Karlsruhe Institute of Technology

Analyzing Application Writeback Behavior

Bachelor's Thesis
submitted by

David Besau

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Daniel Habicht, M.Sc.
	Yussuf Khalil, M.Sc.

December 3, 2024 – April 3, 2025

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, April 3, 2025

Abstract

Many applications with strong data persistence requirements rely on `fsync()` to actively persist data and ensure the consistency of file contents on disk. However, `fsync()` is associated with many problems regarding performance, reliability, and the risk of data loss or corruption. In order to improve the design of interfaces like `fsync()`, more knowledge about how applications use them in practice is required. Building on top of well-established tracing mechanisms for the Linux kernel, we present a novel approach to examining the synchronous writeback behavior of applications and implement it for the ext4 file system. We examine the synchronous writeback behavior of different consumer applications, and present our results using a visualization technique that has not previously been applied to data about synchronous writeback. Based on our results, we provide several insights, including that most synchronization requests cause little to no writeback and that writeback latency is not always proportional to the amount of writeback. Our tooling is based on a modular design and can be extended in various ways to include new data sources and answer different high-level questions about the writeback behavior of applications. By building on our approach, future research can provide a better understanding of application writeback behavior in order to improve the design of both kernel interfaces and applications.

Acknowledgments

First and foremost, I would like to thank my advisor, Daniel Habicht, for guiding me through the process of writing this thesis and always being available to answer my questions with patience.

I would also like to thank Tim Krause as well as Isadora Todorescu for their thorough proofreading and their valuable feedback.

Finally, I would like to express my gratitude to Yūna Kurosawa for always being there for me and giving me the strength to do my very best.

Contents

Chapter 1

Introduction

I/O performance is an important consideration in the design of both hardware and software. In the interplay of hardware and software, the abstraction an operating system (OS) provides to applications plays a crucial role in determining how efficiently and reliably applications can use storage. On modern OSs, buffered I/O has proven to be an important tool in improving I/O performance, and buffered I/O is the default way to perform I/O in many OSs including Linux. However, using buffered I/O comes with drawbacks: If an application relies on the OS to perform writeback in the background, it has no knowledge of when data is actually written to disk. In the case of unbuffered I/O, a failure at any given time can only affect the I/O operation currently in progress, and the application is usually notified of the failure via an error code. With buffered I/O, an I/O error, a power failure, or a system crash can potentially affect any number of I/O operations that are currently buffered by the OS and have not yet completed. Furthermore, it can be difficult for the application to detect I/O errors and recover lost or corrupted data, since errors can occur after the operation has already returned successfully from the viewpoint of the application. To give applications using buffered I/O some control over data writeback, additional OS interfaces are required, which in turn come with their own overhead and potential side effects. This means that the choice of whether to use buffered I/O represents a trade-off between performance on the one side and reliability and data integrity on the other side.

For applications using buffered I/O which require strong persistence of files as well as high performance, there are multiple mechanisms available to achieve a balance in this trade-off. On Linux as well as on POSIX-compliant systems, the most important of these mechanisms are the `fsync()` and `fdatasync()` syscalls. Via these syscalls, an application can request that the kernel flushes the page cache for an open file, i.e., it forces all buffered

writes of that file to disk [23]. These syscalls only return once all writeback operations have completed. While these syscalls by themselves are not enough to provide strong data persistence guarantees, they can be an important tool for applications aiming to ensure that files are in a known, consistent state on disk before continuing [35].

However, under certain circumstances, the performance and reliability of `fsync()` and `fdatasync()` can be problematic. The issues surrounding these syscalls have been the longstanding subject of discussions among developers [15, 43, 16, 73, 13, 18, 14, 75] as well as academic research [7, 59, 41, 6, 64, 53]. Some of the most important issues have been:

1. Active persistence using `fsync()` and related syscalls has a high performance overhead [73, 6, 53, 64, 63, 22, 50, 21].
2. The way `fsync()` and related syscalls handle and report errors is unreliable [23, 14, 13, 43].
3. The behavior of `fsync()` and related syscalls varies across systems and is difficult to understand [59, 23].
4. `fsync()` and similar syscalls are often accidentally called indirectly via library functions [41].
5. The interface provided by `fsync()` and similar syscalls is not well-suited for many modern applications [32, 47].

Shirwadkar, Kadekodi, and Tso [64] have implemented changes in the Linux kernel that improve the performance of `fsync()` and `fdatasync()` significantly on ext4. But some issues remain, for example how to improve error handling and reporting [17]. While alternatives to `fsync()` and related syscalls exist, they each come with their own drawbacks.

Previous research has shown that `fsync()` and similar syscalls can exhibit problematic behavior [59, 41, 6, 64, 53]. Yet, in many cases it is not well documented under which specific conditions these problems occur. Much remains unknown about the behavior of these syscalls and how applications use them in practice. We believe this to be part of the reason why proposals for new interfaces or changes to existing interfaces have often been unsuccessful and known issues have not been resolved. Our contributions in this thesis are twofold: First, we describe the design and implementation of tooling for tracing the synchronous writeback behavior of applications. Using this tooling, we collect data about how different consumer applications use `fsync()` and related syscalls in practice as well as the actual writeback behavior caused by the usage of these syscalls. Secondly, we analyze the collected data to identify concrete situations in which the behavior of `fsync()` and related syscalls is

problematic and discuss potential solutions to these problems. We present a novel approach to visualizing writeback behavior of applications using flame graphs [28]. We also compare our findings to previous research to determine whether we can confirm previous findings or add new insights to them. Our goal is not only that we can provide new insights into the writeback behavior of applications and propose potential solutions to existing problems, but also that the approach and tooling we present can benefit future research and the design of new interfaces for data persistence.

For our experiments, we focus on popular consumer applications because they are widely used, sometimes handle important data, and can exhibit complex I/O behavior, yet they have not been the subject of research to the same degree as, for example, HPC applications. We also think that, due to their interactive nature, consumer applications make it easy to associate a specific user input with expected I/O behavior, and thus serve as intuitive examples to demonstrate our approach. However, our approach is generally applicable to any type of application, as long as it makes use of mechanisms for active persistence.

This thesis is structured as follows: In Chapter 2, we describe the behavior of `fsync()` and other related syscalls and the problems associated with them. We also discuss alternatives to these syscalls and their respective advantages and disadvantages as well as proposed changes to how these syscalls work. In Chapter 3, we give an overview of previous research regarding the issues we present, including proposals to improve the performance of `fsync()` , analyses of how `fsync()` failures are handled, and general surveys of the I/O behavior of applications. We compare our approach to previous work and discuss how previous work has improved some of these issues and where there is still need for further research. In Chapter 4, we present our approach to gaining insights into the writeback behavior of applications and the design of the tooling we use to trace synchronous writeback. In Chapter 5, we describe the implementation of our writeback tracer and how we process the collected data for analysis and visualization. In Chapter 6, we present the results of our analysis and discuss the implications of our findings. Finally, in Chapter 7, we summarize our findings and discuss their implications for future work, as well as the design and usage of mechanisms for data persistence.

Chapter 2

Background

In this chapter, we lay out background information about the `fsync()` and `fdatasync()` syscalls, as well as issues associated with them and alternatives to them. The information in this chapter will serve as a starting point for our subsequent analysis.

First, we outline the general behavior of `fsync()`, `fdatasync()`, and other related syscalls in Section 2.1. In Section 2.2, we discuss common reasons for using `fsync()`, `fdatasync()`, and related syscalls. Section 2.3 gives an overview of the most important problems currently associated with `fsync()` and `fdatasync()`. Finally, we discuss potential methods for solving or mitigating these problems, including existing alternatives as well as proposed changes to `fsync()` and `fdatasync()`, in Section 2.4.

2.1 Behavior of `fsync()` and Related Syscalls

`fsync()`, `fdatasync()` and `msync()` are defined as system interfaces in the POSIX standard [35]. According to POSIX, `fsync()` requests the OS to transfer all data for a given file descriptor to the device storing the corresponding file, and the `fsync()` call only returns once that transfer is finished [35]. The data that `fsync()` causes to be transferred includes any file system information associated with that file. `fdatasync()` has the same effect as `fsync()`, with the exception that, in addition to the file's data, `fdatasync()` only causes file system information to be transferred, which is required for a subsequent successful retrieval of the file's data [35]. `msync()` is called on an address range in memory and requests that the OS transfer all modified data within that range to permanent storage, if said data belongs to a file that is mapped into memory [35]. It should be noted that, while these three syscalls all request the OS to transfer data to a storage device, in the case

of `fsync()` and `fdatasync()`, the nature of this transfer is implementation-defined [35]. This means that it is explicitly allowed that a call to `fsync()` or `fdatasync()` has no effect whatsoever, for example, if the data is already safe from a power failure, or when "the system cannot assure non-volatile storage under any circumstances" [35].

The Linux kernel provides the syscalls `fsync()`, `fdatasync()`, and `msync()` in accordance with the POSIX standard [23, 49]. The Linux man page for `fsync()` and `fdatasync()` adds that a call to one of these syscalls will also cause any caches of the underlying storage device to be flushed [23]. On Linux, the exact behavior of `fsync()` and related syscalls, including any potential side-effects, depends on each individual file system's implementation. However, the man page does note that, in addition to calling `fsync()` or `fdatasync()` on a file, it is required to also call `fsync()` or `fdatasync()` on the parent directory in order to ensure that the directory entry pointing to that file has also reached disk [23].

From this point forward, due to the similarity of `fsync()` and `fdatasync()`, we will use the notion `f*sync()` whenever an argument or explanation applies equally to both syscalls. Furthermore, while `msync()` works in way very similar to `f*sync()`, it is significantly less used and less discussed in literature than `f*sync()`. Therefore, we will focus on `f*sync()` in this thesis. However, we expect that many of the arguments we make also apply to `msync()`.

For the sake of completeness, we will also briefly mention the `sync()` and `syncfs()` syscalls. According to the POSIX standard, `sync()` shall flush all outstanding writes to all file systems [35]. Unlike with `fsync()`, for example, POSIX' `sync()` does not necessarily wait for the data transfer to be completed before returning [35]. However, the Linux version of `sync()` does wait for the data transfer to be completed before returning [66]. Linux adds the `syncfs()` syscall, which works analogously to `sync()`, but only synchronizes a single file system [66]. Thus, the effects of `sync()` and `syncfs()` on Linux are equivalent to calling `fsync()` on every open file on the system and on every file of a file system, respectively, with the exception that `sync()` or `syncfs()` do not specify the order in which changes are written back. For this reason, we will not further discuss these syscalls separately in this thesis. On Linux, there is also the `sync_file_range()` syscall, which allows an application to sync only a part of a file instead of the whole file. We will discuss it in more detail in Section 2.4.1.

2.2 Rationale for Using fsync() and Related Syscalls

The POSIX standard states protection against data loss in the event of a system crash or other failure as the purpose of `fsync()` [35]. In addition, `fsync()` can also be used by applications to ensure that files on disk are in a consistent state. For example, to implement atomic commits and rollback, SQLite can use either a rollback journal or a Write-Ahead Log (WAL), both of which make use of `fsync()` to ensure the consistency of database contents [80]. By default, PostgreSQL's WAL relies on `fdatasync()` to ensure data consistency [79].

The purpose of `msync()`, according to the POSIX standard, is ensuring data integrity of memory-mapped files on disk [35].

2.3 Problems with f*sync()

Researchers, kernel developers, and application developers have pointed out and discussed several problems with `f*sync()`and its different implementations [73, 6, 53, 64, 63, 22, 50, 21, 23, 14, 13, 43, 59, 41, 32, 47]. Since their publication, changes in the Linux kernel have mitigated or fixed some of these issues, while others remain a topic of ongoing discussion. We have compiled a list of the most important issues to date and divided them into five groups of related issues:

1. `f*sync()` has a significant performance overhead [73, 6, 53, 64, 63, 22, 50, 21].
2. `f*sync()` is error-prone [23, 14, 13, 43].
3. The behavior of `f*sync()` varies across systems and is not well understood [59, 23].
4. `f*sync()` is often accidentally called indirectly via library functions [41].
5. The interface provided by `f*sync()` is not suitable for many modern applications [32, 47].

2.3.1 Performance

Regarding the performance of `f*sync()`, we have identified three central problems:

Calls to `f*sync()` are blocking and have a high latency. A call to `f*sync()` has a significant performance overhead for two reasons: First, the overhead of the context switch associated with any syscall, and secondly, the time it takes for the writeback operation to complete. For example, Trudeau has shown that the latency of an `fsync()` call following a single-byte write on a file can range anywhere from 17 ms to 66 ms on HDDs and from 0.14 ms to 9.3 ms on SSDs, using ext4 or XFS, two very common file systems [73]. It should be noted that in the same series of measurements, using `fdatasync()` instead of `fsync()` can reduce the latency by 50% or more, depending on the device [73]. Still, the overhead of both of these syscalls remains significant. For applications that make many calls to `f*sync()`, this can become a significant performance bottleneck.

Due to the way `f*sync()` is implemented in some file systems, a call to `f*sync()` can cause significantly more writeback than the caller intended. This problem exists specifically on the ext4 and F2FS file systems. On ext4, an `f*sync()` call traditionally causes a journal transaction to be committed, which in turn flushes all dirty data in the page cache, thus increasing the performance overhead of `f*sync()` [6]. Park and Shin [53] have demonstrated a way to increase `f*sync()` performance on ext4 by introducing a separate journal to record file-level transactions for `f*sync()` calls, thus relieving the need to commit a transaction for the main journal on every `f*sync()` call. Shirwadkar, Kadekodi, and Tso have implemented changes based on said work of Park and Shin starting with kernel version 5.10, as they describe in "FastCommit" [64]. However, the authors specifically excluded the change causing `f*sync()` not to flush dirty data for all files due to concerns that this change would break compatibility with applications relying on the old behavior, as Shirwadkar explained on the linux-ext4 mailing list [63]. On F2FS, the situation is similar, as an `f*sync()` call involves committing a checkpoint, which in turn requires flushing all dirty data in the page cache [6].

While `f*sync()` performance on other file systems has not been researched as extensively as on ext4 or F2FS, the general issue of `f*sync()` causing writeback of more data than the file `f*sync()` was called on seems to exist on other file systems as well, albeit to different degrees. XFS, for example,

does not require the entire page cache to be flushed for each `f*sync()`, but flushes all transactions up to the last transaction that touched the inode of the file `f*sync()` was called on [22].

Btrfs' implementation of `f*sync()` makes use of a mechanism called *tree logging* to ensure that `f*sync()` calls do not cause more writeback than necessary. Internally, Btrfs uses different types of data structures called *items* to represent file system concepts like inodes, directories, attributes, etc., with each item being stored in one of multiple tree data structure [50]. When an application calls `f*sync()` on a file, Btrfs copies all items related to that file which have been modified since the last transaction commit to a separate tree called the *log tree*, which is then flushed to disk [21]. However, due to the unique architecture of Btrfs, this approach is not directly applicable to other file systems with different internal data structures.

`f*sync()` can have an increased latency and cause additional garbage collection overhead on flash storage. Chang et al. [6] have pointed out that, if `f*sync()` is called frequently, data writes have little chance to be optimized by page caching or request scheduling and merging, resulting in many small random block writes to the underlying storage device. This not only leads to a generally high total time overhead on the I/O path, but can also increase the overhead of garbage collection in flash storage devices [6].

2.3.2 Risk of Errors

While the man page of `fsync()` on Linux states that data will be flushed "so that all changed information can be retrieved even if the system crashes or is rebooted," [23] there is no guarantee that a file's dirty data has been written to disk, even after an `f*sync()` call has returned successfully.

First, even though the man page defines multiple error numbers for use by `f*sync()` on Linux, in some cases, `f*sync()` may return successfully even though an error was encountered during the writeback operation. In particular, when a buffered write operation fails due to a hardware-level error, file systems will respond in different ways, but usually their behavior includes silently discarding the data in the affected pages and marking them as clean [14].

Secondly, even if an error is reported by the kernel, it might not be visible to every application interested in it. This can be the case, for example, if an error occurs during write operations before an application calls `f*sync()`. Current kernels store information about I/O errors in the `address_space` data structures that control the mapping between pages in the page cache and the underlying storage [13]. In theory, this mechanism should enable the first

`f*sync()` call on a file descriptor after an error has occurred to return an error code. However, this only applies to errors that occur after a process' `open()` call on a file [13]. This can make reliable error reporting difficult to achieve for applications consisting of multiple processes. For example, the PostgreSQL server uses a central "checkpointer" process to keep all files in a consistent state on disk. Since the checkpointer process usually does not keep all relevant files open, it might never learn of an error that, for example, occurred during background writeback after another process wrote to a file [14]. In 2018, this issue caused outrage among PostgreSQL developers who learned of this behavior, as well as significant discussion among Linux kernel developers [14].

Furthermore, Larsson [43] pointed out another reason why a successful `f*sync()` call does not guarantee that data has been persisted. They note that, while `f*sync()` requests the storage device to flush its caches, there is no way of telling whether the device actually complied. Thus, they argue, even after an `f*sync()` call returns, the data might still be in a volatile cache on the storage device. While this limitation is worth pointing out when talking about data persistence guarantees, it is beyond the scope of any `f*sync()` implementation, as there is nothing the OS, much less an application, can do to ensure that the storage device has actually persisted the data. Thus, we will not further discuss this issue in this thesis.

2.3.3 Non-Uniform or Badly Understood Behavior

It can be difficult for application developers to anticipate when an `f*sync()` call might be required and what side effects it can have, as the exact behavior of `f*sync()` and how it handles errors can vary not only across POSIX-compliant OSs, but also across different file systems or different versions of the same file system. For example, Rebello et al. [59] mention that, while ext4, XFS, and Btrfs all mark pages clean after `f*sync()` failures, ext4 and XFS keep the latest copy in memory, whereas Btrfs reverts to the previous consistent state. Regarding error handling, Rebello et al. [59] describe the fact that ext4 does not report an `f*sync()` failure immediately in some cases, but instead will fail on a subsequent `f*sync()` call as an example of non-uniform behavior. For applications that use `f*sync()` intensively, the developer would have to be aware of all those differences and possibly employ different `f*sync()` strategies for different file systems in order to use it efficiently and guarantee data integrity.

This problem is only further exacerbated by the fact that documentation on `f*sync()` is scattered and often incomplete. For example, the Linux man page for `fsync()` does not mention that the behavior of `f*sync()`

can vary across file systems [23]. If a developer is looking for information on the exact behavior of `f*sync()` on a specific file system, the file system's documentation often contains little to no information on the topic. In some cases, the only way to learn about the exact behavior is by looking at the file system's source code, which can be difficult to understand. Since every file system provides its own implementation of `f*sync()`, the code path for an `f*sync()` varies between file systems and can branch significantly, depending on mount options and the file system's current state. This can also make it difficult to trace `f*sync()` calls on different file systems, as we noticed while implementing our own tooling as part of this thesis (see Section 5.1).

2.3.4 Accidental Usage

As many popular libraries use `f*sync()` internally, application developers might be unaware of how many `f*sync()` calls they indirectly add to their application by using a library function. For example, consider the SQLite library, which is used by many applications to store data persistently. Like any such library, SQLite generally does not know a specific application's requirements regarding data persistence, and so provides defaults that offer good data persistence and consistency guarantees. In the case of SQLite, this includes a WAL and automatic periodic checkpointing, which requires `fsync()` operations [80]. However, an application developer might in turn be unaware of a library's defaults and explicit or implicit assumptions about their application's data persistence requirements. In the worst case, this could mean that a library actively persists all data passed from the application, while the data is not required to be persisted at all, for example, because it is only cached data. This way, the application, or even other applications on the same machine, might suffer a significant performance decrease due to many unnecessary `f*sync()` calls. Returning to the example of SQLite, consider web browsers on Android, which, like many Android applications, make heavy use of Android's builtin SQLite library. As explained, that library in turn uses `fsync()` to persist data in the background. Kim, Lee, and Won [41] have shown that, in practice, this can cause up to 500 `fsync()` calls for a single page load, with a significant ratio of these calls being caused by cached data or metadata whose persistence is not required.

2.3.5 Unsuitable Interface

The design of interfaces like `f*sync()` means that applications are only able to persist the data of an entire file. This can be adequate for applications that only access small files or that only store data of one type in a single file,

relying on directory structures containing multiple files to represent complex hierarchies of data. However, many modern applications do not conform to this paradigm. Especially for desktop applications aimed at consumers, it has become common practice to store all data belonging to a single project in one file, which improves usability and allows for easier sharing of data between applications as well as between users. Examples of such applications include Office suites like Microsoft Office or LibreOffice, but also editing software for images, videos, or audio. In "A File Is Not a File" [32], Harter et al. point out files created using Microsoft Office applications as an example for this issue. These applications use a file format called Compound File Binary Format (CFBF) to store all data of a single document in one file. Internally, each CFBF file is structured very similarly to an actual FAT file system and contains multiple sub-files [47]. This means that, even when an application is only accessing one of the sub-files, it can only call `f*sync()` on the entire CFBF file, leading to a potential increase in `f*sync()` latency. Harter et al. argue that file systems should be aware of such file-internal structures and provide high-level interfaces for applications to effectively work with file-internal hierarchical data structures [32].

2.4 Existing Alternatives and Improvements

In Section 2.4.1, we describe a number of interfaces that could be used in place of `f*sync()` to achieve data persistence that have been proposed or are already available. In Section 2.4.2, we discuss proposed changes to the inner workings of `f*sync()` or file systems that aim to alleviate or remove some of the shortcomings we have discussed in the previous Section. In addition to these alternatives, there are also some measures that system administrators or manufacturers can use to improve the reliability or performance of `f*sync()`, which we describe in Section 2.4.3. All of these alternatives, changes, and measures come with their own advantages and disadvantages. In the following, we will discuss these aspects in detail.

2.4.1 Alternatives to `f*sync()`

The following are some existing alternatives to active persistence using `f*sync()`.

`sync_file_range()` is an alternative syscall to `f*sync()` which the Linux kernel provides, but is not part of the POSIX standard [67]. It enables finer control than `f*sync()` by allowing an application to specify a range of a file that should be synced instead of the whole file. While, for some applications, this

could be more efficient than syncing the entire file, `sync_file_range()` comes with several disadvantages. As it does not flush metadata, it should only be used by applications "strictly performing overwrites of already instantiated disk blocks" [67]. Contrary to `f*sync()`, `sync_file_range()` also does not flush any volatile hardware write caches and is thus not suited to ensure data integrity [67]. Furthermore, on copy-on-write file systems like Btrfs or ZFS, overwriting an already allocated block is impossible, meaning that `sync_file_range()` cannot be used. For all of these reasons, the man page for `sync_file_range()` cautions that "This system call is extremely dangerous and should not be used in portable programs" [67].

The Linux kernel's `io_uring` interface for asynchronous I/O provides an asynchronous equivalent to both `fsync()`and `sync_file_range()` using the operations `IORING_OP_FSYNC` and `IORING_OP_SYNC_FILE_RANGE`, respectively [37]. `io_uring` uses two ring buffers: Applications queue I/O requests in one ring buffer, and upon completion they can retrieve the result from the other ring buffer. However, since requests can complete in a different order than they were submitted to the submission buffer, this means that the application must wait for a write to complete before queuing the sync request. Otherwise the sync might complete before the write is completed [37]. Furthermore, this interface is only suitable for applications making use of asynchronous I/O.

Using direct I/O by opening a file with the `O_DIRECT` option, applications on Linux can instruct the kernel to bypass the page cache and perform all I/O operations directly on the storage device [51]. This means that there is no need to call `f*sync()` in order to flush the file's page cache, since there is no page cache for the file. However, the `open()` syscall's man page points out that, if data persistence is very important, calling `f*sync()` is still required to make sure that writes have been transferred to the underlying storage device [51]. Alternatively, applications can additionally use the `O_SYNC` or `O_DSYNC` flag when opening the file, which has the same effect as calling `fsync()` or `fdatasync()` after each write operation, respectively [51]. The additional syncing is required because "the `O_DIRECT` flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the `O_SYNC` flag that data and necessary metadata are transferred" [51]. This means that, when a `write()` call on a file opened with `O_DIRECT`, but withouth `O_SYNC` returns successfully, writeback of that file's data and metadata is not guaranteed to be complete [51]. Using `O_DIRECT` without `O_SYNC` also does not automatically flush the storage device's volatile cache as `f*sync()` does [73]. Due to these limitations, MySQL, for example, adds `fsync()` calls even after writes to files opened with `O_DIRECT` [73]. Direct I/O is mostly beneficial

for applications implementing their own caching, or applications that do not need caching for some data as it is mostly written once and rarely read (e.g., Database journals).

2.4.2 Proposed Changes to `f*sync()`

The following are the most notable proposed changes to the behavior or interface of `f*sync()`. They can be divided into changes to improve performance and changes to improve the reliability of error reporting.

Changes to Improve Performance

Multiple publications have addressed the issue of `f*sync()` on ext4 and F2FS flushing more data than the file it is called on [6, 64, 53].

- In "Eager Synching" [6], Chang et al. demonstrate an approach to selectively sync only the dirty data required for future access of a file. Their approach writes this dirty data to a special log space separate from the file system's existing journal, from which all synced files can be restored even in case of a power failure or system crash [6]. The authors report that their approach can reduce `f*sync()` latency by up to 72% on ext4 and 92% on F2FS [6]. However, as the authors have only tested their solution on Android devices, more testing would be required to confirm their findings for other types of systems as well.
- Park and Shin have proposed a very similar approach in "iJournaling" [53]. Their approach involves using a separate journal to quickly record file-level transactions for `f*sync()` calls while still recording normal journal transactions to ext4's existing journal during periodic journaling [53].
- As described in Section 2.3.1, Shirwadkar, Kadekodi, and Tso have implemented changes based on said work of Park and Shin starting with Linux kernel version 5.10, as they described in "FastCommit" [64]. However, the authors specifically excluded the change causing `f*sync()` not to flush dirty data for all files due to concerns that this change would break compatibility with applications relying on the old behavior, as Shirwadkar explained on the linux-ext4 mailing list [63].

In the past, Linux kernel developers have also proposed a version of `f*sync()` for syncing multiple file descriptors at once [15]. This could not only potentially increase performance, as the file system could perform batching on the operations, but would also have the benefit of simplifying active

persistence for application developers [15]. However, such a syscall would be more complex to implement than a conventional `f*sync()`, in case files are located on different file systems, or some files need metadata syncing whereas some do not. No such change has yet been introduced into the Linux kernel.

Changes to Improve Error Reporting

There are some proposed changes addressing the issue of `f*sync()` errors not always being reported to applications reliably.

- Kernel developers have proposed one change to improve the reliability of `f*sync()` error reporting on the linux-ext4 mailing list [14]. They suggest marking a file as being in an error state in its inode to allow every process opening the file to reliably check for errors during an `f*sync()` call. That way, they argue, one process could check for the error without consuming it, so that other processes can see it as well. However, other kernel developers replied that this change would take Linux further away from the POSIX standard, so it is unlikely such a change will happen.
- Ts'o describes a different way of improving error reporting that has been implemented at Google [75]. It is based on a separate process monitoring for I/O errors. In case of an error, that process can instruct a clustered file system to temporarily stop using the corresponding hard drive, while the error is being fixed. However, in our view it is unclear if adding a similar mechanism to the upstream kernel would benefit application developers. First, we expect it would increase application complexity, since the application developer would have to add logic for polling and reacting to messages from the monitoring interface. Secondly, it would add the overhead of context switching for polling messages, which we expect to be undesirable for some applications.

2.4.3 Improving `f*sync()` Performance as an Administrator

System administrators can take steps in order to increase the performance of `f*sync()`-intensive workloads. The following are some of the possible steps.

Choosing the Right File System. As `f*sync()` implementations on different file systems can vary in performance, one way to increase performance for `f*sync()`-intensive workloads is to choose a file system with a relatively efficient `f*sync()` implementation. In this case, performance can be measured in terms of the latency of a single `f*sync()` call or the I/O throughput in `f*sync()`-intensive workloads. The performance of an `f*sync()` implementation can, among other factors, depend on the amount of data and metadata that `f*sync()` writes out as well as the number of flush requests it issues to the storage device [7]. For example, Lee et al. [44] demonstrate that F2FS shows higher performance than other journaling and log-structured file systems on some `f*sync()`-intensive workloads.

Another option is ZFS, on which `f*sync()` only flushes data out to the ZFS Intent Log (ZIL), a temporary cache for writes that have not yet been applied to the file system [81]. By placing the ZIL on a fast separate device (e.g. small but fast flash storage), a so-called Secondary Log Device (SLOG), `f*sync()` latency can be decreased [73, 69].

The limitations of choosing a file system with an efficient `f*sync()` implementation are that the choice of a file system can depend on many other factors. In some scenarios, file systems with better `f*sync()` performance might not be available. For example, F2FS is only suitable for flash storage and mostly found on Android devices, while ZFS might not be available because it is not included in the mainline Linux kernel.

Choosing the Right Storage Hardware. Since `f*sync()` also flushes the storage device's volatile cache, the performance of `f*sync()` can indirectly be improved by improving the performance of the storage device's cache flush operation. There are different kinds of storage hardware that can guarantee flushing their cache in a timely manner, even in the event of a power failure, and can thus safely ignore explicit requests to flush their cache.

One such kind of hardware are RAID controllers with a protected write cache, i.e., a cache protected by a battery that can be flushed even in the event of a power loss. In case of multiple sequential writes interleaved with `f*sync()` calls (e.g., a database log file), a RAID controller can concatenate the writes in its cache, ignoring requests to flush its cache that `f*sync()`issues,

and eventually make one large write from the cache to the actual disks [73]. This comes at the expense that such hardware is expensive and usually only available in enterprise scenarios or high-end consumer workstations.

Similar capabilities have been specified for upcoming hybrid SSDs based on the Compute Express Link (CXL) standard (i.e., SSDs that are both byte- and block-addressable). The CXL specification includes a mechanism called Global Persistent Flush (GPF) that enables reliable flushing of both CPU caches as well as the SSD's cache in case of a power failure. While, at the time of writing, such devices are not yet available, they may in the future provide a viable way to improve performance for `f*sync()`-intensive workloads [11].

Already widely available are SSDs with Power Loss Protection (PLP). They contain capacitors that, in the event of a power failure, can provide enough power to reliably flush the SSD's cache [1]. Thus, the SSD's `FLUSH` command can return immediately without waiting for the writes to complete [7]. But similarly to battery-backed write caches in RAID controllers, such hardware is expensive and usually only available in enterprise solutions or high-end consumer workstations.

Chapter 3

Related Work

In the following, we give an overview of existing work related to our thesis, divided into two categories: Publications on implementation details of `f*sync()` and publications on I/O characteristics of modern applications. Where applicable, we compare the approach of the respective publication with our own approach.

3.1 `f*sync()` Implementation

In "Eager Synching" [6], Chang et al. present a technique that significantly increases the performance of `f*sync()` on ext4 and F2FS. They achieve this by selectively syncing only the dirty data required for accessing the file in the future to a sequential log space, instead of flushing all open files on each `f*sync()` invocation. Using their approach, all files can be restored from the information in the log space, even in case of a power failure or system crash.

In "iJournaling" [53], Park and Shin propose a change to the journaling mechanism of ext4 to improve the performance of `f*sync()`, very similar to the previous work of Chang et al. [6]. Their approach involves using a separate journal for quickly recording the file-level transactions of `f*sync()` calls while still recording normal block-level transactions during periodic journaling.

In "FastCommit" [64], Shirwadkar, Kadekodi, and Tso describe how they have implemented changes based on the work of Park and Shin and successfully merged them into the mainline Linux kernel in version 5.10. While they do not implement all changes proposed by Park and Shin due to concerns about breaking compatibility for existing applications relying on the current behavior of `f*sync()`, they still report that their changes can improve the performance of `f*sync()` by a factor of up to 2.8.

In "Fundamental OS Design Considerations for CXL-based Hybrid SSDs" [31], Habicht et al. demonstrate how existing abstractions in operating systems are not well suited for upcoming CXL-based hybrid SSDs. They propose changes to OS interfaces and resource management of hybrid SSDs that can significantly increase the performance of applications with strong persistence requirements on such hardware. Among other things, they show how `f*sync()` performance can be increased by taking advantage of the unique characteristics of hybrid SSDs, i.e., the ability to be both byte- and block-addressable as well as the presence of a persistent write cache.

Finally, in "Can Applications Recover from fsync Failures?" [59], Rebello et al. analyze the behavior of different `f*sync()` implementations in the event of a failure as well as how applications react. They show how none of the existing failure handling strategies are sufficient, and `f*sync()` failures can lead to data loss and corruption. They highlight as one cause of this problem that the post-failure behavior of file systems is non-uniform and often badly documented.

3.2 I/O Characteristics of Applications

In "A File Is Not a File" [32], Harter et al. examine the I/O behavior of typical productivity and multimedia applications on Mac OS X. Using their findings, they identify multiple I/O patterns common to the examined applications like sequential access being very rare, most files accessed being auxiliary files instead of user files, and most writes being explicitly forced to disk.

The approach of Harter et al. relies entirely on tracing syscalls using DTrace to collect data [32]. While they do not explicitly mention in their paper which syscalls exactly they traced, from looking at the traces they have published [33], we assume that they trace all syscalls made by the application. Since our approach focuses on `f*sync()`, we potentially miss some of the information that would be available using the approach of Harter et al. However, we do not only trace syscalls, but also actual writeback on the file system level caused by `f*sync()`calls. Thus, we expect that we can provide some new insights by comparing our results with those of Harter et al.

We will investigate whether we can make observations similar to those made by Harter et al. regarding synchronous writeback. For example, their claim that most writes are explicitly forced to disk could, if true, have important implications for the design of interfaces for active persistence.

Interestingly, Harter et al. note that, although they initially intended to also trace I/O resulting from memory-mapped files, they find that the amount of memory-mapped I/O is negligible and thus exclude it from their analysis [32]. This corresponds with our observation that the `msync()` syscall is hardly used compared to `f*sync()`.

In "Comparative study on I/O characteristics of mobile web browsers" [41], Kim, Lee, and Won analyze the I/O behavior of different web browsers on Android. Among other things, they examine the writeback behavior and `f*sync()` usage of those apps. To do this, they use a custom tracing tool for Android called *Androtrace*, which they developed themselves [42]. They show that the different browsers display different degrees of efficiency in their I/O behavior. They suggest potential ways to improve the I/O efficiency of such applications, like considering the use of `fdatasync()` over `fsync()`, choosing an efficient database journal mode, and considering carefully which web data to cache.

Androtrace's approach is similar to the one we use for our tooling (see Chapter 4 and Chapter 5), in that both rely on a modified kernel, they are limited to ext4, and that part of the collected data is the same in both cases [41, 42]. For example, both Androtracer and the tooling developed by us record files that are being synced as well as the processes issuing the corresponding sync requests [42]. Androtrace has the advantage of not only recording forced writeback, but also reads and writes in general, as well as more specialized information like flags of I/O operations [41]. However, while Androtrace records what data is being requested to sync, it is unaware of how much data is actually being written back. We expect that, by comparing our data about actual writeback with the results of Kim, Lee, and Won, we can provide some new insights and examine to what degree their findings hold true for modern server and desktop applications on Linux.

Chapter 4

Design

In Section 2.3, we discussed many known problems with the `f*sync()` interface, as well as their causes and situations in which they can occur. Now, our goal with this thesis is to gather data on the synchronous writeback behavior of different applications and to gain a better understanding of this behavior using statistical analyses and visualizations.

4.1 General Approach

We aim to demonstrate a method to gain insights into applications' writeback behavior that can be used to answer the following kinds of research questions:

- What are precise conditions under which a certain problematic behavior of `f*sync()` occurs?
- If problematic behavior occurs, what is its impact, and is that impact relevant to a given use case?
- Can the existing alternatives and improvements to `f*sync()` we discussed help avoid or mitigate the problematic behavior?
- Can we derive any other proposals from our data on how to improve the behavior of `f*sync()` in a given situation? Is there a way the application could be modified to avoid the problematic behavior?
- Can we propose a different interface for persisting data that would be more appropriate for the given use case?

Answering all of these questions for all of the issues we discussed in Section 2.3 is beyond the scope of this thesis. However, we expect that by improving the understanding of applications' writeback behavior, we can lay the groundwork for future improvements of both applications' usage of OS

interfaces for active persistence and the design of these interfaces themselves. Additionally, it is our goal that the tooling we develop can not only aid future research, but can also become a useful debugging tool for kernel and application developers.

To answer our research questions, we decided to gather the following data about the synchronous writeback behavior of different applications:

1. Which files does the application actively sync to disk?
2. Upon a sync request, what data does the OS actually write to disk?
3. How long does it take for the OS to write the data to disk?
4. What code path in the application is responsible for the sync request?

While there are already tools that can trace, for example, on which files an application calls `f*sync()` and how long it takes for that syscall to complete, there is no existing solution to determine which data is actually written back to disk and what time specifically the writeback takes. We expect that such tooling can grant us valuable insights into the strategy an application uses to persist data, and consequently, whether that strategy is adequate or how it could be improved.

On the one hand, consider an application working with data that does not require persistence guarantees. If most sync request of the application do not cause any writeback, this could indicate that the application makes unnecessary sync requests, which could be a performance bottleneck. This could be the case, for example, if the application uses a library function that calls `f*sync()` internally, but the application developer is not aware of that. On the other hand, if the application has very strong persistence requirements, and it is not feasible to determine in advance which files have outstanding writes, persistence guarantees achieved by aggressive syncing will outweigh the performance cost.

If we know the exact location in an application's code from which a sync request originated, we can refer to the application's documentation or source code to take into account any explicitly stated or implicit assumptions the application makes about its need for data persistence. Thus, one of our goals is that, for every `f*sync()` call we trace, our tooling can provide a complete stack trace showing the code path in the application from which the sync request originated. Our analysis will put a special focus on determining and visualizing what percentage of sync requests originate from a given code path as well as what amount of writeback and how much writeback latency that code path causes.

Since there is no existing tool that can directly provide us with the information we intend to analyze, we implemented our own tooling to collect and analyze data, building on top of well established tracing mechanisms in the Linux kernel. The goal of our tooling is to transform the raw data we collect into a format that we can easily run queries on, which will answer high-level questions about the writeback behavior of applications. We put a special focus on using visualizations to gain insights into the data we collect. By using flame graphs [28] to visualize synchronous writeback behavior, we adapt a visualization technique that has been successfully used in the past to analyze application performance, yet has not been used to specifically analyze synchronous writeback behavior.

4.2 Tracing Mechanisms

bpftrace [5] is a recent tracing language for Linux. Using the LLVM compiler suite, bpftrace scripts are compiled to extended Berkeley Packet Filter (eBPF) bytecode, which can be loaded into the kernel. eBPF programs are run in a sandboxed runtime directly within the kernel, and thus can be used to build powerful tracing tools, among other things. An important concept in bpftrace are probes, which represent different events that can be traced [4]. There are probes for tracing mechanisms built into the kernel, such as kprobes or static tracepoints, among various other probe types. bpftrace scripts consist of one or more action blocks, each of which specifies an action to be executed when a given probe is triggered. An action could, for example, be printing some data that the probe provides. Listing 4.1 shows an example of a minimal bpftrace script that prints the process ID of every process that calls `fsync()`.

```
1 tracepoint:syscalls:sys_enter_fsync
2 {
3   printf("Process %d called fsync!", pid);
4 }
```

Listing 4.1: Example of a minimal bpftrace script that prints the process ID of every process that calls `fsync()`.

We decided to use a bpftrace script for tracing, as it allows us to easily collect data from multiple existing tracing mechanisms in the kernel. The script can be easily extended to listen to additional probes, or to selectively filter events to only collect data relevant to our research questions. bpftrace also allows us great control over the output format of our tracing script.

In particular, we utilized existing static tracepoints in the kernel to trace syscalls, as well as additional custom tracepoints in file system code [30] to trace synchronous writeback. We discuss the specifics of the tracepoints we use in Chapter 5. Another feature of bpftrace that proved useful for our purpose is the ability to easily obtain a user-level stack trace for every syscall event we record. This way, it is possible to determine the code paths from which sync requests originate, providing us with an entry point to learn more about what data is being synced and why from the corresponding locations in the application's source code.

4.3 Design Principles

The design of our tooling is centered around the goal of making it as modular and easily extensible as possible. Not only did this allow us to quickly implement and evaluate changes in our tracing approach, but it will also make it easier to extend our tooling to support additional use cases in the future. For example, we wanted to be able to easily integrate additional data sources with as little change to the existing code as possible. To achieve this goal, we decided on two central design principles:

Adhere to the Unix Philosophy [46]. Instead of building one monolithic program with many features, we decided to build a set of small tools that each perform a single task well. This modular approach makes code easier to understand and maintain, and allows us to add new features easily. We also decided that our tools should be entirely CLI-based. Their input and output should be text-based so that they can interoperate with other tools and can be used in scripts.

Separate between a high-level- and a low-level-log. Data collected by tracing tools can be very fine-granular and high-volume. However, data in this form is generally hard to interpret, since the answer to a high-level question can often only be found by inspecting multiple data points, and the high volume of data can make it difficult to identify relevant data points. It would be complicated to run queries on such data that answer the kind of high-level questions we are interested in. Furthermore, if there is data from multiple sources, it needs to be aggregated in some way to be useful. To address this, we decided to use two separate logs to represent our data. The purpose of one of the tools we built is to translate a low-level log into a high-level log.

In the low-level log, each event corresponds to a single event as it was collected from a tracing mechanism. The format of this log is designed to be flexible enough to represent data from different sources. In accordance with the Unix philosophy, the low-level log is human-readable and can be inspected manually. But its purpose is to store all collected data losslessly and in a structured way, without concern about whether it is easy to query or visualize. Space-efficiency is also not an important concern for the low-level log, as it is only a temporary format and can be discarded after the high-level log has been created.

The high-level log, on the other hand, contains events that directly correspond to the high-level questions we want to answer. The purpose of the high-level log is to present data that is relevant to our research questions in a concise way, that is easy to query and visualize. The reason we do not perform the translation from the low-level to the high-level log online as part of the tracing process is that a single high-level log entry can be derived from a large number of low-level log entries, and so converting logs with very large sync requests can require a significant amount of memory and processing time. Thus, performing the translation online would pose the risk of influencing the performance of the application we are tracing.

For example, when an application calls `f*sync()`, the first event we record is the entry of the syscall. This event is followed by separate events indicating that a certain file range was added to the writeback request, that the writeback request was submitted, and that it has completed, respectively. We aggregate all of these low-level events into a single event in the high-level log that directly provides high-level information, like the amount of data that was written back, or the time it took for the syscall to complete, among other things.

4.4 Architecture Overview

Fig. 4.1 shows an overview of the architecture of our tooling. The dataflow resembles a pipeline architecture: A bpftrace script produces the original trace data, which the data collection module converts into a low-level log. In essence, the data collection module transfers the tracing data from a plain text format to a structured format without changing the data itself. The data processing module then converts the low-level log into a high-level log, on which we can run our analysis. Alternatively, the data processing module can export other data formats that can be used to create visualizations or run analyses using third-party tools.

The main component of the data collection module is the event manager, which performs the actual conversion of the tracing data to the low-level log. Basically, the event manager extracts the relevant data from each line of input and uses it to populate a low-level log entry. It then outputs a serialization of the new low-level log entry. On each invocation of the data collection module, the CLI initializes the event manager with the type of input data and the serialization format to use.

The main component of the data processing module is the log converter, which translates the low-level log to a high-level log. To do this, it first iterates over the low-level log to identify all events belonging to a single sync request. It then performs calculations on the data of these events and uses the results to populate a high-level log entry. If the desired output format is a high-level log, it simply outputs a serialization of the new high-level log entry and moves on to the next sync request. However, if the desired output format is a third-party data format, the log converter will pass all high-level log entries on to a separate function that converts the data to the desired format. Again, the purpose of the CLI is to initialize the log converter with the type of input data and the output format to use.

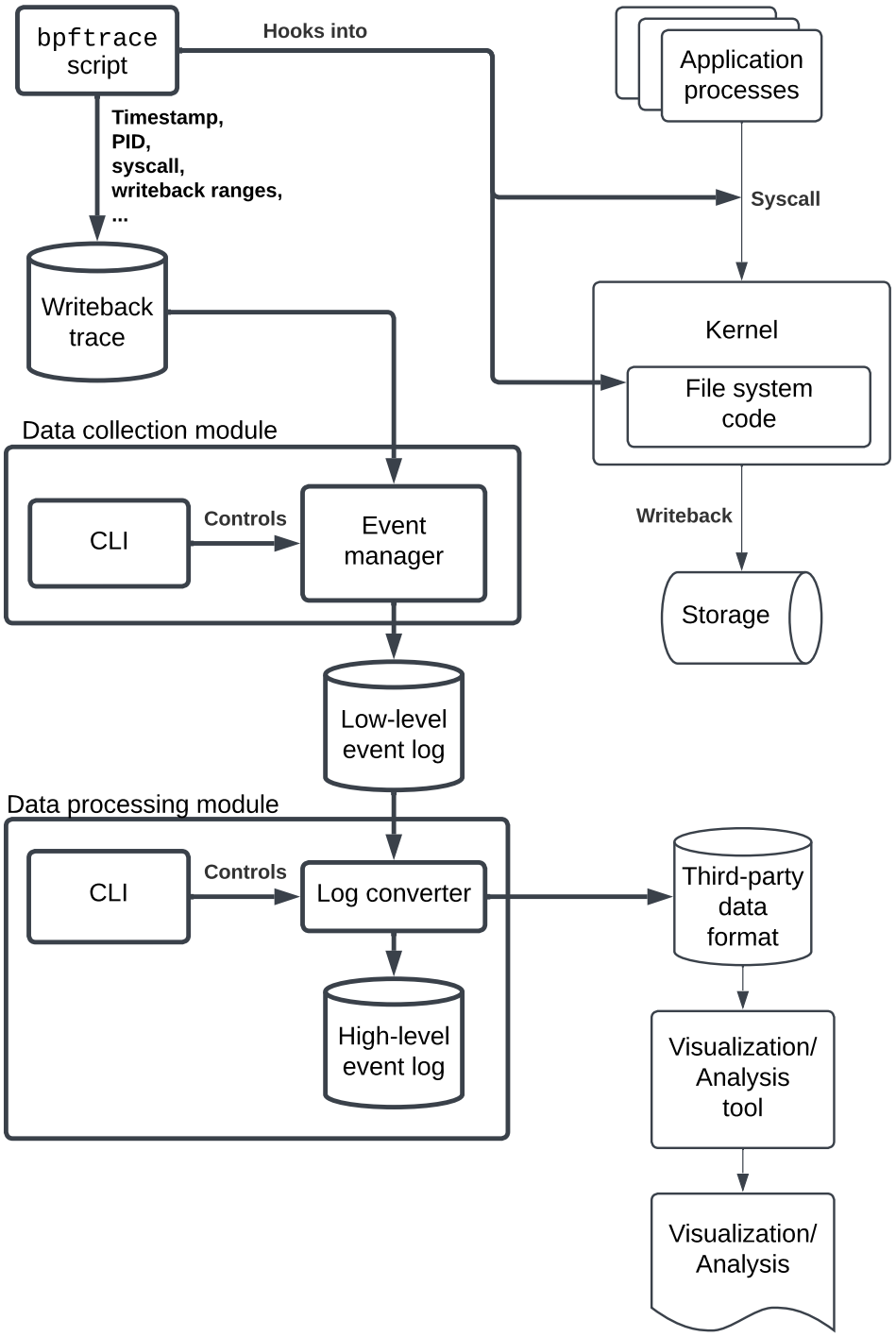


Figure 4.1: Overview of the architecture of our tooling for tracing and analyzing synchronous writeback behavior. Thick outlines indicate the components we implemented. The data collection module is responsible for converting data from different sources to a low-level log. The data processing module converts the low-level log to a high-level log that we can run our analysis on. The data processing module can also support third-party data formats we can use to create visualizations or run analyses using third-party tools.

Chapter 5

Implementation

In this chapter, we describe in detail the mechanisms we use to trace the synchronous writeback behavior of applications, how we implemented our tooling, and which challenges we encountered during the implementation process. We also explain how we create visualizations of the data we collect.

5.1 Tracepoints in the Kernel

One issue we encountered early on is that, as mentioned in Section 2.3.3, the code path in the kernel for an `f*sync()` call is different for every file system. Since the determination of which data is actually written back to disk happens in file system-specific code, our tooling has to hook into file system functions to collect the data we need. Since we did not have the time to implement our tooling for different file systems, we decided to focus on one file system for the purpose of this thesis. We chose ext4, as it is the most widely used file system on Linux and has been the default on many distributions for a long time. However, focusing on a single file system does provide us with a practical advantage: By using a different file system for the root partition of our test system and placing only files belonging to applications we want to trace on a separate ext4 partition, our bpftrace script does not record writeback caused by other applications. While it does still record `f*sync()` calls made by other applications, we can simply ignore any syscall that is not followed by trace events describing the corresponding writeback. Thus, we do not have to manually filter out data from other applications.

We use two sets of tracepoints in the Linux kernel to gather data about synchronous writeback behavior. First, we use existing static tracepoints in the kernel to trace the entry of syscalls. These are available in bpftrace as probes called `tracepoint:syscalls:sys_enter_fsync` and

`tracepoint:syscalls:sys_enter_fdatasync`. Secondly, we used a kernel patch [30] to add custom tracepoints for some ext4-specific functions for which the kernel does not provide tracepoints by default. The most important functions we hook into are:

- `ext4_sync_file()` in `fs/ext4/fsync.c`, which handles writeback for `fsync()`, `fdatasync()`, and `msync()` calls [76].
- `ext4_io_submit()` in `fs/ext4/page-io.c`, which is called to submit an I/O request, that in turn is associated with a BIO [74].
- `io_submit_add_bh()` in `fs/ext4/page-io.c`, which is responsible for adding a folio to a BIO [74].
- `ext4_end_bio()` in `fs/ext4/page-io.c`, which is called when writeback of a BIO has completed [74].

BIOs are the basic unit of I/O for the block layer and lower layers in the Linux kernel [70]. A folio represents a contiguous set of bytes in memory, for example in the page cache [71]. For each sync request, ext4 determines whether the file in question has any dirty pages that need to be written back to disk. If such dirty pages exist, it will allocate an `ext4_io_submit` struct, which is associated with a BIO. Typically, each continuous range of dirty sectors on a block device is represented by a single BIO. For each dirty page, ext4 will then add a folio to the corresponding BIO. That BIO will be submitted via the `ext4_io_submit` struct. If there are multiple separate ranges of dirty pages, the same `ext4_io_submit` struct will be used to submit a separate BIO for each of those ranges. The `ext4_io_submit` struct also contains a pointer to a `writeback_control` struct, which in turn contains information about the writeback request [72]. The kernel patch we use [30] adds a unique ID to each instance of the `writeback_control` struct, which we later use to identify which trace events belong to the same sync request. Based on this process, for each sync request, one of our traces will contain the following events:

- An event for the `f*sync()` call triggering the sync, including a user-level stack trace of the calling process.
- A `sync_start` event, indicating that the function `ext4_sync_file` has been entered, including the path of the file being synced.
- A `bio_add_folio` event for every dirty page that is added to the current BIO.
- A `bio_submit` event, indicating that the BIO has been submitted for I/O.

- A `bio_complete` event, indicating that writeback of the BIO has completed.
- Finally, a `sync_end` event, indicating that the `ext4_sync_file` function has returned. This event also means that the writeback of all BIOs for this sync request has completed.

Each event also contains a timestamp and the process ID of the calling process. If a file does not have any dirty pages at the time of a sync, the `bio_add_folio`, `bio_submit`, and `bio_complete` tracepoints will not be reached. Thus, in that case, the trace will not contain these events for the corresponding sync request. By aggregating all these events, we can later determine what files have been synced, how much data has been written back to disk, how long the writeback process took, and from which code path of the traced application the sync request originated. Listing 5.1 shows an example of a trace created by our bpftrace script.

```
1  TIME   PID   COMM         EVENT           DATA
2  252    53     bio_aof      fdatasync
3      768c __GI_fdatasync+60 (/usr/lib64/libc.so.6)
4      44b1 bioProcessBackgroundJobs+545 (/usr/bin/valkey-server)
5      e168 start_thread+952 (/usr/lib64/libc.so.6)
6      214c __GI___clone3+44 (/usr/lib64/libc.so.6)
7  294    53     bio_aof      start           id 8 start 0 end 4294967295 datasync
       ↪ 1 path /var/lib/valkey/appendonlydir/appendonly.aof.7.incr.aof
8  313    53     bio_aof      bio_add_folio   id 8 bio 0x762 start 1424 len 4096
9  313    53     bio_aof      bio_add_folio   id 8 bio 0x762 start 5520 len 4096
10 321    53     bio_aof      bio_submit      id 8 bio 0x762
11 661    0       swapper/26  bio_complete    bio 0x762
12 856    53     bio_aof      exit            id 8 ret 0
```

Listing 5.1: Example of a trace created by our bpftrace script. The trace contains all events for a single sync request. Values that we use to identify events belonging to the same sync request are highlighted. Timestamps, process IDs, and memory addresses have been shortened to improve readability. Lines 3–6 show a user-level stack trace of the process calling `fdatasync()`.

5.2 Data Collection and Processing

For the data collection and the data processing modules, we chose Python as a programming language, since it allows us to rapidly prototype new functionality.

The data collection module was straight-forward to implement, as every event in the low-level log corresponds to a single line in the output of our tracing script. The data collection module's task is to transfer the data

in each line from a plain text format to the structured format of the low-level log, without changing the data itself. The most important part of this module is a set of regular expressions, which are defined in such a way that exactly one regular expression matches each line of the output of our tracing script. The event manager reads each line from `stdin`, determines which regular expression matches the line, creates a low-level log event from the data in that line, and writes a representation of that event to `stdout`. The default representation uses JSON, as it is easy to serialize and deserialize in Python. However, it is possible to add other serilization formats in the future, if required. As was our design goal, it is also easy to add other data sources to the data collection module, since it essentially requires a set of regular expression for parsing data from that source. For example, we also implemented the ability to parse the output of the `strace` tool, which we initially intended to use for tracing syscalls. Since we could also trace syscalls using our bpftrace script, and bpftrace allows for more flexibility in terms the events we trace and the output format, we did not use this functionality in the end. Still, it serves as a proof-of-concept for adding additional data sources to our tooling.

The program logic of the data processing module is slightly more complex than that of the data collection module. The log converter scans the low-level log for `sync_start` events. Once it finds one, it scans the log backwards from the `sync_start` event's position to find the corresponding syscall event, and forwards to find all other associated events. The reason for starting scanning from the `sync_start` event instead of the syscall event is that the trace can contain syscall events for sync requests on all file systems, but the other events are specific to ext4. If some process unrelated to the application we are tracing calls `f*sync()` on a file on a different file system, the trace will contain the corresponding syscall event, but no other events for that sync requests. Since we want to ignore such lone syscall events, we start scanning from the `sync_start` event, which is always present for sync requests on ext4. Events belonging to the same sync request are identified by the associated process ID, the address of the associated BIO, and a special unique ID that is added to the `writeback_control` struct by the kernel patch we use [30]. When the log converter has found all events belonging to a sync request, it aggregates data from them into a single high-level log event. This includes, for example, concatenating all folios to determine the total writeback range, or calculating the writeback latency using timestamps of the trace events.

Finally, once the log-converter has processed the entire low-level log, it writes the resulting high-level log to `stdout`. The user can choose the output format via a command-line argument. Currently supported are CSV for

further analysis, a human-readable format for debugging purposes, and a format resembling a perf trace, which we use for visualization purposes, as described in Section 5.3.

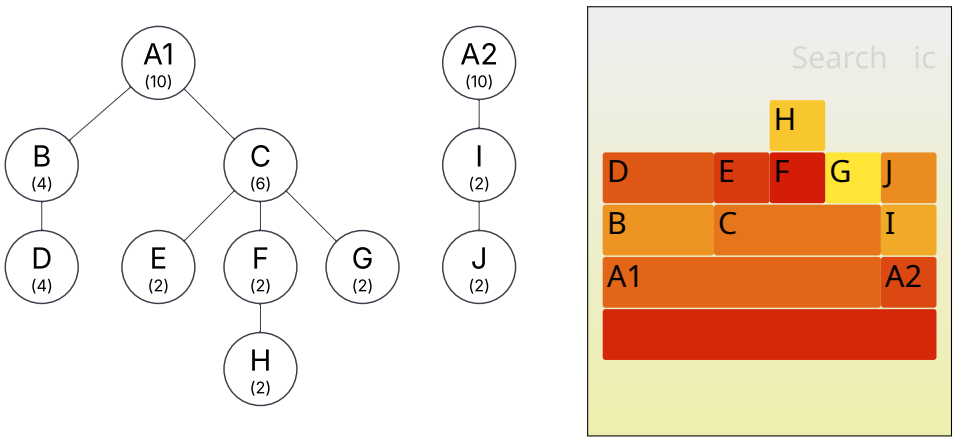
5.3 Visualization

Our design includes the option to convert the high-level log into third-party formats for visualization or analysis with external tools (see Fig. 4.1). The visualization format we determined to be most useful for our purposes are flame graphs. Flame graphs are a visualization technique for hierarchical data introduced by Brendan Gregg [28]. Essentially, a flame graph is a representation of a forest consisting of one or more trees, where each node is associated with a weight. In the flame graph, each node is represented by a horizontal bar. The lowest level of the flame graph consists of a single bar taking up the entire width of the graph, on which the roots of the trees are placed. Each node's children sit on top of that node's bar, and the width of each bar is proportional to the weight of the node in relation to the sum of the weights of itself and its siblings. Fig. 5.1 shows an example of a forest consisting of two trees and the corresponding flame graph.

Flame graphs are commonly used to visualize performance profiling data. For example, CPU flame graphs show the distribution of CPU time spent in different code paths of a program [27], while off-CPU flame graphs conversely show the distribution of time the program spent on a code path while not currently running on the CPU [29]. Gregg has published a tool called `FlameGraph` that can create flame graphs from the output of different tracing or profiling tools [26]. One of the supported input formats of `FlameGraph` is the output of the `perf script` command. We create flame graphs for our data by converting the high-level log into the format of the `perf script` output and then passing it to the `FlameGraph` tool.

Perf is a performance analysis tool for Linux [56]. Using the `perf record` command, the user can record a performance profile of a running program, which is written to a binary `perf.data` file. Using the `perf script` command, the user can then convert this binary file into a human-readable format [55]. The output of `perf script` consists of a series of samples, each of which contains a timestamp, a process ID, a stack trace of the process at the time of the sample, and the number of events counted during the execution of the sampled event. When viewing a flame graph as a representation of forest, one can think of the stack trace of each sample as a path from the root of a tree to a leaf, with the root representing the process of the sample and each subsequent child representing a function call

in the process's stack trace. Thus we can easily create a perf trace from our high-level log by constructing a sample for each log entry, appending the entry's stack trace and setting the number of events to the value of the metric we want to visualize. To visualize the pure count of sync requests, we set the number of events to 1 for each log entry. To visualize writeback or syscall latency, we set the number of events to a log entry's writeback or syscall latency, respectively. Additionally, we can use each log entry's file path in place of the stack trace to visualize the distribution of sync requests across different files. In this case, the resulting flame graph will represent a file system tree of all files that an application has synced. Listing 5.2 shows an example of a perf trace that can be used to create the flame graph in Fig. 5.1(b), as created by the data processing module.



(a) Example forest (b) The corresponding flame graph

Figure 5.1: Example of a forest consisting of two trees (a) and the corresponding flame graph (b). Each node is labeled with its weight in parentheses. The relative width of each bar in the flame graph is proportional to the weight of the corresponding node. Note the unlabeled bar at the bottom of the flame graph, which always takes up the entire width of the flame graph.

```
A1          0   1742913068.657344: 4 cycles:
                    0 D (unknown)
                    0 B (unknown)

A1          0   1742913068.657344: 2 cycles:
                    0 E (unknown)
                    0 C (unknown)

A1          0   1742913068.657344: 2 cycles:
                    0 H (unknown)
                    0 F (unknown)
                    0 C (unknown)

A1          0   1742913068.657344: 2 cycles:
                    0 G (unknown)
                    0 C (unknown)

A2          0   1742913068.657344: 2 cycles:
                    0 J (unknown)
                    0 I (unknown)
```

Listing 5.2: Example of a perf trace that can be used to create the flame graph in Fig. 5.1(b), as created by the data processing module. Each sample's header also contains a process ID (set to 0 here) and a timestamp, both of which can be set to any value, as they do not show up in the resulting flame graph. Each stack trace line also contains an address (set to 0 here) and a location (set to "unknown" here), both of which, again, can be set to any value.

Chapter 6

Evaluation

In this Chapter, we explain how we used our tooling to examine the synchronous writeback behavior of different consumer applications and present the insights we gained from this examination.

In Section 6.1, we describe the setup we used for our experiments with different applications. Then, in Section 6.2, we summarize the most important results of our examination, including visualizations of the data we collected. Finally, in Section 6.3, we interpret our results and discuss their implications.

6.1 Experimental Setup

For our experiment, we use a physical machine with the following specifications:

- Intel Xeon Gold 6130 CPU with 16 cores @ 2.10 GHz and 32 threads
- 96 GiB of DDR4 memory
- 1 TB Samsung Pro 990 NVMe SSD for the OS and test data

For the OS we use Fedora Linux 41 Server Edition with a patched mainline kernel version 6.12 [30]. As mentioned in Section 5.1, we use a separate ext4 partition to store any data accessed by the applications we are examining. The ext4 partition was created and mounted with default options, including `data=ordered`, which means that the journal is only used for metadata [38]. The root partition for the OS uses Btrfs. This way, since the custom tracepoints we use for synchronous writeback tracing [30], including the `sync_start` event, are specific to ext4, we can easily identify and ignore `f*sync()` calls on other file systems, so our results do not contain data from programs other than the ones we are examining.

We trace the following applications:

- Firefox Nightly 138.0a1 (debug build) [25]
- Chromium 136.0.7054.0 (debug build) [8]
- LibreOffice 25.8.0.0.apha0+ (debug build) [34]

Other than activating debug symbols and disabling optimizations, we use the default build configuration provided by each project. We expect that disabling compiler optimizations does not significantly affect our results, since all of the tracepoints we use are in kernel code, and no optimization should influence how often or on which files the application calls `f*sync()`. However, disabling optimizations is necessary to ensure that the stack traces we capture are complete and all symbols can be resolved.

6.1.1 Tracing Web Browsers

Web browsers are among the most important applications for many users of desktop computers. While there has been little research on the exact relationship between storage hardware performance and browser performance, preliminary findings by Park et al. [54] as well as Kim, Agrawal, and Ungureanu [40] suggest that improving the I/O performance of the underlying storage can directly improve page load times in web browsers. Thus, we expect that `f*sync()` performance can also have an impact on browser performance. Modern browsers are highly complex applications with a large codebase, and at runtime they utilize many different threads and processes. If multiple threads perform I/O, understanding the I/O behavior of browsers can be challenging. Analyzing the synchronous writeback behavior of browsers can provide valuable insights that help identify potential performance bottlenecks.

We chose to directly compare the synchronous writeback behavior of two popular web browsers: Firefox and Chromium. We build both browsers from source with debug symbols and without any optimizations so we can capture complete stack traces for all syscalls. Kim, Lee, and Won [41] observe interesting I/O behavior of web browsers on Android. They report observing up to 500 `fsync()` calls per page load. Since their results were specific to mobile devices and, at the time of writing, are almost a decade old, we decided to examine whether we can observe similar behavior on current desktop browsers. Analogously to the experiment of Kim, Lee, and Won, we chose to examine the I/O behavior of web browsers during the process of loading web pages. For our experiment, we selected 20 pages from the website `de.wikipedia.org` and 20 pages from the website `www.kit.edu`. We use a Python script interfacing with the Selenium Framework for

browser automation [62] to automatically load these pages in succession in each of the two browsers. Selenium ensures that the pages are loaded in reproducible way in both browsers. We repeated the experiment ten times in each browser to ensure that our results are statistically significant and reproducible. Furthermore, prior to each run, we cleared each browser's cache and set up a blank profile to ensure that all test runs happen under the same conditions.

6.1.2 Tracing LibreOffice Writer

Office suites are another important class of applications for many users of desktop computers. LibreOffice [34] is a popular open-source office suite that comes pre-installed on many Linux distributions. We expected that Office applications like word processors exhibit different writeback behavior than web browsers since they mainly work with user files and have less need for caching.

We chose to perform an experiment similar to the one Harter et al. [32] conduct with the Pages application from Apple's iWork productivity suite [52]. In our experiment, we open a blank document in LibreOffice Writer and consecutively insert 10 images between 1 MiB and 3 MiB in size. Since we expect most of the synchronous writeback to happen when the user saves the document, we conducted two variants of the experiment: First, we save once after each image is inserted, and secondly, we save only once after all images are inserted. As with the web browsers, both experiments were repeated ten times each to ensure that the results are statistically significant and reproducible.

LibreOffice has two special features for saving documents automatically [61]. The first feature automatically creates a backup copy of the last on-disk version of the document prior to saving a new version, while the second feature, called *AutoRecovery*, periodically saves recovery information that can be used to restore unsaved changes to a document in case of an application crash. Both features were enabled during our experiments, with the interval for the AutoRecovery feature being set to 1 minute. We made sure that at least 1 minute passed between two edits, so that we could observe the writeback behavior of the AutoRecovery feature.

6.2 Results

In the following Sections, we summarize the most important results of our examination of the synchronous writeback behavior of two different kinds of consumer applications.

6.2.1 Writeback Behavior of Web Browsers

In this Section, we summarize the most important results of our comparative examination of the synchronous writeback behavior of the two web browsers Firefox and Chromium.

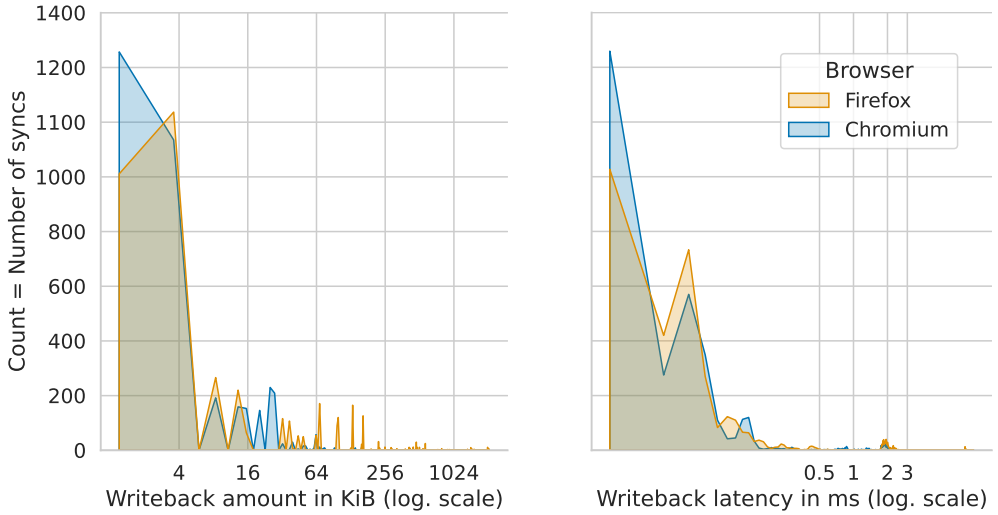
Metric	Firefox	Chromium
Number of unique files synced	65.0	80.6
Average number of syncs per page load	10.40	9.76
Average syscall latency per sync	5.04 ms	4.74 ms
Average writeback latency per sync	0.69 ms	0.43 ms
Average writeback amount per sync	68.23 KiB	10.20 KiB
Average syscall latency per page load	52.41 ms	46.21 ms
Average writeback latency per page load	7.20 ms	4.17 ms
Average writeback amount per page load	710.45 KiB	99.52 KiB
Percentage of syncs without writeback	24%	37%

Table 6.1: Comparison of writeback metrics for Firefox and Chromium. Results are averaged over 3 test runs of 40 page loads each.

Both Firefox and Chromium rely on SQLite for data storage. On Unix, SQLite by default disables the use of `fdatasync()` and uses `fsync()` instead [65]. A comment in the SQLite source code explains, "We do not trust systems to provide a working fdatasync(). Some do. Others do no (*sic*). To be safe, we will stick with the (slightly slower) fsync()" [65]. However, the use of `fdatasync()` can still be enabled via a compiler flag. Chromium's default build configuration sets this flag, while the default build configuration of Firefox does not [48, 9]. Thus, in our experiment we observed that Chromium exclusively uses `fdatasync()`, while Firefox only uses `fsync()`.

Table 6.1 shows a comparison of the two web browsers Firefox and Chromium in terms of the most important metrics we measured. The metrics show that, while both browsers make a similar number of sync requests (10.4 for Firefox and 9.76 for Chromium), Firefox is less efficient in its use of active persistence. For example, it writes back seven times as much data per page load as Chromium and incurs 1.7 times as much writeback latency as Chromium.

For both browsers, we observe that a significant share of syncs does not cause any writeback (24% for Firefox and 37% for Chromium). More generally speaking, a large share of syncs causes only a small amount of writeback and a low writeback latency. This is illustrated in Fig. 6.1.



(a) (b)

Figure 6.1: Distribution of syncs made by Firefox and Chromium in terms of writeback amount in KiB (a) and writeback latency in ms (b). The x-axes are scaled logarithmically.

Fig. 6.2 compares each sync's writeback latency with its writeback amount for both browsers. Contrary to what one might expect, both browsers do not show a clear proportional relationship between writeback latency and writeback amount. However, for Firefox we observe three separate groups of syncs, with each group respectively showing a linear relationship between writeback latency and writeback amount. We cannot see a clear pattern for Chromium, possibly due to the smaller range of writeback amounts we observed.

Fig. 6.3 and Fig. 6.4 show flamegraphs visualizing the frequency with which different function in Chromium and Firefox call `fdatasync()` and `fsync()`, respectively. In both cases, functions belonging to the SQLite library are responsible for a large share of the syncs.

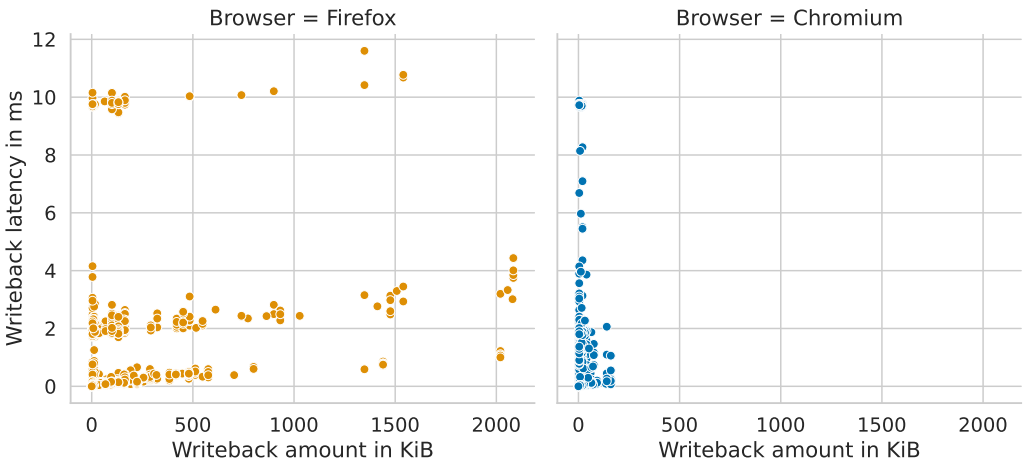


Figure 6.2: Relationship between writeback latency and writeback amount per sync for both Firefox and Chromium.

In Chromium's flame graph (Fig. 6.3), we can identify several causes for calls to SQLite, which together contribute a significant share of the `fdatasync()` calls:

- Usage of the Segmentation Platform, which "is a platform that uses intelligence and machine learning to guide developers for building purpose-built user experience for specific segments of users" [10] (function `segmentation_platform::UkmDatabaseBackend::UpdateUrlForUkmSource` with 13.81% of all samples)
- Writes to the browser history (function `history::HistoryBackend::CommitSingletonTransactionIfItExists` with 12.53% of all samples)
- Storing favicons for websites (function `favicon::FaviconBackend::Commit` with 11.25% of all samples)
- Storing data for error reporting and Network Error Logging [3] (function `net::SQLitePersistentReportingAndNelStore::Backend::DoCommit` with 7.67% of all samples)
- Storing cookies (function `net::SQLitePersistentCookieStore::Backend::DoCommit` with 3.84% of all samples)

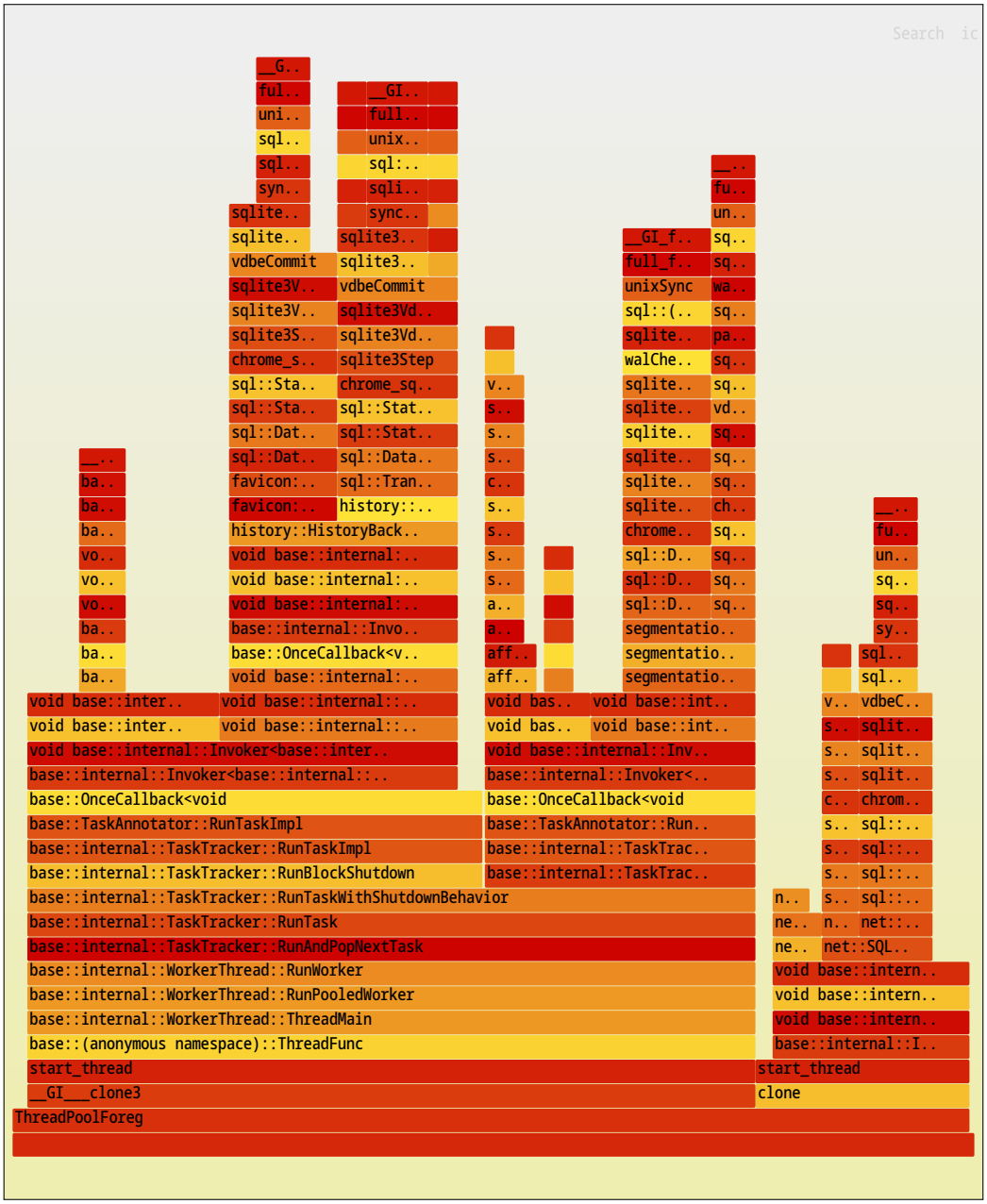


Figure 6.3: Flamegraph showing the frequency with which different functions call `fdatasync()` during page loads in Chromium. Segments taking up less than 3% of the width of the entire graph are omitted to improve readability.

In Firefox' flame graph (Fig. 6.4), we can see that a significant share of syncs is caused by accesses to the IndexedDB API [77] (function `mozilla::dom::indexedDB::(anonymous namespace)::FactoryOp::Run` with 18.57% of all samples) and by setting favicons for websites (function `mozilla::places::AsyncSetIconForPage::Run` with 16.67% of all samples).

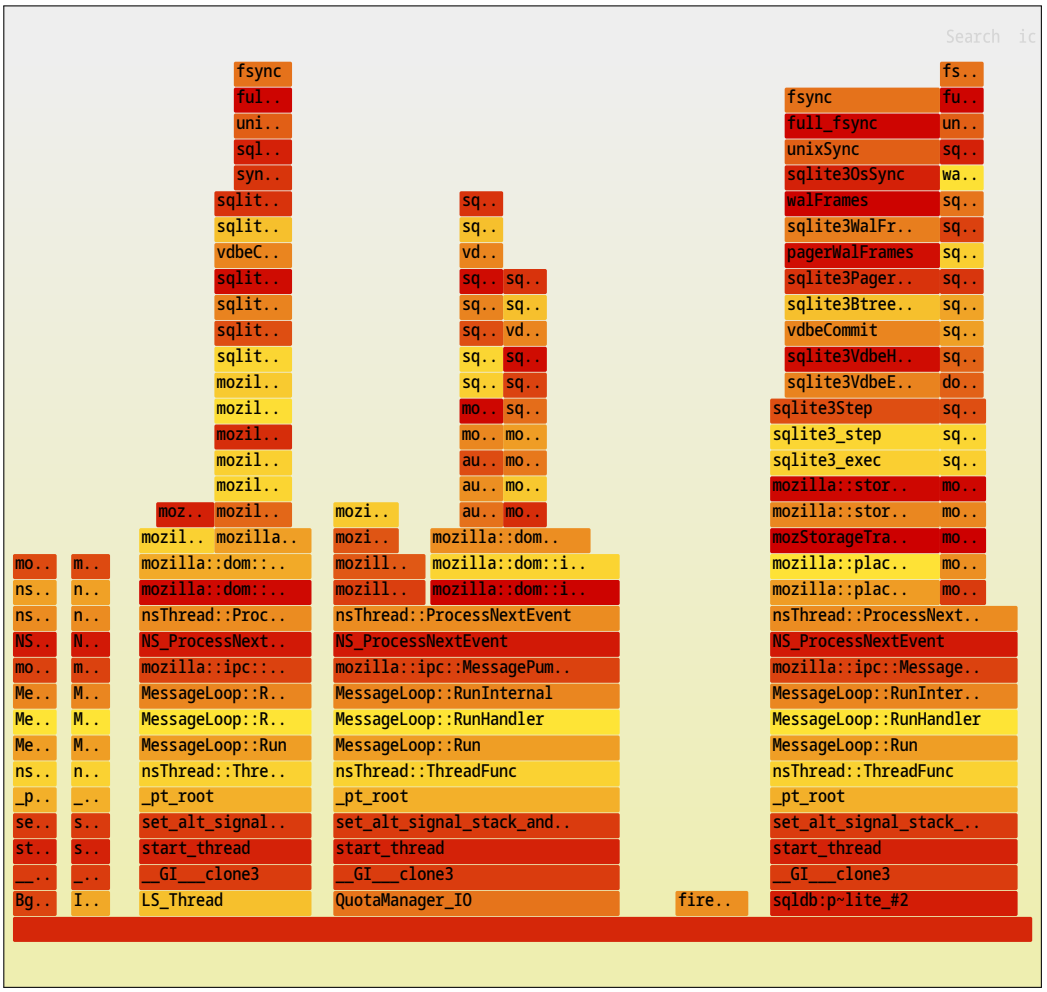


Figure 6.4: Flamegraph showing the frequency with which different functions call `fsync()` during page loads in Firefox. Segments taking up less than 3% of the width of the entire graph are omitted to improve readability.

Additionally, Fig. 6.5 shows the relative writeback latency caused by different functions calling `fdatasync()` during page loads in Chromium. Comparing Fig. 6.3 to Fig. 6.5, we also observe that the count of sync requests originating from a given code path is not necessarily proportional to the writeback latency caused by that code path.

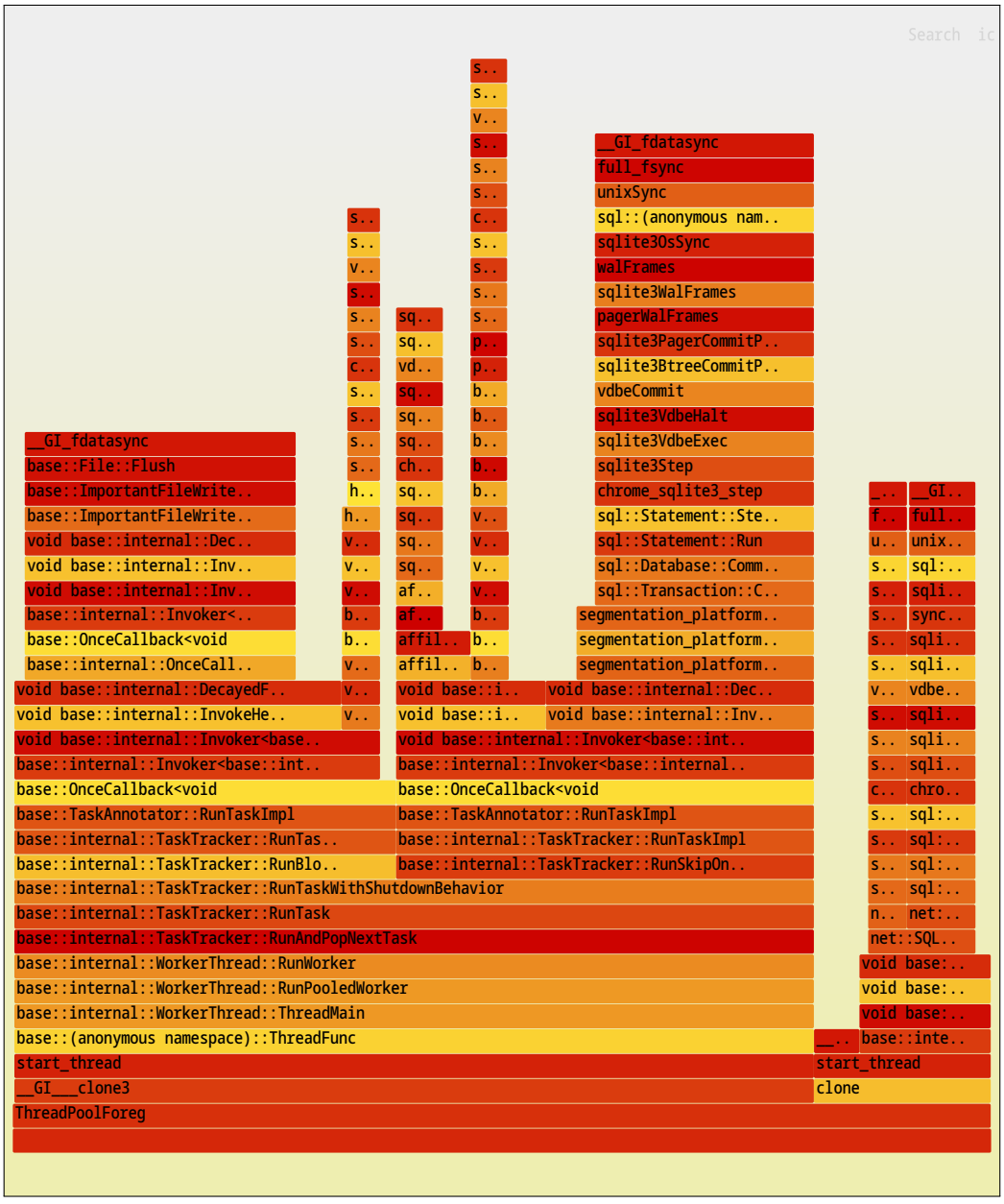


Figure 6.5: Flamegraph showing the relative writeback latency caused by different functions calling `fdatasync()` during page loads in Chromium. Segments taking up less than 3% of the width of the entire graph are omitted to improve readability.

Fig. 6.6 shows the distribution of `f*sync()` calls made by Firefox and Chromium across different files. In line with what we observe in Fig. 6.3 and Fig. 6.4, we see that Chromium makes many syncs to store favicons, browsing history, and data for the Segmentation Platform, while Firefox issues most syncs to the favicons database and website data storage files.

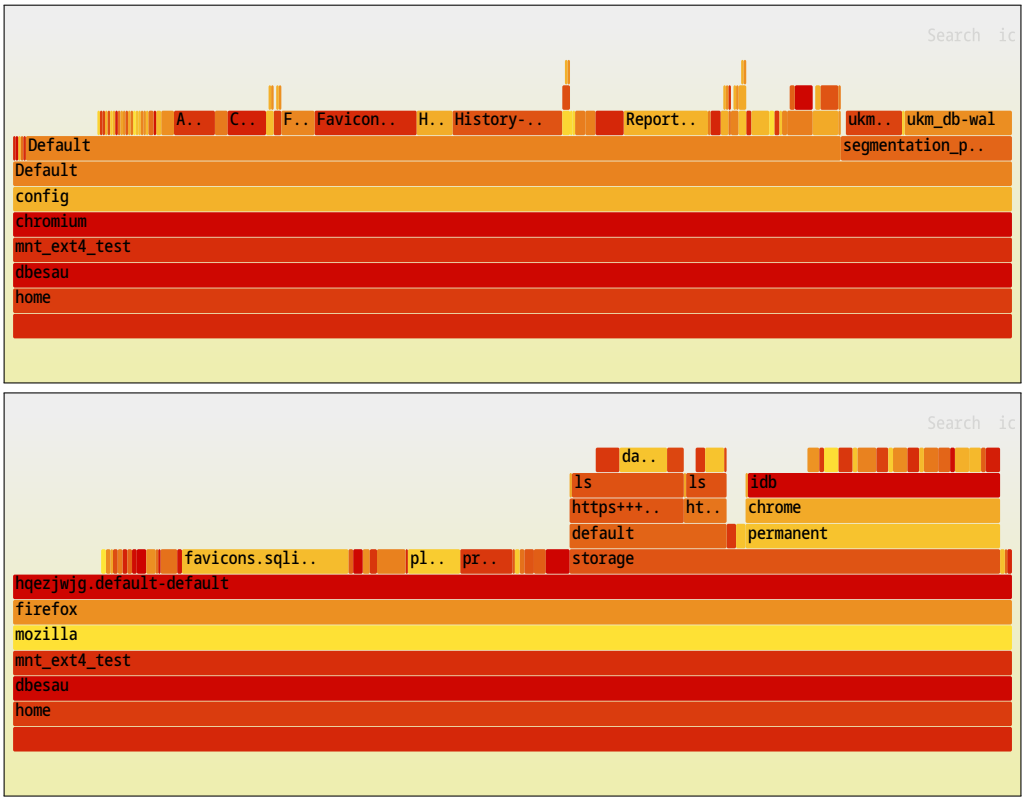


Figure 6.6: Distribution of `f*sync()` calls made by Chromium (top) and Firefox (bottom) across different files. In the bottom graph, the two bars with abbreviated labels to the right of `favicons.sqlite-wal` are for `places.sqlite`, where Firefox stores bookmarks and browsing history and `prefs-1.js`, where Firefox stores user preferences [58].

Finally, we note that both for Firefox and Chromium on the desktop our observations differ significantly from the 500 syncs per page load reported by Kim, Lee, and Won [41]. Without further research, it is difficult to determine the reason for this discrepancy. It is possible that optimizations in the SQLite library as well as the browsers themselves over the past decade have significantly reduced the number of syncs needed per page load. It is also possible that changes in Android's fork of SQLite [57] compared to the upstream version introduce additional `fsync()` calls.

6.2.2 Writeback Behavior of LibreOffice Writer

Metric	Save after each image	Save once after all images
Number of unique files synced	32	14
Percentage of syncs without writeback	52%	53%

Table 6.2: Most important writeback metrics for LibreOffice Writer when consecutively inserting 10 images 1 MiB to 3 MiB in size.

During our experiment, LibreOffice Writer only made `fsync()` calls and never used `fdatasync()`. Table 6.2 shows the most important synchronous writeback metrics for LibreOffice Writer in our experiment. The number of unique files synced when saving after each image (32) is significantly larger than when saving once after all images (14). Looking at the flame graph in Fig. 6.7 showing the distribution of `fsync()` calls across different files, we can see that, apart from the document file the users saves, LibreOffice Writer syncs temporary files in the same directory as that document file as well as configuration files under `/.local/share` used by GTK applications to keep track of recently used files. As with web browsers, a large share of syncs does not cause any writeback (52% when saving after each image and 53% when saving once after all images).

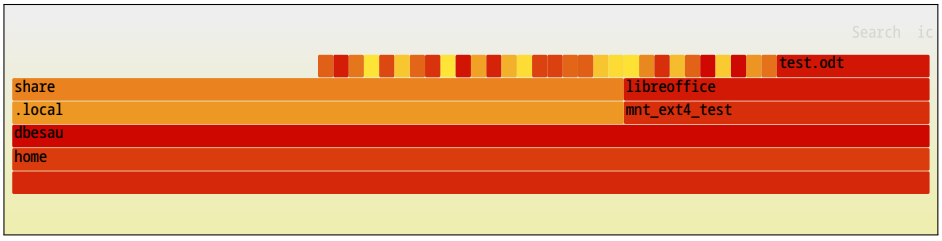


Figure 6.7: Flame graph showing the distribution of `fsync()` calls made by LibreOffice Writer across different files. The small segments in the same directory as `test.odt` are all temporary files created by LibreOffice Writer.

Fig. 6.8(a) shows the writeback amount of `fsync()` calls made by LibreOffice Writer over time. The spikes in writeback coincide with the points in time saved the document and grow linearly over time. When the user saves the document, the following happens: First, if the automatic backup feature is enabled, LibreOffice Writer creates a copy of the last on-disk version of the document in a separate backup directory. Next, LibreOffice Writer calls `fsync()` on a temporary file in the same directory as the document file. This

temporary file contains the current unsaved version of the document, and LibreOffice Writer has continuously updated it as the users made changes. After syncing this temporary file, LibreOffice Writer uses the `rename()` syscall to rename the temporary file to the name of the document file, thus atomically replacing the old version of the document with the new version [60]. Finally, LibreOffice Writer calls `fsync()` on the document file itself, which now contains the new version of the document. Harter et al. [32] observe a similar behavior with the Pages application from Apple's iWork productivity suite [52] and note that this combination of sync and rename operations is a common way for applications to atomically update files, since it ensures that the file is never in an inconsistent state. During our experiment, the first `fsync()` on the temporary file always seemed to cause writeback proportionate to the size of the entire document, while the second `fsync()` on the document file caused no writeback at all. Interestingly, we can observe no `fsync()` calls in between the moments we saved the document, which suggests that the AutoRecovery feature of LibreOffice does not force any writeback. This means that the AutoRecovery feature can help recover unsaved changes in case of an application crash, but not necessarily in case of a power failure. It is unclear to us why the AutoRecovery feature does not use `f*sync()`.

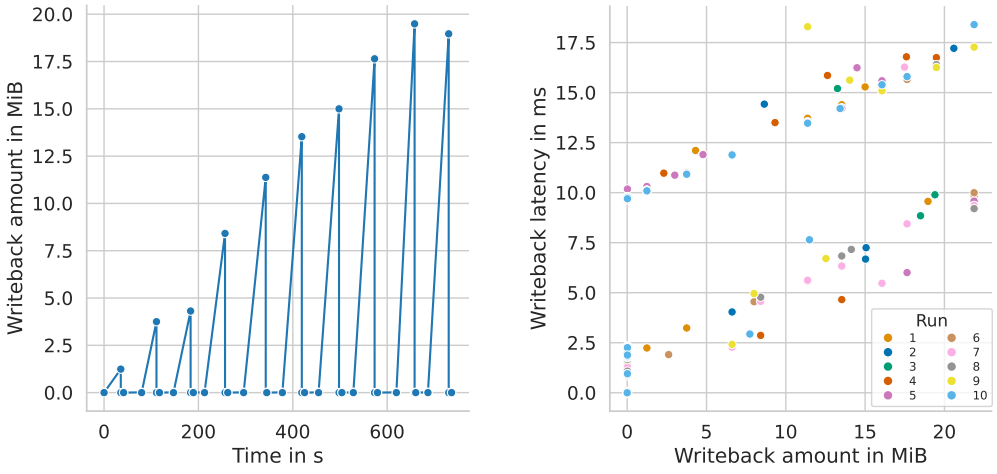


Figure 6.8: Writeback amount of `fsync()` calls made by LibreOffice Writer over time (a), and relationship between the amount of writeback and the writeback latency of `fsync()` calls made by LibreOffice Writer (b) when saving after each image is inserted. Colors in the right graph indicate different test runs.

Fig. 6.8(b) shows the relationship between the amount of writeback and the writeback latency of `fsync()` calls made by LibreOffice Writer. Similar to the results we observed for Firefox, there are multiple separate groups of syncs, with each group showing a linear relationship between writeback amount and writeback latency. However, while we observed three separate groups of syncs with Firefox, we only see two groups with LibreOffice Writer.

Fig. 6.9 shows the distribution of sync calls made by LibreOffice Writer in terms of writeback amount. We can see that, like with web browser, most syncs cause only a small amount of writeback.

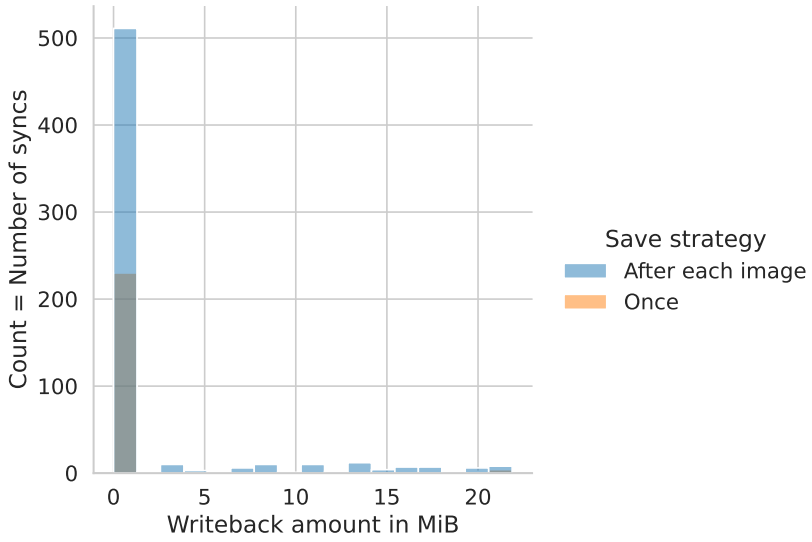


Figure 6.9: Distribution of sync calls made by LibreOffice Writer in terms of writeback amount in MiB.

Finally, in the flame graph in Fig. 6.10 we can see that the `fsync()` calls LibreOffice Writer makes originate from a smaller number of functions than the `f*sync()` calls made by web browsers.

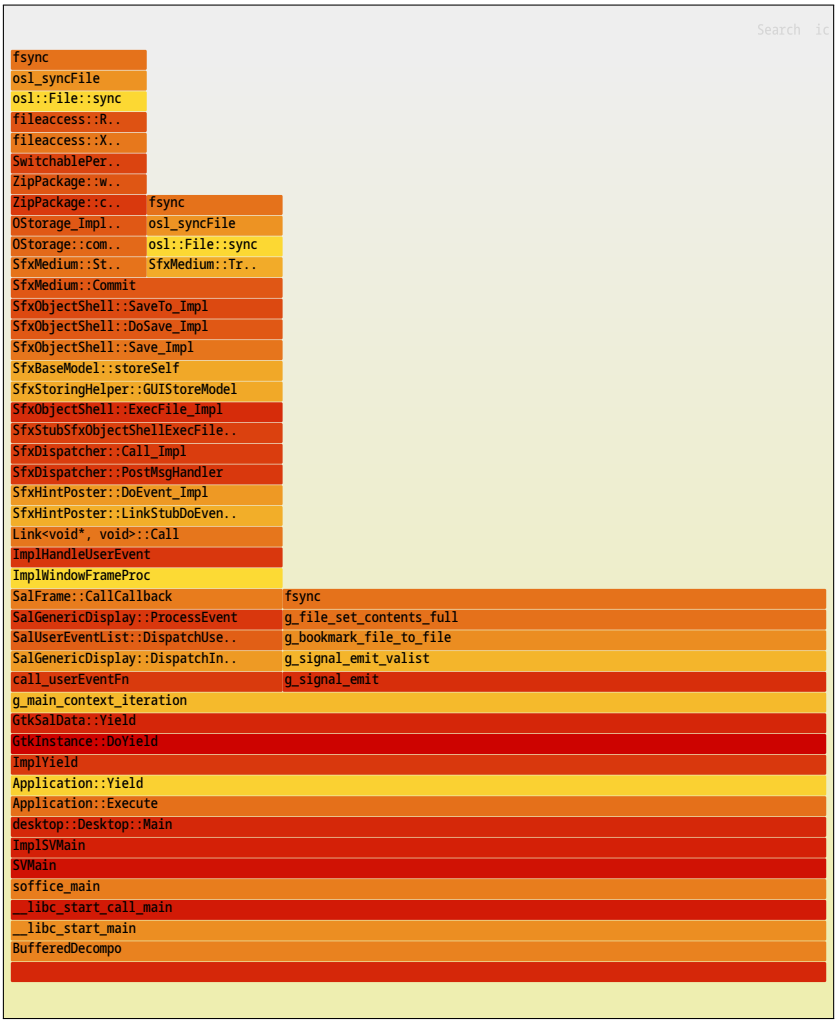


Figure 6.10: Flamegraph showing the frequency with which different functions call `fsync()` in LibreOffice Writer.

6.3 Discussion

From the results of our examination of the synchronous writeback behavior of different consumer applications, we draw the following conclusions:

`fdatasync()` is more efficient than `fsync()`. The results of our examination of Firefox and Chromium show that, while both Browsers make heavy use of active persistence, Chromium does so significantly more efficiently than Firefox. We think that this is due to Chromium using `fdatasync()` instead of `fsync()`, since Table 6.1 shows that Chromium's efficiency advantage can be observed across all metrics, both on the level of individual syncs and on the level of entire page loads.

While it is very unlikely that 7.2 ms of latency influence page loading speed to an extent that is noticeable by the user, it is important to keep in mind that the total latency due to `f*sync()` calls is significantly higher with 52.41 ms in the case of Firefox and 46.21 ms in the case of Chromium. We expect that this latency caused by `f*sync()` calls can, in addition to the many other tasks that browsers perform during page loads, push the total page load time above a threshold that is noticeable by the user. However, we should also note that the values we report per page load are accumulated over multiple threads. Depending to what degree these syncs are parallelized, the actual latency could be lower than the values we report.

This reveals a limitation of our approach, which we did not consider during the design of our experiment: With our current methods of analysis, we do not differentiate between syncs that happen in parallel and syncs that happen in sequence. Since our traces contain process IDs of the threads issuing sync requests, we expect that is is possible to extend our analysis to include statistics and visualizations which grant insights into how syncs are distributed across threads. Due to time constraints, we did not implement such analyses for our current prototype. Taking into account the characteristics of highly multi-threaded applications could be one area in which future research can improve upon our work. Still, our results confirm that choosing `fdatasync()` over `fsync()` can lead to performance improvements, depending on the application.

In Section 6.2.1, we mention that SQLite by default uses `fsync()` instead of `fdatasync()` on Unix due to concerns about broken `fdatasync()` implementations on some systems. It is true that, on some systems like FreeBSD or DragonFly BSD, the behavior of `fdatasync()` is different to the one on Linux and what POSIX specifies, in that `fdatasync()` "does not guarantee that file attributes or metadata necessary to access the file are committed to the permanent storage" [19, 24]. However, on Linux kernel versions 2.3

and later, a working `fdatasync()` implementation according to the POSIX standard is available [23]. Thus, we argue that it would be reasonable for SQLite to default to using `fdatasync()` on Linux.

Most syncs cause little to no writeback. In our results we see that most syncs cause only a small amount of writeback, and that a significant share of syncs does not cause any writeback. This seems to mirror the finding of Harter et al. [32] that for some applications, small writes dominate. For Firefox, Chromium, and LibreOffice Writer, we can add that small syncs dominate. We expect that the distribution of sync sizes can be an important factor to consider in the design of modern interfaces for active persistence.

Not all syncs are equal. Fig. 6.2 shows that, when comparing the write-back latency of syncs that Firefox makes with the amount of writeback they cause, three separate groups emerge. In Fig. 6.8(b), we see a similar pattern for LibreOffice Writer, but with two groups instead of three. This data from our experiments suggests that there is a factor that is more important in determining the writeback latency of a sync than the amount of writeback it causes. After conducting our experiment with Firefox and Chromium, we hypothesized that this factor could be the difference between sequential and non-sequential writeback, i.e. the more scattered a file's dirty pages are, the higher the writeback latency would be, placing it in a higher group. However, after observing similar behavior in LibreOffice Writer, we discarded this hypothesis, since LibreOffice Writer only performs sequential writeback and still exhibits this pattern. We are also confident that the split into separate groups is not caused by different conditions during the test runs, because Fig. 6.8(b) shows that samples from each run test run appear in both groups. We expect that one possible reason for the increased latency in one group is that, at the time of these syncs, ext4 was performing other I/O operations in the background, such as comitting a journal transaction. Unfortunately, we did not have time to test this hypothesis. It could be an interesting question for future research to investigate the reason for the split into different groups of syncs and whether similar behavior occurs on other file systems as well. Whatever the reason for this behavior, we expect it could be an important factor to consider in the design of modern interfaces for active persistence. Perhaps, it could even benefit writeback performance to have separate interfaces for different categories of sync requests.

Many syncs are for auxiliary data. Harter et al. [32] point out that for many applications, including office applications, most files accessed are auxiliary files instead of user files. However, this distinction is not useful for web browsers, since in a workload mainly consisting of loading and viewing different web pages, there are no user files in the same sense as in an office application, yet there is still user data. Thus, we suggest talking about user data and auxiliary data instead of user files and auxiliary files. First, because some applications do not necessary have user files, but still have user data, and secondly, because it is possible that user data and auxiliary data are stored in the same file. For example, we would consider cookies containing session data to be user data and cookies for tracking purposes to be auxiliary data, although both are stored in the same file. Based on this distinction, we observe that many of the syncs Firefox and Chromium make are for auxiliary data. For example, Firefox makes many syncs to the favicons database and files containing website data, while Chromium makes many syncs to the Network Error Log [3] and files belonging to the Segmentation Platform (see Section 6.2.1). We expect that most auxiliary data does not require persistence guarantees and that, consequently, many of the `f*sync()` calls for auxiliary data are not required. However, as long as user data and auxiliary data use the same storage mechanisms, we expect it would be difficult to avoid syncs for auxiliary data. Thus, we suggest that the distinction between user data and auxiliary data should be considered early on in an application's design process, so that user data and auxiliary data can be separated clearly in the file system.

In the case of LibreOffice Writer, we can note that, while it does write auxiliary data like the AutoRecovery information or recently opened documents, it forces none of these writes to disk immediately. LibreOffice Writer only uses `fsync()` when the user actively saves the document, and the entire synced data is user data. It is unclear to us why the AutoRecovery feature does not use `f*sync()`, since it would improve the chances of recovering unsaved changes in case of a power failure, and the performance overhead would be negligible, as by default the AutoRecovery feature only saves every 10 minutes, and the `f*sync()` call could happen in a background worker thread.

6.4 Future Work

With our approach, we have presented a novel way to trace and analyze the synchronous writeback behavior of applications on Linux. It allows developers and researchers to quickly create statistics and visualizations that help answer high-level questions about an application's synchronous writeback behavior. We have also given an overview of the data we collected using our approach and discussed the insights we gained from our analysis of this data. However, many areas remain in which future research can build upon our work and augment our approach. In the following, we outline the most important of these areas, which we have divided into five categories:

1. Extending the scope of our approach
2. Using more advanced analysis and visualization techniques
3. Augmenting our approach with additional data sources
4. Building user-friendly tooling around our approach
5. Using a different approach to answer open questions

6.4.1 Broader Scope

One limitation of our tooling is that we can currently only use it on the ext4 file system. However, this limitation is not inherent to our approach, but rather a consequence of the time constraints on this thesis. We expect that the custom tracepoints we use [30] can be implemented on other file systems as well. There could even be a way to implement the tracepoints in a file system-agnostic way, for example on the Linux kernel's block layer. Comparing data from our approach for the same workloads on different file systems could provide valuable insights into how the choice of file system influences the performance of synchronous writeback. We expect that performing experiments similar to ours on different file systems could also help determine the reason for the split into different groups of syncs we observed (see Section 6.3).

Another way to improve upon our work is to apply our approach to a broader range of applications and workloads. So far, we have examined web browsers and an office application. Even between these two classes of applications, we have observed significant differences in their synchronous writeback behavior. We expect that many other classes of applications exhibit unique writeback behavior as well.

Some types of applications that we could not examine in this thesis, but that we expect to exhibit interesting writeback behavior, are:

- Database management systems (DBMSs) like MySQL or PostgreSQL
- Key-value stores like Valkey
- Web servers like Apache or Nginx
- Mail clients like Thunderbird or Evolution

6.4.2 Analysis and Visualization

As we mention in Section 6.3, we have not implemented an effective way to take into account multiple threads issuing sync requests at the same time in our analysis. However, since our high-level log already contains the process IDs of the threads issuing sync requests as well as a start and end timestamp for each sync request, we expect that it is possible to create a visualization that shows how sync requests are distributed across threads, based on the data we collect using our current approach. We further expect that it is possible to run an analysis on the same data that can more accurately determine the effective performance overhead of synchronous writeback in multi-threaded applications, for example, in browsers during page loads.

In addition, we mention in Section 6.3 that we observe two distinct groups of syncs in our experiment with Firefox and Chromium, but not in our experiment with LibreOffice Writer. We hypothesize that the two groups could be caused by the difference between sequential and non-sequential writeback. Since we already collect data about the ranges of a file that `f*sync()` actually writes back, we expect that it it possible to either confirm or refute this hypothesis with an analysis of the data we collect. However, data from more different applications and workloads would also be needed to draw a definitive conclusion.

6.4.3 Additional Data Sources

Our approach to tracing synchronous writeback can grant valuable insights into the writeback behavior of applications. However, during our analysis, we have identified several areas in which data from additional sources could provide context that helps explain the behavior we observe.

One such area is application-specific performance data. Depending on the workload, some applications can already provide performance data about a specific operation that could be correlated with our data about synchronous writeback to better understand the performance implications of synchronous

writeback. For example, web browser can provide performance data about page load times which we could compare with our data to determine what part of the page load time is caused by synchronous writeback. In other cases, benchmarks can potentially provide application-specific performance data to augment our data.

Another way to augment our data could be to trace more syscalls than just `fsync()` and `fdatasync()`. At the moment, we are only measuring synchronous writeback. However, by also tracing syscalls like `open()` and `write()`, we could determine which writes are being synced by an application and which are not, or which files opened by an application are synced and which are not. It could also be useful to trace the `rename()` syscall, since it is often used in combination with `f*sync()` to atomically update files, as we mention in Section 6.2.2.

Finally, our approach could be expanded to include data not only about synchronous, but also about asynchronous writeback on the file system or block layer level. This way, we could compare the performance of forced writeback and background writeback, for example. In the case of journaling file systems like ext4, it would also be valuable to include data about I/O caused by journal transactions.

6.4.4 User-Friendly Tooling

While our current tooling is fully functional and serves as proof of concept for our approach to tracing and visualizing synchronous writeback, it lacks automation. For example, the entire process from collecting the trace data to creating a visualization currently involves chaining multiple scripts together on the command line. This modular approach is useful for development and debugging, but not for end users. We did create simple helper scripts to automate the process, but they are not robust and require the user to be aware of many assumptions they make about the environment in which they are run, otherwise manual intervention is still required. To improve usability, what is needed is a single robust command line tool wrapping all the functionality we implemented, including sane defaults, helpful error messages, and the ability to recover from errors.

We expect that, if refined, our approach could provide a useful tracing and profiling tool for kernel developers, application developers, system administrators, and researchers. Visualizations are an important part of our analysis. However, our current tooling creates a number of different flame graphs, all in separate files and thus hard to compare. Changing a parameter like the font size or dimensions of the flame graph requires re-running the data processing module, which can be time-consuming and tedious. We expect that the most

useful tool for developers would be a graphical user interface that combines all the visualizations we created and allows the user to interactively change parameters and see the results in real-time. This should also include a way to view different visualizations side-by-side and compare them.

6.4.5 Different Approaches

Our analysis focuses on the performance implications of synchronous writeback. However, as we lay out in Section 2.3.2 and Section 2.3.3, there are other issues with synchronous writeback around the risk of errors and how they are handled. Our approach is not well-suited to examine these issues. While our tooling is technically capable of recording the return value of each syscall and the error code it sets, this is of little use without a way to reproducibly inject errors into different layers of the I/O stack. Furthermore, since error reporting for `f*sync()` calls is not necessarily reliable, we expect that a successful analysis of error handling in synchronous writeback requires gathering additional tracing data from the kernel that would otherwise not be visible to the application. Since some of the most pressing issues with synchronous writeback are related to error handling [59, 12, 13, 14, 18], we expect that such an analysis could provide valuable insights for both kernel developers and application developers.

Chapter 7

Conclusion

Performance concerns about mechanisms for active persistence like `fsync()` have long been the object of research and discussions among developers. However, currently there is only limited insight into how applications make use of such mechanisms, which can serve as a basis for improving the design of kernel interfaces and applications. In this thesis, we have presented a novel approach to tracing and analyzing the synchronous writeback behavior of applications on Linux. Building on top of well-established tracing mechanisms, our approach can combine data from different sources and prepare it in such a way that we can use it to answer high-level questions about an application's writeback behavior. With flame graphs, we have also adapted an existing visualization technique, which has previously not been used for data about synchronous writeback specifically, to visualize data collected using our approach. Our current prototype supports tracing synchronous writeback behavior on the ext4 file system. In our evaluation, we use our tooling to examine the synchronous writeback behavior of web browsers as well as an office application. Through our analysis of the data we collected, we have gained several insights into the synchronous writeback behavior of the applications we examined. Most importantly, we have confirmed that `fdatasync()` is significantly more efficient than `fsync()` in the use cases we examined. Further, we have shown that most syncs cause little to no writeback and the amount of synchronous writeback is not the most important factor in determining writeback latency. Finally, we have presented areas in which future research can build on our work to gain a better understanding of the writeback behavior of applications.

In conclusion, we have not only provided new insights into the synchronous writeback behavior of applications, but we have also built a framework for tracing, analyzing, and visualizing synchronous writeback behavior. The

modular design of our tooling makes it possible to integrate additional data sources and add functionality. Thus, we expect that it can serve as a starting point for further research into application writeback behavior and prove a useful tool for both kernel and application developers.

Glossary

`f*sync()` We use this term to refer to the `fsync()` and `fdatasync()` syscalls whenever an argument applies equally to both of them due to their similar behavior. 1, 8–19, 21–23, 25, 26, 29, 33, 34, 36, 41, 42, 50, 52, 53, 55, 57, 59–61, 68

`fdatasync()` A system interface defined in the POSIX standard [35] and provided by Linux as a syscall in accordance with POSIX [23]. Similar to `fsync()`, with the exception that `fdatasync()` writes back the data of a file and only the metadata required for a future successful read of that file, while `fsync()` writes back the file's data and all metadata. 3, 4, 7–10, 15, 23, 34, 35, 44–49, 51, 55, 56, 60, 63, 65

`fsync()` A system interface defined in the POSIX standard [35] and provided by Linux as a syscall in accordance with POSIX [23]. Given a file descriptor, `fsync()` instructs the OS to write back any dirty data and metadata of that file to the underlying storage device, and only returns once the writeback has completed. v, 1, 3–5, 7–13, 15, 23, 27, 34, 42, 44, 45, 48, 50–55, 57, 60, 63, 65, 67

`io_uring` An interface in the Linux kernel for asynchronous I/O [36]. Applications queue I/O requests in one ring buffer and can retrieve their results from another ring buffer once they have completed. 15

`msync()` A system interface defined in the POSIX standard [35] and provided by Linux as a syscall in accordance with POSIX [49]. `msync()` is used to force dirty writes of memory-mapped files to disk and only returns once the writeback is complete. 7–9, 23, 34

`sync()` A system interface defined in the POSIX standard [35] and provided by Linux as a syscall [66]. `sync()` causes all in-memory modifications to any file systems to be written to disk. The Linux version always waits for the writeback to complete before returning, which is not required by POSIX. 8, 66

`sync_file_range()` A Linux-specific syscall that writes back dirty data within a specified range of a file to disk [67]. The use of `sync_file_range()` is discouraged due to very limited portability and risk of data loss or corruption if used incorrectly. 8, 14, 15

`syncfs()` A Linux-specific syscall [66]. `syncfs()` is similar to `sync()`, with the difference that `syncfs()` only causes in-memory modifications to a specific file system to be written to disk, while `sync()` affects all file systems. `syncfs()` always waits for the writeback to complete before returning. 8

active persistence We use this as a general term to refer to the usage of any interface via which an application explicitly request the OS to write back dirty data of a file to disk. Thus, active persistence can be viewed as a subcategory of the more general term data persistence. 4, 14, 16, 26, 44, 67

BIO A data structure in the Linux kernel. Basic unit of I/O for the block layer and lower layers [70]. 34–36

BPF *Berkeley Packet Filter*. Also known as *BSD Packet Filter*. It was originally introduced as an efficient kernel-level network packet filter to avoid the performance overhead of copying packets to user space before filtering and to replace less efficient packet filters previously used on Unix systems [45]. Eventually, BPF evolved into eBPF. 67

bpftrace A high-level tracing language for Linux [5]. bpftrace scripts are compiled to eBPF bytecode which can be run directly inside the kernel and thus can be used to build powerful tracing tools. 27, 28, 30, 33, 35, 36

CFBF *Compound File Binary Format*. An open file format developed by Microsoft for storing a hierarchical structure of multiple objects or data streams in a single file. Used, for example, by Microsoft Office documents. See [47]. 14

CXL *Compute Express Link*. A standard defining multiple protocols to interface with modern accelerator and memory devices [11]. CXL is also used by upcoming hybrid SSDs [31]. 19, 22, 67

data persistence We use this term to refer to any process of ensuring that data is stored on a non-volatile medium in such a way that it can be successfully retrieved in the future, as well as the state achieved by

such a process. This can be achieved explicitly via mechanisms like `fsync()` (i.e. active persistence) or implicitly via, for example, periodic background writeback by the OS. 5, 14, 26, 66

DBMS database management system 59

eBPF *extended Berkeley Packet Filter.* Developed on the basis of the original Berkeley Packet Filter (BPF), eBPF is not only a packet filter, but a general-purpose solution for running user-defined programs in a sandboxed runtime directly inside the Linux kernel [78]. 27, 66

flame graph A visualization technique for hierarchical data, introduced by Gregg to visualize performance data from tracing and profiling tools [28]. 5, 27, 37–39, 46, 48, 51, 53, 60, 63

folio A data structure in the Linux kernel. Represents a contiguous set of bytes in memory, for example in the page cache [71]. 34, 36

GPF *Global Persistent Flush.* A mechanism defined by the CXL standard that enabled reliable flushing of both CPU and storage device's caches in case of a power failure [11]. 19

hybrid SSD An upcoming type of SSD that is both block-addressable like traditional SSDs and byte-addressable like volatile memory [31]. Hybrid SSDs use a persistent cache to provide high-performance byte-granular access while still serving as persistent storage. 19, 22, 66

kprobe A tracing mechanism included in the Linux kernel [39]. kprobes can be used to dynamically insert breakpoints into arbitrary kernel code and execute custom code once a breakpoint is triggered. 27

LLVM A collection of modular and reusable compiler and toolchain technologies [68]. 27

OS operating system 3, 7, 12, 25, 41, 65–67

PLP *Power Loss Protection.* A feature often found on enterprise-grade SSDs [1]. It means that, in case of a power-failure, capacitors in the SSDs provide enough power to reliably flush the SSD's volatile cache. 19

rollback journal A mechanism used by databases to implement atomic commits and rollbacks [80]. Before a change is made to the database, the original unchanged database contents are recorded in the rollback

journal [20]. If an error occurs during the change, the rollback journal can be used to restore the original database contents. Modern databases often use WAL instead of rollback journals by default. 9

SLOG *Secondary Log Device*. In the context of ZFS, this refers to a separate device used to store the ZIL in order to speed up write caching [69]. 18

static tracepoint In the Linux kernel, static tracepoints are pre-defined hooks into kernel code at strategic locations intended to be used for profiling and debugging [2]. 27, 28, 33

synchronous writeback Writeback of dirty data to disk that an application explicitly requests (e.g., via `f*sync()`) and waits for the completion of. v, 4, 5, 22, 25–28, 31, 33, 41–44, 51, 55, 58–61, 63

WAL *Write-Ahead Log*. A mechanism used by databases to implement atomic commits and rollbacks [80]. Before a change is made to the database, the change is written to the sequential WAL file. Commits in the WAL allow for rollback to a consistent database state in case of a failure. Changes in the WAL are periodically applied to the database in the background. 9, 13, 68

ZIL *ZFS Intent Log*. A Temporary cache used by ZFS to store synchronous writes before they are applied to the file system [81]. 18, 68

Bibliography

[1] *A Closer Look At SSD Power Loss Protection*. Kingston Technology Company. Mar. 2019. URL: https://www.kingston.com/en/blog/servers-and-data-centers/ssd-power-loss-protection (visited on 11/28/2024).

[2] Jason Baron and William Cohen. *The Linux Kernel Tracepoint API*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/core-api/tracepoint.html (visited on 03/24/2025).

[3] Eric Bidelman and Maud Nalpas. *Network Error Logging (NEL)*. web.dev. Jan. 31, 2022. URL: https://web.dev/articles/network-error-logging (visited on 03/29/2025).

[4] *bpftrace(8) Manual Page*. Github. URL: https://github.com/bpftrace/bpftrace/blob/master/man/adoc/bpftrace.adoc#probes (visited on 03/16/2025).

[5] *bpftrace/bpftrace*. Nov. 21, 2024. URL: https://github.com/bpftrace/bpftrace (visited on 11/21/2024).

[6] Li-Pin Chang et al. "Eager Synching: A Selective Logging Strategy for Fast fsync() on Flash-Based Android Devices." In: *ACM Trans. Embed. Comput. Syst.* 16.2 (Dec. 19, 2016), 34:1–34:25. ISSN: 1539-9087. DOI: 10.1145/2930668. URL: https://dl.acm.org/doi/10.1145/2930668 (visited on 10/21/2024).

[7] Gyeongyeol Choi and Youjip Won. "Analysis for the performance degradation of fsync() in F2FS." In: *Proceedings of the 9th International Conference on E-Education, E-Business, E-Management and E-Learning*. IC4E '18. New York, NY, USA: Association for Computing Machinery, Jan. 11, 2018, pp. 71–75. ISBN: 978-1-4503-5485-1. DOI: 10.1145/3183586.3183605. URL: https://dl.acm.org/doi/10.1145/3183586.3183605 (visited on 11/13/2024).

[8] *Chromium*. The Chromium Projects. URL: https://www.chromium.org/Home/ (visited on 04/03/2025).

[9] *chromium*. Git at Google. URL: https://chromium.googlesource.com/chromium/ (visited on 03/29/2025).

[10] *components/segmentation_platform - chromium/src*. Git at Google. URL: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/components/segmentation_platform (visited on 03/29/2025).

[11] *Compute Express Link (CXL) Specification Revision 3.1*. Aug. 7, 2023. URL: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf (visited on 03/23/2025).

[12] Jonathan Corbet. *ext4 and data loss*. LWN. Mar. 11, 2009. URL: https://lwn.net/Articles/322823/ (visited on 10/21/2024).

[13] Jonathan Corbet. *Improved block-layer error handling*. LWN. June 2, 2017. URL: https://lwn.net/Articles/724307/ (visited on 11/13/2024).

[14] Jonathan Corbet. *PostgreSQL's fsync() surprise*. LWN. Apr. 18, 2018. URL: https://lwn.net/Articles/752063/ (visited on 10/21/2024).

[15] Jake Edge. *Asynchronous fsync()*. LWN. May 21, 2019. URL: https://lwn.net/Articles/789024/ (visited on 10/21/2024).

[16] Jake Edge. *Filesystems and crash resistance*. LWN. May 21, 2019. URL: https://lwn.net/Articles/788938/ (visited on 11/12/2024).

[17] Jake Edge. *Handling I/O errors in the kernel*. LWN. June 12, 2018. URL: https://lwn.net/Articles/757123/ (visited on 02/10/2025).

[18] Jake Edge. *PostgreSQL visits LSFMM*. LWN. May 1, 2018. URL: https://lwn.net/Articles/752952/ (visited on 11/13/2024).

[19] *fdatasync(2)*. FreeBSD Manual Pages. URL: https://man.freebsd.org/cgi/man.cgi?query=fdatasync&sektion=2&manpath=freebsd-release-ports (visited on 04/02/2025).

[20] *File Locking And Concurrency In SQLite Version 3*. SQLite Documentation. Mar. 3, 2025. URL: https://www.sqlite.org/lockingv3.html#rollback (visited on 03/24/2025).

[21] *fs/btrfs/tree-log.c*. Version v6.13.2. Oracle. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/btrfs/tree-log.c?h=v6.13.2 (visited on 02/17/2025).

[22] *fs/xfs/xfs_file.c*. Version v6.13.2. Silicon Graphics. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/xfs/xfs_file.c?h=v6.13.2 (visited on 02/12/2025).

[23] *fsync(2)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/fsync.2.html (visited on 10/21/2024).

[24] *fsync(2)*. DragonFly On-Line Manual Pages. URL: https://man.dragonflybsd.org/?command=fsync&section=2 (visited on 04/02/2025).

[25] *Get Firefox browser*. Mozilla. URL: https://www.mozilla.org/en-US/firefox/ (visited on 04/03/2025).

[26] Brendan Gregg. *brendangregg/FlameGraph*. Mar. 20, 2025. URL: https://github.com/brendangregg/FlameGraph (visited on 03/20/2025).

[27] Brendan Gregg. *CPU Flame Graphs*. Brendan's site. URL: https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html (visited on 03/20/2025).

[28] Brendan Gregg. *Flame Graphs*. Brendan's site. URL: https://www.brendangregg.com/flamegraphs.html (visited on 03/20/2025).

[29] Brendan Gregg. *Off-CPU Flame Graphs*. Brendan's site. URL: https://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html (visited on 03/20/2025).

[30] Daniel Habicht. *wb_tracer Kernel Patch*. Version v3. KIT Operating Systems Group, Jan. 28, 2025.

[31] Daniel Habicht et al. "Fundamental OS Design Considerations for CXL-based Hybrid SSDs." In: *Proceedings of the 2nd Workshop on Disruptive Memory Systems*. SOSP '24: ACM SIGOPS 30th Symposium on Operating Systems Principles. Austin TX USA: ACM, Nov. 3, 2024, pp. 51–59. ISBN: 979-8-4007-1303-3. DOI: 10.1145/3698783.3699380. URL: https://dl.acm.org/doi/10.1145/3698783.3699380 (visited on 10/30/2024).

[32] Tyler Harter et al. "A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications." In: *ACM Trans. Comput. Syst.* 30.3 (Aug. 1, 2012), 10:1–10:39. ISSN: 0734-2071. DOI: 10.1145/2324876.2324878. URL: https://dl.acm.org/doi/10.1145/2324876.2324878 (visited on 10/21/2024).

[33] Tyler Harter et al. *The ADvanced Systems Laboratory (ADSL): Traces*. The ADvanced Systems Laboratory (ADSL). URL: https://research.cs.wisc.edu/adsl/Traces/ibench/ (visited on 03/23/2025).

[34] *Home*. LibreOffice. URL: https://www.libreoffice.org/ (visited on 04/03/2025).

[35] "IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))." In: *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (Dec. 2008), pp. 1–3874. DOI: 10.1109/IEEESTD.2008.4694976. URL: https://ieeexplore.ieee.org/document/4694976 (visited on 11/15/2024).

[36] *io_uring(7)*. Linux manual page. URL: https://man7.org/linux/man-pages/man7/io_uring.7.html (visited on 10/30/2024).

[37] *io_uring_enter(2)*. Linux manual page. URL: https://man7.org/linux/man-pages/man2/io_uring_enter.2.html (visited on 11/19/2024).

[38] *Journal (jbd2)*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html (visited on 04/02/2025).

[39] Jim Keniston, Prasanna S. Panchamukhi, and Masami Hiramatsu. *Kernel Probes (Kprobes)*. The Linux Kernel documentation. URL: https://www.kernel.org/doc/html/latest/trace/kprobes.html (visited on 03/24/2025).

[40] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. "Examining storage performance on mobile devices." In: *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*. MobiHeld '11. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2011, pp. 1–6. ISBN: 978-1-4503-0980-6. DOI: 10.1145/2043106.2043112. URL: https://dl.acm.org/doi/10.1145/2043106.2043112 (visited on 03/29/2025).

[41] Myungsik Kim, Seongjin Lee, and Youjip Won. "Comparative study on I/O characteristics of mobile web browsers." In: *2015 IEEE 5th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*. 2015 IEEE 5th International Conference on Consumer Electronics - Berlin (ICCE-Berlin). Sept. 2015, pp. 224–227. DOI: 10.1109/ICCE-Berlin.2015.7391241. URL: https://ieeexplore.ieee.org/document/7391241 (visited on 10/21/2024).

[42] Myungsik Kim, Seongjin Lee, and Youjip Won. *ESOS-Lab/AndroTrace*. Apr. 2, 2024. URL: https://github.com/ESOS-Lab/AndroTrace (visited on 03/23/2025).

[43] Alexander Larsson. *ext4 vs fsync, my take*. Mar. 16, 2009. URL: https://blogs.gnome.org/alexl/2009/03/16/ext4-vs-fsync-my-take/ (visited on 10/21/2024).

[44] Changman Lee et al. "F2FS: A New File System for Flash Storage." In: 13th USENIX Conference on File and Storage Technologies (FAST 15). 2015, pp. 273–286. ISBN: 978-1-931971-20-1. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee (visited on 11/28/2024).

[45] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture (Preprint)." In: USENIX Winter Conference 1993. Dec. 19, 1992.

[46] M. D. McIlroy, E. N. Pinson, and B. A. Tague. *UNIX Time-Sharing System: Foreword*. Vol. 6. Bell System Technical Journal 57. July 8, 1978. 6 pp. URL: http://archive.org/details/bstj57-6-1899 (visited on 03/17/2025).

[47] Microsoft. *Compound File Binary File Format*. Version 11.0. June 25, 2021. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/53989ce4-7b05-4f8d-829b-d08d6148375b (visited on 03/13/2025).

[48] *mozilla-central*. Mozilla Mercurial Repositories. URL: https://hg.mozilla.org/mozilla-central/summary (visited on 03/29/2025).

[49] *msync(2)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/msync.2.html (visited on 11/19/2024).

[50] *On-disk Format — BTRFS documentation*. URL: https://btrfs.readthedocs.io/en/latest/dev/On-disk-format.html#item-types (visited on 02/17/2025).

[51] *open(2)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/open.2.html (visited on 11/20/2024).

[52] *Pages*. Official Apple Support. URL: https://support.apple.com/pages (visited on 03/28/2025).

[53] Daejun Park and Dongkun Shin. "iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call." In: *Proceedings of the 2017 USENIX Annual Technical Conference* (July 12–14, 2017).

[54] Seon-yeong Park et al. "Exploiting Internal Parallelism of Flash-based SSDs." In: *IEEE Computer Architecture Letters* 9.1 (Jan. 2010), pp. 9–12. ISSN: 1556-6064. DOI: 10.1109/L-CA.2010.3. URL: https://ieeexplore.ieee.org/document/5428220/?arnumber=5428220 (visited on 03/29/2025).

[55] *perf-script(1)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man1/perf-script.1.html (visited on 03/20/2025).

[56] *perf(1)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man1/perf.1.html (visited on 03/20/2025).

[57] *platform/external/sqlite*. Git at Google. URL: https://android.googlesource.com/platform/external/sqlite/ (visited on 03/29/2025).

[58] *Profiles - Where Firefox stores your bookmarks, passwords and other user data*. Mozilla Support. Mar. 16, 2025. URL: https://support.mozilla.org/en-US/kb/profiles-where-firefox-stores-user-data (visited on 03/29/2025).

[59] Anthony Rebello et al. "Can Applications Recover from fsync Failures?" In: *ACM Trans. Storage* 17.2 (June 15, 2021), 12:1–12:30. ISSN: 1553-3077. DOI: 10.1145/3450338. URL: https://dl.acm.org/doi/10.1145/3450338 (visited on 10/21/2024).

[60] *rename(2)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/rename.2.html (visited on 04/02/2025).

[61] *Saving Documents Automatically*. LibreOffice Help. URL: https://help.libreoffice.org/latest/en-GB/text/shared/guide/doc_autosave.html (visited on 03/28/2025).

[62] *Selenium*. Selenium. URL: https://www.selenium.dev/ (visited on 03/25/2025).

[63] Harshad Shirwadkar. *Re: [PATCH v10 5/9] ext4: main fast-commit commit path*. E-mail. Oct. 26, 2020. URL: https://lore.kernel.org/linux-ext4/CAD+ocbws2J0boxfNA+gahWwTAqm8-Pef9_WkcwwKFjpiJhvJKw@mail.gmail.com/ (visited on 02/12/2025).

[64] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. "Fast-Commit: resource-efficient, performant and cost-effective file system journaling." In: *Proceedings of the 2024 USENIX Annual Technical Conference* (July 10–12, 2024).

[65] *sqlite/src/os_unix.c*. URL: https://github.com/sqlite/sqlite/blob/62d9d70eddda991bd3dedb55c1beb5a23fb6cae8/src/os_unix.c#L3580 (visited on 03/29/2025).

[66] *sync(2)*. Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/sync.2.html (visited on 11/15/2024).

[67] *sync_file_range(2)*. Linux manual page. URL: https://man7.org/linux/man-pages/man2/sync_file_range.2.html (visited on 11/16/2024).

[68] *The LLVM Compiler Infrastructure Project*. URL: https://llvm.org/ (visited on 03/24/2025).

[69] *To SLOG or not to SLOG: How to best configure your ZFS Intent Log*. TrueNAS Blog. Mar. 4, 2016. URL: https://www.truenas.com/blog/o-slog-not-slog-best-configure-zfs-intent-log/ (visited on 03/24/2025).

[70] Linus Torvalds. *include/linux/blk_types.h*. Version v6.12. URL: https://web.git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/blk_types.h?h=v6.12 (visited on 03/19/2025).

[71] Linus Torvalds. *include/linux/mm_types.h*. Version v6.12. URL: https://web.git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/mm_types.h?h=v6.12 (visited on 03/19/2025).

[72] Linus Torvalds. *include/linux/writeback.h*. Version v6.12. URL: https://web.git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/writeback.h?h=v6.12 (visited on 03/19/2025).

[73] Yves Trudeau. *Fsync Performance on Storage Devices*. Percona Database Performance Blog. Feb. 8, 2018. URL: https://www.percona.com/blog/fsync-performance-storage-devices/ (visited on 10/21/2024).

[74] Theodore Ts'o. *fs/ext4/page-io.c*. Version v6.12. URL: https://web.git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/ext4/page-io.c?h=v6.12 (visited on 03/19/2025).

[75] Theodore Ts'o. *Re: fsync() errors is unsafe and risks data loss*. E-mail. Apr. 12, 2018. URL: https://lwn.net/Articles/752112/ (visited on 11/13/2024).

[76] Stephen Tweedie et al. *fs/ext4/fsync.c*. Version v6.12. URL: https://web.git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/ext4/fsync.c?h=v6.12 (visited on 03/16/2025).

[77] *Using IndexedDB - Web APIs*. MDN. Oct. 27, 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB (visited on 04/03/2025).

[78] *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. eBPF Documentation. URL: https://ebpf.io/what-is-ebpf/ (visited on 11/21/2024).

[79] *Write Ahead Log*. PostgreSQL Documentation. Jan. 1, 2012. URL: https://www.postgresql.org/docs/8.1/runtime-config-wal.html (visited on 11/15/2024).

[80] *Write-Ahead Logging*. SQLite Documentation. URL: https://www.sqlite.org/wal.html (visited on 10/30/2024).

[81] *ZFS ZIL and SLOG*. TrueNAS Documentation. URL: https://www.truenas.com/references/zilandslog/ (visited on 11/20/2024).