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

Linux/Android FS/MM/Storage Workshop

Kyu-Jin Cho

Seoul National University

Transition of Filesystem Design

- File system designs have evolved in response to technology
  - High-performance storage devices
  - Large number of cores in the machine

Transition of Filesystem Design

- File system designs have evolved in response to technology
  - High-performance storage devices
  - Large number of cores in the machine

- In-kernel vs. Userspace

In-kernel Filesystem

- Native performance

- Low safety from crash
- Complex kernel interface
- Hard to add new functionality

Transition of Filesystem Design

- File system designs have evolved in response to technology
  - High-performance storage devices
  - Large number of cores in the machine

- In-kernel vs. Userspace

In-kernel Filesystem

- Native performance

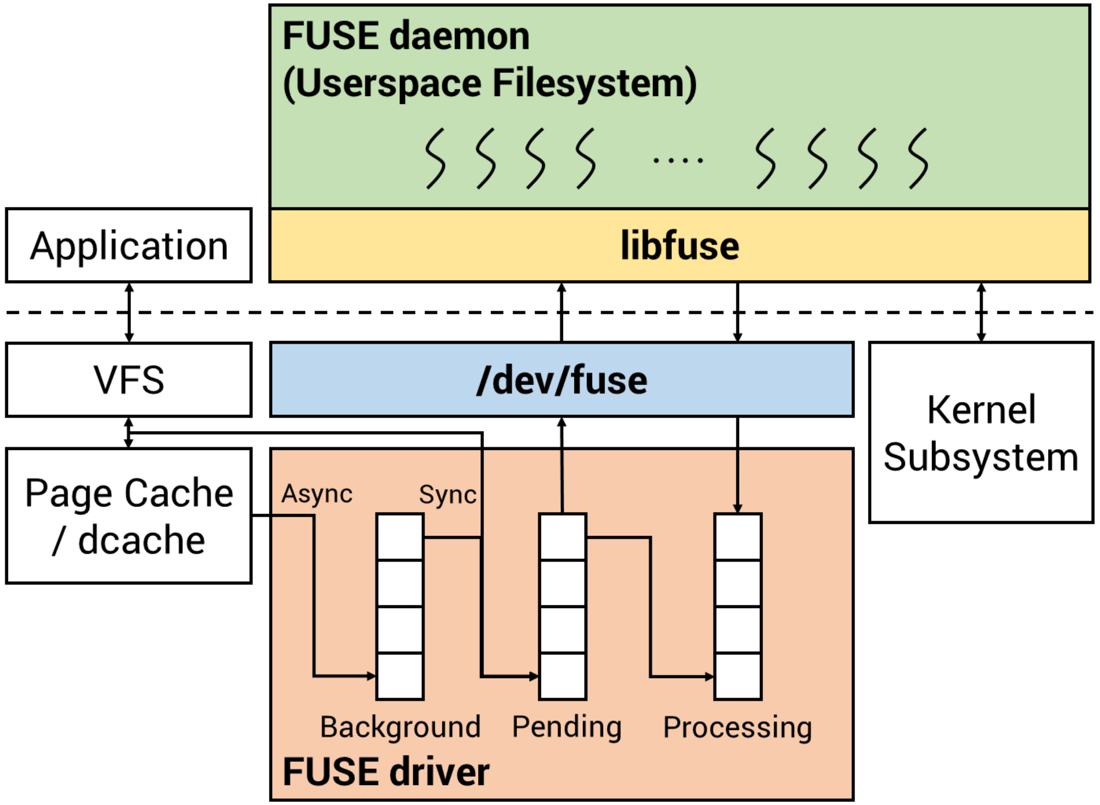
- Low safety from crash
- Complex kernel interface
- Hard to add new functionality

Userspace Filesystem

- High safety from crash
- Easy to maintain and develop
- High portability

FUSE (Filesystem in Userspace)

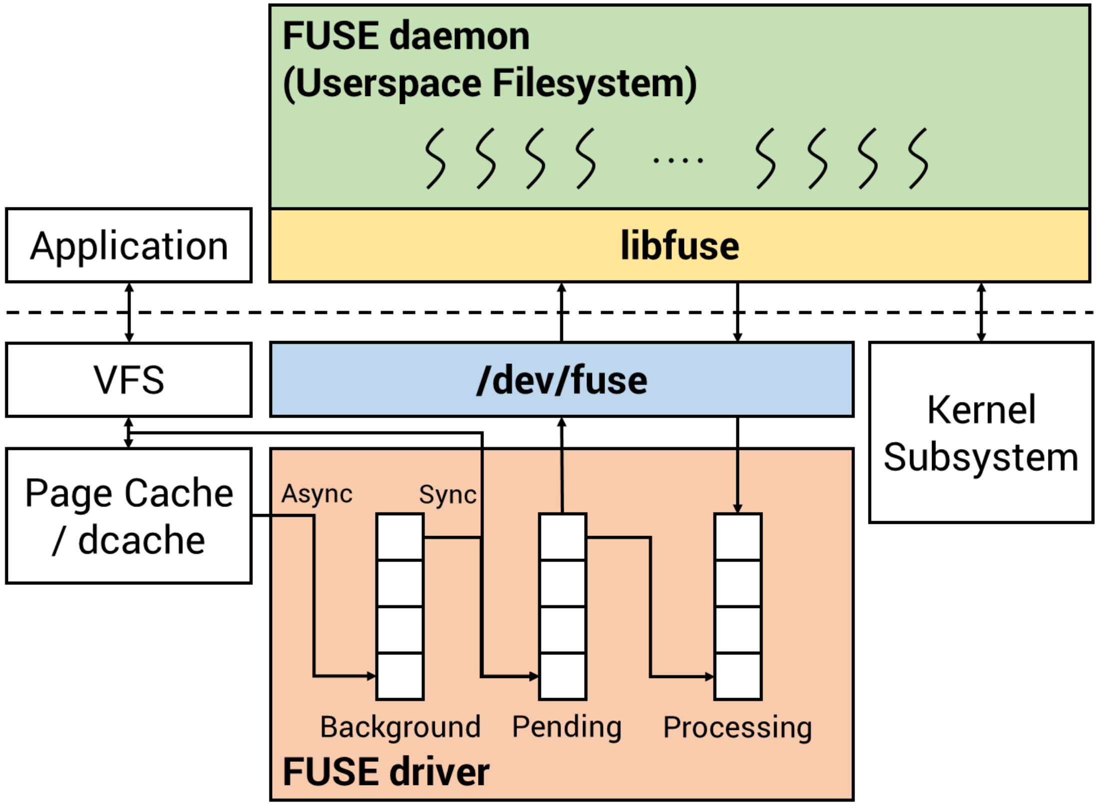
- FUSE consists of two main components:
  - ***FUSE driver*** within the kernel
  - ***FUSE daemon*** within the userspace



FUSE (Filesystem in Userspace)

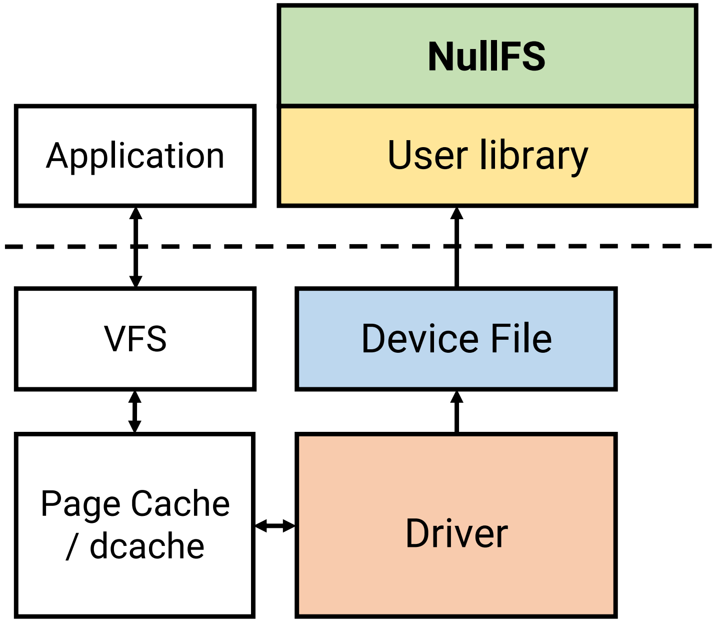
- FUSE consists of two main components:
  - ***FUSE driver*** within the kernel
  - ***FUSE daemon*** within the userspace

- FUSE driver has 5 types of queues:
  - ***Pending*** queue for synchronous requests
  - ***Background*** queue for asynchronous requests
  - ***Processing*** queue for in-flight requests
  - ***Interrupt*** queue
  - ***Forget*** queue

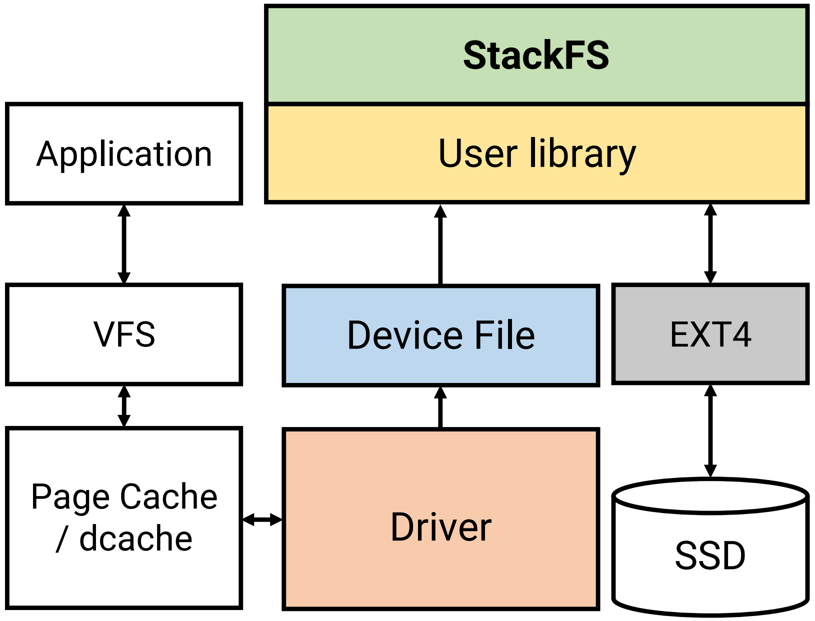


FUSE-based Filesystem

- Userspace filesystem tested
  - **NullFS**: a very simple filesystem which only supports the LOOKUP on the root directory
  - **StackFS**: a stackable filesystem that forwards incoming filesystem operations to an underlying in-kernel filesystem

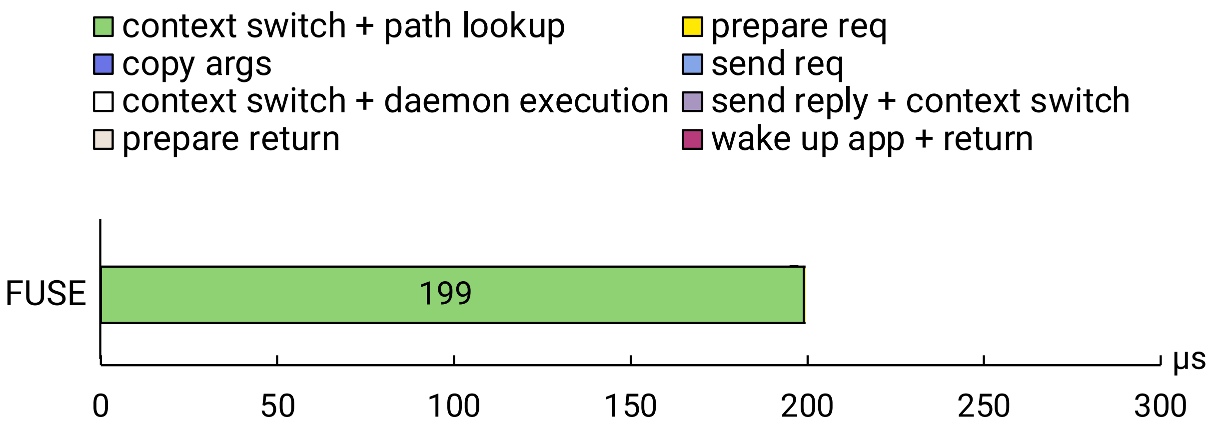


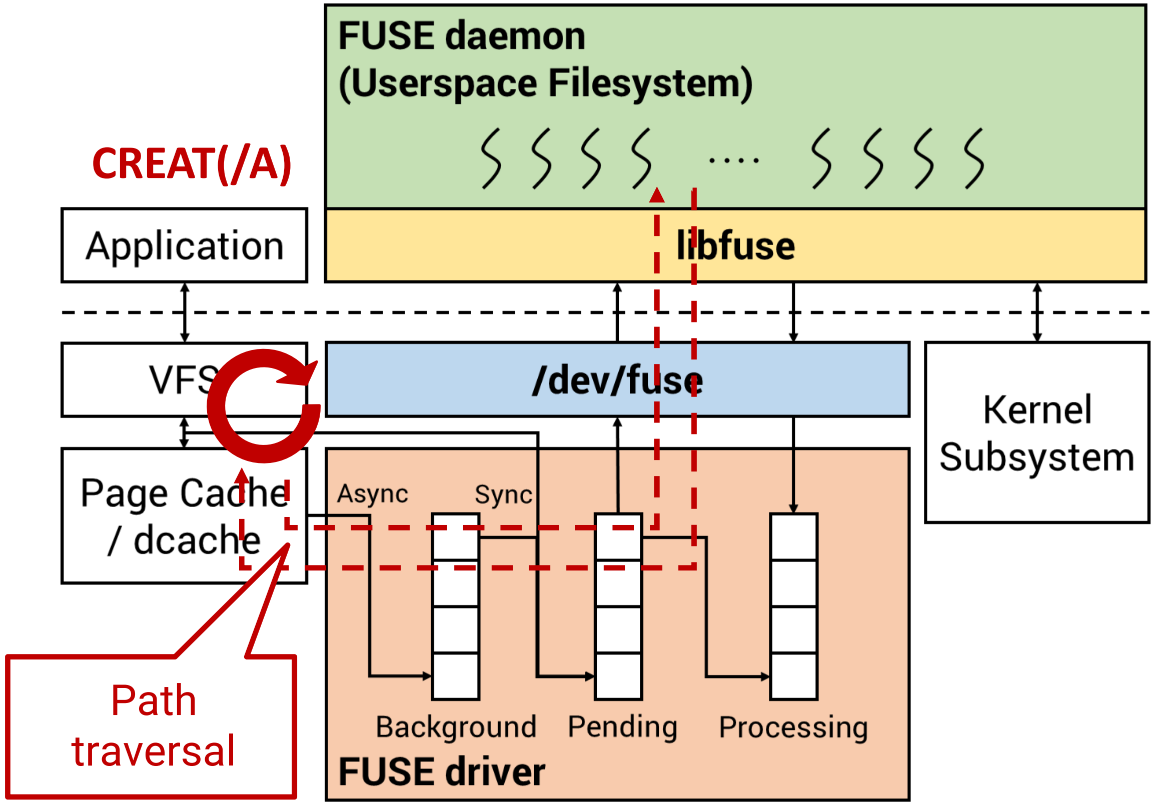
<NullFS>



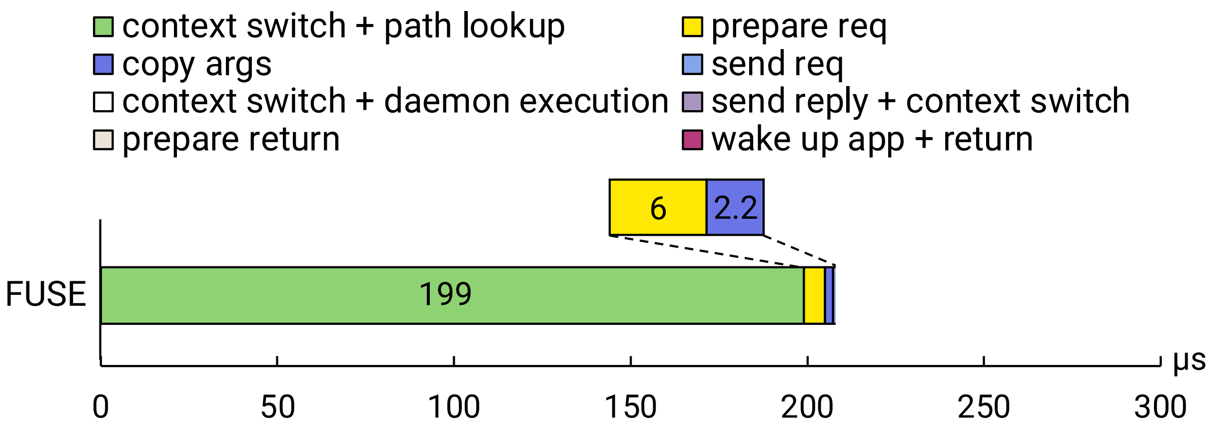
<StackFS>

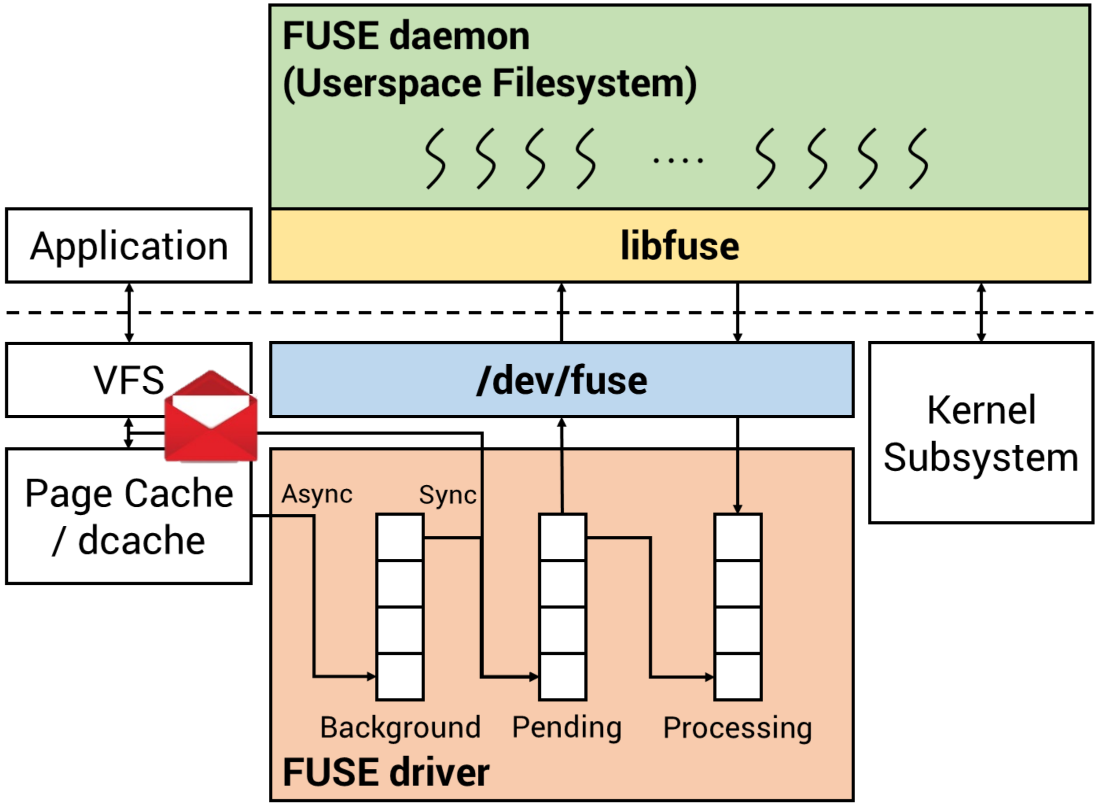
Overhead #1: Latency of FUSE



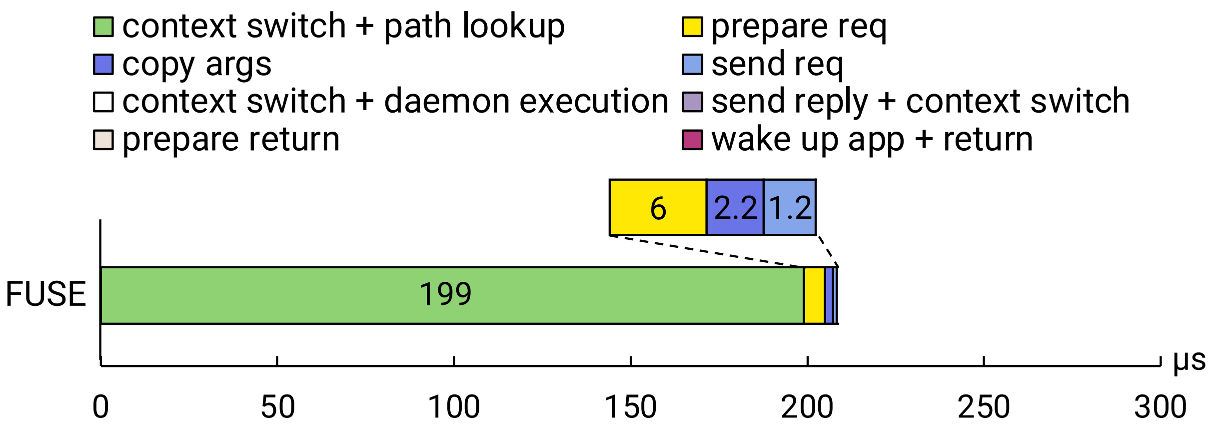


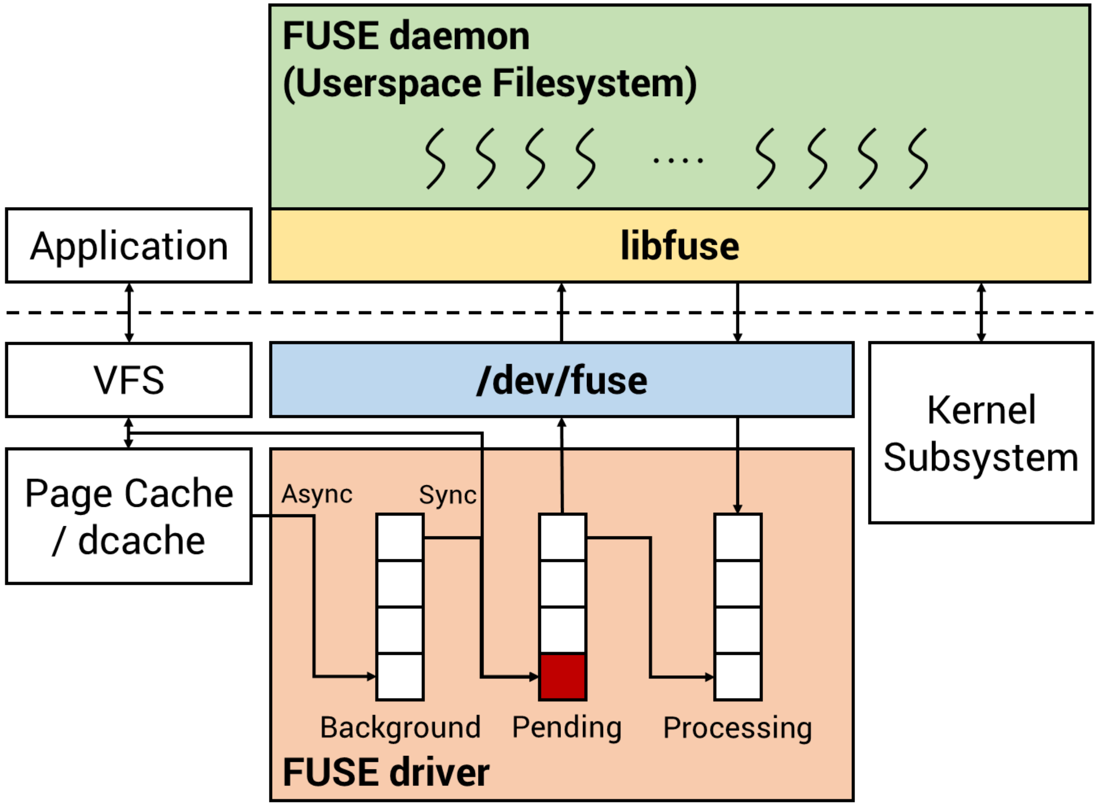
Overhead #1: Latency of FUSE



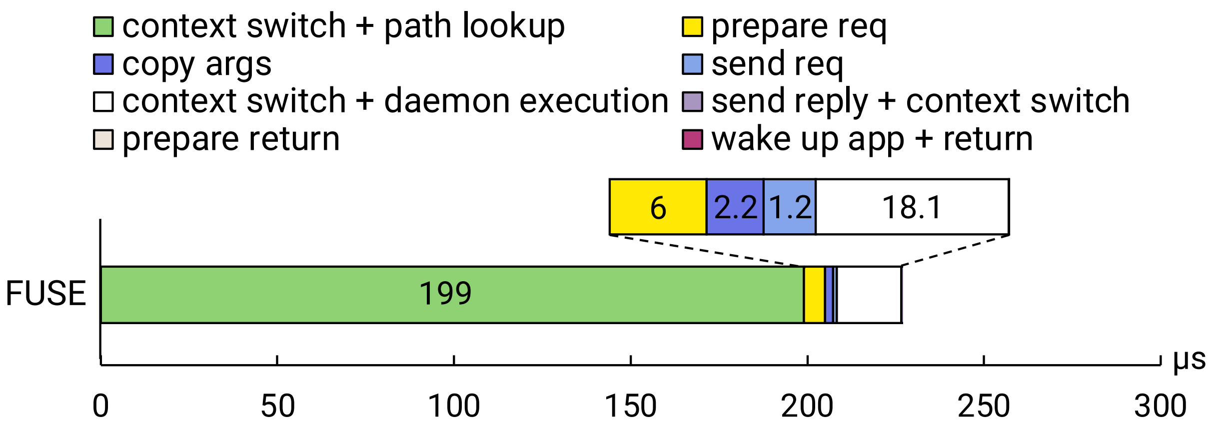


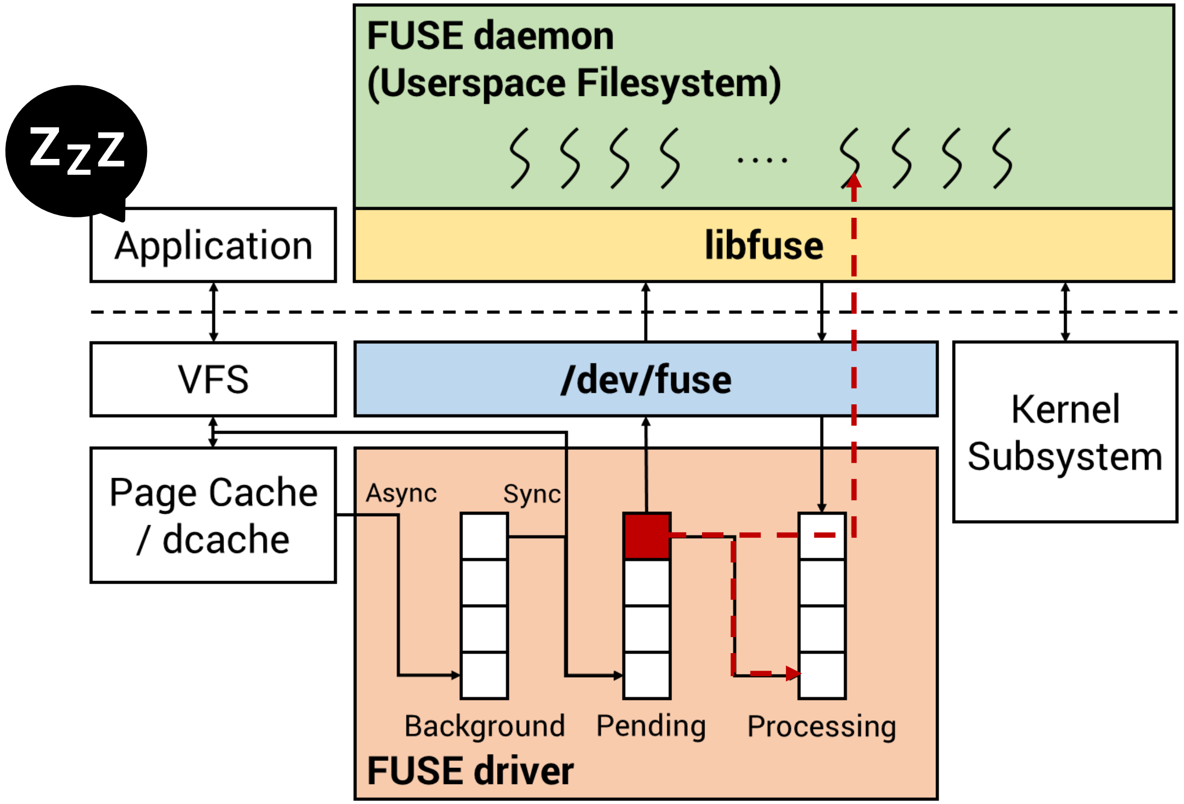
Overhead #1: Latency of FUSE



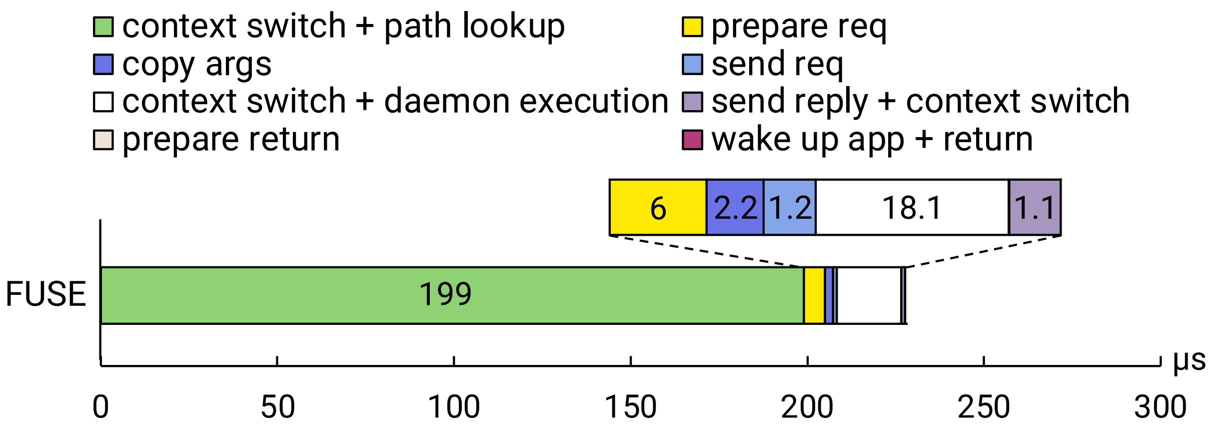


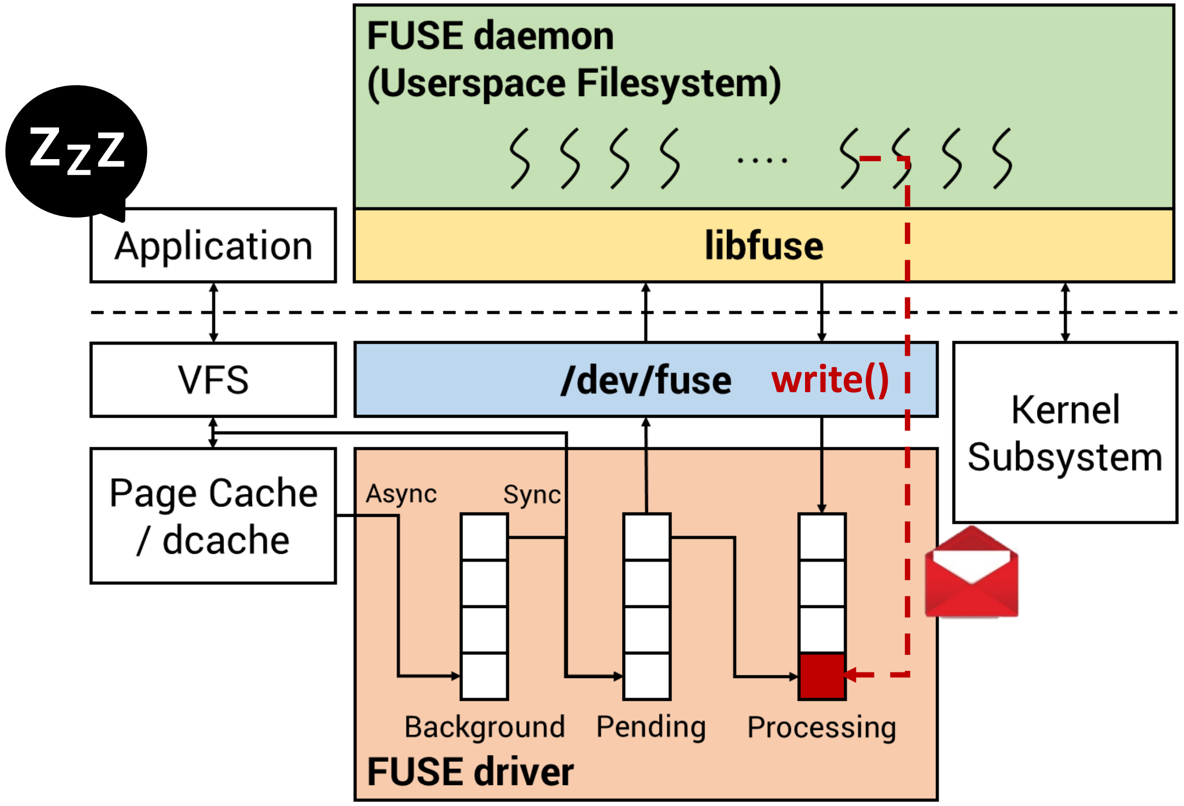
Overhead #1: Latency of FUSE



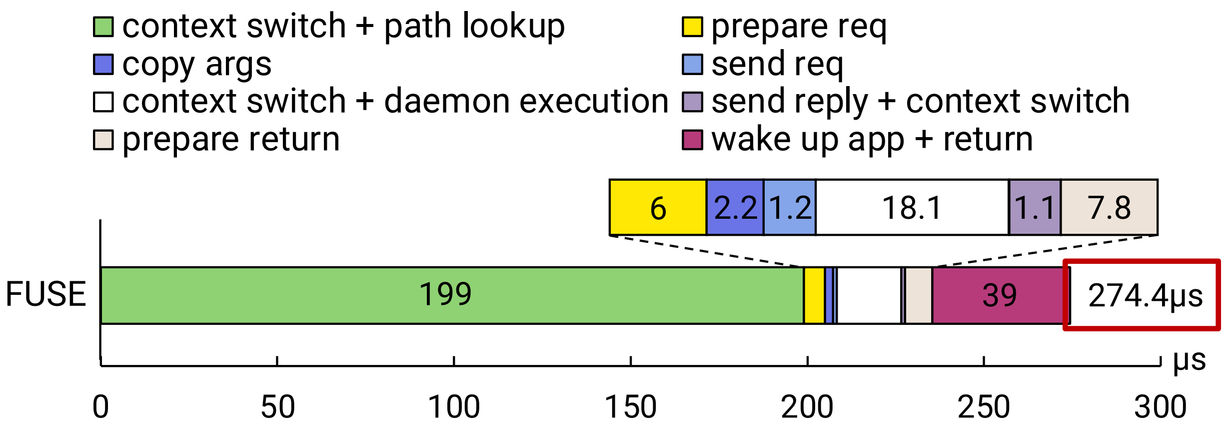


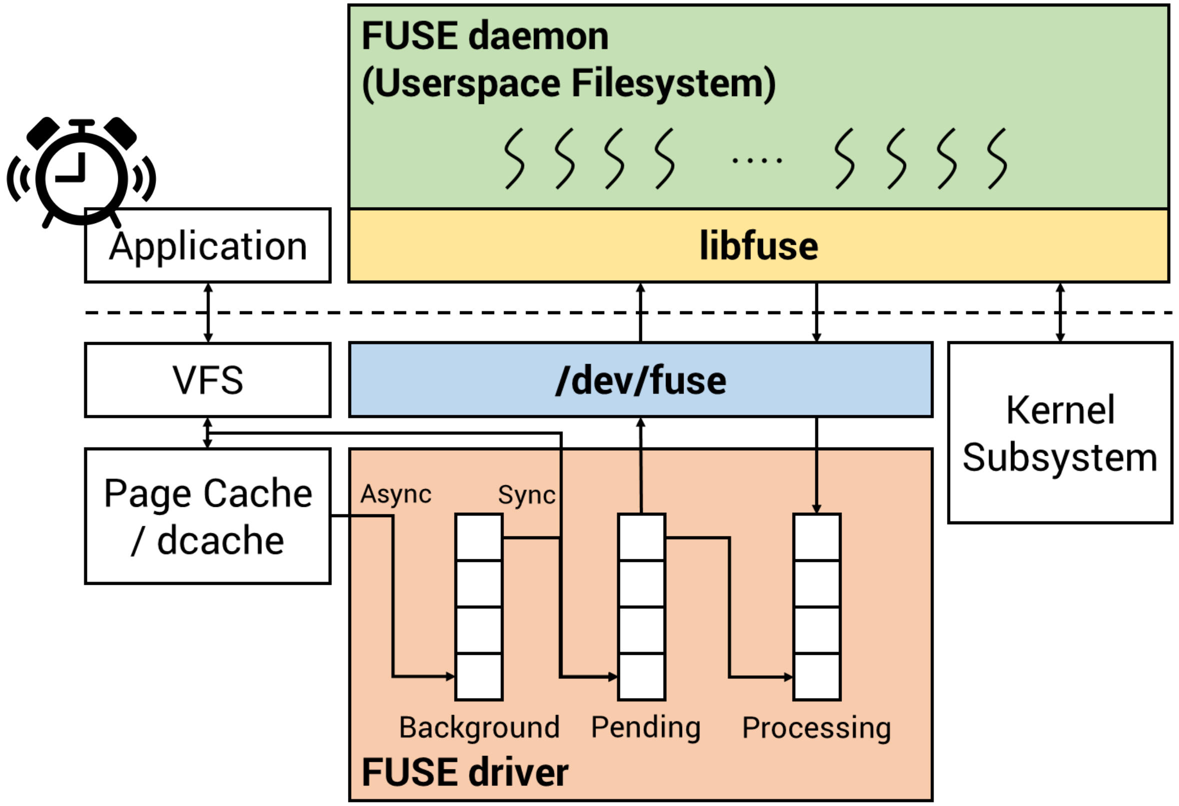
Overhead #1: Latency of FUSE





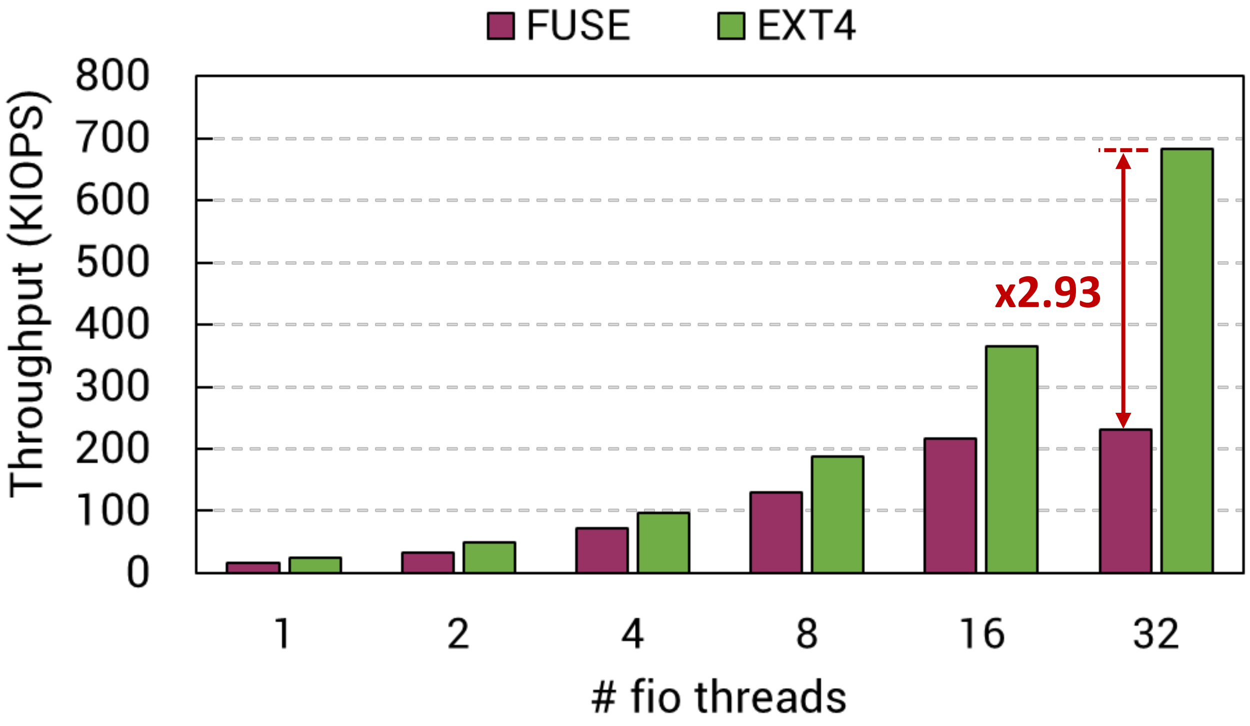
Overhead #1: Latency of FUSE





Overhead #2: Scalability Issue

- A single pending queue in FUSE fails to harness the full throughput potential of a high-performance device



<Scalability of random read on StackFS over EXT4 (FUSE) vs. native EXT4>

RFUSE

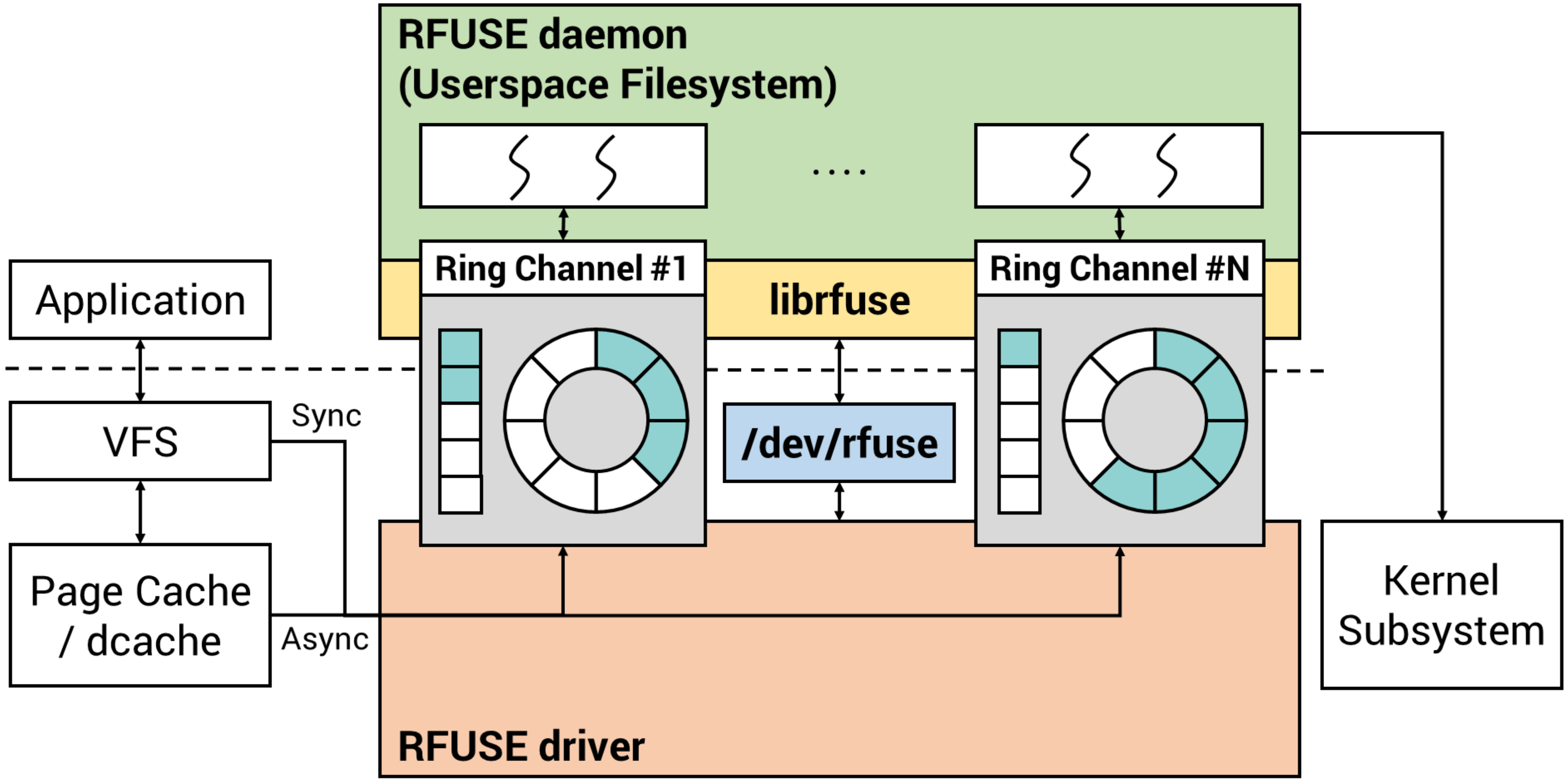
- A userspace filesystem framework designed to support a modern hardware environment with high-performance and scalability

1. **Scalable kernel-userspace communication**
   - Per-core, NUMA-aware ring channels
   - Worker thread management

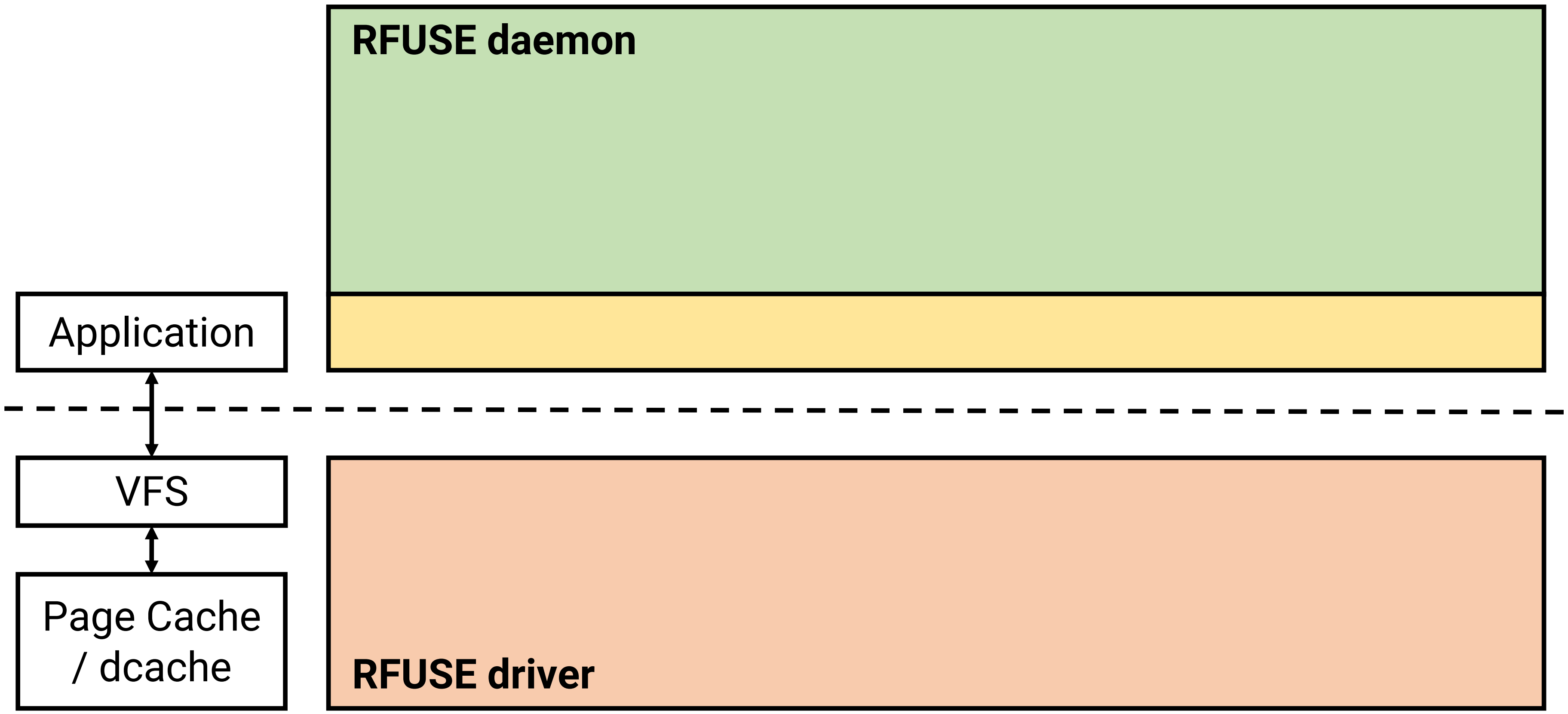
2. **Efficient request transmission**
   - Hybrid polling
   - Load balancing of asynchronous requests

3. **Full compatibility with existing FUSE-based filesystems**

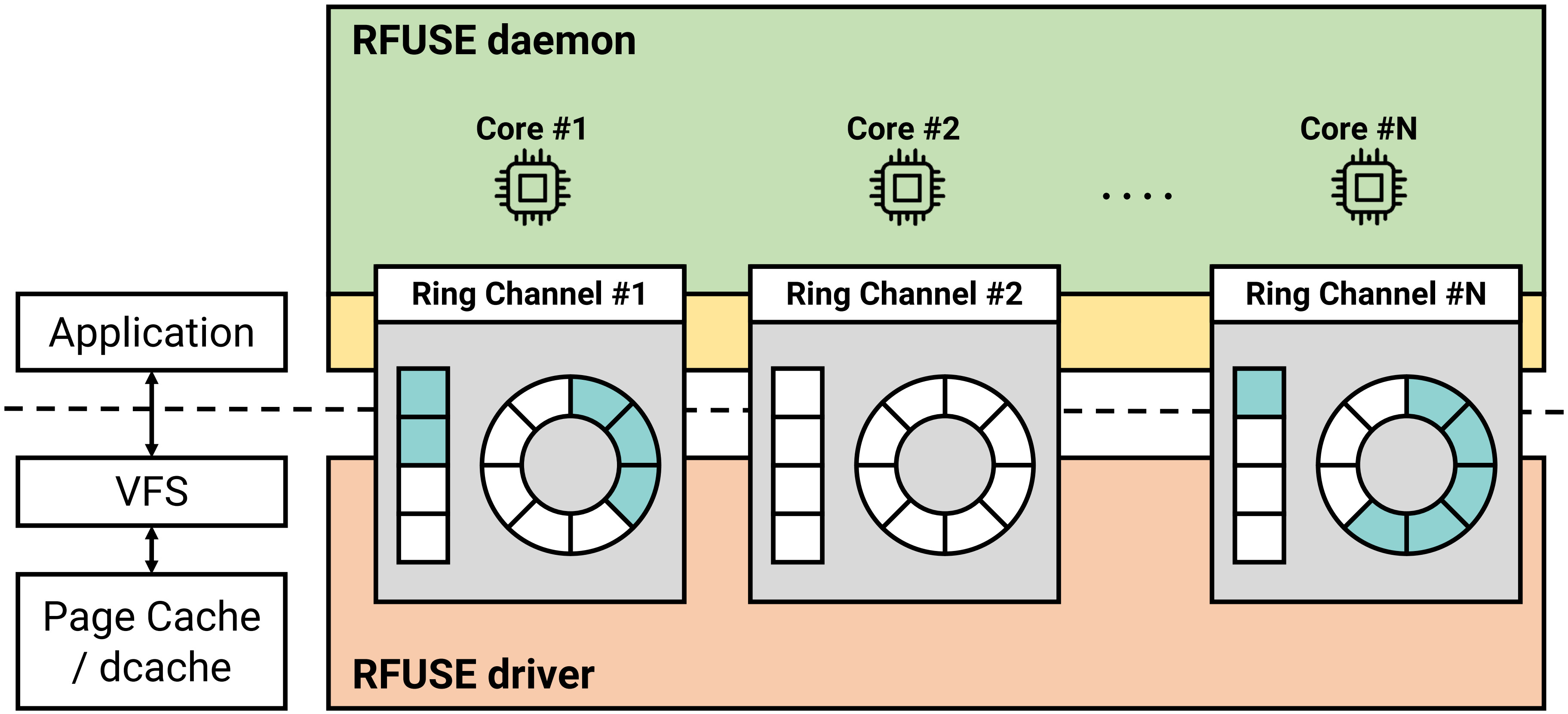
RFUSE Architecture



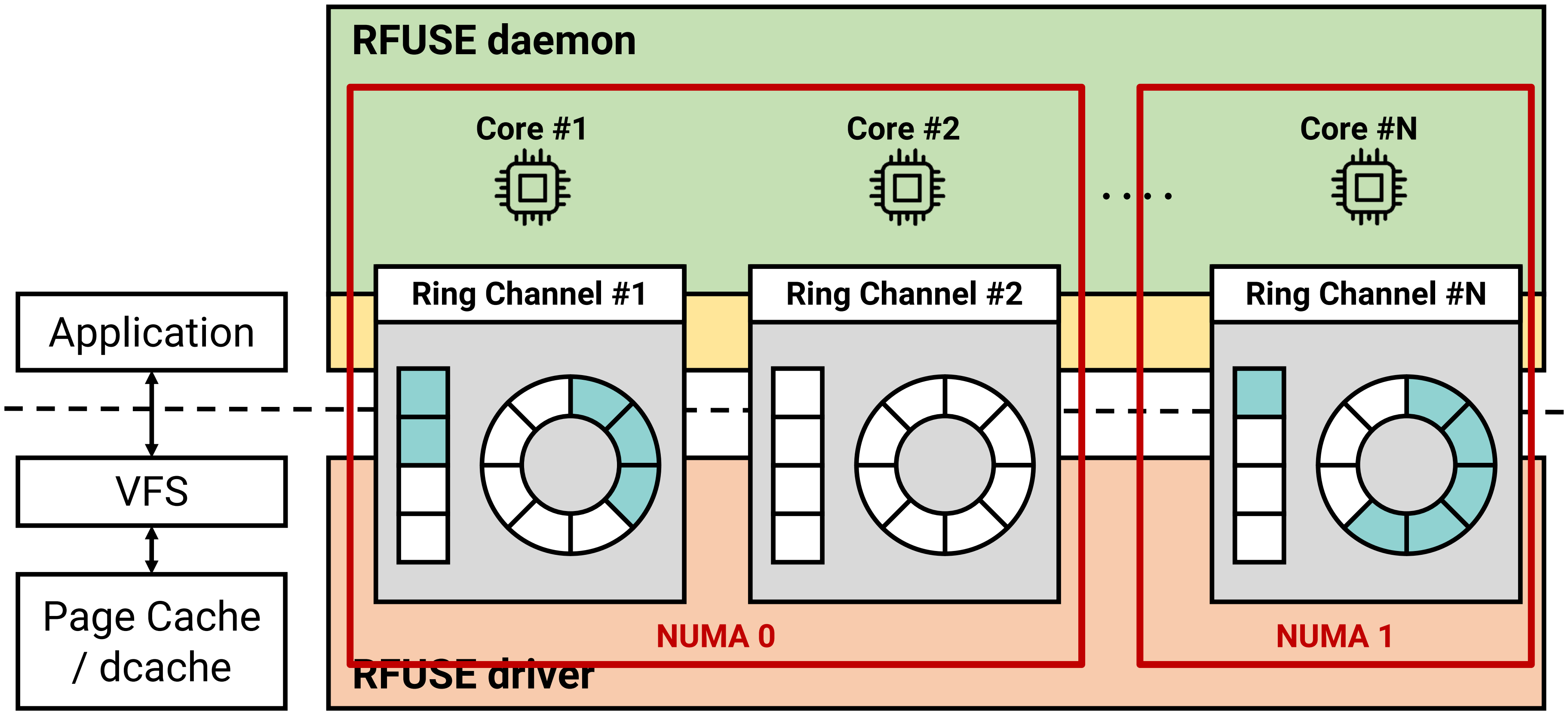
Scalable Communication



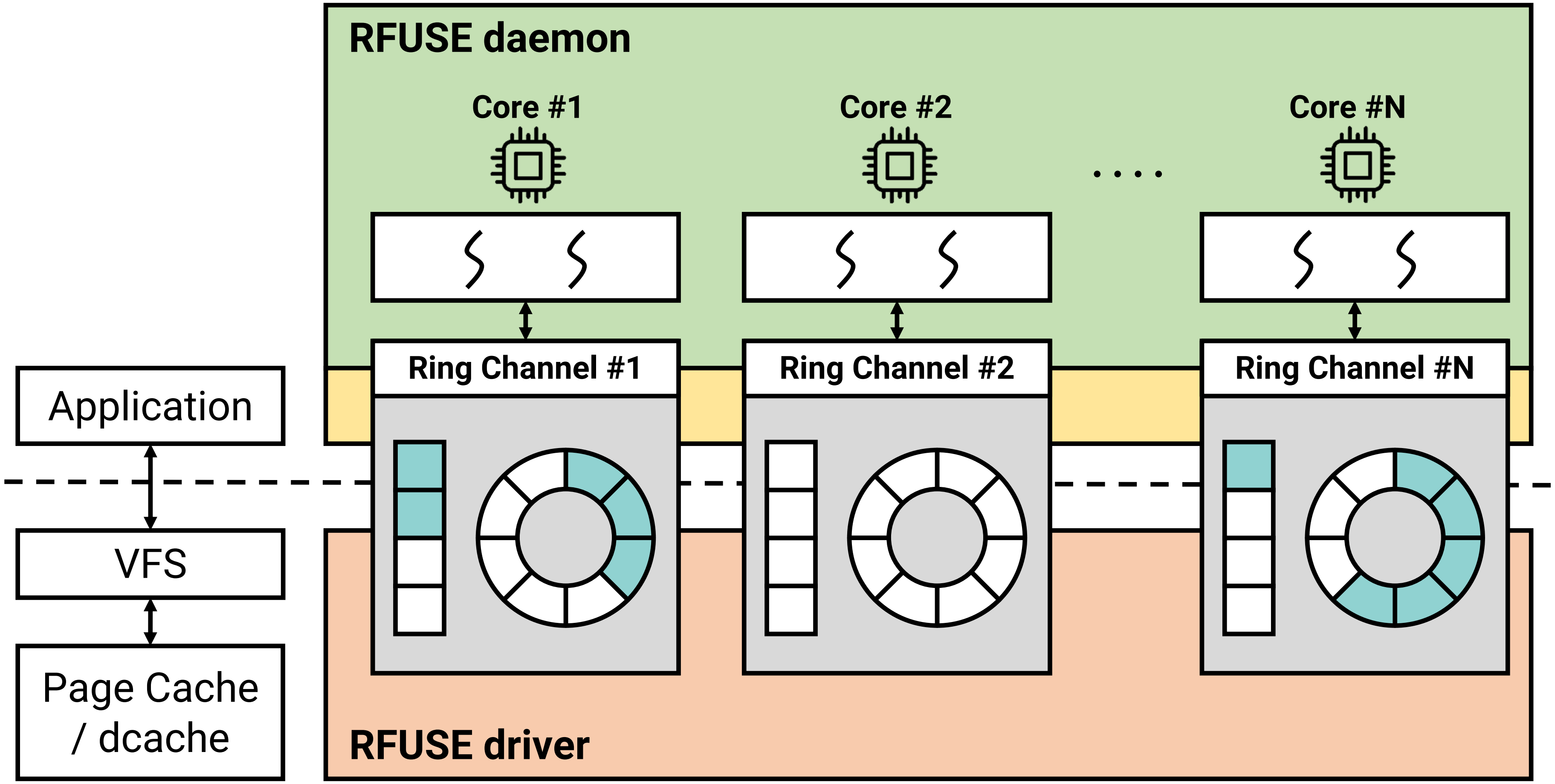
Scalable Communication



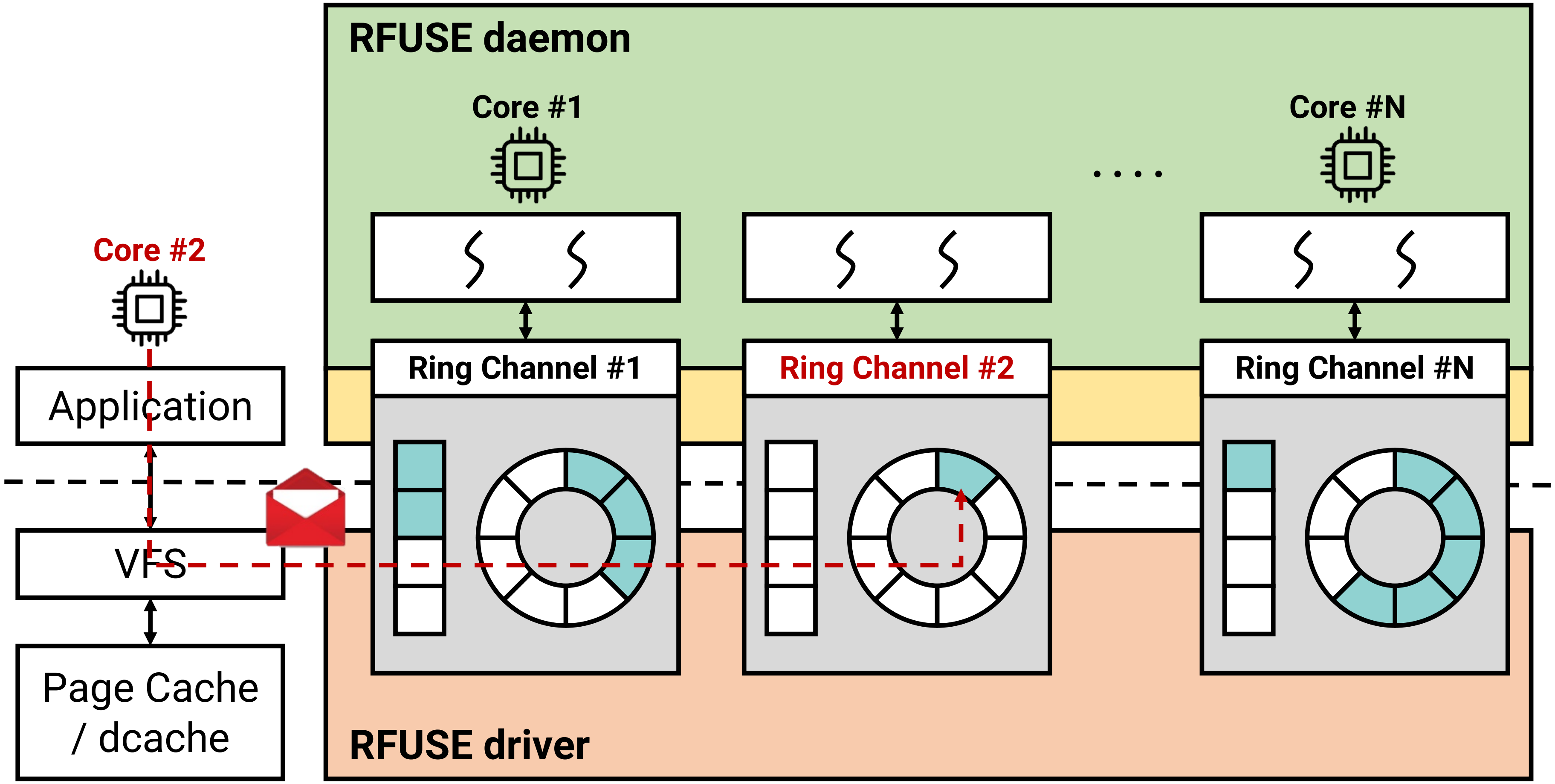
Scalable Communication



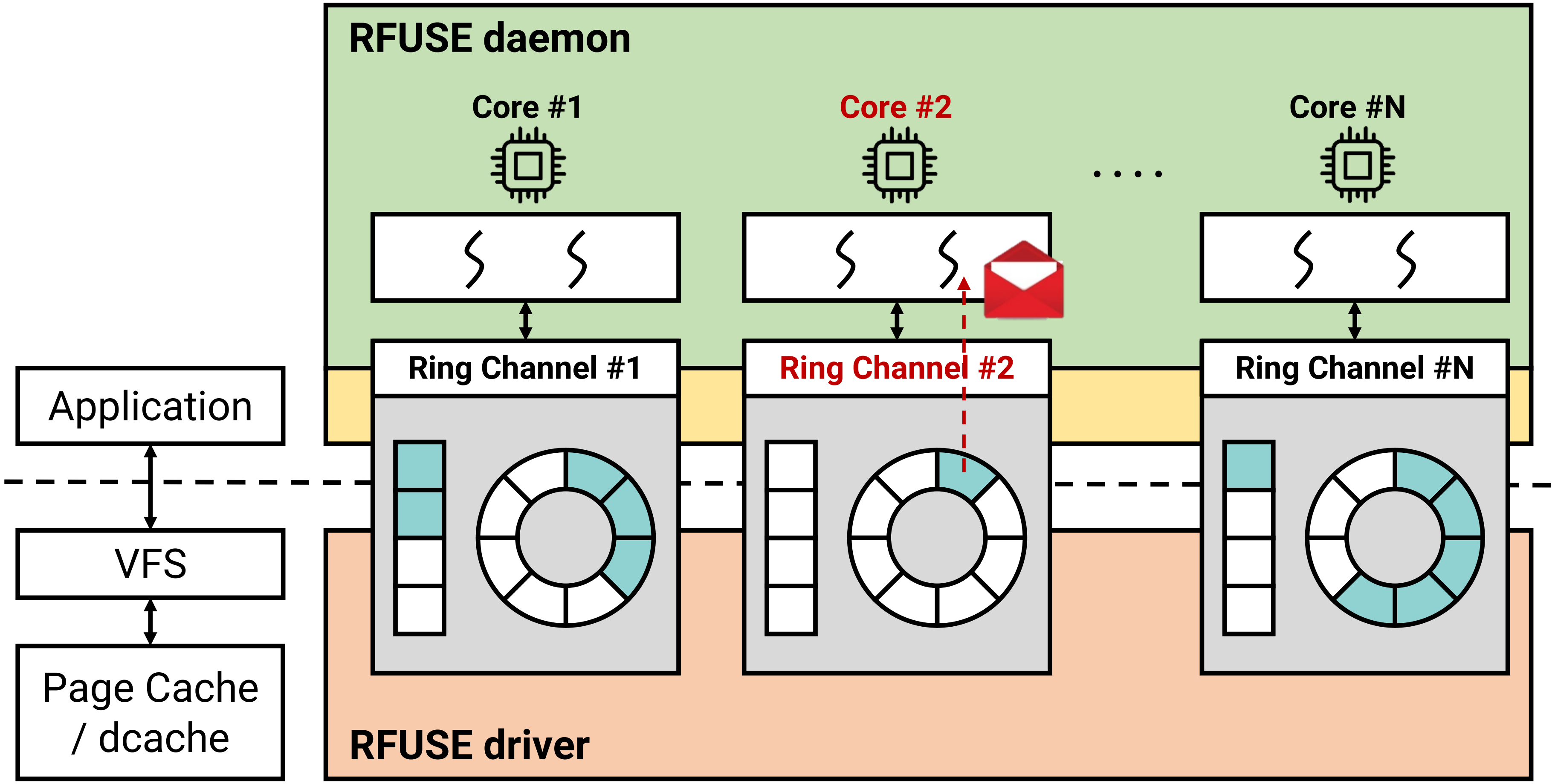
Scalable Communication



Scalable Communication

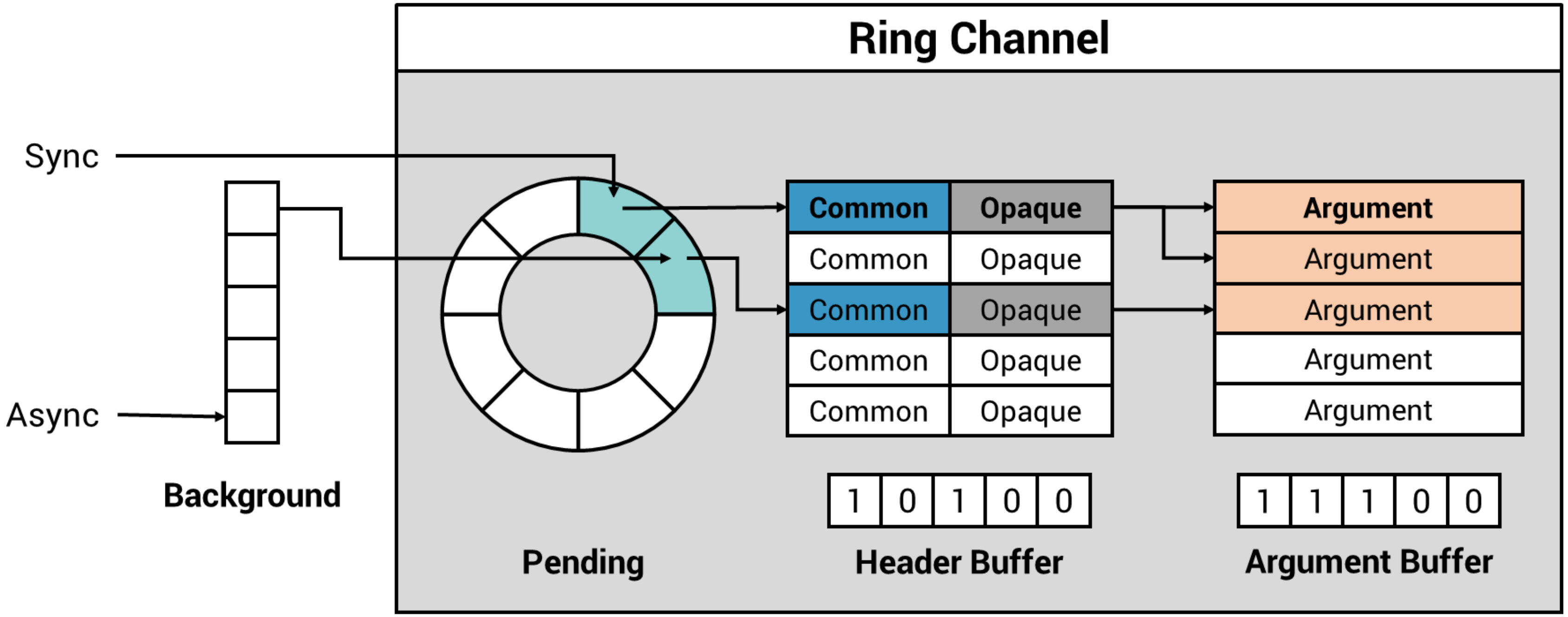


Scalable Communication



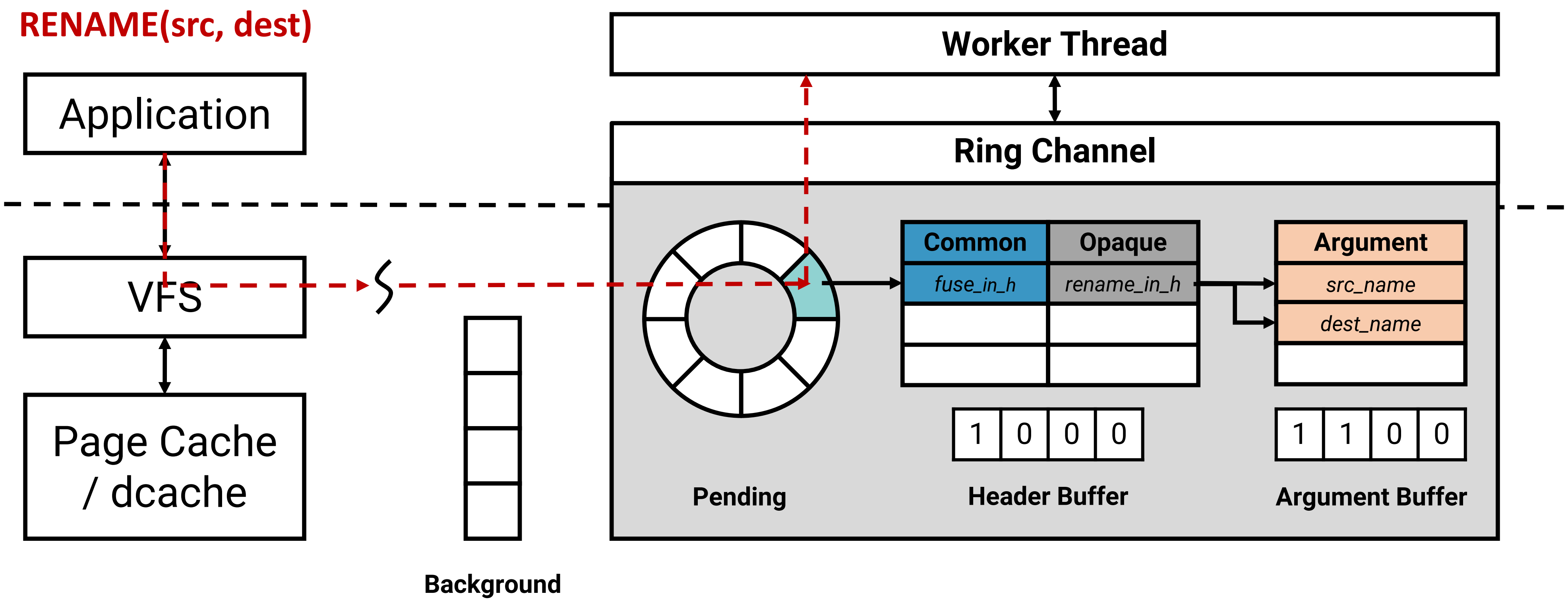
Efficient Request Transmission

- RFUSE utilizes the ring buffer structure similar to the *io_uring* interface, specifically to meet the needs of the FUSE framework.



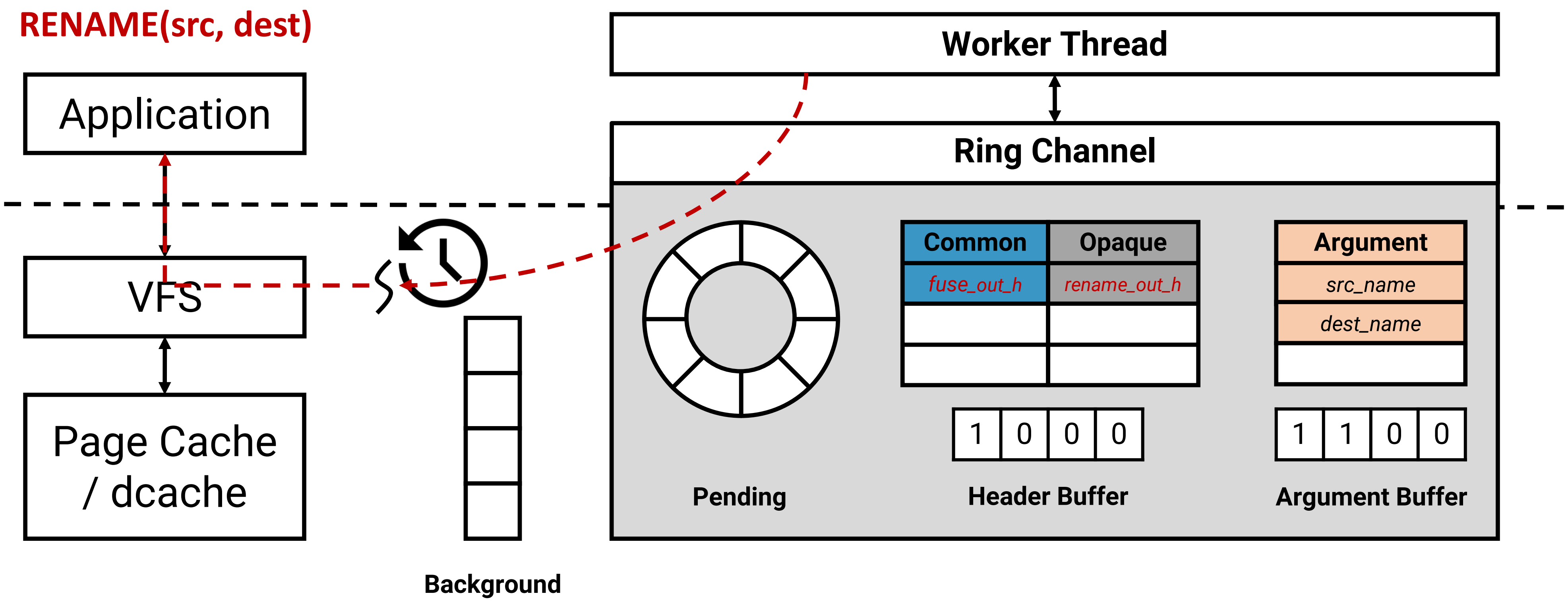
Efficient Request Transmission

- Hybrid polling mechanism



Efficient Request Transmission

- Hybrid polling mechanism



Full Compatibility with FUSE

- The modifications to make use of the ring channels:
  - The FUSE kernel driver
  - The layer of libfuse that handles message communication

- No modifications of all FUSE APIs exposed to developers
  - Both high-level FUSE API and low-level FUSE API
  - Splicing I/O interface

```
struct fuse_operations {
        .getattr     = …
        .readlink    = …
        .mkdir       = …
        ... }
```

```
struct fuse_lowlevel_ops {
        .init         = …
        .destroy      = …
        .lookup       = …
        ... }
```

- **Users do not need to rewrite their FUSE-based filesystem code when using RFUSE.**

Evaluation Setup

- Hardware Setup

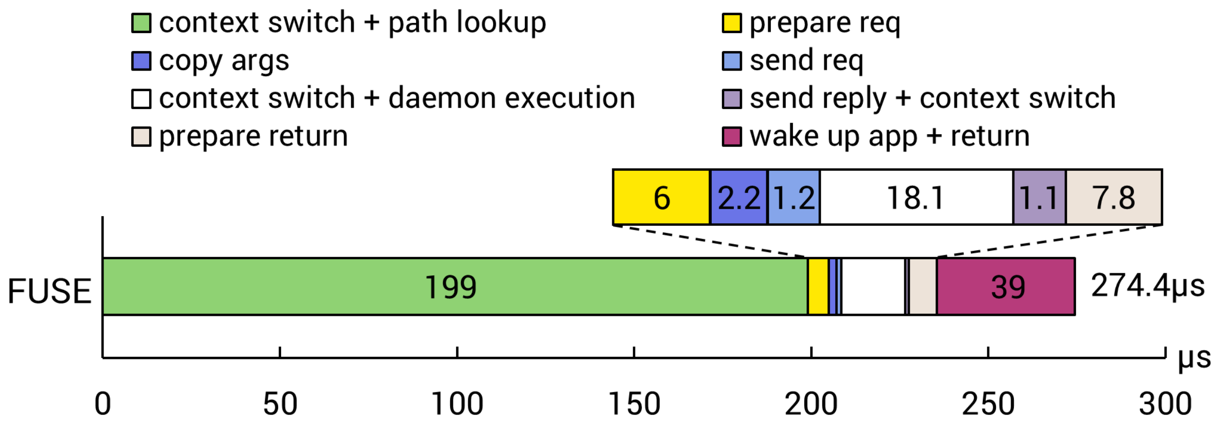
Machine	Dell PowerEdge R750xs
CPU	2 x Intel(R) Xeon(R) Silver 4316 CPUs (80 logical cores)
DRAM	DDR4 256GB
Disk	2TB Fadu Delta PCIe 4.0 SSD
OS	Ubuntu 20.04.3 LTS
Linux Kernel	v5.15.0

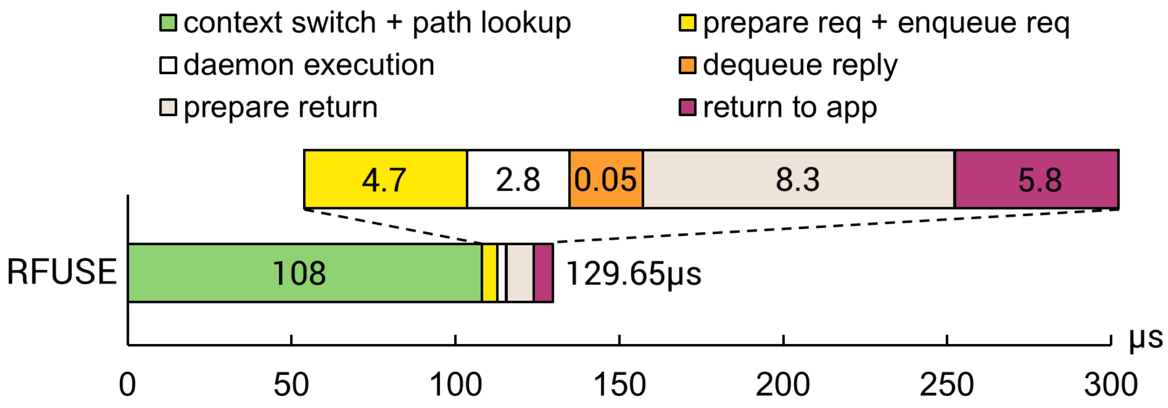
- Frameworks compared
  - FUSE
  - EXTFUSE [1] : Extended FUSE using eBPF
  - XFUSE [2] : FUSE with multiple pending queue (emulation)

[1] *Ashish Bijlani, et al.* Extension Framework for File Systems in User space, USENIX ATC '19
[2] *Qianbo Huai*, et al. XFUSE: An Infrastructure for Running Filesystem Services in User Space, USENIX ATC '21

Latency Breakdown

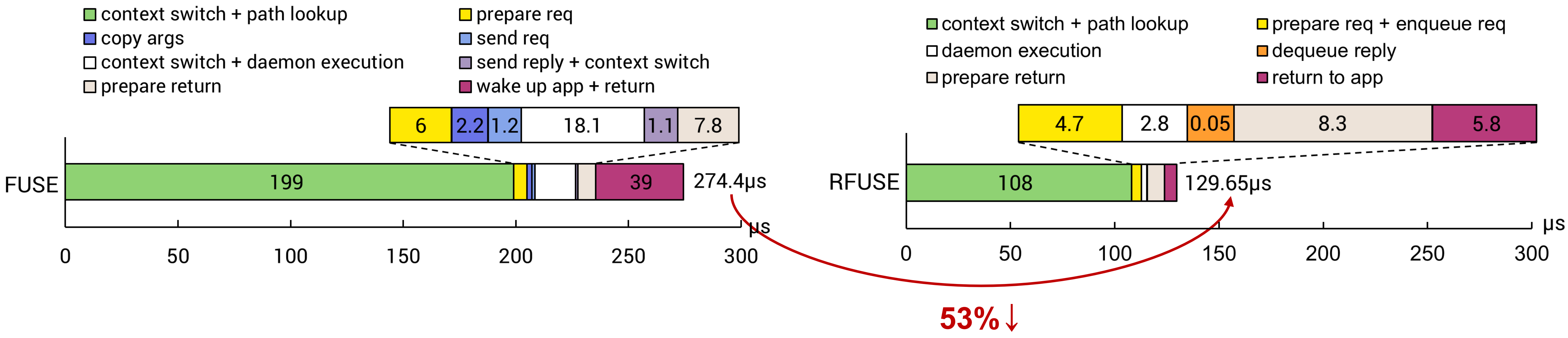
- CREAT() on root directory of NullFS, which promptly returns without performing any action





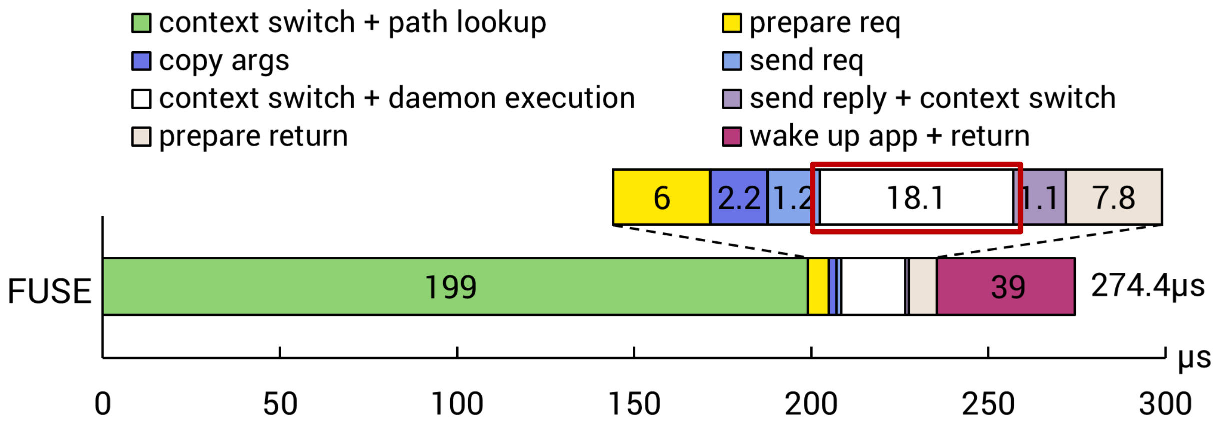
Latency Breakdown

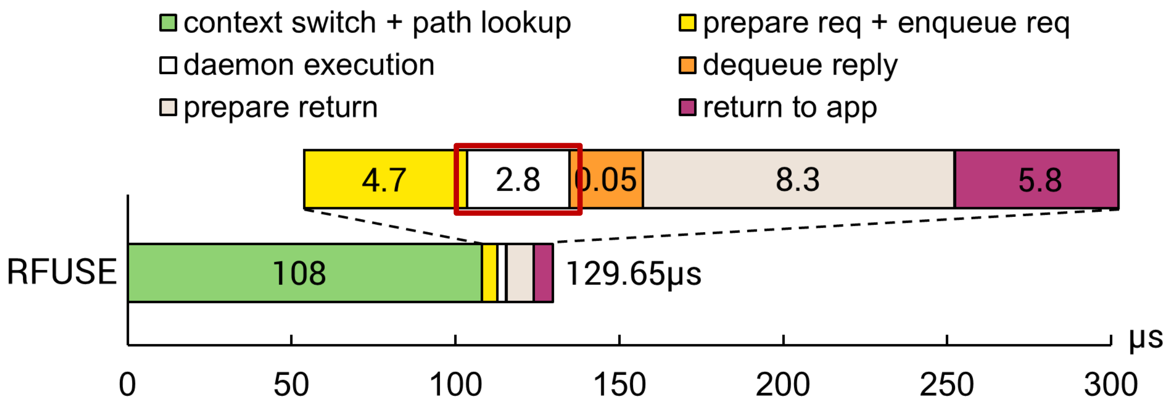
- CREAT() on root directory of NullFS, which promptly returns without performing any action
- RFUSE demonstrates a 53% lower latency than FUSE



Latency Breakdown

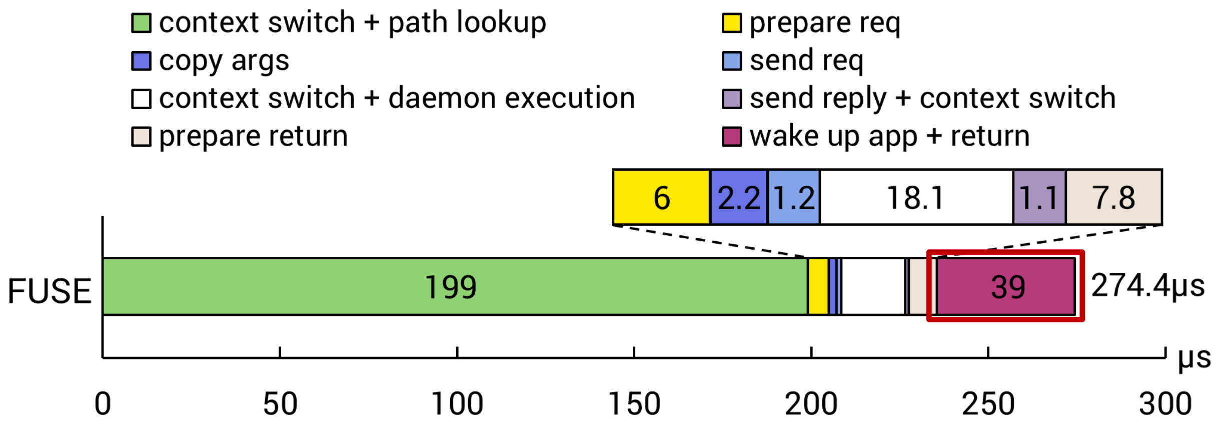
- CREAT() on root directory of NullFS, which promptly returns without performing any action
- RFUSE demonstrates a 53% lower latency than FUSE
  1. No context switches when processing requests and replies

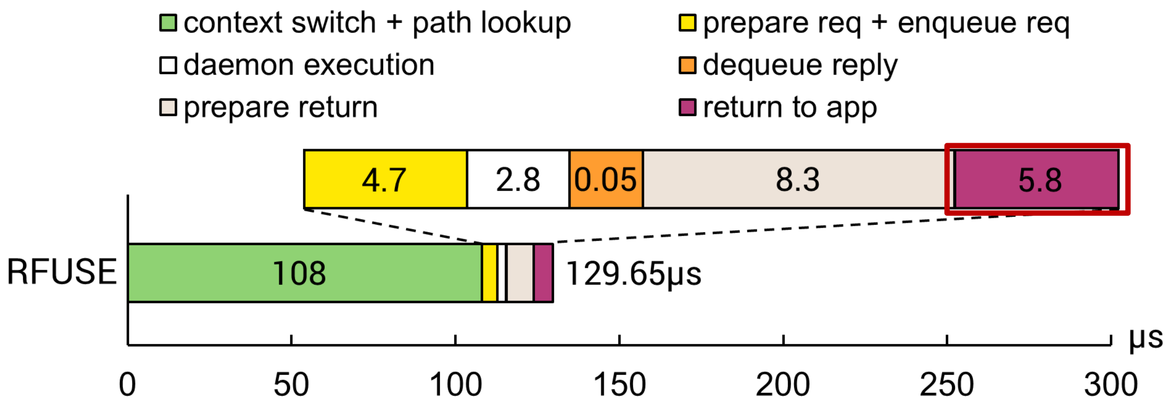




Latency Breakdown

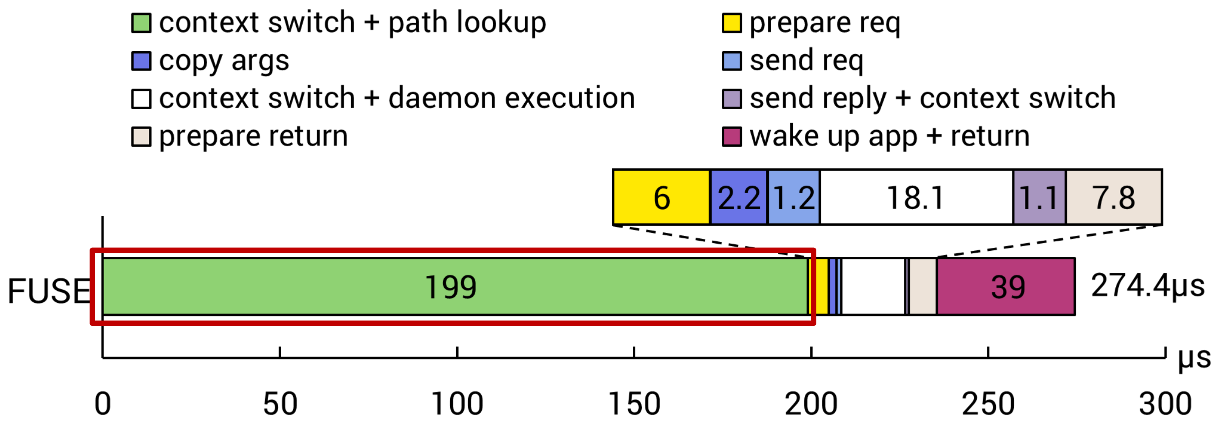
- CREAT() on root directory of NullFS, which promptly returns without performing any action
- RFUSE demonstrates a 53% lower latency than FUSE
  1. No context switches when processing requests and replies
  2. Low wake-up overhead within the kernel driver

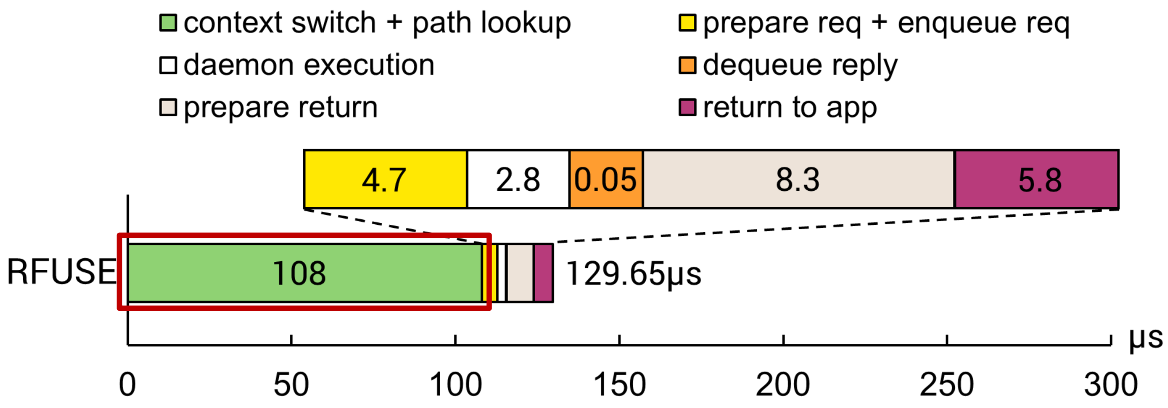




Latency Breakdown

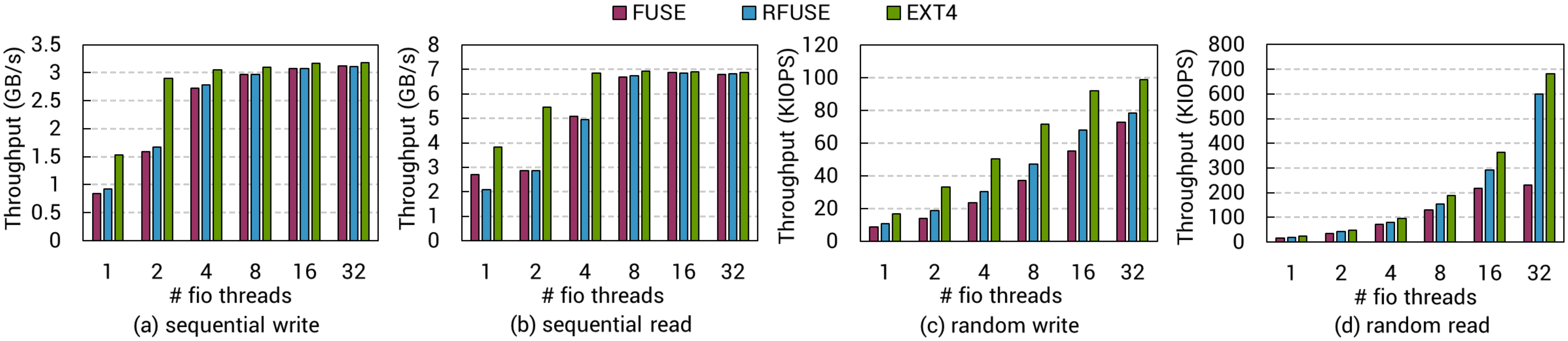
- CREAT() on root directory of NullFS, which promptly returns without performing any action
- RFUSE demonstrates a 53% lower latency than FUSE
  1. No context switches when processing requests and replies
  2. Low wake-up overhead within the kernel driver
  3. Short execution time for path traversal to verify the existence of subdirectories





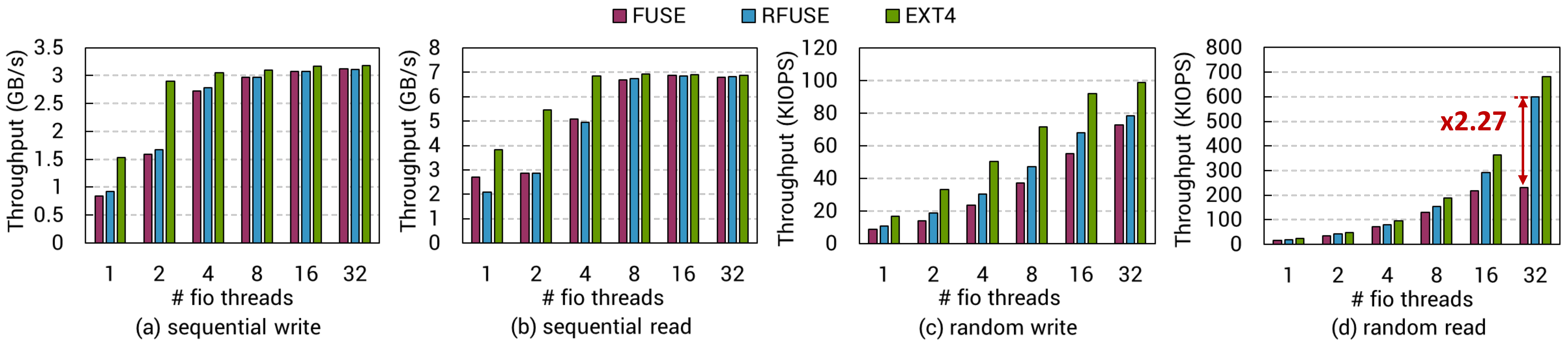
I/O Scalability

- FIO benchmark on StackFS while increasing the number of threads
  - Sequential I/O with 128KB size
  - Random I/O with 4KB size
  - 128GB file size in total



I/O Scalability

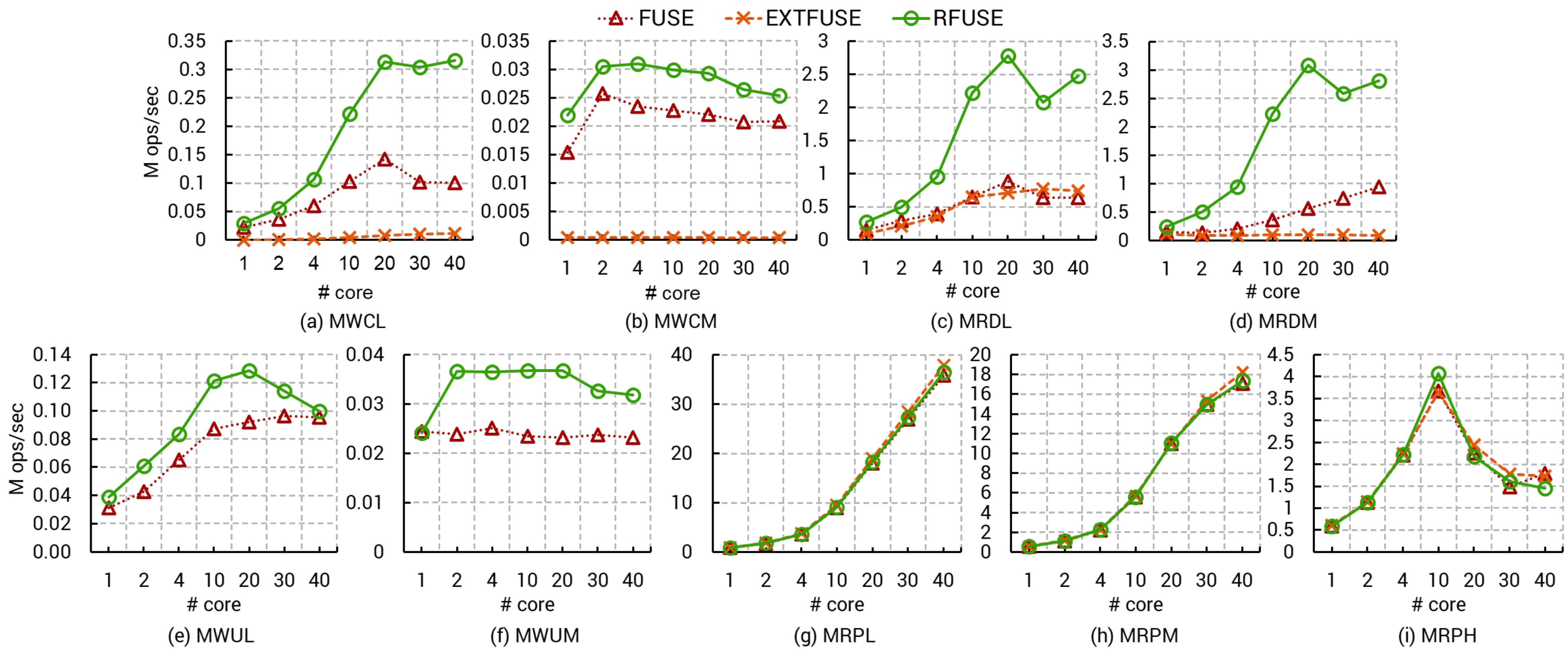
- FIO benchmark on StackFS while increasing the number of threads
  - Sequential I/O with 128KB size
  - Random I/O with 4KB size
  - 128GB file size in total



Metadata Operation Scalability

- FXMARK benchmark on StackFS

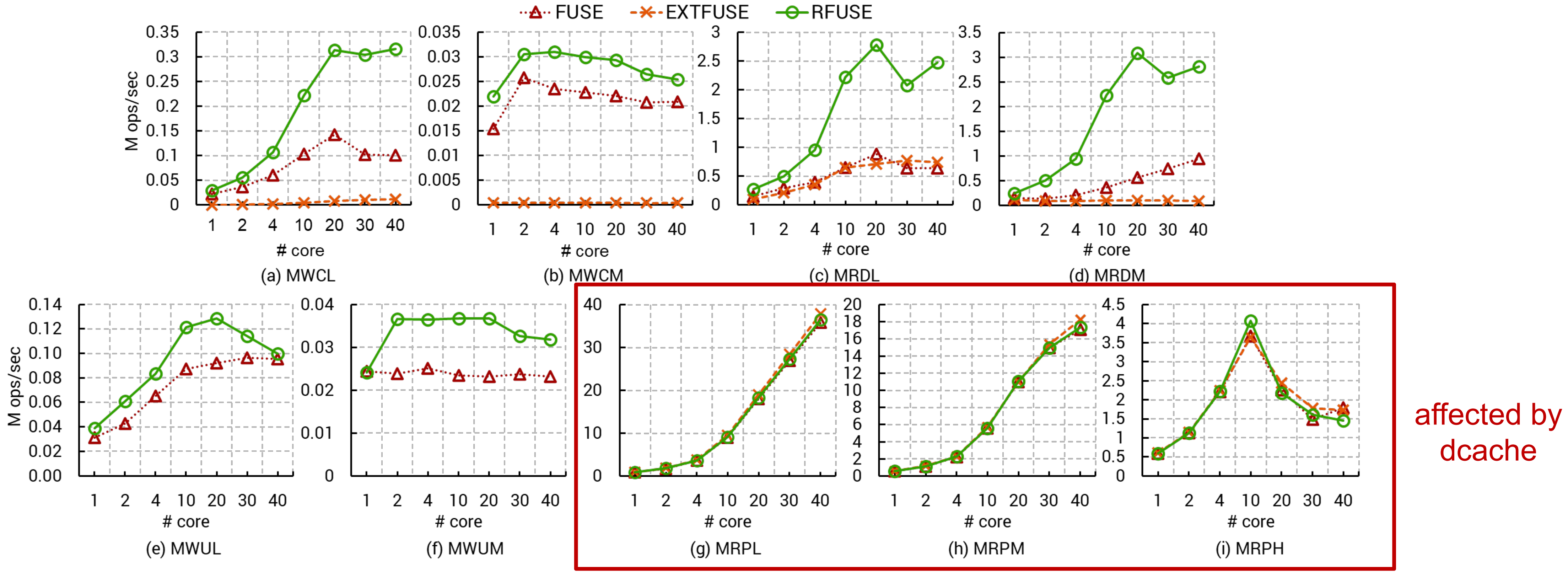
Workload	Description
MWCL	Create empty files in a private directory
MWCM	Create empty files in a shared directory
MRDL	Enumerate a private directory
MRDM	Enumerate a shared directory
MWUL	Unlink empty files in a private directory
MWUM	Unlink empty files in a shared directory
MRPL	Open and close private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



Metadata Operation Scalability

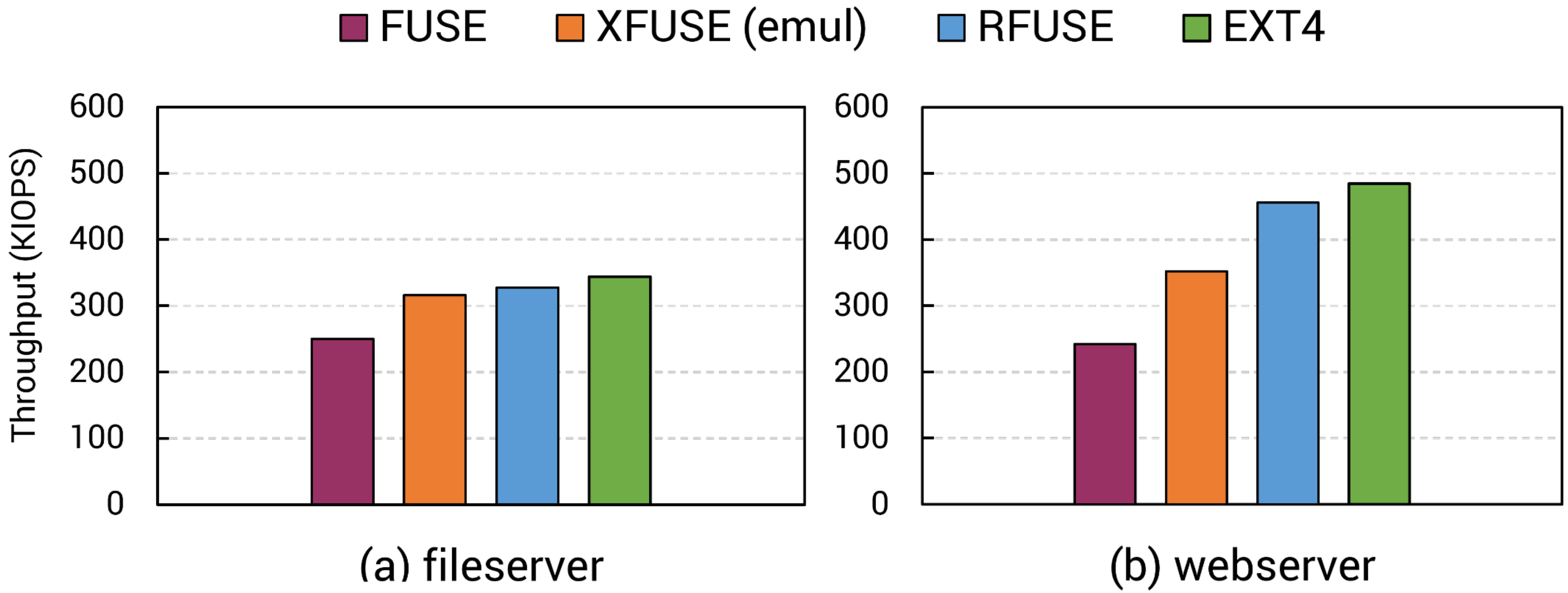
- FXMARK benchmark on StackFS

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



Macro-benchmarks

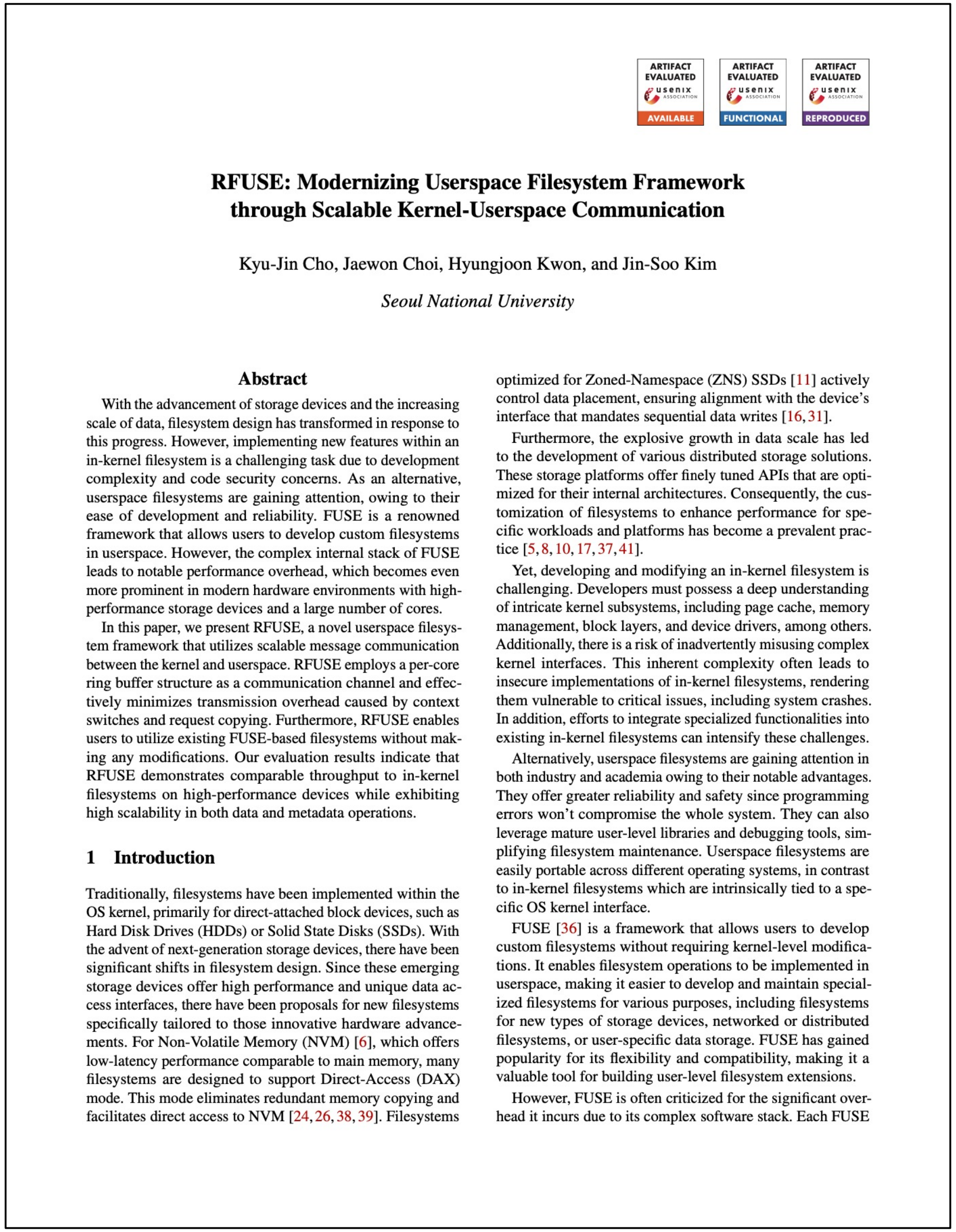
- Filebench benchmark on StackFS
  - fileserver: 200K files using 50 threads
    - Create file with a size of 128KB and then expanded through 16KB APPEND operations
  - webserver: 1.25M files using 100 threads
    - Create file with a relatively small size of 16KB and read whole file heavily



In the paper…

- More Details about RFUSE:
  - Transmission of ring channel Information
  - Load balancing of asynchronous requests
  - Memory usage of ring channels
  - …

- More Experiment Results:
  - FIO benchmark on Fuse-nfs
  - Macro benchmarks
  - Factor analysis of RFUSE
  - CPU utilization
  - …



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

Conclusion

- **RFUSE:** A userspace filesystem framework designed to support a scalable communication between the kernel and userspace

- RFUSE can provide **high-performance and scalability** on a modern hardware environment

- Source code is available at Github: https://github.com/snu-csl/rfuse

Thank you