystems, including page cache, memory management, block layers, and device drivers, among others. Additionally, there is a risk of inadvertently misusing complex kernel interfaces. This inherent complexity often leads to insecure implementations of in-kernel filesystems, rendering them vulnerable to critical issues, including system crashes. In addition, efforts to integrate specialized functionalities into existing in-kernel filesystems can intensify these challenges.

Alternatively, userspace filesystems are gaining attention in both industry and academia owing to their notable advantages. They offer greater reliability and safety since programming errors won't compromise the whole system. They can also leverage mature user-level libraries and debugging tools, simplifying filesystem maintenance. Userspace filesystems are easily portable across different operating systems, in contrast to in-kernel filesystems which are intrinsically tied to a specific OS kernel interface.

FUSE [36] is a framework that allows users to develop custom filesystems without requiring kernel-level modifications. It enables filesystem operations to be implemented in userspace, making it easier to develop and maintain specialized filesystems for various purposes, including filesystems for new types of storage devices, networked or distributed filesystems, or user-specific data storage. FUSE has gained popularity for its flexibility and compatibility, making it a valuable tool for building user-level filesystem extensions.

However, FUSE is often criticized for the significant overhead it incurs due to its complex software stack. Each FUSE

request, originating from the Virtual File System (VFS) layer, must undergo multiple steps before finally reaching the userspace implementation. During this process, FUSE incurs several context switches between the kernel and userspace and memory copy overhead. Also, the single queue used by the FUSE driver to dispatch filesystem requests to the userspace FUSE daemon prevents FUSE from achieving scalable performance. These overheads become even more prominent in modern hardware environments with a large number of cores and high-performance devices.

Numerous efforts have been made to mitigate the inherent overhead in FUSE [3, 15, 23]. These approaches primarily focus on enhancing communication between the kernel and userspace, aiming for performance on par with in-kernel filesystems. However, they are only partially effective, since they share the FUSE's fundamental design that relies on a single queue. Moreover, they often require developers to either reimplement the filesystem functions or introduce new implementations, which makes them incompatible with existing FUSE-based filesystems.

In this paper, we introduce RFUSE, a novel userspace filesystem framework designed to support scalable communication between the kernel and userspace. RFUSE is specifically engineered to mitigate the overheads in FUSE's internal architecture and offers improved support for modern hardware environments. To achieve this, RFUSE leverages a ring buffer data structure, commonly used for efficient message passing, to facilitate kernel-userspace communication. RFUSE has the following three design goals:

- **Scalable kenel-userspace communication.** RFUSE employs per-core, NUMA-aware ring channels, ensuring that requests transmitted across distinct channels are delivered free from lock contention. This approach maximizes the parallelism of request processing, resulting in high scalability.
- **Efficient request transmission.** RFUSE maps the ring channels as shared memory between the kernel and userspace and uses hybrid polling to efficiently transmit requests and replies. This approach effectively reduces context switches and request copy overheads.
- **Full compatibility with existing FUSE-based filesystems.** RFUSE provides the same set of APIs as FUSE, allowing existing FUSE-based filesystems to run seamlessly on RFUSE without any modifications.

To demonstrate RFUSE's scalability in a contemporary hardware environment, we carried out a series of experiments, comparing the results with other userspace filesystem frameworks. Our evaluation shows that RFUSE effectively reduces communication latency by 53%. In addition, RFUSE exhibits significantly better performance in the majority of I/O workloads. Especially, RFUSE achieves 2.27x higher throughput than FUSE in the random read workload. Furthermore,

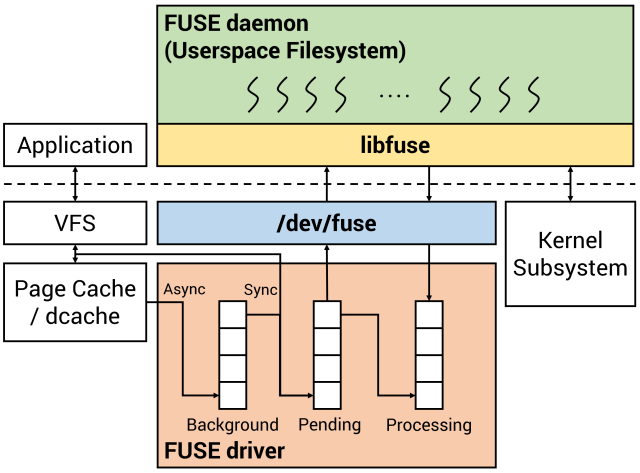

Figure 1: The internal architecture of the FUSE framework. For brevity, the *forget* queue and the *interrupt* queue are omitted in this figure.

RFUSE achieves better scalability than other frameworks in both data and metadata operations. Under the several macrobenchmarks that simulate real-world use cases, RFUSE demonstrates high performance comparable to the in-kernel filesystem. The source code of RFUSE is publicly available at https://github.com/snu-csl/rfuse.

The rest of the paper is organized as follows. We first present our background and motivation in Section 2. Section 3 describes the design of RFUSE and Section 4 shows the experimental results. We briefly introduce related work in Section 5 and conclude the paper in Section 6.

2 Background and Motivation

2.1 FUSE (Filesystem in Userspace)

FUSE enables unprivileged users to develop their own filesystems without modifying the kernel. Figure 1 illustrates the internal architecture of the FUSE framework. FUSE consists of two main components: the *FUSE driver* within the kernel and the userspace *FUSE daemon* created when the FUSE-based filesystem is mounted.

When the FUSE driver is loaded, it creates a particular device, */dev/fuse*, which acts as an intermediary between the Virtual File System (VFS) and the FUSE-based filesystem. Internally, the FUSE driver has five types of queues: *background*, *pending*, *processing*, *forget*, and *interrupt*. The first three queues are used to route requests for filesystem operations to the FUSE daemon. The forget queue is for interaction with the directory cache (dcache), while the interrupt queue handles interrupt requests, which are generated when the kernel needs to interrupt an ongoing filesystem operation.

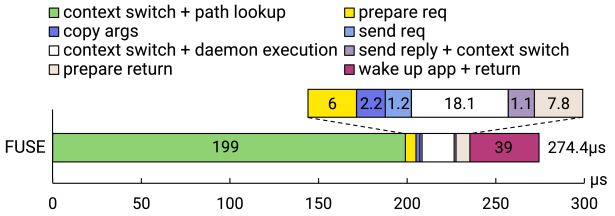

Figure 2: Latency breakdown for processing an empty filesystem operation (CREATE) in FUSE.

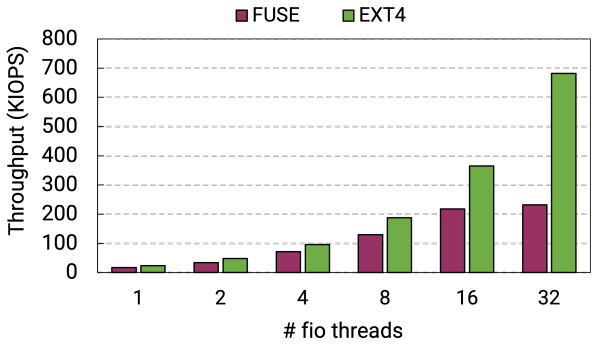

Figure 3: Scalability of random read throughput on StackFS over EXT4 (FUSE) vs. native EXT4.

When applications initiate a file operation on a FUSE-based filesystem, VFS sends the request to the FUSE driver. The driver then enqueues the request in the appropriate queue depending on whether it is a synchronous or an asynchronous request. Synchronous requests are immediately added to the pending queue. In contrast, asynchronous requests, such as read-ahead or write-back requests, are initially put into the background queue before making their way to the pending queue. The FUSE driver limits the number of asynchronous requests in the pending queue to prevent interference from bulk asynchronous requests. This strategy is particularly beneficial for preserving the responsiveness of synchronous requests that are usually latency-sensitive.

When a FUSE-based filesystem is mounted, the FUSE daemon is initiated and establishes a communication channel by performing an `open()` system call on */dev/fuse*. Subsequently, the FUSE daemon creates a worker thread that performs a `read()` system call on */dev/fuse* to retrieve file operation requests. If there are no pending requests, the thread sleeps in a wait queue managed by the FUSE driver, until it receives further requests. Otherwise, the FUSE driver responds to the `read()` system call by returning the first request in the pending queue. Once the worker thread parses the request, it executes the corresponding operation according to the opcode.

A FUSE request consists of the common header, the operation-specific header, and argument(s). The common header contains the essential information required by all operations, such as the opcode and flags that denote the request's status. The operation-specific header includes the additional information specific to each operation. For metadata operations, the argument usually denotes the name of the target file(s), whereas for data operations, it indicates the required data for I/O. Both the FUSE driver and the FUSE daemon exchange these information by performing `read()` and `write()` system calls on */dev/fuse*. A FUSE reply also contains the common header and the operation-specific header. In FUSE, the headers for the request and reply are named *in_header* and *out_header*, respectively.

A FUSE daemon can have multiple worker threads. When the FUSE daemon finds no more remaining threads to receive a request from the FUSE driver, it spawns a new worker thread before handling the received request. There is no explicit limitation on the number of worker threads in FUSE, but it is implicitly controlled by the limitation imposed on the number of asynchronous requests that can reside in the pending queue.

2.2 Overheads in FUSE

Although FUSE provides high flexibility in developing userspace filesystems, its complex stack leads to notable performance overhead.

Latency overhead. As a first step, we conducted a latency analysis of the CREATE operation on NullFS. NullFS is a userspace filesystem we developed, which simply returns zero for any filesystem operation executed in userspace, except for the LOOKUP operation on the root directory. Figure 2 presents the latency breakdown of an empty CREATE operation, as observed in our experimental setup (see Section 4.2). The graph illustrates the various stages of the operation, highlighting the time taken at each step.

First, we can see that accessing the VFS layer and path lookup occupies 72% of the total time. Within the VFS layer, the kernel performs iterative path traversal starting from the root directory to check the existence of subdirectories and files. This path-name resolution process results in several LOOKUP operations directed towards the FUSE daemon in userspace. Hence, the latency during the initial path lookup phase (highlighted in green) encompasses the time taken for multiple rounds of context switches between the kernel and userspace. Second, the context switch and request copy overhead between the kernel driver and the FUSE daemon is not negligible. Even though NullFS does nothing but return the result, the userspace execution took as long as 18.1 μsec, due to the context switch overhead. Third, Figure 2 illustrates a significant overhead, amounting to 39 μsec, when waking up the application process that awaits a response from the FUSE daemon.

Several optimizations have been proposed to address the aforementioned latency issues in FUSE. Android 12 intro-

duced FUSE-passthrough [3] to achieve the performance of FUSE comparable to direct access to the in-kernel filesystem. With FUSE-passthrough, the FUSE driver directly forwards the READ/WRITE requests to the underlying filesystem. However, this approach bypasses the FUSE daemon, thereby sacrificing FUSE's ability to support custom userspace filesystem functions. For this reason, FUSE-passthrough is only effective for stackable filesystems that pass the unmodified requests directly to the underlying filesystem.

Another interesting approach is EXTFUSE [15]. It extends the FUSE framework, enabling the userspace filesystem to register simple eBPF [12] code snippets into the kernel. This allows various filesystem functionalities to be executed directly within a safe sandboxed environment in the kernel, avoiding costly context switches between the kernel and userspace. However, EXTFUSE requires filesystem developers to craft new functionalities within the constraints of eBPF, including limited code size, bounded loops, restricted access to kernel data, constrained pointer usage, and so on.

Bandwidth overhead and scalability issues. In FUSE, all requests from the VFS layer are placed into a shared pending queue, leading to severe lock contention, especially when multiple threads execute filesystem operations simultaneously. Not only does this design fail to harness the full bandwidth potential, but it also acts as a roadblock in the development of scalable userspace filesystems.

We ran the `FIO` benchmark to assess the scalability of random read throughput in FUSE, varying the number of `FIO` threads from 1 to 32[1]. Figure 3 contrasts the throughput of the native EXT4 filesystem with that of StackFS over EXT4. StackFS [4] is a userspace filesystem built on top of FUSE that merely passes filesystem operations to the underlying kernel filesystem (EXT4 in this experiment). Figure 3 shows that the throughput of StackFS fails to scale once the number of threads exceeds 16, while the throughput on the native EXT4 filesystem increases linearly. We note that even with a small number of threads, StackFS's bandwidth lags behind that of EXT4. We believe that the single queue-based communication in FUSE prevents StackFS from attaining scalable performance.

Recently, XFUSE [23] proposes the use of multiple communication channels to increase parallelism in FUSE. However, just adding more queues does not completely resolve the lock contention. Furthermore, the inherent context switch overhead from the original FUSE design still remains.

2.3 Motivation

Our work is inspired by io_uring [19], an efficient I/O interface introduced by the Linux kernel to address the limitations of the native asynchronous I/O interface. The io_uring interface is built around two primary elements: the Submission Queue (SQ) that holds I/O requests placed by applications, and the Completion Queue (CQ) that contains the results of those I/O requests. Typically, io_uring notifies the kernel of the submission of a new I/O request and fetches completion events from the kernel by calling the `io_uring_enter()` system call. However, io_uring offers an additional feature called *polled I/O mode* to eliminate systems calls for low latency devices. In this mode, a dedicated kernel thread monitors the submission queue while the user application polls the completion queue. The polled I/O mode enables io_uring to operate without frequently making system calls.

At its core, io_uring provides a shared memory-mapped ring buffer between the kernel and userspace to process messages to/from block devices. Using a ring buffer offers numerous advantages. First, messages (request commands or completion entries) can be enqueued into the ring buffer atomically with constant-time complexity. This capability allows the ring buffer to handle burst messages with low latency, yielding high throughput. Second, the ring buffer can be easily scaled to handle increased throughput by either enlarging its size or adding more ring buffers. Especially, a separate ring buffer can be allocated for each CPU core to minimize potential lock contention.

A comparable architecture is also employed as the communication interface between CPUs and peripheral devices. For instance, the NVMe protocol [13] utilizes a pair of ring buffers, Submission Queue (SQ) and Completion Queue (CQ), to interact with the NVMe SSDs. Similarly, Ethernet NICs (Network Interface Cards) employ Transmit (TX) and Receive (RX) ring buffers to manage outgoing and incoming network packets.

In this paper, we propose RFUSE, a novel and scalable FUSE framework that leverages a collection of ring buffers for communication between the in-kernel FUSE driver and the userspace FUSE daemon. RFUSE strives to enhance the scalability of the FUSE framework and reduce both context switch and request copy overheads by deploying ring buffer-based, per-core communication channels between the kernel and userspace. Another goal of RFUSE is to maintain the same interface as FUSE so that existing FUSE-based userspace filesystems can be executed easily over RFUSE.

3 Design

RFUSE utilizes the ring buffer structure for scalable communication between the kernel and userspace, similar to the io_uring interface. We could not directly utilize io_uring because io_uring performs request submission in the user-to-kernel direction, which does not align with the FUSE structure where kernel-to-user submission is necessary. Furthermore, as io_uring has its own kernel context, we find it challenging to facilitate flexible optimizations within the FUSE structure.

Instead, we have designed a novel *ring channel* based on a ring buffer structure, specifically to meet the needs of the FUSE framework. In this section, we delve into the mechan-

[1] The experimental setup is same as in Figure 10 (d)

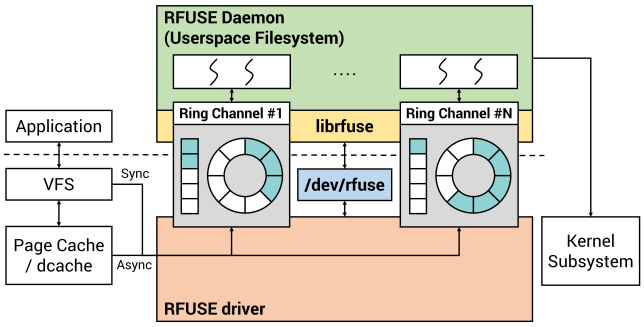

Figure 4: The overall architecture of RFUSE

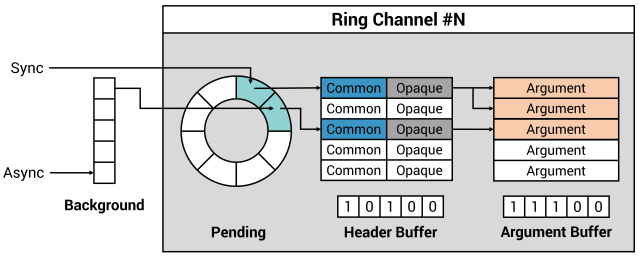

Figure 5: Components in a ring channel. For brevity, the *forget* and *interrupt* ring buffers are omitted in this figure.

ics of our ring channels and describe the design challenges associated with them.

3.1 Overall Architecture of RFUSE

RFUSE is designed to maximize performance and scalability in modern hardware environments that are equipped with many CPU cores and high-performance devices. Figure 4 depicts the overall architecture of RFUSE. Similar to FUSE, RFUSE consists of two main components: the in-kernel *RFUSE driver* and the userspace *RFUSE daemon*. However, unlike FUSE which relies on a single queue for communication between the kernel and userspace, RFUSE employs a ring channel-based message passing mechanism for each core.

When the RFUSE driver is loaded, a ring channel is created for each core in the machine along with a special device */dev/rfuse*. This architecture is intended to boost throughput by enabling parallel processing of filesystem operation requests. When a user mounts an RFUSE-based filesystem, the RFUSE daemon maps the memory region of these ring channels into the user's virtual address space using `mmap()`. This allows the userspace filesystem to exchange messages with the kernel without any context switch (see Section 3.2 for details).

When the RFUSE driver forwards a request to the RFUSE daemon, it determines the appropriate ring channel for request delivery based on the CPU core ID where the current thread is scheduled. For example, if an application thread issuing a filesystem operation runs on core 3, the RFUSE driver transmits the corresponding request to the RFUSE daemon via ring channel #3.

RFUSE allocates the memory for ring channels and their associated components in consideration of NUMA locality. When a ring channel is allocated to a different NUMA node, every access during request submission and completion incurs remote NUMA memory access penalties, resulting in substantial latency. To mitigate this, RFUSE allocates each ring channel to memory on the same NUMA node as its corresponding CPU core. This ensures that the RFUSE daemon does not access memory from a different NUMA node while processing requests.

Replacing the single queue in FUSE with per-core ring channels looks seemingly straightforward to improve performance, but it introduces several design challenges. In the following subsection, we examine the components of the ring channel and its internal operations in more detail. Section 3.3 explains how RFUSE manages worker threads on per-core ring channels. We delve into how RFUSE mitigates context switch and thread wake-up overhead through hybrid polling in Section 3.4. Section 3.5 examines RFUSE's strategies for load balancing in the face of burst asynchronous requests. Section 3.6 describes how the RFUSE daemon and the kernel driver communicate with each other using logical identifiers. The memory overhead caused by the ring channels is analyzed in Section 3.7. Lastly, Section 3.8 outlines the extensions we made in RFUSE to ensure compatibility with existing FUSE-based filesystems.

3.2 Scalable Kernel-Userspace Communication

Figure 5 illustrates the internal components of a ring channel that connects the RFUSE driver and the RFUSE daemon. Each ring channel has three ring buffers: pending, forget, and interrupt. In addition, there are two separate buffers and a background queue exists for each ring channel. Similar to FUSE, synchronous requests are enqueued directly into the pending ring buffer, while asynchronous requests are initially added to the background queue. These asynchronous requests are subsequently moved to the pending ring buffer to prevent them from exceeding the predefined maximum capacity of that buffer.

In contrast to FUSE, which sends a request in response to the system call, RFUSE utilizes a *header buffer* and an *argument buffer*. Each entry in the header buffer consists of a common header and an opaque header. The common header contains the common information for all operations such as

an opcode and a completion flag. During request submission, the opaque header holds an operation-specific header. Upon returning the result from userspace, RFUSE reuses the same header buffer entry as an out header. This approach allows RFUSE to deliver the request's outcome to the RFUSE driver efficiently.

These components of a ring channel are mapped to the virtual memory area (VMA) of the RFUSE daemon when an RFUSE-based filesystem is mounted. This establishes a shared memory space between the kernel and the RFUSE daemon. Through these shared ring buffers, the kernel can interact with the RFUSE daemon without the need to allocate and copy a request for every filesystem operation.

For example, let us consider a scenario where the VFS layer forwards a CREATE request to the RFUSE driver. The RFUSE driver first retrieves the index of an empty entry from the header buffer. Then, the driver fills the common parameter in the common header part and uses the opaque header part as *create_in_header* which is the operation-specific header of the CREATE request. Additionally, since the CREATE operation requires a filename as an argument, the driver gets a single entry from the argument buffer and records its index in the common header. After the preparation of the request, the driver enqueues the index of the header buffer entry into the pending ring buffer and increments the tail pointer. When the RFUSE daemon dequeues from the pending queue, it retrieves the index of the header buffer and parses the header to perform the appropriate userspace filesystem operation. In the case of CREATE, it returns two operation-specific out headers: *entry_out_header* containing metadata for the created file, and *open_out_header* containing file descriptor information. These are returned by reusing the opaque header and argument entry, which are used for request submission and the reply is transmitted by setting the completion flag in the common header. This approach significantly reduces the need to allocate and copy for each of requests and replies and makes efficient communication between the kernel and userspace.

RFUSE uses bitmaps for both the header buffer and argument buffer to track the allocation status of entries in these fixed-sized buffers. When all the bits in the bitmap are set, indicating that no further requests can be added to the buffer, application threads will go into a sleep state, waiting for the completion of previously submitted requests. Upon request completion, RFUSE resets the corresponding bit in the bitmap and awakens one of the threads that is in a sleep state, awaiting its turn.

3.3 Worker Thread Management

For each ring channel, the RFUSE daemon creates dedicated worker threads responsible for handling the requests received from that channel. A worker thread is bound to the corresponding CPU core by setting its CPU affinity to the same core ID as the assigned ring channel.

To completely eliminate lock contention among worker threads, it is natural to have only one worker thread per ring channel. However, this single-thread approach can negatively impact the performance. For instance, when a time-consuming operation such as FSYNC is in progress, other requests must wait until the FSYNC operation finishes. Creating as many worker threads as required, as is done in FUSE, is also not a viable option. This is because the worker threads associated with a ring channel are affinitized to the same CPU core, leading to substantial contention on that particular core.

Considering these constraints, RFUSE permits multiple workers per ring channel but caps the maximum thread count. Because the RFUSE daemon spawns only a small number of worker threads (two, by default) within a ring channel, contention on a single core remains limited. Note that there is no lock contention among worker threads operating on different ring channels.

3.4 Hybrid Polling

In FUSE, the communication between the FUSE driver and the FUSE daemon relies on `read()`/`write()` system calls on the */dev/fuse* device. When the worker threads in the FUSE daemon no longer have incoming requests to handle, or when application threads are waiting for a response from the FUSE daemon, they go into a sleep state until an event wakes them up. Using system calls leads to frequent context switches, and the event-wait mechanism between processes adds noticeable delays on the order of microseconds. This can result in significant overhead, particularly for metadata operations which typically require short latency.

Similar to the polled I/O mode in io_uring, RFUSE also supports a polling mechanism. In RFUSE, the worker threads poll the head pointer of the pending ring buffer in userspace for incoming requests, while the application threads monitor the completion flag of their submitted requests in the header buffer, waiting for a response. The use of polling eliminates not only the context switches caused by system calls, but also the delays associated with awakening threads from the sleep state. However, if polling is used in a naive manner, it can lead to the wastage of CPU resources. This inefficiency is further exacerbated in RFUSE, where both kernel and userspace threads running on the same CPU core.

As a solution, RFUSE adopts a *hybrid polling* approach. There is a user-defined period (50μsec, by default) during which a thread can perform busy-waiting idly. If the application thread in the polling state exceeds this period, it will enter the sleep state, waiting for the completion flag to be set. For requests that can be quickly handled by the userspace implementation, the application thread can receive a reply during polling and return promptly. Otherwise, for requests with longer latency, it will enter the sleep state, thus avoiding unnecessary CPU wastage. The worker threads in the RFUSE daemon also behave similarly; if there are no incoming re-

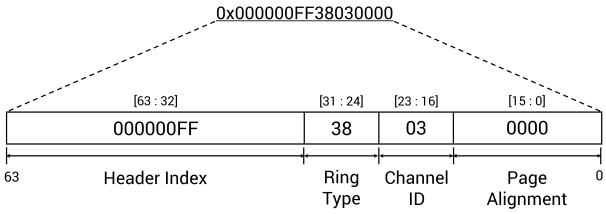

Figure 6: Encoding ring channel information in a 64-bit integer.

quests while polling the pending ring buffer, they will sleep in the wait queue.

3.5 Load Balancing of Asynchronous Requests

In the RFUSE driver, asynchronous requests are handled in a manner similar to FUSE, where they are first added to the background queue for congestion control before being transferred to the pending ring buffer. This design aims to minimize the impact of burst asynchronous requests on synchronous operations.

However, such a policy poses a problem when it is combined with RFUSE's ring selection strategy. Because RFUSE chooses the ring channel based on CPU core ID, a large number of asynchronous requests can overwhelm a single ring channel, especially for read-ahead or write-back requests that are generated in bursts by a single kernel thread. Furthermore, given the limited number of worker threads allocated to each ring, the throughput of asynchronous operations can be significantly affected. To address the skewed distribution of asynchronous requests, RFUSE implements a load-balancing policy when congestion occurs.

When enqueuing asynchronous requests into the background queue, RFUSE identifies congestion and attempts to perform load balancing based on the following two criteria: (1) when the number of requests waiting in the background queue exceeds the maximum number of asynchronous requests that can reside in the pending ring buffer, and (2) when there is a thread in the sleep state due to the prolonged execution time within the RFUSE daemon. If congestion is detected in a ring channel, RFUSE schedules the incoming asynchronous requests onto different ring channels in a round-robin fashion. This helps alleviate the load on the congested ring channel and maximize the utilization of multiple ring channels, thus increasing the overall throughput.

3.6 Transmission of Ring Channel Information

The RFUSE daemon needs to identify the locations of in-kernel data structures such as ring buffers, header buffers, and argument buffers for the following internal operations: (1) mapping the components of a ring channel in the VMA by performing `mmap()` on */dev/rfuse* during the initialization phase, (2) identifying data pages prepared for READ/WRITE requests from application threads, (3) transitioning to a sleep state on the wait queue associated with the ring buffer by `ioctl()` when the worker thread needs to stop its polling, and (4) waking up an application thread by `ioctl()` that has entered a sleeping state while waiting for completion.

However, the userspace RFUSE daemon cannot know the exact addresses of those data structures since they are allocated and managed by the kernel driver. Therefore, rather than relying on physical addresses, the RFUSE daemon utilizes logical identifiers, such as ring channel IDs, ring buffer types, and header buffer indexes, to communicate with the kernel driver. Through these logical identifiers, the userspace daemon can communicate more securely as they do not need to directly communicate via physical addresses. When the `mmap()` system call is invoked, these logical identifiers are encoded and then passed to the kernel driver using the 64-bit `offset` parameter of the `mmap()` system call.

Figure 6 depicts an example of how to encode ring channel information in a 64-bit integer. We exclude bits [15:0] due to page alignment constraints in the `offset` parameter of the `mmap()` system call. We use bits [23:16] to indicate the ID of the ring channel and bits [31:24] for ring buffer types. The remaining bits [32:63] are used to specify an entry index within the header buffer. For the `ioctl()` system call, this information is passed as the third parameter.

3.7 Memory Usage of Ring Channels

Throughout the lifespan of a userspace filesystem, ring channels remain mapped to the RFUSE daemon, retaining memory until the filesystem is unmounted. The number of ring channels matches the number of CPU cores, with both the ring buffer and the header buffer having the same number of entries. Due to some operations such as RENAME that require two arguments, the argument buffer has twice as many entries as the ring buffer. With these considerations, we can calculate the total memory usage due to ring channels as follows:

$$MemUsage = N_c \times N_r \times (S_p + S_f + S_i + S_h + 2 \times S_a) \quad (1)$$

where N_c and N_r denote the number of cores and the number of entries in the ring buffer, respectively. S_p, S_f, and S_i represent the entry size of the pending, forget, and interrupt ring buffer, respectively. Finally, S_h and S_a indicate the entry size of the header buffer and the argument buffer, respectively.

By default, RFUSE uses the following parameter values (in bytes): N_r = 4096, S_p = 4 (*integer index to the header buffer*), S_f = 32, S_i = 8, S_h = 256 (*the common and opaque header size*), and S_a = 256 (*the maximum length of the file name*). Considering an 80-core machine with 256GB of memory, the estimated memory footprint of ring channels is approximately 250MB. Given that this accounts for about 0.1% of the to-

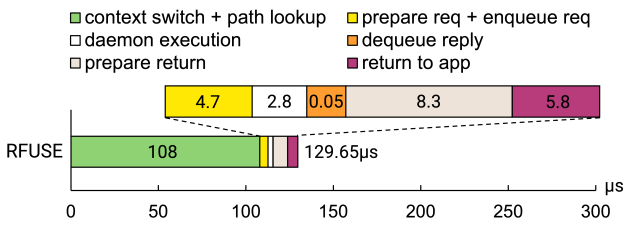

Figure 7: Latency breakdown for processing an empty filesystem operation (CREATE) in RFUSE.

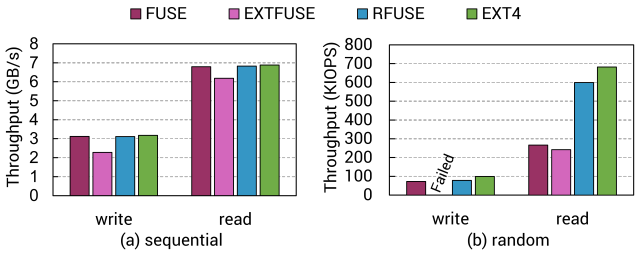

Figure 8: FIO throughput of StackFS and native EXT4.

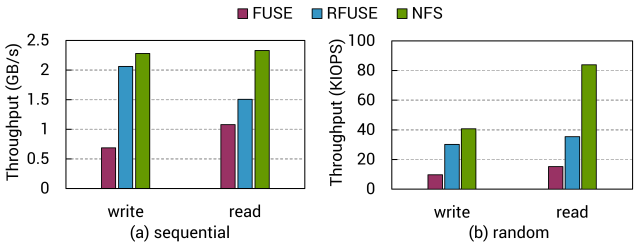

Figure 9: FIO throughput of Fuse-nfs and in-kernel NFS.

tal memory size, we believe this level of memory usage is acceptable.

3.8 Compatibility with FUSE

To make use of the ring channels, we have modified the FUSE kernel driver and the low-level layer of *libfuse* that handles message communication. In RFUSE, the READ/WRITE handlers in the kernel driver, previously used for message communication in FUSE, are now dedicated solely to data transmission for I/O requests.

Nevertheless, RFUSE retains all FUSE APIs exposed to developers of userspace filesystems. RFUSE also provides the same splicing I/O interface as FUSE, enabling data transfer between two in-kernel buffer without data copy into userspace. Thus, RFUSE ensures full compatibility with existing FUSE-based filesystems. Users do not need to rewrite their FUSE-based filesystem code when using RFUSE. The only action required is to re-link their filesystems with the *librfuse* library.

Since requests are submitted based on the CPU core ID, RFUSE requests can be executed out-of-order. Nevertheless, RFUSE ensures the same level of correctness as FUSE regarding request ordering. While FUSE utilizes a single communication queue, the userspace FUSE daemon may have multiple worker threads. This implies that simultaneous enqueuing of dependent requests may yield varying outcomes depending on the userspace filesystem implementation within FUSE. Consequently, the ordering of requests transmitted in parallel should be managed either by the VFS layer or through FSYNC-like operations initiated by applications.

4 Evaluation

4.1 Experimental Setup

Hardware setup. We used two types of testbeds to conduct our experiments. The first testbed is a Dell PowerEdge R750xs server equipped with two Intel(R) Xeon(R) Silver 4316 CPUs (80 logical cores in total) and 256GB of DDR4 memory. This testbed is also equipped with a 2TB Fadu Delta PCIe 4.0 SSD and a Mellanox ConnectX-6. Note that unless otherwise explicitly specified about the machine configuration, all experiments were carried out using this testbed. The second testbed is a Supermicro 7049GP-TRT server with two Intel(R) Xeon(R) Gold 5218R CPUs (80 logical cores in total) and 256GB of DDR4 memory. This testbed is equipped with Mellanox ConnectX-5. For the experiment on Fuse-nfs in Section 4.3.1, we used this testbed as the client and the first testbed as the server. Both testbeds run Ubuntu 20.04 LTS with the Linux kernel version 5.15.0.

FUSE frameworks tested. We conduct a comparative analysis of RFUSE against other userspace filesystem frameworks, specifically FUSE [36] v3.10.5 and the latest version of EXT-FUSE [15] available on GitHub. Additionally, we have developed an emulated version of XFUSE [23], as its source code is not in the public domain. This emulation encompasses multiple FUSE communication channels corresponding to the number of CPU cores and the adaptive waiting strategy that dynamically adjusts the busy-wait period within the FUSE driver. We have excluded the RAS feature for supporting online upgrades of user-level filesystems, as it does not significantly impact filesystem performance.

User-level filesystems tested. For our experiments, we consider three userspace filesystem implementations: NullFS, StackFS [4], and Fuse-nfs [2]. To analyze and contrast the latency associated with request handling in FUSE and RFUSE, we implemented a very simple userspace filesystem called NullFS. NullFS only supports the LOOKUP operation on the root directory, and it merely returns zero for all other operations. StackFS is a stackable userspace filesystem that forwards incoming filesystem operations to an underlying

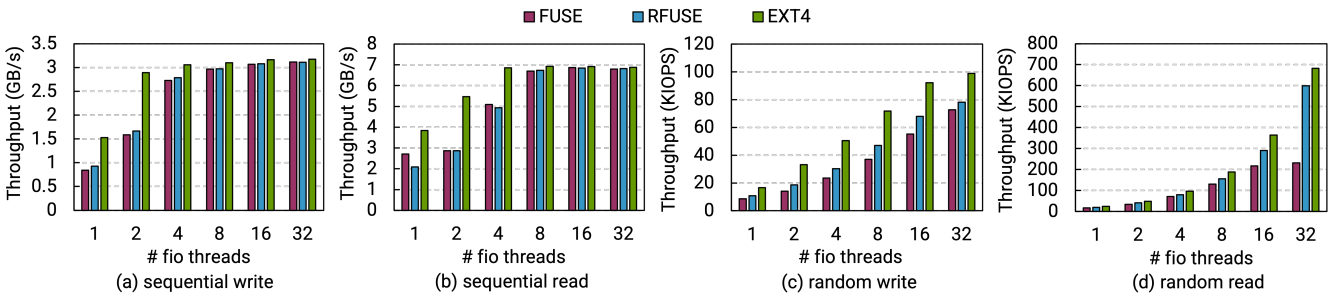

Figure 10: Scalability with FIO benchmark

in-kernel filesystem, such as EXT4. We evaluate the performance of StackFS on top of the EXT4 filesystem across three different frameworks, FUSE, EXTFUSE, and RFUSE, as well as that of the native EXT4 filesystem within the kernel. Fuse-nfs is a userspace implementation of the Network File System (NFS) client using the *libnfs* user-level library. Since EXTFUSE lacks a ported implementation of Fuse-nfs, our comparison focuses on Fuse-nfs running on FUSE and RFUSE, in addition to the in-kernel implementation of NFS.

4.2 Latency Breakdown

Figure 7 depicts the latency breakdown of a CREATE operation to NullFS on the root directory, which promptly returns without performing any action in RFUSE. In comparison to the same operation's latency in FUSE, as illustrated in Figure 2, RFUSE demonstrates a 53% lower latency. The substantial improvement in latency can be attributed to three primary factors.

First, RFUSE eliminates the need for context switches when processing requests and results. By accessing the pending ring buffer, RFUSE can retrieve the requests to be executed and quickly return the results by setting the completion flag of the corresponding entry in the header buffer. Thus, RFUSE improves the time taken in userspace by 6.46x compared to FUSE (highlighted in white in Figure 2 and Figure 7).

Second, RFUSE effectively minimizes the wake-up overhead within the kernel driver using a hybrid polling technique. After sending a request, the application thread polls the completion flag for a certain duration. Since NullFS returns the result instantly upon receiving a request, the completion is detected while the application thread is still polling, enabling immediate result retrieval.

The last factor is the improved time required for path traversal to verify the existence of subdirectories and files. As mentioned in Section 2.2, the path-name resolution initiated by the VFS layer triggers internal LOOKUP operations to the FUSE daemon along the path of the target file. Each of these LOOKUP operations results in a round trip between the kernel and userspace. Due to the reduced latency in processing a single request in RFUSE, LOOKUP operations are executed faster than in FUSE, considerably decreasing the time taken for path-name resolution.

4.3 Micro-benchmark

4.3.1 FIO Performance

To demonstrate RFUSE's ability to deliver high throughput, we perform the FIO benchmark [1] on StackFS and Fuse-nfs. The FIO benchmark is executed using 32 threads, varying both the data access pattern and the request size. For sequential I/O workloads, we use a request size of 128KB and invoke FSYNC at the end of the sequential writes. For random I/O workloads, we use a 4KB request size and trigger FDATASYNC after every write operation during random writes. Each FIO thread operates on a 4GB file with a total file size of 128GB. We also conducted the FIO benchmark using the splicing I/O interface of FUSE and RFUSE. However, we omit the results as they did not show significant differences. Note that we were unable to measure the random write throughput of EXTFUSE as it returned errors in our testing environment.

Figure 8 displays the FIO results for the native EXT4 filesystem and StackFS deployed on various frameworks. In Figure 8(a), both FUSE and RFUSE exhibit comparable throughput to EXT4 for both sequential read and write workloads. This is because, for sequential reads, the data is prefetched into the page cache through read-ahead operations, and for sequential writes, the written data is collected in the page cache before being written back in bulk. However, EXTFUSE exhibits lower throughput even for sequential workloads compared to other frameworks. EXTFUSE provides a functionality similar to fuse-passthrough, allowing I/O operations to be directly passed to the underlying filesystem via eBPF. However, this functionality was not available in the open-source version of EXTFUSE on GitHub, limiting its performance capabilities.

In Figure 8(b), RFUSE shows performance comparable to the native EXT4 filesystem for random workloads. In particular, RFUSE achieves 2.27x higher throughput than FUSE in

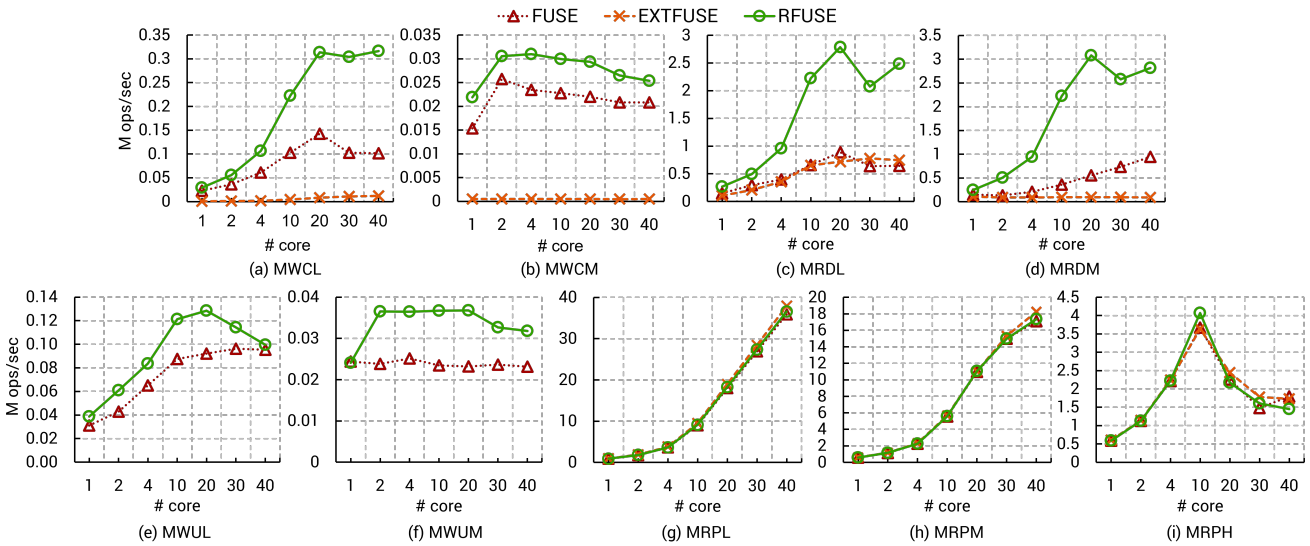

Figure 11: Scalability with FXMARK metadata operations

random reads. This is due to the effectiveness of RFUSE's hybrid polling mechanism in reducing the context switch and wake-up overhead. Considering that 4KB I/O operations typically have short execution times, FIO threads can receive the results while they are polling for them.

We conduct the same workloads on in-kernel NFS and Fuse-nfs deployed on both FUSE and RFUSE. Although it may be difficult to make a direct comparison between in-kernel NFS and Fuse-nfs due to the inherent differences in their client implementations, we consider the results from in-kernel NFS as a reference point for theoretical maximum performance. In Figure 9, RFUSE achieves higher throughput than FUSE across all workloads due to its scalable communication interface and a reduction in the average latency.

4.3.2 I/O Scalability

To investigate RFUSE's scalability compared to FUSE, we conducted experiments on StackFS with the same workloads in Section 4.3.1. We gradually increased the number of FIO threads from 1 to 32, and the results are presented in Figure 10. We have omitted the results of using splicing I/O as they followed a similar trend to those in Figure 10.

For sequential workloads, EXT4 demonstrates significantly higher throughput at lower thread counts. This can be attributed to the inherent characteristics of the FUSE and RFUSE frameworks that require communication between the kernel and userspace. Achieving sufficient throughput with fewer threads is challenging due to the communication overhead, even with the assistance of the page cache and read-ahead operations. However, when the number of threads exceeds 8, RFUSE exhibits throughput comparable to EXT4 due to increased parallelism.

Workload	Description
MWCL	Create empty files in a private directory
MWCM	Create empty files in a shared directory
MRDL	Enumerate a private directory
MRDM	Enumerate a shared directory
MWUL	Unlink empty files in a private directory
MWUM	Unlink empty files in a shared directory
MRPL	Open and close private files in a directory
MRPM	Open and close arbitrary files in a directory
MRPH	Open and close the same file in a directory

Table 1: Summary of metadata operation in FXMARK.

In Figure 10(b), we can observe that the sequential read throughput of RFUSE is lower than that of FUSE when using only one FIO thread. During the execution for sequential reads, read-ahead is performed to prefetch data. However, in RFUSE, this operation can lead to congestion on the ring channel, triggering RFUSE to initiate load balancing. Consequently, when there is only one thread, RFUSE experiences a minor performance decline due to the overhead associated with request reallocation. Nevertheless, with a higher number of threads, the increased parallelism allows RFUSE to achieve throughput comparable to EXT4.

For random workloads, RFUSE demonstrates higher throughput than FUSE while increasing the number of threads. Notably, in the random read workload, the throughput of FUSE ceased to scale beyond 16 threads, while RFUSE continues to show the scalable throughput. RFUSE exhibits better scalability due to its utilization of per-core ring channels. In addition, as mentioned in Section 4.3.1, the reduction in con-

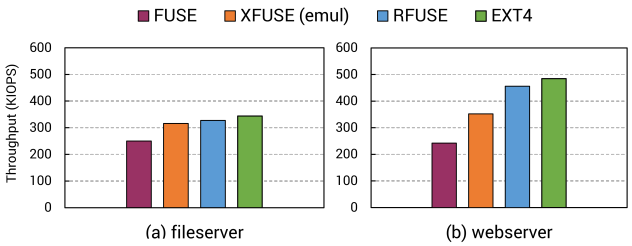

Figure 12: Thoughput of `filebench` workloads

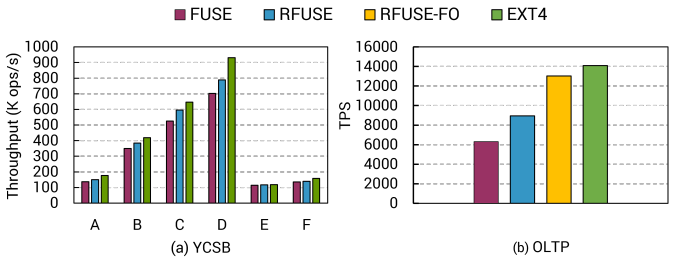

Figure 13: Thoughput of `YCSB` benchmark on RocksDB and `sysbench OLTP` benchmark on PostgreSQL

text switch and wake-up overhead enabled by hybrid polling significantly enhances RFUSE's overall performance.

4.3.3 Metadata Operation Scalability

To evaluate the scalability of RFUSE when performing metadata operations, we ran the `FXMARK` benchmark [29] on StackFS. Table 1 summarizes the details of the `FXMARK` benchmark used in the experiment. We used all the metadata workloads defined in `FXMARK`, with the exception of the RENAME workloads named `MWRL` and `MWRM`, as StackFS does not support RENAME operation. Note that we failed to measure the throughput of `MWUL` and `MWUM` on EXTFUSE, as it resulted in errors in our environment.

Figure 11 depicts the scalability of metadata operations for the evaluated userspace filesystem frameworks. The results show that RFUSE consistently demonstrates superior scalability compared to FUSE and EXTFUSE across the various workloads. RFUSE leverages per-core, NUMA-aware ring channels, enhancing the parallelism of metadata operations and eliminating inter-NUMA accesses, which could lead to high latency. Furthermore, RFUSE's hybrid polling proves particularly effective in metadata operations, because most of these operations can be completed quickly. This allows RFUSE to achieve both high scalability and superior throughput in workloads with contention for shared resources compared to other frameworks.

We note that EXTFUSE shows lower scalability, especially in `MWCL` and `MWCM`. It is possibly due to the key-value maps used for storing custom data structures in EXTFUSE, which may not be designed to scale effectively. For the workloads `MRPL`, `MRPM`, and `MRPH`, all the evaluated frameworks show similar throughput and scalability. This is because these workloads operate on a directory structure with a depth of five, where path-name resolution becomes the primary operation. As this operation heavily depends on the dcache in the VFS layer, there is little variation among the frameworks.

4.4 Macro-benchmarks

Filebench. We performed the `filebench` benchmark [35] on StackFS using predefined workloads, namely, `fileserver` and `webserver`, which contain a mixed set of data and metadata operations. The `fileserver` workload simulates the behavior of a file server that serves files to multiple clients. Files are initially created with a size of 128KB and then expanded through 16KB APPEND operations. We executed the `fileserver` workload using 200,000 files and 50 threads. The `webserver` workload mimics the behavior of a web server that serves web pages and files to clients over the Internet. Files in the `webserver` workload are created with a relatively small size of 16KB. We executed the `webserver` workload with 1.25M files using 100 threads. For both workloads, the unit size of the READ operation was set to 1MB.

We present the results of these workloads in Figure 12. In both workloads, RFUSE outperforms FUSE and XFUSE in throughput and shows performance comparable to EXT4. XFUSE exhibits superior performance compared to FUSE due to increased parallelism and its adaptive waiting strategy. However, XFUSE still suffers from context switching and request copying overhead, resulting in lower throughput compared to RFUSE. RFUSE, on the other hand, leverages communication through a ring channel, effectively eliminating these overheads and achieving higher performance than other frameworks when handling a mixed set of operations.

YCSB. To evaluate an application-level performance of each framework on a real-world workload, we deployed RocksDB [7] on StackFS and measured a throughput using the `YCSB` benchmark [18]. For the `YCSB` workloads in Figure 13(a), we initially load 50M KV pairs and run each `YCSB` workload with a uniform distribution. The results indicate that RFUSE can attain significant performance improvements compared to FUSE, demonstrating throughput akin to EXT4 across all `YCSB` workloads.

OLTP. We also deployed PostgreSQL [21] on StackFS and measured a TPS (Transactions Per Second) using the `sysbench OLTP` benchmark [25]. For the `OLTP` workload, we load 50M rows across 10 tables before running the benchmark. In Figure 13(b), RFUSE demonstrates a 42% higher TPS compared to FUSE. The results indicate that RFUSE can handle transaction processing more effectively compared to FUSE, owing to the enhanced parallelism by per-core ring

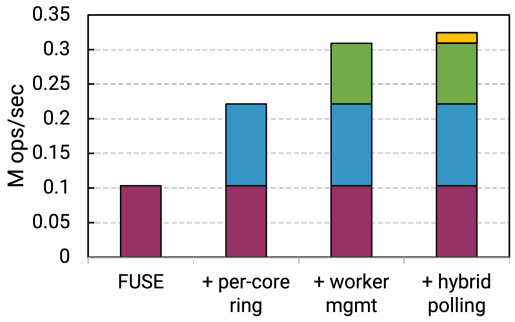

Figure 14: Impact of each technique in RFUSE

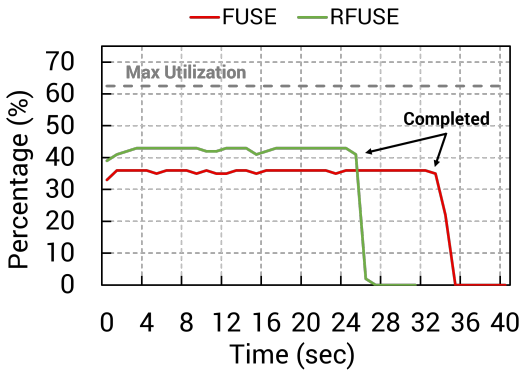

Figure 15: CPU utilization for the `fileserver` workload

channels. We can also observe that the TPS results of the frameworks are relatively lower than EXT4. This is primarily due to frequent FSYNC operations induced by logging in the `OLTP` workload. For the FSYNC operation, both FUSE and RFUSE initially writeback dirty pages of the target file to the userspace daemon and wait for the completion of all pending WRITE requests before dispatching the FSYNC request. This incurs significant processing overhead for OLTP workloads on the frameworks, leading to lower throughput compared to EXT4. To validate this, we measured the performance of RFUSE after turning off the FSYNC option in the PostgreSQL (labeled as RFUSE-FO). The result demonstrates that the TPS of RFUSE-FO is nearly on par with that of EXT4.

4.5 Factor Analysis

To assess the influence of the proposed techniques incorporated into RFUSE, we measured the throughput of the `fxmark:MWCL` workload on StackFS. Figure 14 illustrates how throughput varies as we introduce each technique one by one to FUSE. When we add per-core ring channels, we observe 2.2x higher throughput compared to the native FUSE, thanks to the enhanced parallelism. Furthermore, the management of worker threads with a CPU affinity yields a noteworthy improvement by mitigating inter-NUMA accesses. Finally, applying hybrid polling not only reduces latency but also leads to an observed improvement in throughput, while reducing contention within CPU cores.

4.6 CPU Utilization

Lastly, we measured the CPU utilization while executing the `fileserver` workload used in Section 4.4 on StackFS. Figure 15 displays the variations in CPU utilization for both FUSE and RFUSE. Considering that the `fileserver` workload operates with 50 threads on our 80-core machine, the theoretical maximum CPU utilization is 62.5%.

Owing to its hybrid polling mechanism, RFUSE exhibits roughly 7% higher CPU usage during execution compared to FUSE. However, due to RFUSE's higher throughput on the `fileserver` workload, we can see that RFUSE has a shorter execution time than FUSE. From an energy consumption perspective, despite its architecture based on hybrid polling, RFUSE is thought to consume either less or a comparable amount of energy as FUSE.

5 Related Work

Library Filesystem. A Library Filesystem (*libFS*, for short), provides a set of APIs implementing filesystem functionalities in the form of a user-level library. To access the filesystem service, applications must be directly linked to libFS during compile time. LibFS typically does not provide the standard POSIX interface. Instead, it offers filesystem APIs optimized either for specific application data access patterns or for the underlying storage platforms. Due to these benefits, many distributed filesystems are designed in the form of libFS. Examples include *libhdfs* for the Hadoop Distributed File System (HDFS) [17], *libcephfs* for CephFS [37], and many more tailored for large-scale storage systems [22, 28, 33, 41]

However, using libFS may pose some challenges. Since they do not adhere to any standardized API, applications using the POSIX API cannot directly utilize those filesystems. Also, application developers need to be familiar with the intentions and specifics of the target libFS, complicating its seamless integration with the application. Finally, a change in the filesystem API or the implementation of new funtionality require either rewriting or recompiling the application.

System Call Hooking. Several state-of-the-art NVM (Non-Volatile Memory) filesystems [14, 20, 24, 30] are implemented using a system call hooking mechanism, which allows them to directly access the NVM without going through the kernel by intercepting system calls. This is typically achieved through LD_PRELOAD [32] which is an environment variable provided by the dynamic linker,

allowing users to specify shared libraries to be loaded prior to initiating the program execution. By intercepting *libc* [9] with LD_PRELOAD, one can create a userspace filesystem using a custom library that redefines system call wrappers related to filesystem functions. However, recent studies warn about the pitfalls of implementing a userspace OS subsystem using the LD_PRELOAD hook. For example, zpoline [40] argues that LD_PRELOAD is designed to hook function calls, not system calls. System calls that are internally invoked through the `syscall` or `systenter` instruction in *libc* cannot be successfully hooked by LD_PRELOAD. This can lead to unexpected behaviors, such as FD inconsistency [27, 40], as they disrupt the synchronization between the kernel and userspace subsystems.

Restartable Userspace Filesystem. Although userspace filesystems are easy to use, a crash in a userspace filesystem remains a significant concern. Recovery from a sudden crash requires manual intervention, potentially causing disruption in services to users throughout the recovery period. Re-FUSE [34] introduces extensions into the FUSE framework for transparent and correct filesystem restart following a crash. Moreover, XFUSE [23] not only provides transparent restart capabilities but also supports online upgrades, allowing the integration of new features into the FUSE-based userspace filesystem with minimal service downtime. Such restartability feature enhances the deployment of userspace filesystems in production environments.

6 Conclusion

RFUSE is a userspace filesystem framework supporting scalable kernel-userspace communication. By harnessing per-core, NUMA-aware ring channels, RFUSE minimizes contention between worker threads and achieves high scalability. The ring channels shared between the kernel driver and the RFUSE daemon also enable RFUSE to perform efficient message transmission without the need for request copying. Moreover, a hybrid polling mechanism of RFUSE effectively reduces the costly context switches. Since RFUSE maintains the same set of APIs as FUSE, existing FUSE-based filesystems can be used without any modifications. Our evaluation results shows that RFUSE can seamlessly support modern hardware environment with its superior throughput and high scalability.

Acknowledgments

We would like to thank our shepherd, Youyou Lu, and the anonymous reviewers for their valuable feedback. This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773 and No. RS-2023-00222663), and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korea government (MSIT). This work was also supported in part by a research grant from Samsung Electronics.

References

[1] fio: Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[2] fuse-nfs: A FUSE module for NFS. https://github.com/sahlberg/fuse-nfs.

[3] FUSE passthrough. https://source.android.com/docs/core/storage/fuse-passthrough.

[4] fuse-stackfs. https://github.com/sbu-fsl/fuse-stackfs.

[5] Gluster Docs. https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/.

[6] Non-Volatime Memory (NVM). https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[7] RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb.

[8] S3FS: FUSE-based file system backed by Amazon S3. https://github.com/s3fs-fuse/s3fs-fuse.

[9] Standard C libraries on Linux. https://man7.org/linux/man-pages/man7/libc.7.html.

[10] SwiftFS: a userspace filesystem to mount OpenStack container in Swift. https://github.com/wizzard/SwiftFS.

[11] Zoned Namespace (ZNS) SSDs. https://nvmexpress.org/specification/nvme-zoned-namespaces-zns-command-set-specification/.

[12] eBPF: extended Berkley Packet Filter. https://www.iovisor.org/technology/ebpf, 2017.

[13] NVMe Express Base Specification. https://nvmexpress.org/specifications/, 2017.

[14] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027, 2020.

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

[16] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[17] Dhruba Borthakur et al. HDFS Architecture Guide. *Hadoop apache project*, 53(1-13):2, 2008.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[19] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[21] Joshua D Drake and John C Worsley. *Practical PostgreSQL*. " O'Reilly Media, Inc.", 2002.

[22] Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, and Jiwu Shu. SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 915–928, 2023.

[23] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. XFUSE: An Infrastructure for Running Filesystem Services in User Space. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 863–875, 2021.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[25] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[26] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[27] James Lembke, Pierre-Louis Roman, and Patrick Eugster. DEFUSE: An Interface for Fast and Correct User Space File System Access. *ACM Transactions on Storage (TOS)*, 18(3):1–29, 2022.

[28] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, 2022.

[29] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, 2016.

[30] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. Simurgh: a fully decentralized and secure NVMM user space file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[31] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. ZenFS+: Nurturing Performance and Isolation to ZenFS. *IEEE Access*, 11:26344–26357, 2023.

[32] Kevin Pulo. Fun with ld_preload. In *linux. conf. au*, volume 153, page 103, 2009.

[33] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[34] Swaminathan Sundararaman, Laxman Visampalli, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Refuse to crash with Re-FUSE. In *Proceedings of the sixth conference on Computer systems*, pages 77–90, 2011.

[35] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[36] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on*

File and Storage Technologies (FAST 17), pages 59–72, 2017.

[37] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[38] Matthew Wilcox. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/.

[39] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[40] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, 2023.

[41] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6, 2015.

A Artifact Appendix

A.1 Abstract

RFUSE is a novel userspace filesystem framework that utilizes scalable message communication between the kernel and userspace using a per-core ring channel as a communication channel. This artifact comprises the RFUSE source code and scripts utilized in the benchmarks presented in the paper. RFUSE is implemented by modifying both the user-level library and the kernel driver of FUSE. Furthermore, this appendix provides comprehensive instructions on accessing our artifact and reproducing the results achieved in our research.

A.2 Scope

The following items represent major claims that our artifact allows to validate. For detailed descriptions and insights into the relationship between the artifact and experiments, please refer to `claims.md`

(Claim 1): *For sequential I/O operations, all frameworks and EXT4 show similar throughput due to the aid of page cache in the kernel. For random I/O operations, RFUSE demonstrates higher throughput than FUSE due to the hybrid polling mechanism in reducing context switch and wake-up overhead.*

(Claim 2): *RFUSE scales well for common data operations due to its utilization of per-core ring channels.*

(Claim 3): *RFUSE scales well for common metadata operations due to enhancing the parallelism of metadata operations and eliminating inter-NUMA accesses. For `MRPL`, `MRPM` and `MRPH` workloads, all frameworks and EXT4 show similar scalability due to the aid of dcache in the kernel.*

(Claim 4): *For `filebench` macro workloads, RFUSE outperforms FUSE and shows performance comparable to EXT4, which indicate that RFUSE is well-suited for handling a mixed set of operations.*

(Claim 5): *RFUSE demonstrates shorter latency than FUSE on NullFS due to the reduction of communication overheads.*

A.3 Contents

The submitted artifact consists of 5 components:

1. The *kernel drivers*, which contain the kernel driver codes for both FUSE and RFUSE.
2. The *user-level libraries*, which contain the user-level library for both FUSE and RFUSE.
3. The *linux kernel* source code (v5.15.0).
4. The *filesystems*, which include the source code of NullFS and StackFS.
5. The *benchmarks*, which are used in the experiments in the paper.

A.4 Hosting

The source code of RFUSE is publicly available at `https://github.com/snu-csl/rfuse` and the latest version of RFUSE is uploaded on the *master* branch.

A.5 Requirements

A.5.1 Hardware Requirements

We evaluated RFUSE on the machine equipped with Fadu Delta PCIe 4.0 SSD and 80 logical cores. For machines with older PCIe generation devices and the small number of cores, the benchmarks may not show similar results we present in the paper, but we believe the overall trends should be similar.

A.5.2 Software Requirements

We developed the RFUSE kernel driver compatible with Linux kernel version 5.15.0. To ensure the correct compilation of our artifact, please verify that your machine's kernel version matches v5.15.0.

All provided instructions are tailored for the Ubuntu OS distribution. If you intend to utilize a different Linux distribution, adjust the environment setup instructions based on the specific distribution you are using.

A.6 Set-up

This section provides concise instructions for setting up the environment and installing RFUSE from scratch. For comprehensive details including steps for mounting user-level filesystems, please refer to `README.md`.

1. Git clone our repository. The rest of the instructions assume you are in the project directory.
2. Install the Linux kernel v5.15.0 and reboot using the installed Linux kernel.
   (a) cd linux && make menuconfig
   (b) Configure `CONFIG_FUSE_FS`=m
   (c) make-kpkg –initrd –revision=1.0 kernel_image kernel_headers
   (d) cd .. && dpkg -i *.deb
   (e) Update grub to load the kernel v5.15.0 and reboot.
3. Configure the number of ring channel as the number of CPU cores in the machine.
   (a) vi lib/libfuse/include/rfuse.h driver/rfuse/rfuse.h

(b) Change the value of `RFUSE_NUM_IQUEUE` in each file to the number of cores in machine.

4. Compile and install the user library and kernel driver of RFUSE.

   (a) cd lib/librfuse && ./librfuse_install.sh

   (b) cd driver/rfuse && make && ./rfuse_insmod.sh

5. Add the location of the library to tell the dynamic link loader where to search for the library.

A.7 Experiments

For artifact evaluation, we have provided convenient scripts to execute the benchmarks used in our experiments. Please refer to `bench/README.md` for detailed instructions. Note that this guideline assumes the use of a machine with an additional storage device for conducting the experiments.