XFUSE: An Infrastructure for Running Filesystem Services in User Space

Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen, *Alibaba Group*

https://www.usenix.org/conference/atc21/presentation/hsu

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

XFUSE: An Infrastructure for Running Filesystem Services in User Space

Qianbo Huai[*], *Windsor Hsu*[*], *Jiwei Lu*[*], *Hao Liang*[†], *Haobo Xu*[*] *and Wei Chen*[*]
[*]*Alibaba Group*
Sunnyvale, California
USA
[†]*Alibaba Group*
Shenzhen, Guangdong
China

Abstract

Implementing the filesystem in user space reduces development complexity [28, 30] and decreases dependency on the underlying OS platform. Implementing the filesystem at the user level as opposed to inside the OS kernel, however, has traditionally meant lower performance [11, 17, 22]. This performance overhead is increasingly limiting with high performance storage devices based on new persistent memory technology (e.g. 3D XPoint [13]) and advanced networking techniques (e.g. RDMA [14]). User space file systems have also been associated with poor reliability, availability and serviceability (RAS) characteristics [26]. As a result, there is a tendency to consider user space filesystems as prototypes and proof-of-concepts. In this paper, we systematically analyze the concerns with deploying user space filesystem to provide production file storage services. We present XFUSE, a filesystem in user space framework that addresses the performance and RAS concerns, and that enables file storage services to be effectively deployed at the user level. Our performance analysis indicates that XFUSE enables filesystem requests made through standard kernel interfaces to be processed at the user level with latency in the 4 microseconds range, and offers throughput exceeding 8 GB/s.

1 Introduction

User space code is generally easier to develop and maintain than kernel code. Thus filesystems with advanced functionality tend to be developed as user space filesystems (e.g. [3, 5, 24, 31, 32]). However, because filesystems are traditionally incorporated into the OS kernel, applications have been largely developed using standard kernel filesystem interfaces. This means that user space filesystems will incur additional performance overhead from passing messages between the kernel and the user space filesystem [11, 17, 22]. This overhead is especially limiting with the use of high performance storage devices and networking that routinely offer latency in the microseconds range and throughput of several GB/s [13, 14].

The use of user level networking [16, 18] and I/O helps to reduce the overall impact of crossing into user space to reach the filesystem but the performance hit is still significant. In particular, metadata operations such as stat and other operations that benefit from filesystem caching may be fast in kernel mode filesystems, but will be slower in user space filesystems due to additional communication cost between the operating system kernel and user space file systems [22].

Moreover, ensuring reliability, availability and serviceability (RAS) for user space filesystems has additional complexity because the rest of the system may continue to operate when the filesystem is down. For example, if a user space filesystem aborts, the application using the filesystem may continue to execute and wait for its requests to be completed. This partial failure possibility requires additional handling but it also provides a basis for the user space filesystem to be upgraded without disrupting the application. Such non-disruptive upgrade facilitates the deployment of new releases in production environments. There has been some previous work [26] on supporting restartable user space filesystems. However, the prior work requires significant kernel changes and do not support advanced filesystem features such as direct I/O and multi-threading.

Workloads are increasingly executed on the cloud in virtual machines(VMs) and sandboxed containers [4, 8, 12] to enable efficient resource utilization and increase agility. Cloud service providers can offer additional storage services to their customers through a storage client such as a filesystem gateway to object storage, optimized NAS client, etc. For the aforementioned reasons, it is advantageous to implement this storage client in user space. Furthermore, if the storage client can be deployed outside of the customer VMs and in the VM host, cloud service providers will be able to clearly separate the storage client from the user VM and independently manage the client as part of the storage service. There has been some recent work in this area. For example, virtio-fs [9] builds upon FUSE to allow VMs to access a host filesystem directory.

In this paper, we focus on enabling the deployment of pro-

duction storage services using user space filesystems. We evaluate prior approaches to understand what remains to be done to make this a reality. We propose XFUSE, a software framework patterned after FUSE [7] that addresses the performance and RAS concerns generally associated with user space filesystems. We believe that the improvements that XFUSE provides over FUSE can be applied to FUSE-based approaches such as virtio-fs [9] to better support running the user space filesystem in the VM host as opposed to inside the VMs that are running user applications.

Our contributions include:

- Systematically analyze the path a kernel filesystem request takes from being issued by an application to being handled by a user space filesystem and have the results communicated back to the application.
- Design and implement an optimized framework for user space filesystem that is backward compatible with FUSE and that takes advantage of the growing number of cores available on modern systems to achieve low latency and high throughput for fast storage devices.
- Demonstrate that such a framework enables kernel filesystem requests to be processed in user space with latency in the 4 microseconds range and throughput exceeding 8 GB/s.
- Extend the framework to provide features such as support for online upgrade and crash recovery that are critical for deploying user level filesystems in production.

The rest of this paper is organized as follows. In the next section, we survey related work. In Section 3, we introduce the design and implementation of XFUSE. In Section 4, we evaluate XFUSE performance. Section 5 concludes this paper.

2 Background and Related Work

FUSE is a widely adopted framework to support a filesystem in user space [7]. Figure 1 shows its highlevel architecture. FUSE consists of a Linux kernel module and a user space library that is linked to a user space filesystem daemon process. Conceptually, the FUSE kernel module consists of two parts: filesystem handler which is the code that processes filesystem operations including the code that invokes VFS, and device handler which interacts with the user space filesystem via a special device, /dev/fuse.

The filesystem handler and device handler run in different process contexts. The former runs in the context of the application process while the latter runs in the filesystem daemon process context. Communication between the two parts is coordinated through kernel events and incurs context switch cost. More specifically, there is a context switch when an application process sends a filesystem request to the filesystem daemon process, and there is another context switch when

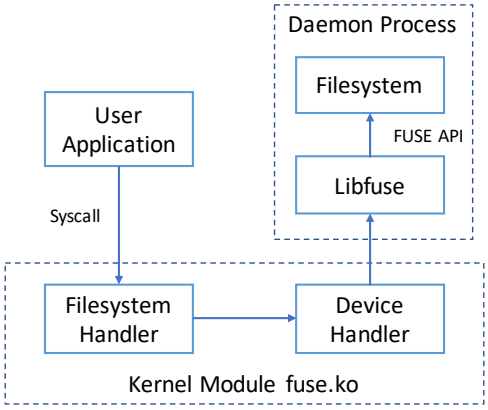

Figure 1: FUSE (and XFUSE) Architecture.

the filesystem daemon process responds to the application's request. Therefore each filesystem operation submitted by the application incurs at least two context switches with FUSE. In addition, FUSE has a single pending request queue from which threads of a filesystem daemon process pick up incoming requests for processing. Under load, this global queue becomes a source of contention.

There has been a large body of work on analyzing and improving the performance of FUSE (e.g. [11, 17, 22, 30]). ExtFUSE [11], for instance, extends FUSE so that a filesystem in user space can register a piece of simple code in the OS kernel to handle selected filesystem operations without incurring a context switch. There have also been alternatives to FUSE. For example, AVFS [1] uses the environment variable LD_PRELOAD [15, 21] to intercept libc POSIX API entry and invoke filesystem operations without context switch.

More recently, ZUFS [10] has been proposed as an alternate framework for user space filesystem that eliminates one data copy by having the kernel module copy data directly from the source to the destination. ZUFS and XFUSE share many of the same performance goals. We attempted to evaluate ZUFS but encountered issues. Our queries, as well as those from others, on the ZUFS project page on GitHub went unanswered. It appears that ZUFS is no longer maintained.

There has also been work on implementing the filesystem as an embeddable library running in user space (e.g. NVFUSE [6]) and on providing the filesystem as a separate user space process that communicates with each application through a private communication channel (FSP [19]). These approaches require applications to be rebuilt. They also bypass the VFS layer which provides important functionality such as as permission checking, file sharing coordination and buffer management.

The idea of transparently restarting the filesystem upon failure is explored in [25] and a specific framework that provides support for restartable user space filesystems is proposed in [26]. This framework, however, requires significant kernel

changes and does not support advanced filesystem features such as direct I/O and multi-threading [26]. The shadow driver concept proposed in [27] inspired us to keep track of incoming requests and eventually led to the crash restart algorithm proposed in this paper.

There has been a lot of recent interest in enabling VMs (and sandboxed containers [4, 8, 12]) to share a host directory. In particular, virtio-fs [9] builds upon FUSE to allow a VM to access files on the host. It provides a direct access (DAX) mode whereby data in the host page cache can be mapped into a VM and then accessed directly from within the VM. For non-DAX mode operations, virtio-fs is essentially FUSE with virtio [23] as the channel between the filesystem and device handlers. From this perspective, we believe that the improvements XFUSE provides over FUSE should be applicable to virtio-fs as well.

3 XFUSE

XFUSE is designed to achieve low latency and high throughput for fast storage devices, and scale with the increasingly large number of CPU cores available in today's systems. Besides supporting user space filesystems in general, XFUSE is specifically designed for deploying the following types of filesystems:

- Filesystems that use high speed storage devices such as those based on persistent memory technology. To effectively leverage the very low read and write latency (microseconds range) that these devices offer, software overhead must not dominate.

- Filesystems that use SSDs and distributed storage systems based on high performance network technology such as RDMA. Such storage systems can deliver low I/O latency (on the order of 100 microseconds) and high throughput (several GB/s).

- Filesystems that are used in a production environment where availability of service is critical and disruption of users should be kept to a minimum. In particular, recoverable faults and maintenance activities such as filesystem upgrade should not materially impact service to the user.

FUSE has been widely adopted to deploy user space filesystem. Thus XFUSE is designed to be backward compatible with FUSE. To end users, an XFUSE mount is almost identical to a FUSE mount. To filesystem developers, XFUSE supports the FUSE API and extends it to enable additional functionality such as support for online upgrade and crash restart. Just as with FUSE, XFUSE consists of a kernel module, xfuse.ko, and a library, libxfuse.a, that needs to be linked into the filesystem daemon.

3.1 Performance

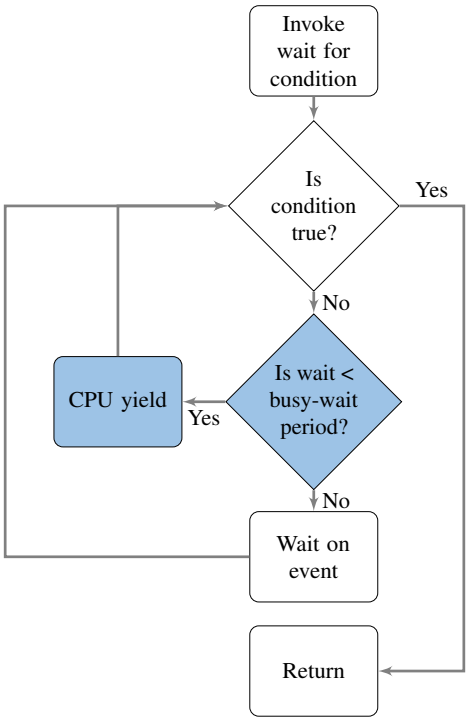

Figure 2: Busy-Event Wait.

3.1.1 Adaptive Waiting

XFUSE is patterned after FUSE [7] and has a similar architecture. Figure 1 can be used to describe the XFUSE architecture as well. From the figure, a FUSE request flows from the user application to the filesystem daemon process through the filesystem handler and the device handler. After the filesystem has processed the request, it sends a response in the reverse direction through the device handler and the filesystem handler. FUSE uses kernel events to coordinate the flow between the filesystem and device handlers. The latency for a FUSE request thus includes the time required by the filesystem daemon to handle the request and the time for two event waits - one by the device handler to obtain an incoming request and the second by the request handler to obtain the response from the filesystem daemon.

A kernel event notification takes on the order of a few microseconds to be delivered. For requests that can be handled quickly by the filesystem, the overhead of two event waits is very costly. For example, the latency of filesystem metadata requests that operate on in-memory data is dominated by the two event waits. The event waits also mask the low-latency benefit provided by high performance storage devices such as those based on persistent memory technology.

To avoid this overhead, XFUSE introduces an initial period of busy waiting or busy polling to the event wait. This wait scheme is depicted in Figure 2. If the condition being sought by a process is satisfied during the busy waiting period, the process continues without incurring the cost of an event wait. However, if the condition being sought is not met within the busy-wait period, the scheme falls back to regular event wait. We term this scheme *busy-event wait*.

With XFUSE, the device handler running in a filesystem daemon thread checks for new requests in a busy loop for a short period of time. If a pending request arrives during this busy waiting period, it is found and handled immediately by the daemon thread. If no new request arrives within the short period of time, the thread falls back to waiting on an event that is signalled whenever a request is submitted. The device handler sends a reply back to the filesystem handler in the same manner. If the filesystem daemon can handle a request fast enough, the filesystem handler may still be busy waiting when the reply becomes available. In this case, the end-to-end latency as observed by the application process can be as low as 3-4 microseconds.

The effectiveness of busy waiting has some dependence on whether the application threads and filesystem daemon threads are running on the same or different CPUs. We evaluate this factor in Section 4.1.2. In setting the busy-wait period, there is a tradeoff between request latency and CPU utilization which affects throughput. The busy-wait period should be set based on the performance characteristics of the filesystem and the underlying storage system. Later in Section 4.1.1, we evaluate this parameter and find that a busy-wait period on the order of 10 microseconds achieves a good balance for systems based on persistent memory technology as well as those based on SSD.

To further reduce CPU consumption and potentially increase throughput, we can dynamically adjust the busy-wait period based on the latency observed in the system. In particular, if the actual time required to service a request exceeds the busy-wait period, attempting to busy wait is futile and only wastes CPU resources. We thus experimented with a simple scheme that dynamically disables busy waiting whenever the last observed latency exceeds the busy-wait period by more than the net overhead of event wait, and reenables it whenever the last observed latency falls below this threshold. We refer to this scheme as *adaptive busy-event wait*. Results reported in Section 4.1.1 show that adaptive busy-event wait is very effective at avoiding unnecessary busy waiting, thereby increasing throughput.

3.1.2 Increased Parallelism

With the growing number of cores available on modern systems, increasing the number of requests that can be processed in parallel is the key to increasing throughput. XFUSE enables multiple filesystem daemon threads to work on different requests in parallel, and supports the asynchronous processing model to enable each thread to handle multiple concurrent requests.

More specifically, XFUSE provides multiple communication channels between the filesystem handler and device handler. Each channel has two queues. One is the free queue containing request slots that can be used for new requests. The other is the processing queue containing requests that are inflight. The filesystem handler selects a channel for an incoming request using a channel selection policy. At the other end of each channel is a filesystem daemon thread waiting for new requests. There can be multiple requests in the processing queue per daemon thread. XFUSE also ensures that threads working on different channels do not have lock contention between them. The number of channels and the number of request slots per channel determine the maximum concurrency that XFUSE can deliver to the filesystem daemon. In order to make effective use of the multiple channels and associated filesystem daemon threads that XFUSE supports, thread placement and channel selection policies are important considerations. We discuss and evaluate these policies later in Sections 4.1.2 and 4.1.3.

In contrast, FUSE uses a single request queue to hold incoming requests. When there are many concurrent accesses, the single request queue may become a source of contention and limit the throughput that FUSE can achieve. As we shall see later in Section 4, XFUSE is able to drive much higher throughput than FUSE through increased parallelism.

3.2 RAS

3.2.1 Online Upgrade

Scheduling service downtime to perform an upgrade is very disruptive in production settings. By operating outside of the kernel, a user space filesystem can potentially be upgraded without disrupting the application. Such a non-disruptive or online upgrade capability will ease the introduction of new features and big fixes, and facilitate the deployment of user space filesystems in production environments.

When a filesystem daemon terminates, all the file descriptors to the special device /dev/fuse are closed. This leads the kernel to unmount the filesystem automatically. In order to keep the kernel from unmounting the filesystem during an upgrade of the filesystem daemon, XFUSE provides a monitor service to hold all the XFUSE device file descriptors while the old filesystem daemon exits and a new daemon executing the upgraded software takes over to serve running applications.

XFUSE includes a library, libxfuse to facilitate the interaction between the filesystem daemon and the XFUSE monitor service. Libxfuse extends the FUSE libfuse library with new functionalities and APIs that allow user space filesystems built with it to support online upgrade.

Figure 3 illustrates the XFUSE-assisted filesystem online

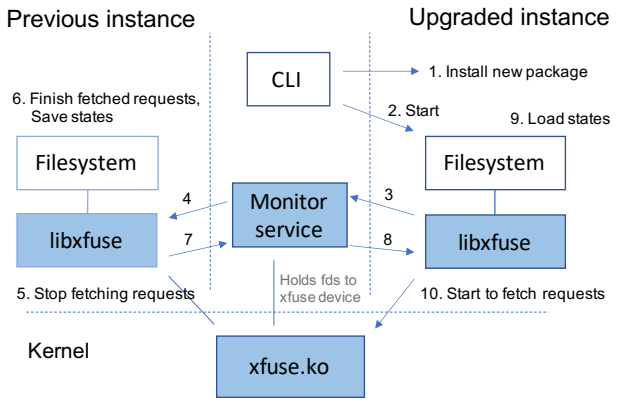

Figure 3: Filesystem Online Upgrade Workflow.

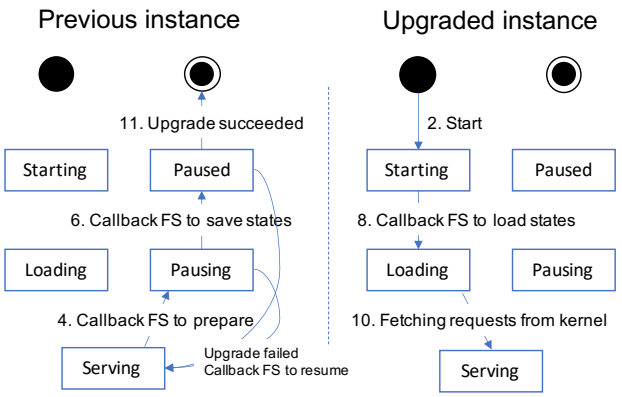

Figure 4: State Transitions during Filesystem Online Upgrade.

upgrade workflow. Figure 4 shows the XFUSE state transitions in both filesystem daemons during the upgrade process. The numbered actions in the two figures match. Certain details, such as error handling, are omitted to reduce clutter.

1. The upgrade workflow starts with an operation to update the filesystem software package. In Figure 3, this operation is triggered through a CLI (command line interface) command to install the new software package.

2. After installation, a filesystem daemon process running the new software is started. This filesystem daemon initializes its XFUSE stack with parameters to interact with the XFUSE monitor service, and provides libxfuse with a number of callback functions. The usage of those callbacks is depicted in Figure 4.

3. Upon starting, libxfuse connects to the XFUSE monitor service on behalf of the filesystem daemon and fetches all the file handles to the XFUSE device. As discussed earlier, having the monitor service create and hold XFUSE device handles ensures that the filesystem service can remain online to applications during the upgrade.

4. When a filesystem daemon is running, it maintains a communication channel with the XFUSE monitor service. Through this channel, the monitor service notifies the current filesystem daemon that it is being upgraded.

5. Libxfuse in the current filesystem daemon stops fetching new requests from the kernel. However, XFUSE device read operations issued moments earlier may still be returning from the kernel. The filesystem may also be replying to just-processed requests. Libxfuse waits for all pending requests to drain.

6. The filesystem continues processing requests. When libxfuse is certain that there are no more inflight requests, it notifies the filesystem to save its transient state. As an optimization, libxfuse can notify the filesystem earlier in Step 4 to start preparing for an upgrade so as to reduce the time needed to save the transient state after incoming requests are paused.

7. Libxfuse notifies the XFUSE monitor service on behalf of the current filesystem daemon that it is paused and that its transient state has been saved. The saved state includes open file handles, file locks and all other runtime data necessary for a new filesystem daemon to take over and continue serving running applications.

8. The XFUSE monitor service notifies the new filesystem daemon that the saved state is ready to be loaded.

9. Libxfuse in the new filesystem daemon calls back into the filesystem to load the saved state and prepare to handle incoming requests.

10. The new filesystem daemon starts to fetch requests from the kernel. Past this point, the previous filesystem daemon exits. The online upgrade has been successfully completed.

We have implemented a user space filesystem with the ability to save and restore its transient state, and integrated it with XFUSE to enable us to upgrade the filesystem software with minimal user impact. The filesystem and XFUSE components are deployed and serving production workloads. In Figure 5, we plot the performance as observed by the fio benchmark [2] operating against the filesystem as the filesystem is upgraded. At 34 seconds into the plot, the upgrade is initiated. This triggers a series of steps to verify and install the new software package. A filesystem daemon process running the new software is then started. Throughout this time, the current filesystem daemon continues to serve I/O requests. At 55 seconds into the plot, the current filesystem daemon is asked to stop processing incoming requests. By 57 seconds into the plot, the current filesystem daemon has completed all

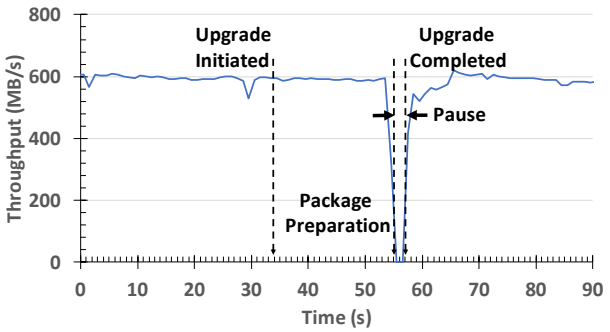

Figure 5: Throughput During Online Upgrade of Filesystem.

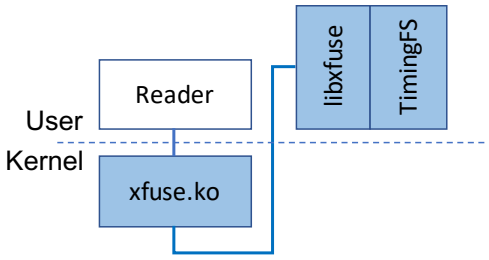

Figure 6: Experimental Setup for Parametric Analysis.

pending requests and saved all of its transient state. The new filesystem daemon loads the saved transient state and starts to serve incoming requests. At this point, the old filesystem has served its purpose and terminates itself. There is a ramp up in performance as the new filesystem establishes its cache.

3.2.2 Crash Restart

The online upgrade process forms the basis for supporting crash restart of the filesystem daemon process. Just as for online upgrade, in order to support crash restart, the filesystem must be able to remember its transient state and reinstate it via the new filesystem daemon instance. After the new instance is up and has restored the previous state, it can start to replay any pending requests. The kernel module does not know the exact point at which the filesystem daemon process crashed and restarted. It simply resends all the pending requests that it has already sent to the previous instance and for which it has not received the response.

The new filesystem daemon instance, however, cannot simply re-process all the requests because filesystem requests are not idempotent. For example, suppose a user successfully removes a file *P*. If the file removal operation were to be applied again, it would fail because the file *P* no longer exists.

To handle crash restart in the face of such non-idempotent requests, the filesystem daemon saves information about recent requests in a request-response table. This table records the response for requests that have already been processed by the filesystem daemon. For each incoming request, the filesystem daemon replies with the saved response if the request is recorded in the table. Otherwise it processes the requests normally. To uniquely identify a request, the kernel module assigns a request ID, which encodes a consecutive sequence ID, to each incoming request. To ensure that IDs are assigned consecutively, the filesystem daemon needs to persist the largest request ID it has processed. After a restart, the new daemon instance passes this value as a mount flag to the kernel.

The size of the request-response table is bounded by the number of requests that may be inflight. Conceptually, the XFUSE kernel module keeps a queue of the requests that it has received the response for. When sending the next request, it dequeues a completed request and sends its request ID along with the new request. The filesystem daemon uses the completed request ID to prune its request-response table. In practice, the tracking of completed requests and their responses is done on a per channel basis to avoid introducing points of contention in this process, Each channel only needs to keep track of up to queue depth number of recent requests.

4 Performance Evaluation

To evaluate the performance characteristics of XFUSE, we first use a controlled environment to explore the different aspects of XFUSE individually. The goal of this analysis is to systematically understand how these aspects are affected by policy choices and tuning parameters, and to project the performance that can potentially be achieved by a filesystem daemon that is optimized for XFUSE. We then measure actual system performance on real systems when the filesystem requests are looped through XFUSE and FUSE, and compare the results with directly accessing kernel-mode EXT4 [20].

4.1 Parametric Analysis

In this section, we use the experimental setup depicted in Figure 6. The setup is designed to provide a controlled environment where various aspects of XFUSE can be isolated and the associated parameters can be tuned systematically. Because existing user space filesystems have not been optimized beyond what FUSE can drive, this setup also serves to project the kind of performance that XFUSE can potentially achieve with a user space filesystem that is optimized for it. The setup consists of a reader composed of multiple reader threads each synchronously reading 4 KB of data from a random location in a large file. These read requests are sent via XFUSE to TimingFS, a simple filesystem daemon that emulates the timing characteristics of persistent memory and SSD.

TimingFS supports only readdir, getattr and read operations. Readdir and getattr operations are supported only to the extent necessary for a filesystem to be mounted and a file within the filesystem to be opened for read. The main purpose of TimingFS is to respond to read requests of the opened file. In

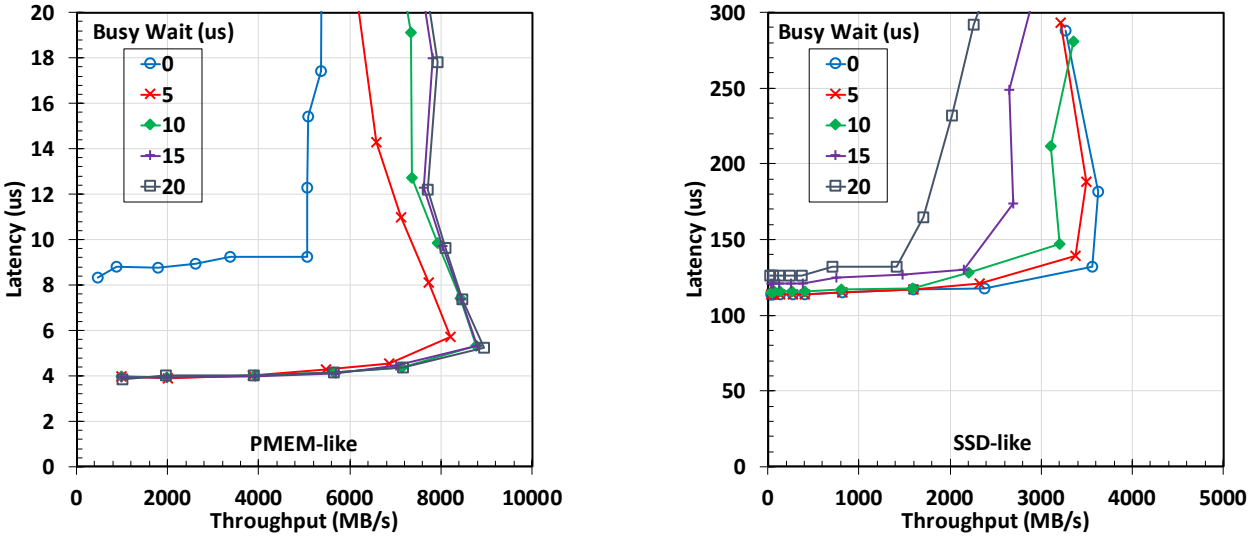

Figure 7: Performance with Busy-Event Wait.

the persistent memory mode, TimingFS handles the file read requests by immediately copying the requested amount of data from the specified location in a memory pool standing in as the file. In the results, this case is denoted as PMEM-like. In the SSD mode, TimingFS enqueues the read request and a worker thread handles the enqueued request after a delay of 100us. This case is denoted as SSD-like in the results. We use a file size of 1 GB to ensure that the memory footprint exceeds the L1 and L2 processor caches as is typically the case for I/O requests because of the amount of data involved. For 4 KB requests, ensuring that the data does not reside solely in the processor caches adds about 1 us to the read latency.

All the experiments in this analysis were performed using dedicated servers running Linux 4.19.91 on the Alibaba Cloud. Each server has dual Intel(R) Xeon(R) Platinum 8163 CPUs operating at 2.50GHz for a total of 48 physical cores. We restricted the experiments in this section to using the first 24 physical cores.

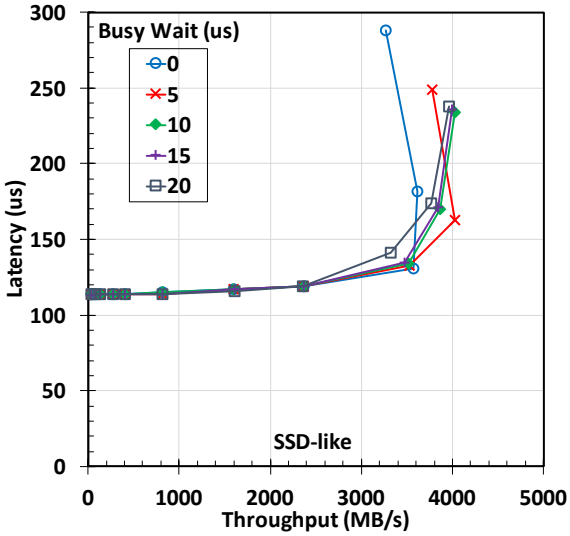

Figure 8: Performance with Adaptive Busy-Event Wait.

4.1.1 Waiting Strategy

Figure 7 summarizes the effect of introducing an initial period of busy waiting to event wait. Note that with a busy-wait period of 0, busy-event wait degenerates into event wait. Observe that adding on the order of 10 us of busy waiting is very effective at lowering latency and increasing throughput for PMEM-like storage. On the other hand, adding busy waiting is not effective for SSD-like storage. In fact it degrades performance significantly for SSD-like storage, and the degradation increases with the busy-wait period. This is because the service time with SSD-like storage exceeds the busy-wait period so that the busy waiting is wasted and only serves to drive up CPU utilization which affected the throughput.

We next investigate enabling busy waiting only when it is beneficial. The basic idea is that if the current wait is going to be longer than the busy-wait period, we should skip the busy waiting and go straight to event waiting. We use a simple method of predicting the current wait time based on the observed latency of the last completed wait. Now, the last wait could have been completed by an event notification so a conservative threshold for enabling busy waiting is that the last observed latency is within the busy-wait period plus the net overhead of event wait. In other words, we disable busy waiting when the last observed latency exceeds the busy-wait period by more than the net overhead of event wait and reenable it when the last observed latency falls below this threshold. From Figure 9, the latency achieved with busy

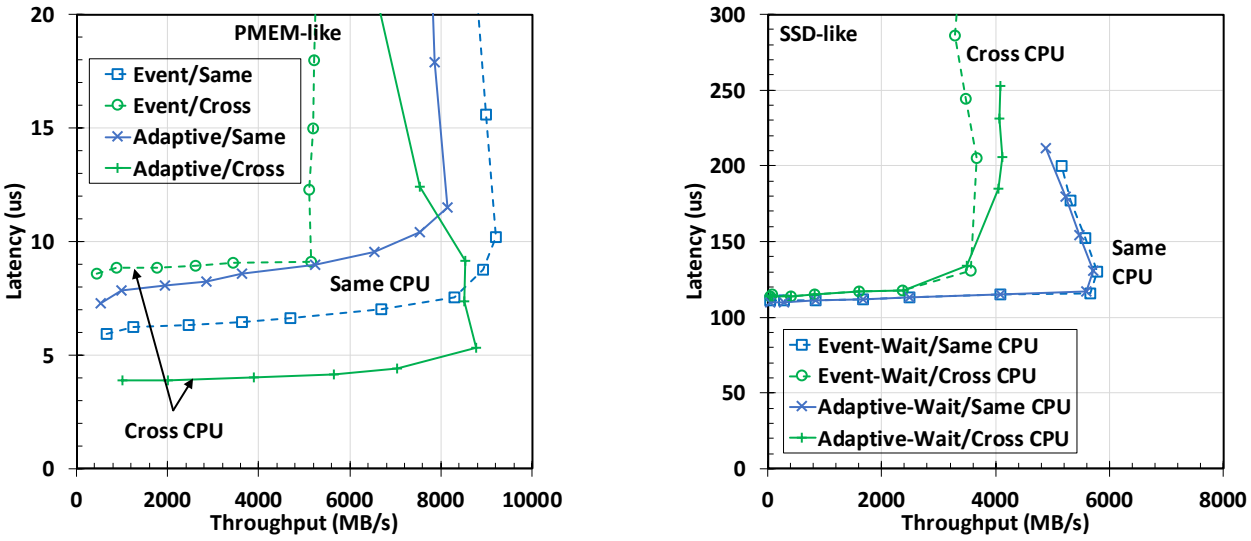

Figure 9: Effect of Thread Placement.

waiting can reach just under 4 us while that with event waiting is around 9 us, suggesting that the net event overhead is about 5 us in our experimental setup.

As shown in Figure 8, with such an adaptive busy-event wait scheme, the latency becomes relatively insensitive to the busy-wait period because futile busy waiting is avoided. With adaptive busy-event wait, a busy-wait period in the 10 us range woks well for both the PMEM-like and SSD-like configurations. Observe further that throughput is increased beyond event-wait even though the service time with SSD-like storage far exceeds the busy-poll periods under consideration. Recall that there are two waits in the system. One is by the device handler to obtain an incoming request and the second is by the request handler to obtain a response from the filesystem daemon process. For SSD-like storage, the device handler running in the filesystem daemon thread to pick up incoming requests benefits from the busy waiting when under load. Thus adaptive busy-event wait offers a performance benefit even for relatively slow SSD-like storage.

4.1.2 Thread Placement

The overhead of busy waiting and event waiting depends on whether the application threads and filesystem daemon threads are executing on the same CPU. If the application thread and corresponding filesystem daemon thread run on different CPUs and use busy waiting to send messages to each other, the number of context switches will be greatly reduced. On the other hand, if the two threads are scheduled on the same CPU, context switches between the 2 threads are needed to accept the request and receive the response. Furthermore, the scheduler typically implements a per CPU run queue so that local CPU event notification is likely to be delivered faster. There are also cache locality considerations when it comes to scheduling the application threads and filesystem daemon threads, but our evaluation shows that this effect is secondary.

In Figure 9, we investigate the effect of thread placement on performance by controlling the CPU on which each application and filesystem daemon thread runs, and the mapping of requests to channels. Specifically, we affine each application thread to a CPU and map each request to a channel based on the CPU ID of the application thread issuing the request. On the filesystem daemon side, we consider two cases - one where the thread listening on a channel is affined to the same CPU as the application thread whose requests are mapped to that channel (denoted same CPU), and the second where the listening thread is affined to a different CPU (denoted cross CPU). Observe that for PMEM-like storage, cross CPU busy waiting offers the lowest latency of just under 4 us, outperforming same CPU busy waiting by between 3-5 us. On the other hand, same CPU event waiting significantly outperforms cross CPU event waiting. In the SSD-like case, the long service time with SSD-like storage means that busy waiting does not improve latency. Thus for SSD-like storage, the lowest latency is obtained by scheduling the application threads and corresponding filesystem daemon threads on the same CPU.

In some production environments such as those where large server farms are used to provide a specific set of services to many customers, the thread placement on each server can be controlled as we have done in these experiments. When PMEM-like storage is used in these types of environments, the application and corresponding filesystem daemon threads should be placed on different CPUs and adaptive busy-event wait should be used so as to achieve a significant improvement in latency. For SSD-like storage, the application and corresponding filesystem daemon threads should be placed

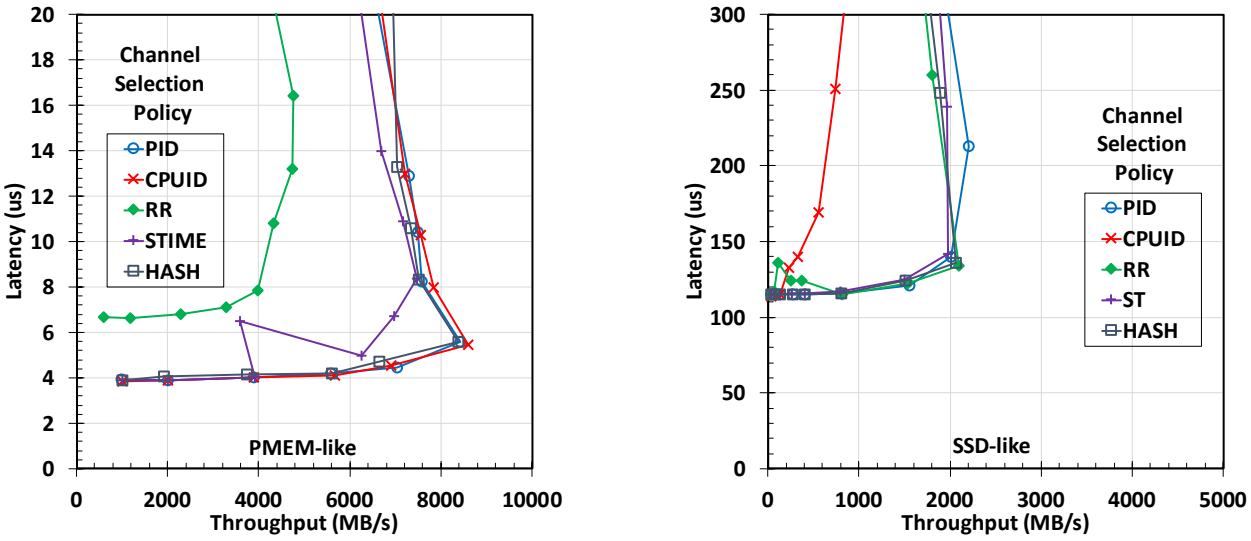

Figure 10: Effect of Channel Selection Policy.

on the same CPU. In this case, adaptive busy-event wait does not improve performance but does not hurt either.

4.1.3 Channel Selection

In other environments where thread placement cannot be explicitly managed, the performance achieved depends on how the scheduler maps the application and filesystem daemon threads to CPUs, and on how the requests are mapped to the channels. In Figure 10, we evaluate various channel selection policies when the thread placement is left entirely up to the scheduler.

The best strategy in this case is to evenly distribute requests across the channels so that multiple filesystem daemon threads can be brought to bear on handling the requests. There is one caveat in that if the policy keeps switching to an idle channel, it will render busy waiting ineffective. In the figure, RR denotes the round robin channel selection policy. In the PMEM-like case where busy waiting can be effective, RR performs worse than other policies because it keeps rotating to a channel where the corresponding filesystem daemon thread is no longer busy waiting. In order to leverage busy waiting, the channel selection policy needs to have a bias towards a specific channel.

We also evaluate using the CPU ID, thread ID and thread start time to map requests to channels. Depending on where the scheduler places the application threads, using the CPU ID to select channels may result in a very skewed channel distribution. Thread start time may also collide and result in a less than balanced distribution. We find that using thread ID is much more stable in terms of yielding an even distribution because the thread ID is at least unique. To further avoid the risk of a skewed distribution, we can use hashing techniques on the thread ID to avoid colliding on the same channel. The HASH selection algorithm uses 3 hash functions to identify candidate channels and selects the channel with the shortest queue. In the case of a tie, it always selects the first candidate channel to avoid defeating busy waiting. HASH consistently avoids skewed distribution.

4.1.4 Performance Potential

XFUSE is designed to provide an efficient conduit between the kernel file system interface and a user space filesystem daemon. The performance that it can deliver ultimately depends on the capability of the user space filesystem. Here, we use our simple filesystem daemon, TimingFS, to project the best-case performance that XFUSE can achieve with a user space filesystem that is optimized for it. We configure XFUSE based on the analysis results discussed above. Specifically, we use adaptive busy-event wait with a busy-wait period of 10 us and an event overhead of 5 us. We also set up XFUSE with 24 channels and corresponding TimingFS threads, one for each of the physical cores that are used for the experiments.

Figure 11 presents the results. Notice that in the PMEM-like configuration, XFUSE is able to achieve latency in the 4 us range and throughput exceeding 8 GB/s. FUSE, on the other hand, has a latency of 10 us and throughput of less than 2 GB/s. Across both PMEM-like and SSD-like configurations, XFUSE significantly outperforms FUSE both in terms of latency and throughput. The improvement is especially dramatic for the PMEM-like configuration because the storage is so fast that even small overheads become hugely significant.

4.2 System-Level Performance

In this section, we use system-level benchmark workloads to evaluate the performance potential of XFUSE. The experi-

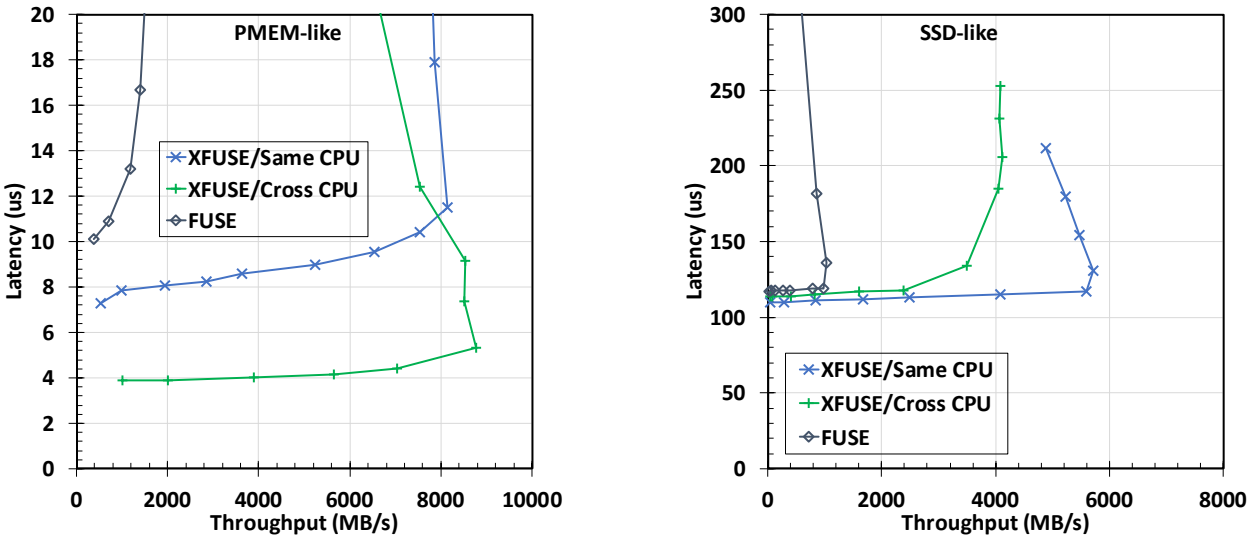

Figure 11: Performance of XFUSE and FUSE with TimingFS.

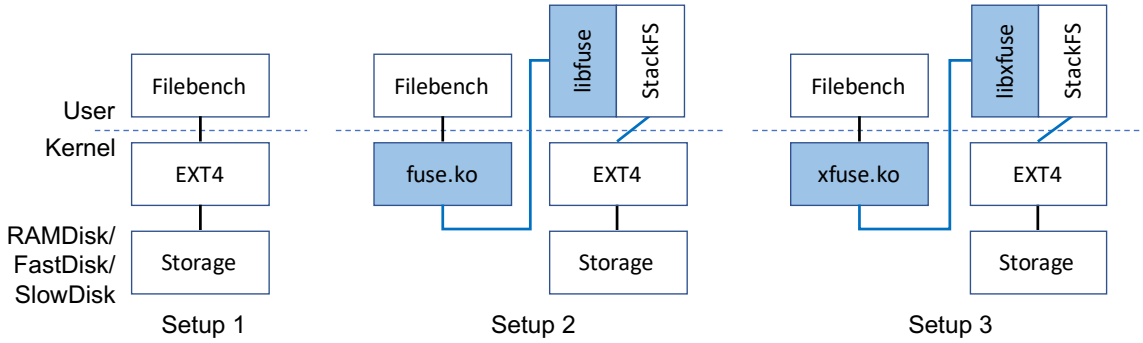

Figure 12: Experimental Setup for Evaluating System-Level Performance.

mental setup is depicted in Figure 12. The setup is designed foremost to provide a common basis for comparing XFUSE with FUSE and regular kernel-mode EXT4, and errs on the side of being conservative for XFUSE. Specifically, in this setup, XFUSE and FUSE rely on StackFS [30] to route calls back to kernel-mode EXT4. StackFS is a simple stackable passthrough filesystem originally developed to evaluate FUSE performance. StackFS has not been reworked to take advantage of the low-latency and high parallelism that XFUSE can deliver.

As discussed in [30], existing techniques such as the kernel page cache and FUSE's readahead are effective at masking the performance of user space filesystems in several types of workload. In this section, we present results focusing on those cases where FUSE has a significant gap with kernel-mode EXT4, namely the Filebench random read, file create and webserver workloads, as previously defined and made public [29, 30]. We ran these workloads as is and without specifying CPU affinity.

We performed the measurements using dedicated servers running Linux 4.19.91 on the Alibaba Cloud. The FUSE version used is 3.6.1. Each server has 48 physical cores (dual Intel(R) Xeon(R) Platinum 8163 CPUs operating at 2.50GHz) and 768GB of memory. All the experiments in this section were limited to using the first 24 physical cores and 256 GB of memory. The remaining 512 GB of memory was allocated as a RAM disk. We configured XFUSE based on the analysis results presented in the previous section, meaning that we used adaptive busy-event wait with a busy-wait period of 10 us and an event overhead of 5 us. We also set up XFUSE with 24 channels and corresponding StackFS threads, one for each of the physical cores.

The first config uses the RAM disk as the storage device. This config is meant to illustrate the potential performance achievable when using XFUSE to access a user mode persistent memory based filesystem. We refer to this setup as RAMDisk. In the second config, we use the fastest cloud disk available. The average latency for a 4 KB read is about 115 us

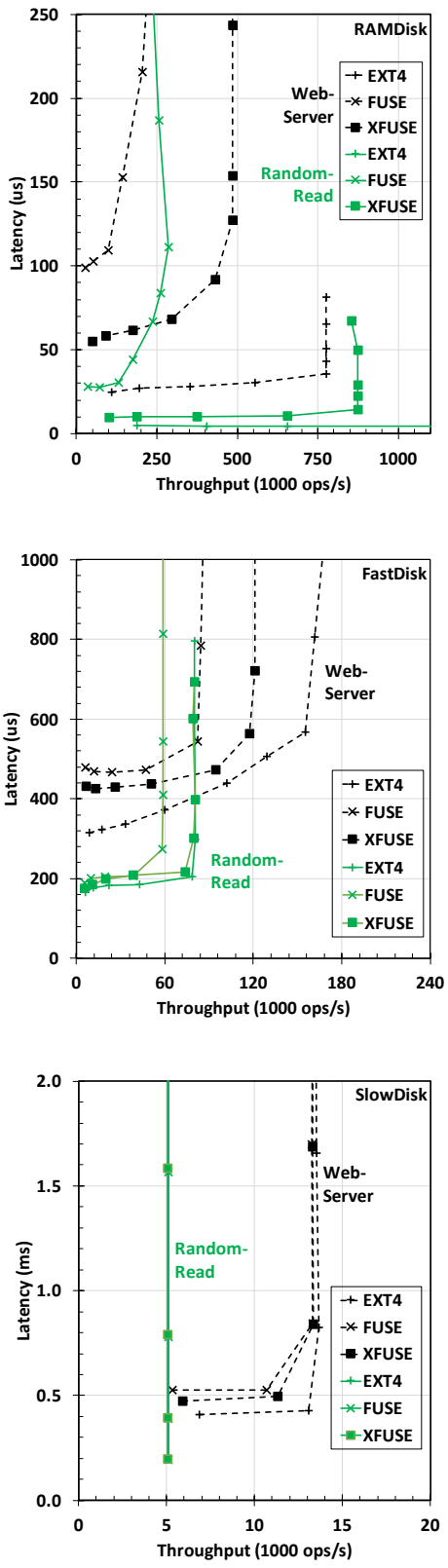

Figure 13: Performance of Random-Read and Web-Server Workloads.

on this disk. Based on its provisioned size, the throughput for this disk is capped at 80K IOPS. We refer to this setup as FastDisk. In the third config, we use the slowest cloud disk available. The average 4 KB read latency is about 250 us for this disk and it is limited to 5K IOPS. We refer to this setup as SlowDisk.

Figure 13 summarizes the performance results for the Random-Read and Web-Server workloads. With RAMDisk, kernel-mode EXT4 outperforms FUSE by a large margin across the 2 workloads. XFUSE closes the gap significantly both in terms of latency and throughput. For Random-Read, latency with XFUSE comes in at 9 us versus 28 us with FUSE. Throughput with XFUSE exceeds 874K ops/s while FUSE can only achieve 285K ops/s. This represents a 3x improvement. The improvement is similarly large for the Web-Server workload. The gap with kernel-mode EXT4 is still significant as expected because the XFUSE results are obtained by looping requests to user space and back into kernel-mode EXT4. We expect the gap to be reduced with an actual filesystem daemon running in user space that is optimized for XFUSE.

Observe that as the storage is slower, the gap between kernel-mode EXT4 and FUSE shrinks as does the benefit of XFUSE over FUSE. With SlowDisk, there is virtually no performance difference between XFUSE and FUSE. This is to be expected because when the storage is slower, the performance is bottlenecked more by the storage than by the conduit to user space. For FastDisk, which is the storage that performance-critical workloads are most likely to be using, XFUSE offers significant benefit over FUSE. For example, for the Web-Server workload, latency with XFUSE is 425 us versus 466 us with FUSE, and throughput is 125K ops/s versus 86K ops/s with FUSE. Note that the throughput of FastDisk is capped at 80K IOPs based on its provisioned size. For the Random-Read workload, XFUSE is able to deliver the full throughput of FastDisk, matching the throughput achieved by kernel-mode EXT4.

The results for the File-Create workload are summarized in Figure 14. To reduce clutter, the results for FastDisk are not presented. XFUSE outperforms FUSE for both FastDisk and RAMDisk, but not by as large a margin as in the case of the Random-Read and Web-Server workloads. For the File-Create workload, the throughput with kernel-mode EXT4 is several times higher than that with either FUSE or XFUSE. The File-Create workload creates millions of small files and each create requires several calls into StackFS to perform getattr and pathname lookups. StackFS in turn dynamically allocates memory and uses internal global locks and states to translate the requests it receives back to POSIX calls to the underlying EXT4. These roundtrips between the kernel and StackFS as well as the implementation of StackFS restrict the performance of File-Create with FUSE and XFUSE. Notice further from the figure that the File-Create workload does not scale in performance beyond 4 threads on kernel-mode EXT4. This limits the benefit that XFUSE can provide over

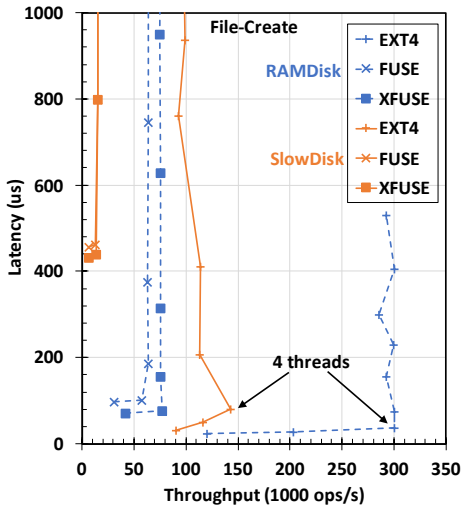

Figure 14: Performance of File-Create Workload.

FUSE because the increased parallelism that XFUSE provides cannot be brought to bear on this workload with EXT4 as the backing filesystem.

The results underscore the point that XFUSE is designed to provide an efficient conduit between the kernel filesystem interface and a user space filesystem. The performance that it can achieve depends ultimately on the capability of the user space filesystem. Our measurements indicate that even with an existing FUSE-targeted StackFS, XFUSE offers significant performance improvement over FUSE. We anticipate that the improvement will be even more pronounced with user space filesystems that are designed to take advantage of the low latency and high parallelism that XFUSE can deliver.

5 Conclusion

This paper presents XFUSE, a user space filesystem framework that addresses the performance and RAS concerns generally associated with user space filesystems. XFUSE is backward compatible with FUSE and takes advantage of the growing number of cores available on modern systems to achieve low latency and high throughput for fast storage devices. XFUSE can enable filesystem requests made through standard kernel interfaces to be processed at the user level with latency in the 4 microseconds range, and offers throughput exceeding 8 GB/s. XFUSE also provide features such as support for online upgrade and crash recovery that are critical for deploying user level filesystems in production.

Acknowledgments

This paper would not have been possible without the collaboration of our storage and kernel teams. We are grateful to our shepherd and anonymous reviewers for helping us improve the paper, and to the many who have worked on FUSE and on whose shoulders we stand. We are working towards contributing XFUSE to the community.

References

[1] *AVFS - A Virtual File System*. http://avf.sourceforge.net.

[2] *fio - Flexible I/O tester rev. 3.27*. https://fio.readthedocs.io/en/latest/fio_doc.html.

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

[4] *gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere*. https://gvisor.dev.

[5] *A network filesystem client to connect to SSH servers*. https://github.com/libfuse/sshfs.

[6] *NVMe based File System in User-space*. https://github.com/nvfuse/nvfuse.

[7] *The reference implementation of the Linux FUSE (Filesystem in Userspace) interface*. https://github.com/libfuse/libfuse.

[8] *Secure and fast microVMs for serverless computing*. https://firecracker-microvm.github.io.

[9] *virtio-fs*. https://virtio-fs.gitlab.io.

[10] *zero-copy file-system feeder. A Linux module which dispatch kernel's VFS commands to user-space server*. https://github.com/NetApp/zufs-zuf.

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

[12] Jay Chen. *Making containers more isolated: An overview of sandboxed container technologies*, (accessed January 12, 2020). http://unit42.paloaltonetworks.com/making-containers-more-isolated-an-overview-of-sandboxed-container-technologies.

[13] Intel Corp. *Intel® Optane™ technology for data centers*, (accessed January 12, 2020). http://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html.

[14] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, 2016.

[15] Shi Zhan Feng Dan Zhao Heng and Yao Yingying. LD_PRELOAD based file system management. *Journal of Huazhong University of Science and Technology (Natural Science Edition)*, 2010.

[16] Yukai Huang, Jinkun Geng, Du Lin, Bin Wang, Junfeng Li, Ruilin Ling, and Dan Li. LOS: A high performance and compatible user-level network operating system. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 50–56, 2017.

[17] Shun Ishiguro, Jun Murakami, Yoshihiro Oyama, and Osamu Tatebe. Optimizing local file accesses for FUSE-based distributed storage. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 760–765, 2012.

[18] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation*, pages 489–502, 2014.

[19] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File systems as processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[20] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.

[21] Kevin Pulo. Fun with LD_PRELOAD. In *linux. conf. au*, volume 153, 2009.

[22] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, 2010.

[23] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[24] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *USENIX Conference on File and Storage Technologies (FAST'02)*, 2002.

[25] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. Membrane: Operating system support for restartable file systems. *ACM Transactions on Storage (TOS)*, 6(3):11, 2010.

[26] 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.

[27] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4):333–360, 2006.

[28] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

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

[30] Bharath Vangoor, Kumar Reddy, 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.

[31] 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.

[32] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.