Universiteit Leiden
The Netherlands

Opleiding Informatica

Enhancing Prefuse-SSHFS Performance Through Page and Metadata Caching

Leon Monster

Supervisors:
Dr. K.F.D. Rietveld
Dr. B.J.C. van Werkhoven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2025

Abstract

File systems are essential components of modern computing, enabling efficient data sharing across processes by organizing data on storage devices. Developing new file systems is complex, as they are typically part of the kernel. To simplify this process, Linux provides FUSE; an interface that allows one to implement file system operations in user space. Unfortunately, FUSE-based file systems suffer from significant performance overhead due to frequent context switches.

Prefuse aims to solve this problem by hooking system calls and directly routing them to the user space file system code, avoiding the costly context switches. While Prefuse demonstrates promising performance improvements, it exhibits increased latency when used with `SSHFS`.

This thesis investigates the cause of this latency by firstly reproducing the results of the Prefuse-`SSHFS` experiment. Next, we prove that the increase in latency is caused by a lack of page and metadata cache through qualitative and quantitative experiments. Finally, a basic page and metadata cache are implemented and their performance is measured.

The addition of caching to Prefuse-`SSHFS` results in significant performance improvements of up to 84%. However, in case of the `passthrough` file system, the same caching mechanism leads to reduced performance due to redundant caching: image-based FUSE file systems already leverage the kernel's page cache. These findings highlight the need to evaluate caching strategies in the context of each specific workload and file system, rather than applying them uniformly.

Contents

1 Introduction

File systems have been around since the 1960's and are a fundamental part of modern day computing [1]. Without them, multi-user and multi-process systems as we use nowadays would not be possible. The file system makes the storage device usable by giving it structure and by utilizing metadata. The type of structure and the kind of metadata depends on the specific file system. File systems also allow applications to share mass storage by making sure that processes do not access the storage in ways that would lead to race conditions, data corruption or data loss.

Typically, file systems are part of the operating system, residing in kernel space, as opposed to most other processes that run in user space. This has several disadvantages, one of which is the difficulty of development. Having your code run in kernel mode means that there is no memory protection, and any memory corruption, whether in user space of not, will cause the machine to crash [2]. Being part of the kernel also means that your code needs to be platform independent as operating systems are expected to work on multiple different architectures. To make the development of file systems easier, Linux comes with FUSE (Filesystem in USErspace) [3]. It exposes a simple interface through which one can easily come up with their own file system. It is widely in use, with popular file systems like `SSHFS` [4] and `s3fs` [5].

FUSE, however, has proven to be slow [6] in comparison with native file systems, due to its overhead, particularly in context switching. Over the years, several solutions have been proposed to reduce the overhead of FUSE [7, 8, 9, 10, 11], the most recent of which is Prefuse [12]. Prefuse allows users to run a file system as a dynamic library, located completely in userspace. This reduces overhead of context-switching, thereby improving performance. Prefuse outperforms FUSE and RFUSE [10] and is compatible with most client programs, making it an interesting option for running FUSE file systems with greater performance. This is especially the case for file system implementation that are not backed by an image file, such as network file systems.

However, when benchmarking `SSHFS` using the `filebench` [13] fileserver benchmark, Prefuse unexpectedly does not outperform FUSE, only achieving slightly higher throughput and lower overall latency. It was also found that using Prefuse in combination with `SSHFS` results in a significant increase in latency for most individual file system operation, as opposed to `SSHFS` with FUSE. In this thesis we will investigate the increased latency of Prefuse-`SSHFS` for the `filebench` benchmark.

1.1 Research Questions

We aim to answer the following research questions:

RQ1. What is causing the increase in latency of Prefuse-`SSHFS` in comparison with FUSE-`SSHFS`?

RQ2. What methods can be used to decrease the latency of Prefuse-`SSHFS` and can the latency be decreased to be on par or to surpass that of other state-of-the-art FUSE implementations?

Besides answering these questions, this thesis has contributed to the source code of Prefuse,

available on GitHub[1], through the addition of a simple page cache, metadata cache, bug fixes and documentation.

1.2 Thesis overview

This thesis is structured as follows: in Section 2 an overview of relevant concepts and technologies will be given. Related work will be discussed in Section 3. Section 4 presents an analysis of the latency problem, while Section 5 outlines design choices made during the implementation of the page and metadata cache, along with the reasoning behind them. Section 6 presents an experimental evaluation of the implemented caches. In Section 7 limitations of Prefuse and the page cache can be found, as well as suggestions for future work. Finally, Section 8 contains the conclusions drawn from the experiments.

This thesis was written as part of the Bachelor's programme in Computer Science at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University. The research was conducted under the supervision of dr. K.F.D. Rietveld, whose insights were instrumental throughout the process.

[1] https://github.com/sholtrop/prefuse

2 Background

This section provides an overview of relevant concepts and technologies needed to understand the problem and its solution. While some of these concepts apply to a range of operating systems, it is important to note that the following techniques are explained with a Linux-based operating system in mind.

2.1 Operating Systems

An operating system (OS) is a collection of software that manages the resource allocation of the system. This includes CPU time, memory, input devices and file systems. The OS functions as the middle man between applications and the system's hardware, providing an interface through which users can easily interact with the system's resources. Besides resource allocation, the operating system also acts as a process management entity, which includes tasks like process scheduling and inter process communication (IPC). Task scheduling is an research field on its own, with multiple different techniques that all have their strengths and weaknesses [14]. Task scheduling allows the system to work on multiple tasks at once instead of waiting for a task to finish, as was commonly done in the advent of computing.

Having multiple processes running at once does raise some security concerns, given that the processes run in the same physical memory. Therefore, modern day CPU's support virtual memory, giving each process its own address space, seperated and protected from other processes. This is done by a mapping, called a pagetable, from the virtual memory address a process uses to a physical memory address.

Each time the OS switches from one process to another process, it performs what is called a context switch. Here the state of the process currently being executed is stored and a previously stored state of another process is restored. Context switches are usually computationally expensive and their frequency should therefore be minimized.

Context switches also occur as a result of system calls made by user space programs. Although modern processors support multiple execution modes, operating systems conceptually operate using two main privilege modes: user mode and kernel mode. User mode, also known as user space, is a restricted environment in which regular applications run, with limited access to system resources. In contrast, kernel mode grants the operating system full access to the system's resources, such as the hardware and memory. System calls are a way in which an application can request a service from the operating system, such as reading a file, writing to a file or deleting a file. Once a system call is made, a context switch occurs to transition from the user space application to kernel mode, allowing the kernel to handle the system call. Once the kernel is done, there is another context switch, this time from kernel mode to user mode, and the result of the system call is returned to the application.

Reading data from a storage device is a costly operation. This is why operating systems tend to cache data that has been read previously in the random access memory (RAM). The difference between reading from RAM and reading from disk storage can be illustrated with the following:

suppose the latency of a single CPU instruction is scaled to 1 second, then a single read from RAM will take about 6 minutes and a read from rotational disk storage will take a stunning 1-12 months [15]. Reads from solid state drives (SSD's) are much faster, but would still take between 2 and 6 days, highlighting the necessity of caching. Most CPU's also implement some form of caching which is even faster then reading from RAM, however, operating systems have very little control over these caches.

2.2 File Systems

File systems (FS) are entities that govern file organization and access. They provide a data storage that allows applications to share mass storage. Without a file system, applications could access all data on the storage device, potentially leading to racing conditions, data corruption or even data loss. There are many file system designs and implementation, each with their strengths and weaknesses. Examples of file systems include `FAT`, `NTFS`, `APFS`, `ext4` and `XFS` [16, 17, 18, 19]. The most common approach to structuring data on storage devices is hierarchical [1], distinguishing between two entities: files and directories. Most file systems also store metadata about files and directories, such as size, creation time, owner, and more.

Modern operating systems allow the use of multiple file systems. This is made possible by the virtual file system (VFS), which exposes a uniform interface to access different types of file systems. It allows user application to interact with different types of file systems using the same application interface (API), regardless of the underlying file system format. This is achieved by defining a set of generic operations and data structures that all supported file systems must implement. Once a user application performs a system call, such as a `read`, the VFS maps this system call to calls specific to the file systems implementation. The VFS allows user applications to make use of a variety of file systems, without requiring changes to the source code.

Besides local file systems, as mentioned before, there are also network based file systems, running on top of the VFS. These file systems allow files to be stored on remote machines and can be accessed over a network as if they were stored on the local machine. Examples of network file systems include `SSHFS` [4] (see also Section 2.4), Amazon's `s3fs` [5] and Google's `GFS` [20].

2.3 FUSE

Like mentioned in Section 1, developing a file system that resides in kernel space may prove to be a difficult task. This is why FUSE has been developed. FUSE is a software interface for Unix-like operating systems that allows users to easily build their own file system. The file system implementation is run in user space and FUSE bridges the gap between the kernel and the source code. It does this by exposing a `struct` of function pointers [21], which can be implemented by the user. FUSE's functions include, but are not limited to: `getattr`, `readlink`, `mkdir`, `rmdir`, `rename`, `chmod`, `open`, `read`, `write`, `opendir`, `readdir`, `access`, `create` and `lseek`.

FUSE consists of three modules: the FUSE kernel module (`fuse.ko`), a userspace library (`libfuse`) and a mount utility (`fusermount`). Once an application performs a file system operation, such as reading a file, a system call is send to the kernel, via the VFS to the FUSE kernel module. The

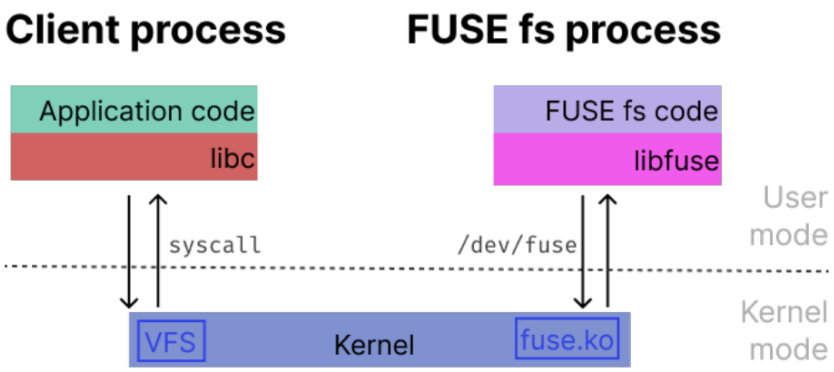


Figure 1: FUSE's architecture [12].

kernel module forwards the call to `libfuse` in userspace, which handles the actual file system operation. The result is returned in the same way: from `libfuse` back to `fuse.ko` in kernel mode, via the VFS to the user application in user space. Figure 1 provides a visual representation of this process.

As illustrated in Figure 1, each file system operation done on a FUSE-based file system involves at least four context switches. As explained in Section 2.1, these context switches cause overhead, which should be minimized to improve performance. FUSE has been proven to be relatively slow [6], primarily due to this overhead. Nevertheless, FUSE remains widely adopted due to its flexibility and ease of use.

2.4 SSHFS

One of the FUSE-based network file systems is `SSHFS` [4]. `SSHFS` allows the user to mount a remote file system using the SSH File Transfer Protocol (`SFTP`). It comes with all major Linux distributions and has been in production use across a wide range of systems. Using `SSHFS` requires only invoking the executable with a mountpoint and remote host. Once mounted, the user is able to access files under the mountpoint as if they were stored locally, even though they physically reside on a remote machine.

Once an application makes a system call to a file under the `SSHFS` mountpoint, the kernel's VFS handles the system call and routes it, via the FUSE kernel module, to the `SSHFS` user space process. Next, the `SSHFS` user space process translates the generic file system operation into `SFTP` commands, which are then send to the remote. On the remote side, the `sftp-server` daemon processes these commands and interacts with the actual file system. The response is then packaged and send back via `SFTP` to the local machine, through `SSHFS`, the FUSE kernel module and the VFS, back to the user application. An visual representation of this process can be found in Figure 2.

2.5 zpoline

`zpoline` is a novel technique introduced by Yasukata et al. [22] that allows for system call hooking on x86-64 CPU's. System call hooking is the process of intercepting system calls before they get to

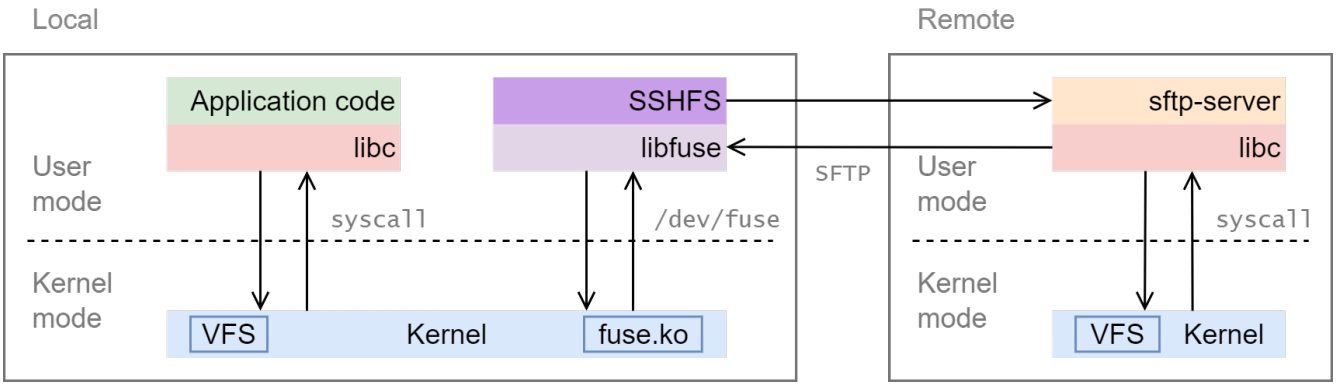


Figure 2: SSHFS's architecture.

the kernel, and alter or augment their behavior in some way. Hooking is used in many applications such as debuggers and benchmarkers. There exist a number of solutions that allow for system call hooking, such as `strace`/`ptrace` [23, 24] and `eBPF` [25], however `zpoline` has little overhead, allows for extensive hooking and does not require changes to the source code of the user space program, among other advantages [22].

`zpoline` operates by performing binary rewriting of user space binaries, replacing system call instructions with short jumps to custom trampoline code. This trampoline code redirects execution to a user-defined handler which can optionally invoke the original system call. To enable this replacement, `zpoline` requires modification of the `/proc/sys/vm/mmap_min_addr` Linux kernel tunable; otherwise the low memory addresses needed for the jump targets may be inaccessible. While `zpoline` introduces some startup overhead due to its dynamic binary rewriting, this cost is offset by its low runtime overhead and high performance during system call interception.

2.6 Prefuse

Prefuse is the result of the master thesis of Sjors Holtrop [12]. It allows one to run a file system as a dynamic library, fully in user space, completely avoiding context switches, leading to a significant increase in performance. Prefuse is backwards compatible with existing FUSE file systems and offers great compatibility with client programs, thanks to its use of `zpoline`. According to the thesis, it has far better performance than FUSE, making it an interesting option to use in a production environment.

Prefuse consists of three main parts: `zpoline`, `libprefuse` and `FUSE-file-system-as-a-library` (`FFAAL`), as can also be seen in Figure 3. It works as follows: before the application runs, `zpoline` binary rewrites the system calls and configures them to jump to `libprefuse`'s hook function. Once the application makes a system call, the system call is not sent to the kernel, but rather to `libprefuse`'s hook function. Next, `libprefuse` checks if the system call is meant for a file under the mountpoint of a FUSE file system. If this is not the case, the system call is forwarded to the kernel, thus not affecting other system calls that do not use the mountpoint. Otherwise, Prefuse converts the system call according to FUSE's requirements and invokes the corresponding FUSE

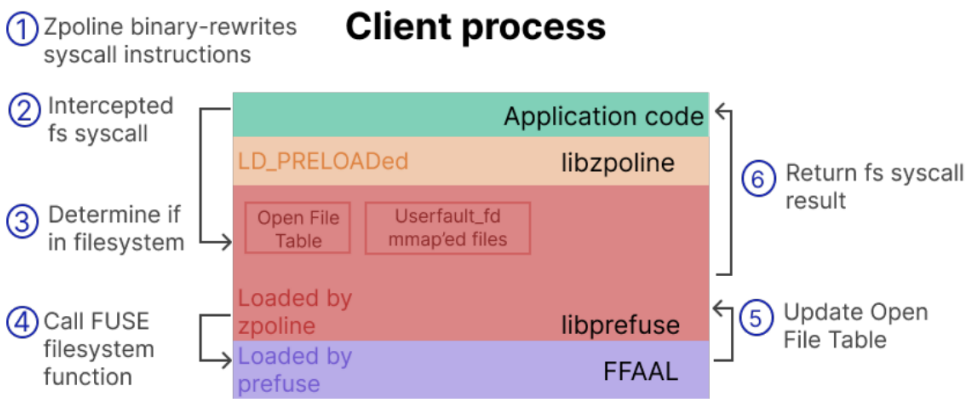


Figure 3: Overview of Prefuse's data flow [12].

function. The result of the FUSE function is then returned via `libprefuse` to the application.

2.7 Caching Interplay

Caching interplay refers to the coordination among various caching layers, determining what data is cached, where it is cached, how long it is retained and who is responsible for invalidation or refreshing. A precise understanding of these responsibilities is essential for interpreting the latency behavior as discussed in Section 4.2. In this section, we focus on the caching behavior of FUSE-based image and network file systems, and examine how Prefuse influences these mechanisms.

As illustrated in Figure 1, all file system operations to a FUSE file system, regardless of it being image- or network-based, are routed through the kernel. In the case of an image-based FUSE file system, the image must first be read from the underlying storage device into memory before FUSE can access its contents. Most operating systems treat file system images as regular files, thus FUSE can issue `open`, `read`, `pread` and `close` system calls on the image just like any other file. These system calls are handled by the kernel, which may serve data directly from the page cache if the image, or parts of it, have been accessed recently.
When a user process makes a system call targeting a file under the FUSE mountpoint, the kernel forwards the request to the FUSE daemon. The daemon then reads the relevant data from the image file, interprets it, and sends the result back to the kernel. The kernel, in turn, delivers the result to the application via the VFS. Meanwhile, it caches the result in the page cache, so that subsequent accesses to the same data range can be served directly from the cache, bypassing both the storage layer and the FUSE daemon entirely [26].

When coupled with Prefuse, these files are still being cached by the kernel. Prefuse is only responsible for routing the system calls directly to the FUSE file system code, which in turn uses regular system calls like `pread`. The key difference is that Prefuse always invokes the FUSE daemon, which then issues regular system calls, which may or may not be served from the kernel's cache, depending on the current state of the cache.

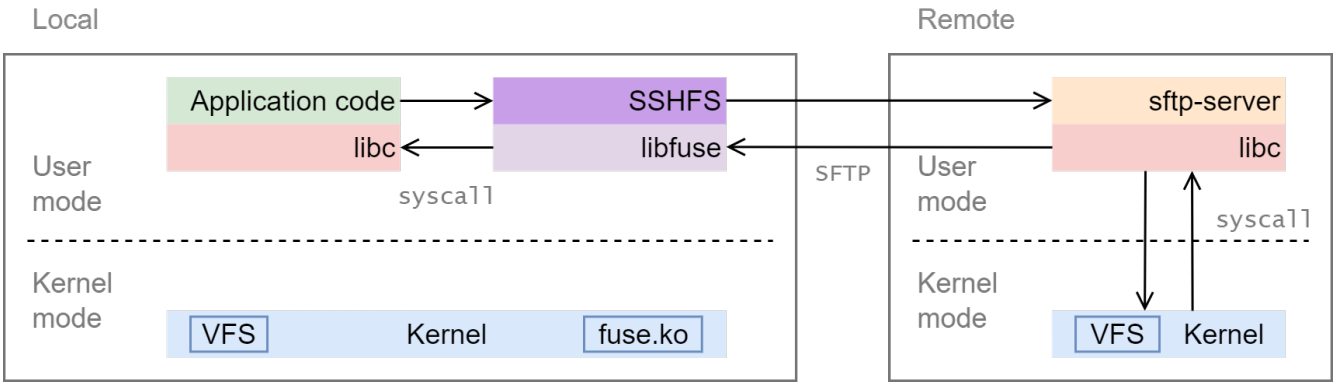


Figure 4: SSHFS's architecture in combination with Prefuse.

In the case of network-based FUSE file systems, the FUSE implementation packages each system call into a network message, formatted according to the specific protocol used, and sends it to a remote machine. On the remote end, a user space process receives the request and issues corresponding system calls to its kernel to fulfill it. The remote kernel handles the operation and may cache data as it deems appropriate. Once the result is ready, it is transmitted back over the network to the originating machine, via the file system code, through the kernel to the user application. Despite the data originating from a remote source, the local kernel is still able to cache the received content since the result is routed through the VFS. If the application accesses a page that remains in the local cache, the kernel is able to serve the request directly [26], bypassing the need to query the remote machine, thereby reducing latency and improving performance. As a result, the same data may be cached on both the remote and the local machine (not without risks, see also Section 7.2).

Using Prefuse in combination with network-based FUSE file systems changes the caching mechanics. Since system calls bypass the local kernel, it can no longer cache data retrieved from the remote machine. As a result, every system call triggers a new request to the remote machine, increasing latency and degrading overall performance. However, the remote kernel still performs caching of data in the same way as with FUSE. A visualization of the system call path for `SSHFS` in combination with Prefuse can be found in Figure 4.

3 Related Work

This section contains a selective overview of prior work, aimed at improving or measuring FUSE's performance. In addition to research specifically on FUSE, we also include relevant studies on caching techniques and recent developments in this field.

3.1 To FUSE or Not to FUSE

As mentioned in the introduction, FUSE is widely known for its performance limitations. Vangoor et al. [27] conducted a comprehensive study on FUSE's performance for a wide range of workloads, hardware configurations and FUSE settings. The results indicate that performance degradation ranged from unnoticeable to -83%, even with FUSE optimizations enabled. Additionally, they observed an increase in CPU utilization of up to 31%, largely due to overheads such as context switching.

One of the key FUSE settings leveraged in the paper was the write-back cache option, which, when enabled, enhances performance by batching write operations. While this thesis does not implement a full write-back strategy, it adopts a write-through approach instead. Specifically, writes in Prefuse are applied to both the cache and the backing storage synchronously. This strategy reduces the amount of cache entries that need to be invalidated, thus increasing the hit-ratio and improving performance. Due to the in-process nature of Prefuse, features like background flushing are not feasible. Section 5 outlines the implementation details of the cache, while Section 7.1 discusses its trade-offs and limitations in more detail.

3.2 RFUSE

RFUSE is a novel user space file system framework by Kyu-Jin Cho et al. [10] designed to enhance the performance of FUSE-based file systems by optimizing communication between the kernel and user space. RFUSE introduces a new form of communication between kernel and user space. It employs a per-core ring buffer as a communication channel, minimizing transmission overhead caused by context switches and data copying. RFUSE is compatible with existing FUSE-based file systems without any modifications, making adoption and integration simple. Besides the compatibility with existing file systems, RFUSE also achieves a throughput close to that of the in-kernel file system `ext4`, while reducing latency up to 53% compared to traditional FUSE implementations.

3.3 LDPFUSE

LDPFUSE [9] is a framework by the same author as Prefuse that allows custom file systems to be compiled as a dynamic library. It works by exporting file system functions with the same name as those provided by `glibc`. By pre-loading the library, these custom implementations overwrite the original file system calls in the source code. Since the file system resides in the same address space as the client process, context switches are no longer needed. Pre-loading can be done by setting the `LD_PRELOAD` environment variable, instructing the dynamic linker to pre-load the given shared libraries before others. This mechanism enables one to overwrite functions with a custom implementation. However, `LDPFUSE` has some major limitations. Most importantly, it can't intercept

all file system operations, and it doesn't work with binaries running in secure execution mode. These issues were a big part of what inspired the development of Prefuse.

Similar to LDPFUSE, Prefuse also makes use of the `LD_PRELOAD` environment variable. However, instead of rewriting function implementations, it is used to load `zpoline`, which performs binary rewriting of system calls. See Section 2.5 for more details.

3.4 CuttleFS

CuttleFS is one of the results of a research paper by A. Rebello et al. [28] in which the researchers investigate how three Linux file systems (`ext4`, `XFS` and `Btrfs`) behave in the presence of failures. The commonalities and differences between these file systems are studied. Besides file systems, the team also looked at how five widely used applications (`PostgreSQL`, `LMDB`, `LevelDB`, `SQLite` and `Redis`) handle `fsync` failures. An `fsync` failure occurs when the file system's attempt to persist updates to the backing storage using the `fsync` system call fails, indicating that recent changes may not have been safely stored and the file system consistency could be compromised. The key finding: none of the failure-handling methods were sufficient, with `fsync` failures causing data loss and corruption.

In order to deterministically cause `fsync` failures, one of the deliverables of the paper by A. Rebello et al. is `CuttleFS`. It is a purpose-build user space file system with its own page cache that allows for emulation of `fsync` failure behaviors, giving the user full control over fault injection. While both Prefuse, as extended by this thesis, and `CuttleFS` implement a fine-grained user space page cache, their motivations differ. `CuttleFS` is engineered to emulate a variety or file system failures, whereas Prefuse is designed to support custom file system logic and performance optimizations. This paper also confirms our motivation for not implementing a write-back cache: we can not rely too heavily on `fsync`. See Section 7.1 for more details.

3.5 Unified Buffer Cache (UBC)

One of the fundamental challenges in operating system design is the separation between the file system buffer cache and the virtual memory page cache. This led to redundant caching, inefficient memory usage, and increased complexity in maintaining consistency between the two. Silvers [29] addressed this problem with his implementation of the Unified Buffer Cache (UBC) for NetBSD, which unified the file system and VM caches into a single mechanism. This approach reduced data duplication and simplified cache coherency by allowing both subsystems to share the same physical pages for file data. This was later implemented in other kernels, including Linux.

While it is not possible to unify caches at the physical memory level due to our userspace-based approach, we do cache both data and metadata in the same class. Our cache can be tuned as seen fit, as opposed to in-kernel caches like UBC, making it an attractive feature for many use cases.

3.6 TriCache

Feng et al. [30] propose `TriCache`: a user transparent, multi-level block cache to enable high performance out-of-core processing. It addresses the limitations of traditional operating system page caches, particularly their inefficiency on cache misses and lack of scalability, by offering a virtual memory abstraction, built on top of a block interface. The results show that `TriCache` can outperform the Linux kernel's page cache, while staying compatible with existing systems so it can be integrated without significant changes.

The research conducted by Feng et al. illustrates that other systems are also moving cache management from the operating system into user space, in order to achieve better performance and to give more fine-grained control over the cache. In the case of Prefuse, implementing a user space page cache was a somewhat constrained design decision, given that we bypass the kernel entirely. However, `TriCache` shows that user space caching is not inherently a performance compromise, but is able to match or even outperform the kernel's page cache.

4 Analysis

In this section, an analysis is conducted to identify the cause of the increased latency in Prefuse-SSHFS. First, the results from the Prefuse thesis will be validated, followed by an investigation into the latency through three experiments.

4.1 Reproducibility Study

Before we can analyze and address the significant latency observed with SSHFS on Prefuse, we need to be able to reproduce the performance results as presented in [12]. Due to the scope of this thesis, only the filebench benchmark with the fileserver workload will be reproduced. This process was rather time-consuming due to technical issues arising from the combination of fuse2 and Ubuntu 22.04, as well as the use of an incorrect filebench version; the benchmark requires version 1.5.alpha1 or newer. Prior versions do not support cvars which are used in the fileserver workload.

The benchmark was run using the same adjusted fileserver workload as used in the Prefuse thesis: 3000 files with file sizes following a Gamma distribution (mean: 128KiB, shape: 1.5), 50 threads, mean read/write size: 1MiB, mean append size: 128KiB. The workload ran for 60 seconds, and we took the average of 100 runs.

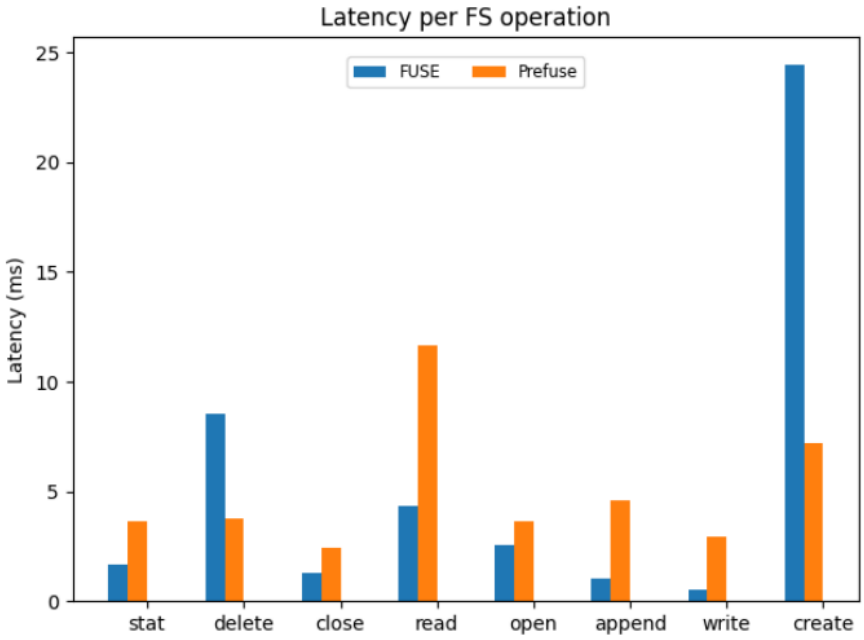
As shown in Figure 5, the latency in the reproduced benchmark closely matches the original in terms of magnitude. In both experiments, the operations stat, close, open, append, and write exhibit higher latency when using Prefuse compared to FUSE. delete and create on the other hand perform better with Prefuse. There are some minor differences in the raw latency values. For instance, read with FUSE has a latency of around 5ms in the original benchmark, whereas it increases to around 7ms of latency in the reproduced experiment. This can be explained by a difference in hardware used for the benchmark: the original results were obtained using a Samsung 980 PRO M.2 NVMe SSD in full passthrough, while the reproduced benchmark was run using a standard Kingston A400 SATA SSD.

These results suggest that the original benchmark was valid and that its findings can be reproduced.

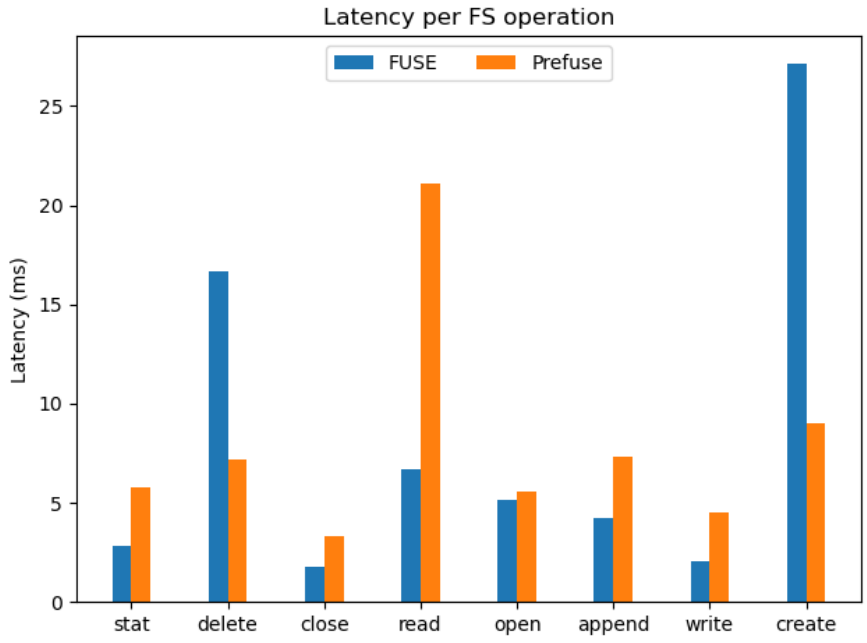
4.2 Latency Analysis

After successfully verifying the findings presented in the Prefuse paper, we proceeded to investigate the root cause of the increase in latency observed in Prefuse. We hypothesize that the increase in latency for Prefuse is due to the lack of a page cache and metadata cache. In contrast, FUSE leverages the kernel's caching abilities, significantly reducing access times, as explain in Section 2.7.

To investigate this hypothesis, we have conducted a series of experiments using a slightly modified version of the test_program found in the Prefuse source code under examples/sshfs/simple_read. Unlike the original version, which reads 512 bytes with no offset, our modified version reads a large 1GiB file in chucks of 1MiB. The reads start at an offset of 0 which is incremented by 1024 bytes for each subsequent read. In total, 100 read system calls are performed, and the total time taken



(a) Original experiment [12]



(b) Reproduced experiment

Figure 5: Latency per file operation: original (a) and reproduced (b)

is recorded. The entire experiment is run 100 times and the results are averaged. Note that the average is calculated across the total times of 100 separate experiment runs, each consisting of 100 reads, rather than across the 100 individual read calls within a single run. With this setup, the reads start at offset 0 and continue up to offset $99 \times 1024 = 101376$. The final byte read is then $101376 + \texttt{1MiB} = 1149952$, or approximately `1.15MB`. This ensures the accessed data lies beyond the first page, as opposed to the original `test_program`, increasing the chance of observing cache effects.

In the following experiments we make use of flags. When using `open` on a file, flags can be provided to influence how the file is accessed. For example, flags like `O_RDONLY` and `O_WRONLY` only allow reading the file or writing to the file respectively. The `O_DIRECT` flag tries to minimize cache effects of the I/O to and from the file, meaning file operations are done directly to and from user space buffers. This flag is interesting for our experiments since it allows us to compare FUSE's performance with and without caching.

4.2.1 Number of Reads

By inserting a `print` statement in the source code of `SSHFS` and recompiling it, we were able to count the number of `SSHFS_read` operations performed during the execution of the `test_program`.

As shown in Table 1, Prefuse performs the expected 100 reads, while FUSE performs only 10. By adding the size and offset to the `print` statement, we observed that FUSE is not reading in chunks of `1MiB` as specified in the `test_program`, but rather in `128KiB` chunks. The offsets are also not as expected, with the first read reading at offset 0 and subsequent reads reading at increments of `128KiB`, indicating that previous reads are saved and only data that has not previously been read is requested. This can be confirmed by calculating the cached byte range: $10 \times \texttt{128KiB} = 1310720$, meaning bytes up to 1310720 are cached. The last byte read, at offset 1149952, falls within the cached range, confirming that all necessary data was already cached, and no additional pages needed to be loaded.

In contrast to FUSE, Prefuse reads in chunks of `1MiB` at the offsets specified in the `test_program`. Besides the difference in read behavior, there is also a significant difference in performance: FUSE completes the test approximately 98% faster than Prefuse.

Set-up	Number of reads	Avg. time (ms)	Std. Dev. (ms)	`fincore` page count
FUSE	10	8.513	0.677	368
FUSE + `O_DIRECT`	100	372.620	69.398	336
Prefuse	100	357.898	63.713	560
Prefuse + `O_DIRECT`	100	359.282	69.526	560

Table 1: Results of `SSHFS` read experiments: read count, latency analysis and page cache presence across different set-ups.

4.2.2 FUSE with O_DIRECT

To investigate the impact of the `O_DIRECT` flag, and by extension the kernel's page cache, the experiment as described in Section 4.2.1 was repeated, only this time the file is opened with the `O_DIRECT` flag enabled. In Table 1 we can clearly see that the performance of FUSE gets significantly worse, closely matching Prefuse's performance. This can be attributed to the local kernel no longer caching the results from the remote, resulting in a query to the remote for every read operation, causing significant overhead.

The `O_DIRECT` flag does not seem to influence the performance of Prefuse. This suggests that Prefuse does not benefit from the local cache. If Prefuse were to benefit from the local page cache, enabling the `O_DIRECT` flag should lead to a decrease in performance, which is not observed as the difference in the average time for Prefuse with and without `O_DIRECT` is not statistically significant ($p = 0.8835$).

4.2.3 Page Cache Presence

`fincore` [31] is a Linux utility that counts the number of pages of file content being resident in memory. By using this tool on our `1GiB` file, immediately after executing the `test_program`, we can determine whether the file's content remain cached in memory.

`mincore` [32] is a Linux system call that checks which pages of a memory-mapped file are currently resident in memory. By mapping the file into memory and invoking `mincore`, we can obtain a page-level view of the file's presence in the page cache.

Using these tools, we examined the caching behavior of the file across different configurations. Surprisingly, the file was cached in all cases tested. This behavior is likely due to how the experiment is setup: the remote endpoint is `localhost`, meaning that `sftp-server` runs on the same machine as the test program. Because `sftp-server` relies on regular system calls, it allows the kernel to cache data according to its standard policies, which is why `fincore` is able to find the file in the page cache. Notably, the use of the `O_DIRECT` flag does not appear to influence the caching behavior on the remote machine, as the file was still cached even in setups using this flag. An additional observation is that Prefuse shows around 200 more pages cached than FUSE. Since each page is `4KiB`, this corresponds to roughly `800KiB` of additional data held in memory. To investigate this difference, we used `mincore` to determine which pages of the file were cached, aiming to rule out the possibility that the beginning of the file had been evicted in the FUSE setup. The results indicated that no such evictions occurred. Instead, Prefuse appears to cache a larger portion of the file overall. The reason for this isn't obvious and could be an interesting point to explore further.

To prevent `sftp-server` from invalidating our results, we repeated the experiment using a dedicated remote machine. Although we were unable to find the `1GiB` file in the kernel's page cache using `fincore`, FUSE without `O_DIRECT` still outperformed the other configurations significantly. Manual page presence checks using `mincore` did provide meaningful data, as shown in Table 2. These results clearly demonstrate that only FUSE is leveraging the page cache, while the other configurations do not. Notably, the number of cached pages differs between Table 1 and Table 2 for the FUSE set-up, likely due to differing read behaviors between `sftp-server` and `SSHFS`. We chose not to include

timing averages since communication between two devices over a network is prone to variability, whereas using the `localhost` loopback interface is rather consistent.

Set-up	`mincore` page count
FUSE	288
FUSE + `O_DIRECT`	0
Prefuse	0
Prefuse + `O_DIRECT`	0

Table 2: Results of `mincore` page count for `SSHFS` reads using two distinct machines.

These three experiments have proven that FUSE is able to utilize the local kernel's page cache, resulting in significantly higher performance compared to Prefuse. This highlights the need for an in-process page cache for Prefuse to close the performance gap.

5 Design and Implementation

In this section, any design choices made during the implementation of the page and metadata cache and the reasoning behind them are explained.

5.1 Prefuse Page Cache

From Section 4.2 it is clear that FUSE leverages the kernel's page cache, improving performance significantly, whereas Prefuse does not. To make Prefuse a viable alternative to FUSE, it must at least match FUSE's performance. The most straightforward approach is to implement an in-process page cache, which is exactly what we have done. We introduced a new class `PageCache` which serves as a wrapper around our underlying caching data structure. This class includes additional fields such as the number of entries and the page size, as well as convenient methods for straight forward interation with the cache. It also keeps track of key metrics like cache hits, misses and invalidations. Since Prefuse is used as a library, printing cache statistics on exit is non-trivial. Rust does not support destructors for static classes, so to overcome this limitation, the cache uses shared memory to persist its statistics. These statistics are then printed on exit of the helper binary of Prefuse.

The data structure used is the `moka::sync::Cache` [33] class. Initially, we attempted using the `lru::LruCache` [34], however this implementation requires a lock for each read or write operation. Since Prefuse is designed to support multi-threading, our initial tests revealed that this particular LRU cache caused significant performance degradation due to threads blocking while waiting for locks. In contrast, `moka::sync::Cache` supports fully concurrent lock-free reads, though writes still require a mutex lock, making it much better suited for our requirements. `moka::sync::Cache` offers two eviction policies: the default `TinyLFU`, which is a combination of LRU and LFU, and a standard LRU policy. We have opted to use LRU as eviction policy, due to its simplicity and predictable eviction behavior, which aligns well with typical file system access patterns.

The cache is of type `<PageKey, Page>`, where `PageKey` is a tuple consisting of a filename and a page number relative to the start of the file. Ideally, the key would be a tuple of an inode and the page number; however, obtaining the inode requires issuing another system call, which would add overhead to the cache, reducing its efficiency. The `Page` struct contains a vector holding the data and a `size` field indicating how much of this data is valid. This `size` field was added to support reading from partial pages in the cache, ensuring that reads do not extend beyond the end of the file. The cache is initialized once on the first time it is accessed, also called lazy initialization, and is static so that all threads share the same cache instance.

The cache is used as follows: during a `read` system call that is delegated to Prefuse, we first verify that caching is enabled. If it is, we determine the page numbers that correspond to the requested read range. To determine which memory pages are affected by the `read` operation, we compute the index of the first and last page touched. This is done as follows:

```
let page_first = offset as usize / page_size;
let page_last  = (offset as usize + count - 1) / page_size;
```

Here, `page_first` represents the index of the first memory page accessed by the read. It is calculated by dividing the byte `offset` (where the read begins) by the `page_size` of the page cache and casting this to an integer. `page_last` gives the index of the last page that the read ends on. This is calculated by taking the offset plus the number of bytes to read (count), subtracting 1 (since offsets are 0-based), and then dividing by the page size, again casting to an integer.

For each of the page numbers in this range (inclusive), we check whether the page is present in the cache. If the page is cached, the requested bytes are copied from the cached page into the buffer and we continue with the next page. If the page is not found in the cache, a regular, full page `read` is performed and the result is copied to a new `Page` struct, which is then inserted into the cache. The relevant portion of the result is then copied into the user-provided buffer. `Write` operations follow a similar logic, with the key difference that they do not issue a `read` in the case of a page miss. Currently, the caching method used is `write-through`. Ideally the cache would be `write-back`, however this is a non-trivial task. See Section 7.1 for a more in-depth discussion. On `write` system calls we have opted to not invalidate the pages in the cache, but rather update them, after which the write is also done to the backing storage. This reduces the number of evictions, thus improving performance.

5.2 Prefuse Metadata Cache

Initial tests using Prefuse's page cache showed promising results, with significantly improved latency compared to when the page cache was disabled. However, further investigation into FUSE's read policy showed that FUSE does not read in chuncks of `1MiB`, as specified in the `filebench` workload. Instead, it reads in smaller batches (pages), with the final read often being of seemingly arbitrary size. Our hypotheses is that FUSE uses the metadata of a file to avoid reading past its size, which causes the final read operation to have a variable length. We aimed to replicate this behavior to explore whether metadata caching could further improve Prefuse's performance. Therefore, we added another `moka::sync::Cache` to our `Cache` class with the file name and FUSE's `stat` struct as the key-value pair.

To utilize the metadata cache, the `prefuse_read` function had to be altered slightly: on each page cache miss, it tries to retrieve the file size from the metadata cache. If the file size is cached, we verify that the requested read does not extend beyond the end of the file (EOF). In such cases, we read only up to EOF, cache the result if needed and return immediately. If the file size is not available in the cache, we invoke the `prefuse_stat` function, which is used for the `fstat` system call, to obtain the file size. Although this introduces some overhead, it does not cause a noticeable increase in latency since file systems, `SSHFS` included [4], often perform directory and file attribute caching themselves. The main performance benefit comes from avoiding unnecessarily large reads beyond EOF. Avoiding repeated `fstat` calls provides an additional, though smaller, improvement.

To maintain cache consistency, entries are invalidated upon file changes. Since Prefuse does not yet implement all system calls, this currently applies to `chmod`, `chown`, `release`, `unlink` and `write` system calls. In addition to the `read` system call utilizing the metadata cache, the `stat` system calls have also been modified to use cached data whenever it is available.

5.3 Configuration

To allow for enabling and the configuration of both caches, we have extended Prefuse's configuration file with fields such as `page_size` and `page_cache_size`. An overview of the configuration file extension can be found in Figure 6.

```
# Enable write-though page and metadata cache, default: false
cache = true
# Number of entries in the page cache, default: 5000
page_cache_size = 7500
# Size of pages to cache in bytes, default: 128KiB
page_size = 262144
# Number of entries in the metadata cache, default: 5000
meta_cache_size = 2500
```

Figure 6: Prefuse's extended configuration file.

6 Experiments

In this section we evaluate the performance of Prefuse with caching enabled against FUSE and Prefuse without caching. This is done by making use of various workloads where metrics like latency, throughput and execution time are measured.

6.1 Experiment Setup

Hardware. All experiments were performed on the same machine, running a Ryzen 5 2600x @3.6GHz, 16GB of dual channel RAM @3200MT/s and a Kingston A400 SATA SSD.

Software. The experiments were run on Ubuntu 24.04 LTS using WSL2. The `filebench` version used is version 1.5alpha3, as found in the `RFUSE` repository [10]. This specific version is custom compiled to allocate more shared memory to support bigger workloads.

6.2 Page Cache Benchmark

To interpret the results of experiments done on Prefuse with caching enabled, it is important to know the characteristics of the actual page- and metadata cache. For this, we have written multiple benchmarks that interact directly with the cache implementation, bypassing Prefuse entirely. The benchmarks do not rely on a file system, as all operations are performed in RAM. They are implemented using the `criterion` benchmark library, which automatically performs statistical analysis and visualizes the results. Each benchmark records the time taken to perform 1000 operations (such as reads or writes) in a single iteration, which in turn is repeated 100 times. The benchmarks use a page cache with `1000` entries and `128KiB` pages. An overview of all benchmarks and their behaviors can be found in Table 3.

Benchmark	Description
`cache_write`	Write 1000 pages to an empty cache.
`cache_overwrite`	Write 1000 pages to an already filled cache.
`cache_read`	Read 1000 existing pages from the cache.
`cache_read_miss`	Read 1000 non-existing pages from the cache.
`metacache_write`	Write 1000 file stat structs to an empty cache.
`metacache_overwrite`	Write 1000 file stat structs to an already filled cache.
`metacache_read`	Read 1000 existing file stat structs from the cache.
`metacache_read_miss`	Read 1000 non-existing file stat structs from the cache.

Table 3: Overview of Page Cache and Metadata Cache Benchmarks

The benchmark results can be found in Table 4. It shows that writing to a cache takes considerably longer than reading from it. This is due to the fact that reading from the `moka::sync::Cache` does not require a lock, whereas writing to it does. From the difference between `cache_write` and `cache_overwrite` we can see that evicting an entry also add overhead. Looking at Table 4, it is clear that both caches perform very well, with single operations only taking between 0.1 and 1 µs on average.

Benchmark	Mean (µs)	Std. Dev. (µs)
cache_write	928	55
cache_overwrite	1041	67
cache_read	223	11
cache_read_miss	155	9
metacache_write	1162	74
metacache_overwrite	1192	78
metacache_read	266	21
metacache_read_miss	185	10

Table 4: Benchmark results: mean execution time and standard deviation (in microseconds) for 1000 operations, averaged over 100 runs.

6.3 File Server Workload

This benchmark will prove whether Prefuse with page caching enabled improves the per operation latencies as shown in Figure 5. The workload used is the same as the one used in the results verification experiment: 3000 files with file sizes following a Gamma distribution (mean: `128KiB`, shape: 1.5), 50 threads, mean read/write size: `1MiB`, mean append size: `128KiB`. The workload ran for 60 seconds, and we took the average of 100 runs. The metadata cache is kept constant at 5000 entries, which is sufficient since the workload uses 3000 files.

6.3.1 SSHFS

Page cache configuration has a significant impact on the performance of an application. Therefore, we first conduct an experiment to find the optimal page size/cache size ratio for this particular workload in combination with `SSHFS`. In order to decrease the duration of this experiment, we took the average latency for the `read` operation of only 10 runs for each combination of page size and cache size. We tested the following page sizes: `32KiB`, `64KiB`, `128KiB`, `256KiB` and `512KiB`. Cache sizes tested were: `5000`, `7500`, `10000`, `125000` and `15000`.

The results, shown in Figure 7, reveal that smaller pages (`32KiB` and `64KiB`) are insufficiently sized for effective caching in this workload. We also observed a trend: increasing the number of cache entries generally reduces the latency per operation. This is expected behavior, since caching more data results in more hits and thus less cache miss overhead. The best performance was achieved using `512KiB` pages with a cache size of `12500` entries, resulting in an average latency of 6.3ms per operation. Despite this, we have opted to use the second best performing combination of `256KiB` pages with a cache size of `7500`. This decision is made due to practical considerations: a `512KiB` x `12500` cache would consume up to `6.3GiB` of memory, while the chosen configuration (`256KiB` x `7500`) limits the cache footprint to approximately `1.9GiB`. This trade-off ensures that Prefuse performs well, without excessively consuming system resources.

Now that we have found a good configuration for the page cache, we can repeat the experiment as described in Section 4.1, only this time with the Prefuse cache enabled. The metadata cache size is kept constant at `5000` entries. We also conduct additional runs with only the metadata cache

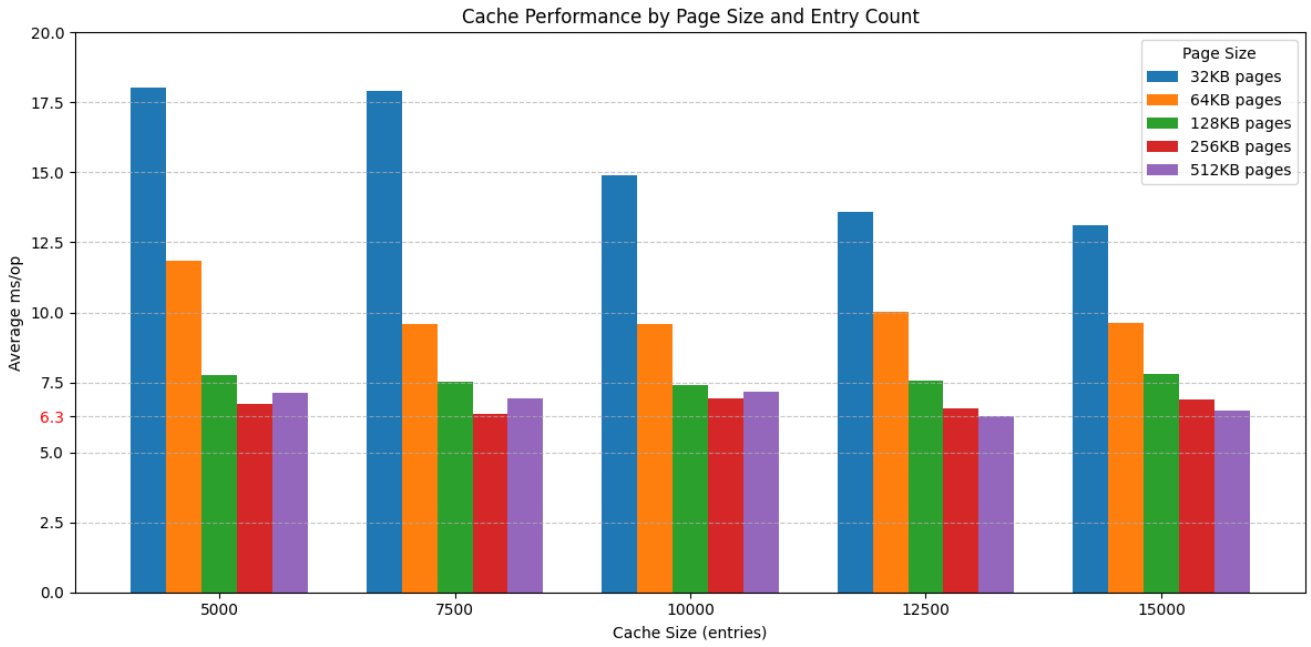


Figure 7: Cache performance by page size and entry count for the fileserver workload in combination with `SSHFS`.

enabled, as well as only the page cache, to isolate and compare their individual and combined effects.

The results, as presented in Figure 8, look very promising, showing a notable performance improvement across all file system operations when using both metadata and page cache. This outcome was somewhat unexpected, given that the cache was specifically designed to improve the latency for the `stat` and `read` system calls. These proved to be accelerated, with a relative acceleration for the `stat` and `read` system calls of 52% and 84% respectively. `stat` now matches FUSE's performance, whereas `read` is almost twice as fast.

We initially expected the latency for `write` to increase, due to the additional overhead of writing to both the page cache as well as the backing storage device. This overhead is evident when comparing the performance of `write` with only the metadata cache enabled (which avoids write-through) to that with only the page cache enabled. Contrary to our expectations, the latency for `write` and other file system operations, besides `stat` and `read`, also decreased when using the combined cache. We hypothesize that the performance improvements in `stat` and `read` help reduce overall I/O pressure. This, in turn, frees up system resources and allows other threads to execute more efficiently, resulting in broader performance gains beyond the operations directly targeted by the cache.

Furthermore, it is clear that most of the performance gain of the `read` operation can be attributed to the page cache. Interestingly, the performance of `stat` with only the metadata cache enabled is almost equal to that achieved with only the page cache. At the same time, both caches appear to

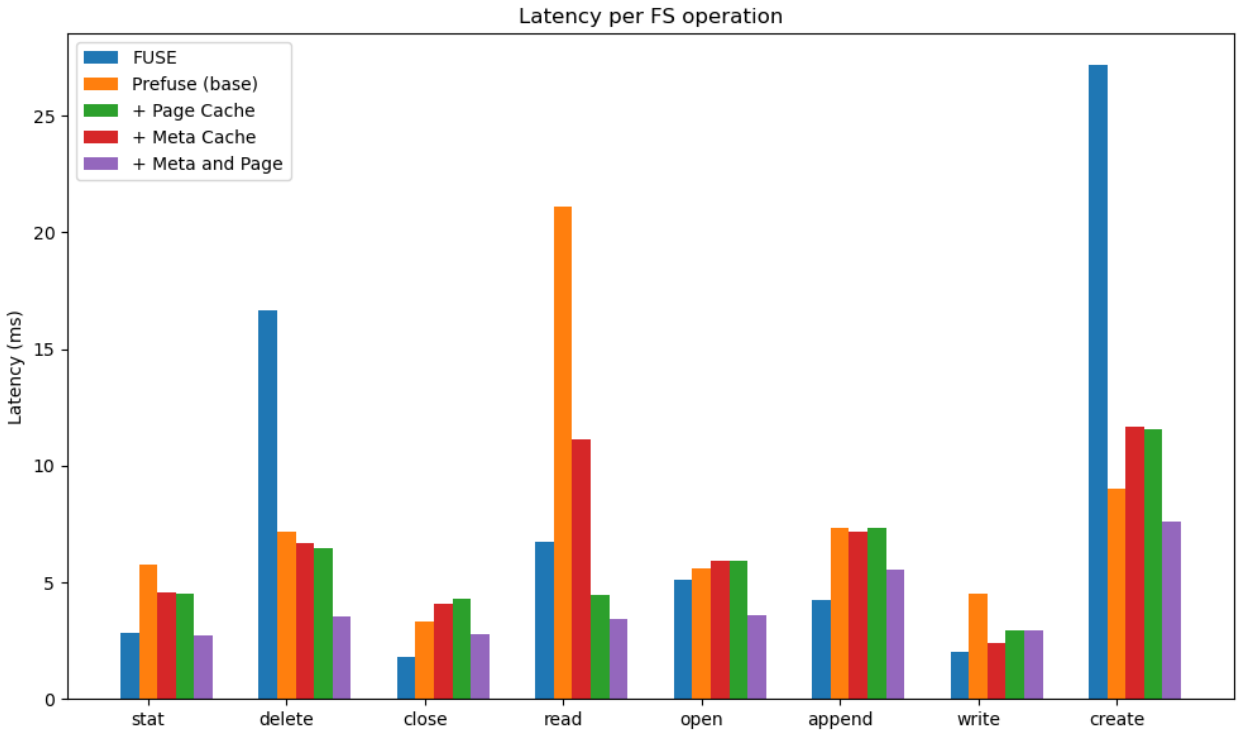


Figure 8: Latency per file system operations using FUSE, baseline Prefuse, and Prefuse with page and metadata caching enabled (`SSHFS`).

introduce overhead, as operations not directly influenced by the cache, such as `close`, `open` and `create`, slow down.

6.3.2 Passthrough

Since the underlying file system significantly affects system performance, it is important to repeat the previous experiment using alternative file systems. For this purpose, we have selected the `passthrough` file system. Initial tests revealed that the cache configuration used in the previous experiment was not a good fit for this file system. Therefore the cache configuration experiment was also repeated. All other parameters were kept consistent with the original setup.

The results shown in Figure 9 indicate that, unlike `SSHFS`, the `passthrough` file system performs worse with larger page size when page caching is enabled. For our setup, the optimal cache configuration was found to be `64KiB` pages with a total of `12500` pages, resulting in a maximum memory footprint of approximately `0.8GiB`. Using this configuration, we repeated the latency experiment for the fileserver workload with `passthrough` as backing file system. The experiment follows the same methodology as descibed in Section 6.3.1, with the only difference being the file system used. Based on the previous experiment, the combined cache configuration consistently outperformed the individual caches. Therefore, we have chosen to test only the combined cache in this experiment. The FUSE results are excluded from Figure 10 to maintain readability, as its much larger scale (1-5ms compared to 0-150µs) would distort the visualization.

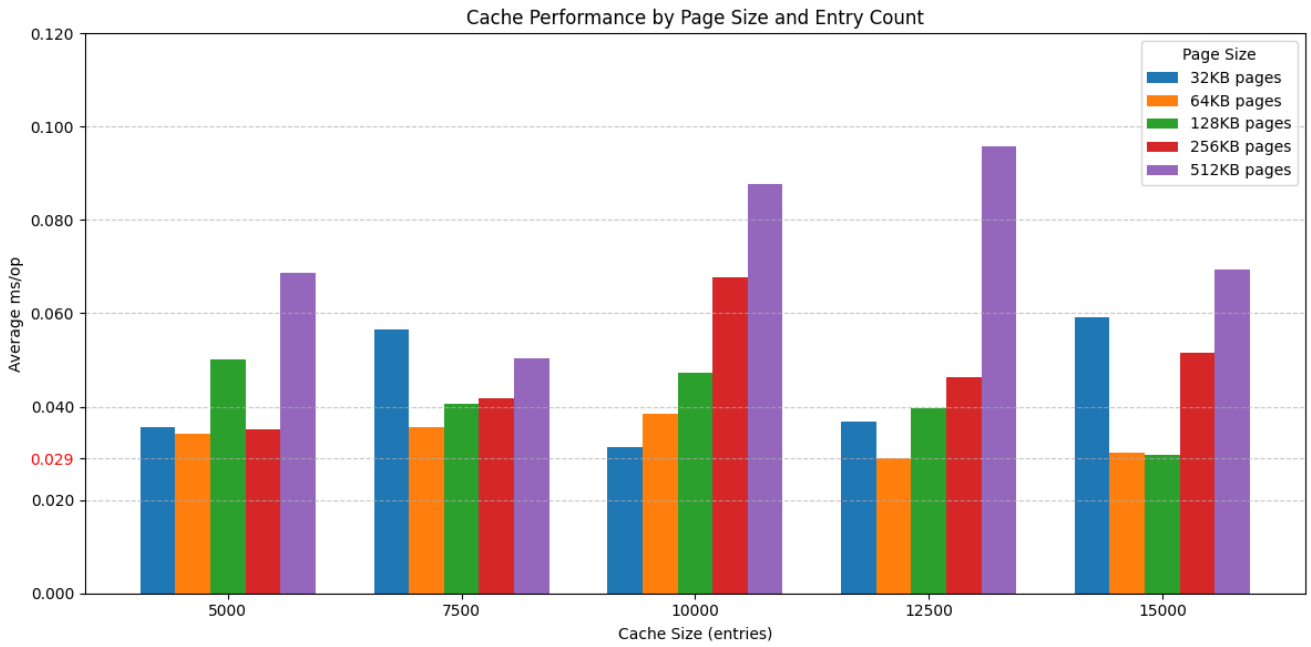


Figure 9: Cache performance by page size and entry count for the fileserver workload in combination with `passthrough`.

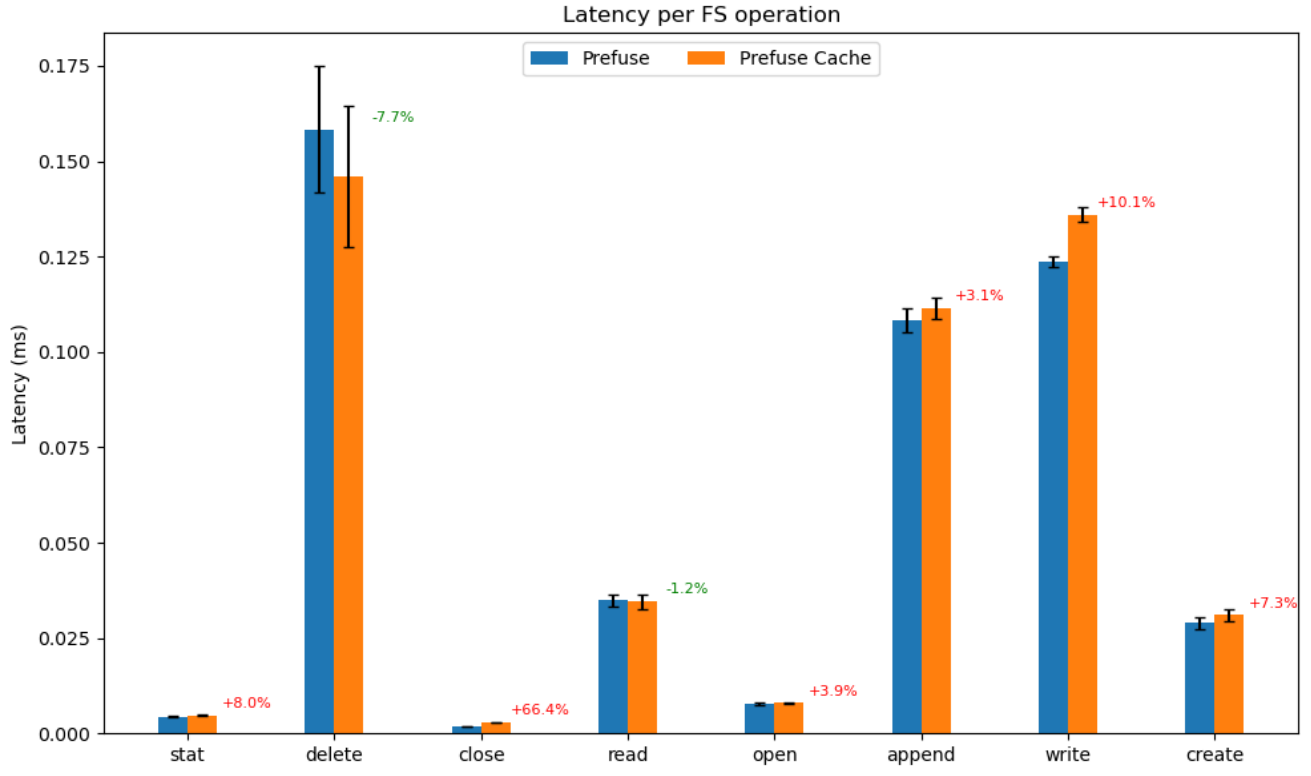


Figure 10: Latency per file system operation for Prefuse and Prefuse with page and metadata cache enabled (`passthrough`). Error bars represent the standard error of the mean (SEM).

From the results shown in Figure 10, it is clear that the `passthrough` file system does not benefit from our page and metadata cache. Only the `read` and `delete` system calls show slight improvement, the latter likely not due to our cache, but rather the randomness in the workload's fileset. For `stat`, the overhead introduced by the cache actually outweighs the benefits, making it about 8% slower when caching is enabled. The `write` operation also sees a performance drops, which is logical since it can not take advantage of the page cache. Instead, the added latency is due to the overhead of writing to both the cache and the backing storage. Despite `filebench`'s millisecond time resolution, the observed 66.4% increase in latency for the `close` operation is supported by very low standard errors (3.25×10^{-5} vs. 4.71×10^{-5}), suggesting the result is statistically significant. However, the absolute difference is minimal (approximately 1µs) and is unlikely to have any practical impact. The underlying cause of this increase remains unclear.

The underwhelming results are likely due to the kernel caching files read by `passthrough`, resulting in double caching, both by Prefuse as well as the kernel. As outlined in Section 2.7, Prefuse forwards system calls under the mountpoint to `passthrough`, which invokes regular system calls and allows the kernel to cache the requested data. Prefuse then caches that same data in user space, resulting in redundant caching and overhead. We validated this hypothesis by conducting an experiment similar to that in Section 4.2.3, using `passthrough` as the backing file system. When Prefuse was run without the `O_DIRECT` flag, 416 pages were found in the kernel's page cache. In contrast, enabling `O_DIRECT` resulted in no cached pages, confirming that `passthrough` leverages the kernel's page cache by default.

7 Discussion

Throughout the course of this thesis, we encountered several limitations of Prefuse and its cache. In this section, we outline and discuss the most notable ones.

7.1 Flushing Dirty Pages on Exit

Initially the page cache was only used by `read` system calls, while `write` operations simply invalidated any pages affected. This led to a decrease in hit-ratio and therefore we researched the idea to make the cache fully write-back. This entails writing to the cache, marking the page as dirty and then return the result of the `write` operation to the application. A separate thread then occasionally writes all dirty pages to the backing storage. Typically, flushing dirty pages is managed by the kernel. The kernel retains memory mappings even after a process exits, ensuring that all dirty pages are eventually written back to the disk.

Prefuse, however, runs entirely in user space as a library (`libprefuse`) and therefore lacks a centralized entry or exit point. The cache is implemented as a static class, which means it has no destructor or similar mechanism to flush dirty pages when the process ends. Even if such a mechanism were introduced, it would still not guarantee correctness in cases where the process terminates unexpectedly, for example due to a signal.

Another method that could be used to flush dirty pages is to force the underlying file system to make a `fsync` (or a similar) system call before it exits. However, as demonstrated by A. Rebello et al. [28] (see Section 3.4), this method is not foolproof: failures of the `fsync` sytem call still result in data loss and corruption. Moreover, not all file systems explicitly call `fsync` on exit by default, `SSHFS` being one of them, which requires us to modify their source code; an undesirable solution.

The author of the Prefuse paper [12] also acknowledges this limitation in the context of memory-mapped files. He similarly concluded that, in user space, it is not possible to provide the same guarantees as the kernel when it comes to writing back dirty pages.

To improve the hit-ratio of the page cache, we have opted to write to both the cache and the backing storage. While this approach introduces some overhead to `write` system calls, it ensures cache consistency and allows subsequent reads to benefit from cached data, improving overall performance for read-heavy workloads.

7.2 SSHFS Page Caching

While caching can significantly improve performance for `SSHFS`, it also introduces a risk of data inconsistency. Since the cache operates locally, any changes made on the remote side, such as by another user or process, are not reflected in the cache. As a result, cached data might become stale, leading to potential mismatches between what the application sees and what is actually written on the backing storage. This undermines the reliability of the file system in environments where multiple clients or processes may access and modify shared data. Therefore, caching in combination with `SSHFS` should be used with caution.

It is worth noting that this issue is not specific to Prefuse-`SSHFS`, but also applies more broadly to network-based file systems. As described in Section 2.7, the kernel maintains its own data cache for FUSE file systems. However, the kernel is unaware that the underlying file system is remote and potentially mutable by other clients, which further increases the risk of stale data.

7.3 Future Research

Future work could focus on extending the use of the cache to support more file system operations, such as `append`, `truncate`, and other modifications that currently bypass caching logic. This would further improve performance and consistency across a wider range of workloads.

Additionally, implementing the same write-through mechanism the page cache has for the metadata cache could reduce the number of entry invalidations, resulting in reduced latency for all operations that use file metadata in some way.

Another area worth exploring is reducing the cache overhead for the `read` system call. Currently, pages are page-sized and zeroed on initialization, whereas we might want to store a partial page, leading to unnecessary memory usage and overhead. Investigating techniques such as lazy allocation could help reduce this overhead, further increasing performance.

Finally, another important direction would be to explore cache invalidation mechanisms, such as time-based expiration or remote change detection, to reduce the risk of stale data in environments where files can be modified outside of the current client, like mentioned in Section 7.2. These improvements would make the caching system more robust and adaptable to real-world use cases.

8 Conclusion

The increase in per-operation latency for Prefuse when used with `SSHFS`, as identified in the Prefuse paper [12] and confirmed by our measurements (Figure 5), is primarily due to the absence of a caching mechanism. This finding directly answers Research Question 1.

While introducing a page cache reduced `read` latency to approximately one quarter of the original, we aimed to more closely replicate the kernel's caching behavior by leveraging file metadata, such as file size. This led to the implementation of a dedicated metadata cache. Together, these two caches significantly improved performance for operations that make direct use of cached data, most notably `read` and `stat`, which saw speedups of 84% and 52% respectively. However, these performance gains were only observed in the context of the `SSHFS` file system. When the same caching mechanism was tested on the `passthrough` file system, no major improvements were observed; in fact, most operations showed a slight decline in performance. The passthrough FUSE file system directly accesses files stored on the local file system, thereby already leveraging the local kernel's page cache. Consequently, adding a user-space cache via Prefuse results in redundant (double) caching without any measurable performance benefit, and should therefore be avoided in such cases.

This answers Research Question 2: the implementation of a page cache and metadata cache in Prefuse improves the performance of specific operations, namely those that directly benefit from caching, within the `SSHFS` file system. However, these improvements do not generalize to all operations or file systems.

In conclusion, adding a metadata cache and page cache to Prefuse significantly improved performance for the `SSHFS` file system, while it had an adverse effect on the `passthrough` file system. This highlights the importance of empirically evaluating the impact of caching and its configuration for each specific workload and backing file system before deployment. As we have seen in this thesis, user-space caching should be avoided for FUSE file systems that can leverage the local kernel's page cache, as this introduces redundant caching without tangible performance benefits. This includes file systems that directly access files that are stored on the local file system. The impact on performance of image-based file systems that access an image file, rather than a regular file, on the local file system, will be investigated in future work. Nonetheless, we believe that further extending the cache's capabilities, applying it to a broader range of system calls, and reducing its overhead could make it a valuable enhancement to Prefuse.

References

[1] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), page 213–229, New York, NY, USA, 1965. Association for Computing Machinery.

[2] Rusty Russell. Unreliable guide to hacking the linux kernel. https://docs.kernel.org/kernel-hacking/hacking.html. Accessed: 2025-05-06.

[3] Fuse. https://docs.kernel.org/filesystems/fuse.html. Accessed: 2025-05-06.

[4] Libfuse/sshfs. https://github.com/libfuse/sshfs. Accessed: 2025-05-06.

[5] s3fs-fuse. https://github.com/s3fs-fuse/s3fs-fuse. Accessed: 2025-05-06.

[6] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. Performance and resource utilization of fuse user-space file systems. *ACM Trans. Storage*, 15(2), May 2019.

[7] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W Yu. Direct fuse: Removing the middleman for high-performance fuse file system support. *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, 2018.

[8] A. Bijlani and U Ramachandran. Extension framework for file systems in user space. *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[9] S. Holtrop. Linker-directive based file system in userspace: Introducing ldp fuse. Bachelor's thesis, LIACS, Leiden University, 2022.

[10] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. RFUSE: Modernizing userspace filesystem framework through scalable Kernel-Userspace communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 141–157, Santa Clara, CA, February 2024. USENIX Association.

[11] Qianbo Huai, Windsor W. Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. Xfuse: An infrastructure for running filesystem services in user space. In *USENIX Annual Technical Conference*, 2021.

[12] S. Holtrop. Prefuse: Fuse-file-systems-as-a-library for a faster, consistent, backwards-compatible fuse. Master's thesis, LIACS, Leiden University, 2025.

[13] Filebench - a model based file system workload generator. https://github.com/filebench/filebench. Accessed: 2025-03-10.

[14] Abraham Silberschatz. *Operating System Concepts*. John Wiley & Sons, Chichester, England, February 2009.

[15] Brendan Gregg. *Systems performance*. Prentice Hall, Philadelphia, PA, October 2013.

[16] Overview of fat, hpfs, and ntfs file systems. https://learn.microsoft.com/en-us/troubleshoot/windows-client/backup-and-storage/fat-hpfs-and-ntfs-file-systems. Accessed: 2025-05-09.

[17] Role of apple file system. https://support.apple.com/nl-nl/guide/security/seca6147599e/web. Accessed: 2025-06-18.

[18] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. *Usenix Association*, 32(3):25–30, 2007.

[19] R.Y. Wang and T.E. Anderson. xfs: a wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 71–78, 1993.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *19th ACM Symposium on Operating Systems Principles*, 2003.

[21] Fuse operations struct reference. https://libfuse.github.io/doxygen/structfuse__operations.html. Accessed: 2025-05-12.

[22] 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, Boston, MA, July 2023. USENIX Association.

[23] strace. https://man7.org/linux/man-pages/man1/strace.1.html. Accessed: 2025-05-13.

[24] ptrace. https://man7.org/linux/man-pages/man2/ptrace.2.html. Accessed: 2025-05-13.

[25] ebpf documentation. https://ebpf.io/what-is-ebpf/. Accessed: 2025-05-13.

[26] Fuse i/o modes. https://docs.kernel.org/filesystems/fuse-io.html. Accessed: 2025-06-26.

[27] 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, Santa Clara, CA, February 2017. USENIX Association.

[28] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? *ACM Trans. Storage*, 17(2), June 2021.

[29] Chuck Silvers and The NetBSD Project. Ubc: an efficient unified i/o and memory caching subsystem for netbsd. *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*, 2000.

[30] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. Tricache: A user-transparent block cache enabling high-performance out-of-core processing with in-memory programs. *ACM Trans. Storage*, 19(2), March 2023.

[31] Masatake Yamato. fincore. https://man7.org/linux/man-pages/man1/fincore.1.html. Accessed: 2025-05-16.

[32] mincore. https://man7.org/linux/man-pages/man2/mincore.2.html. Accessed: 2025-06-24.

[33] Cache in moka::sync: A thread-safe concurrent synchronous in-memory cache. https://docs.rs/moka/latest/moka/sync/struct.Cache.html. Accessed: 2025-05-17.

[34] Lrucache. https://docs.rs/lru/latest/lru/struct.LruCache.html. Accessed: 2025-05-17.