

Vrije Universiteit Amsterdam



Bachelor Thesis

Exploring Redis Persistence Modes: Introducing AOFURing, an `io_uring` AOF Extension

Author: Darko Vujica (2701541)

1st supervisor: Tiziano De Matteis
daily supervisor: Krijn Doekemeijer, Zebin Ren
2nd reader: Alexandru Iosup

*A thesis submitted in fulfillment of the requirements for
the VU Bachelor of Science degree in Computer Science*

August 16, 2024

Abstract

In an era where real-time data processing and high-throughput applications are integral to industries such as finance, e-commerce, healthcare, and gaming, ensuring high performance in data storage systems is crucial. Redis, widely adopted for its speed and versatility, is central to these environments, supporting applications ranging from caching and session storage to real-time analytics and data streaming. As the demands on these systems grow, optimizing I/O operations is increasingly important. With faster storage technologies and the need for low-latency operations, traditional I/O mechanisms are becoming bottlenecks.

The key scientific questions addressed in this thesis are: 'How does the integration of `io_uring` in Redis AOF persistence affect performance metrics?' and 'What impact does this integration have on data correctness and durability?'

To answer these questions, this thesis introduces AOFURing, an extension of Redis's AOF persistence mode that leverages the `io_uring` API to optimize I/O operations. By comparing AOFURing with existing persistence modes, this work provides insights into the trade-offs between throughput, latency, and resource utilization. Additionally, it explores how such a design, and Asynchronous I/O (AIO) in general, might affect data correctness and durability.

AOFURing's performance showed mixed results. While it outperformed AOF (`fsync=always`) with more than 6 times higher RPS, it underperformed compared to AOF (`fsync=everysec`) and AOF (`fsync=no`), with a reduction in RPS by up to 3.4%. This improvement came with the drawback of a significant increase in CPU usage, around 189% on average. Although AOFURing proved reliable in terms of data correctness, it poses a potential risk of data loss during system failures due to the asynchronous nature of `io_uring`.

The code for AOFURing is openly available at: https://github.com/daraccrafter/Thesis-Redis-IO_Uring.git.

Qualitative

That's functional
Correctness
Requirement

3 gap.

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem Statement	2
1.3	Research Questions	3
1.4	Research Methodology	3
1.5	Thesis Contributions	3
1.6	Plagiarism Declaration	4
1.7	Thesis Structure	4
2	Background on Redis and io_uring	5
2.1	RDB	5
2.2	AOF	7
2.2.1	AOF Log	8
2.2.2	AOF Rewrite	9
2.2.3	Loading Data in AOF	9
2.2.4	Redis AOF Flow of Execution	10
2.3	io_uring	11
2.4	Liburing	14
3	Design of AOFUring	15
3.1	Submission of SQEs	16
3.1.1	SQE Setup and Linking	16
3.2	Processing of SQEs	17
3.3	Completion of CQEs	17
3.4	Data Durability of AOFUring	18

CONTENTS

4	Implementation of AOFUring	20
4.1	Development Environment and Tools	20
4.2	AOFUring Configuration	21
4.3	AOFUring Ring Initialization	22
4.4	Execution Flow of AOFUring	22
4.5	Submitting Submission Queue Entries	23
4.5.1	AOFUring Write Function	23
4.5.2	AOFUring Fsync Function	24
4.6	Processing Completion Queue Entries	25
5	Evaluation	27
5.1	Benchmark Environment	27
5.2	Benchmarking Process	27
5.3	Performance Analysis	29
5.4	Resource Consumption	33
5.5	AOFUring Data Correctness Test	35
6	Related Work	36
7	Conclusion	38
7.1	Answering Research Questions	38
7.2	Limitations and Future Work	39
	References	40
A	Reproducibility	43
A.1	Abstract	43
A.2	Artifact Check-list (Meta-information)	43
A.3	Description	44
A.3.1	How to Access	44
A.3.2	Software Dependencies	44
A.4	Installation	44
A.4.1	Cloning the Repository	44
A.4.2	Ubuntu & Debian-based	44
A.4.3	Manual Dependency Installation	45
A.5	Evaluation and Expected Results	46
A.5.1	Benchmarks	46

CONTENTS

A.5.2	Data Correctness Test	48
A.5.3	Plotting	48

CONTENTS

1

Introduction

1.1 Context

In the era of technology, efficient data storage systems are critical. Businesses gather extensive customer data, which they rely on in order to grow their customer base. Therefore, fast-performing data storage systems are essential for a multitude of applications, especially in the domain of High-Performance Computing (HPC) and Data Intensive Scalable Computing (DISC). As the volume of data to be processed expands, systems require storage systems which have the capacity to handle large amounts of data without failures. Furthermore, the manageability of complex systems becomes more achievable as we simplify and enhance storage mechanisms, reducing human error and the complexity of Information and Communication Technology (ICT) services (1). ✓

Among the various data storage solutions, Redis stands out as a widely adopted in-memory data structure store, known for its speed and flexibility (2). It is commonly used in scenarios requiring rapid data access and real-time analytics, making it an integral part of many big data and HPC environments. Providing robust and scalable storage solutions that support critical sectors such as industry, healthcare, and government, makes advanced computing technology more accessible and beneficial to society as a whole (3). Redis addresses this need by offering versatile storage options that ensure both performance and durability. To achieve this, Redis offers two distinct persistence modes: snapshotting (4) (Redis Database) and journaling (5) (Append-Only File). Append-Only File (AOF) proves to be more reliable; however, Gottesman et al. state that this mechanism raises a problem regarding performance as it is quite slow (6). ✓

Recent advancements in storage systems introduce `io_uring`, a modern asynchronous I/O API. A recent study finds that `io_uring` promises to reduce latency and improve

what's
less
reliable?

1. INTRODUCTION

throughput, by reducing the inefficiencies of traditional system call mechanisms (7). Hence, exploring the integration of `io_uring` into Redis's AOF mechanism to enhance its *write* and *fsync* performance is a crucial step forward. The primary challenge of integrating `io_uring` into Redis AOF lies in ensuring compatibility with existing functionalities. The interface significantly affects functionality because standard I/O operations rely on system calls like *fsync* and *write*, which work differently on `io_uring`. Shifting to `io_uring` involves rethinking how Redis handles these I/O operations. | good. ✓

This research presents a design for incorporating `io_uring` into the AOF persistence mode, referred to as AOFUring. The study examines the throughput and latency benefits, potential, and limitations of this design. Additionally, a performance analysis is conducted, comparing various AOF configurations (section 2.2), RDB, and AOFUring.

1.2 Problem Statement

In Redis, the AOF journaling mechanism guarantees data durability by logging every *write* operation, ensuring recoverability in case of system failures by replaying the log on reboot. Conventional implementations of journal persistence modes use POSIX I/O, which is reliant on system calls such as *write* and *fsync*, impose significant performance overhead. Moreover, studies state that the AOF mechanism raises a performance issue due to logging transactions to an append-only log file (6). This causes significant performance impact as persistence is not made until the AOF file is flushed to the disk, slowing down the process. Therefore, providing a fast and reliable in-memory key-value store that can compete with traditional on-disk databases is a challenging endeavor.

On the other hand, the snapshot persistence mode Redis Database (RDB) performs comparably better, although there is a higher risk of data loss due to the intervals in which the snapshots are taken. While this paper does not focus on improving RDB performance, the comparison of the AOF optimization with Redis's standard persistence mode provides a better insight on advancing its performance.

The primary challenge of this thesis is to optimize Redis AOF using asynchronous I/O to match the performance of the default Redis (RDB). While this thesis does not delve deeply into the data durability of such system, it lays the groundwork for future research into optimizing asynchronous I/O stores. | This is in the abstract

1.3 Research Questions

To evaluate the performance, we break down the problem of assessing the implementation of `io_uring` into the following research questions:

- **(RQ1)** *How does the performance of AOFUring compare to traditional Redis persistence modes?* This research aims to compare the performance of AOFUring with both the Redis AOF (in three different configurations) and **default RDB persistence modes**. Since RDB is considered the gold standard for Redis persistence, this comparison will provide a clearer visualization of improvements in AOF throughput and latency. Evaluating the performance trade-offs between AOFUring, standard AOF, and RDB is crucial for determining whether `io_uring` can enhance AOF to be closer in performance to RDB. By directly comparing the standard AOF with the `io_uring`-enhanced AOF, we aim to measure the progress and effectiveness of `io_uring` in mitigating performance bottlenecks caused by system calls.
- **(RQ2)** *What impact does AOFUring have on data correctness, and durability?* This research question aims to investigate how the integration of `io_uring` into Redis AOF (AOFUring) **affects these critical aspects**.

1.4 Research Methodology

The research aims to evaluate and enhance Redis performance through a comprehensive approach. Initially, various Redis configurations will be set up and benchmarked to identify performance bottlenecks and understand how these configurations influence throughput and latency. This will be followed by an exploration of `io_uring`, along with `liburing`, a library designed to facilitate the use of `io_uring`. The goal is to assess their potential benefits for improving Redis AOF performance. Based on the findings, a design plan will be developed to optimize Redis performance. Finally, implementing `io_uring` in the Redis AOF persistence mode according to the design, and a series of performance benchmarks will be conducted to address the research questions and evaluate the impact of these optimizations.

Refⁿ

1.5 Thesis Contributions

This thesis makes contributions to the performance optimization in key-value stores:

1. INTRODUCTION

- **Conceptual Contribution:** This thesis introduces a design for integrating `io_uring` into Redis’s AOF persistence mode, addressing performance limitations associated with AOF. This integration paves the way for further experimentation with asynchronous I/O in in-memory datastores. In the future, this approach could help in experimenting with techniques for lowering the data loss risk, providing faster and more reliable in-memory database options.
- **Experimental Contribution:** Experiments benchmark for the latency and throughput of Redis persistence mechanisms and AOFUring. The impact of `io_uring` on Redis’s resource consumption, especially CPU and memory usage. The thesis provides guidelines for conducting reproducible performance benchmarks and resource utilization experiments.
- **Artifact Contribution:** The `io_uring` integration in Redis AOF is developed as an open-source project, available to the research community at the following URL: https://github.com/daraccrafter/Thesis-Redis-IO_Uring.git.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

1.7 Thesis Structure

A high-level overview of the problem of modern data persistence, the arising issues, and their potential solutions are provided in this chapter. Chapter 2 delves into the background knowledge of Redis data persistence, detailing the AOF and RDB mechanisms. Chapters 3 and 4 present the design and implementation of Redis AOF data persistence with `io_uring` integration. Next, Chapter 5 reports on the results and evaluation, covering all benchmarks. Lastly, Chapter 6 contextualizes the work within related research.

2

Background on Redis and `io_uring`

Redis is an in-memory key-value store primarily used as a cache or a database. One notable feature of Redis is its robust data persistence mechanisms. These mechanisms ensure that the data remains intact and can be recovered even in the event of a system failure, or shutdown. Understanding the different persistence methods used by Redis is crucial for optimizing its performance. In this section, we will explore the two primary persistence mechanisms employed by Redis: RDB (Redis Database) and AOF (Append-Only File). By examining their advantages and limitations, we aim to provide a comprehensive comparison that highlights their suitability for different use cases and their impact on performance and data integrity.

2.1 RDB

RDB (Redis Database) is a persistence mechanism in Redis that creates point-in-time snapshots of the in-memory database. This method allows Redis to save the state of the database at specific intervals. RDB is typically used in scenarios where fast startups are crucial, and occasional data loss is acceptable. For example, it is ideal for environments where read-heavy operations dominate and where data can be reconstructed if lost. RDB offers several advantages that make it suitable for specific use cases (8):

- **Fast Restarts:** RDB files are compact and can be quickly loaded into memory, providing a fast way to restart Redis instances. ✓
- **Portability:** RDB files can be easily copied to create backups or to migrate data across different servers. ✓

2. BACKGROUND ON REDIS AND IO_URING

- **Increasing Throughput:** RDB optimizes Redis performance by offloading persistence tasks to a forked child process. This ensures that the main Redis process remains focused on handling client requests without performing any disk I/O operations. ✓

However, RDB also has some limitations that need to be considered (8):

- **Data Loss Risk:** Since snapshots are taken at intervals, any changes made between snapshots are lost if a failure occurs.
- **Infrequent Updates:** If snapshots are not taken frequently, the restored data may be outdated, reflecting the state of the database at the time of the last snapshot. This becomes problematic if a system failure occurs before the next snapshot, as the in-memory datastore will be reconstructed from the most recent snapshot, potentially containing stale data.

RDB snapshots are created using the **SAVE** or **BGSAVE** commands (8):

- **SAVE:** This command blocks the main Redis thread, which writes the snapshot to disk. While it ensures consistency, it can impact performance, especially for large datasets.
- **BGSAVE:** This command forks a child process to create the snapshot, allowing the main Redis thread to continue processing commands. This approach minimizes disruption to Redis's in-memory processing.

RDB snapshots can be configured in the Redis configuration file (`redis.conf`) using the **save** directive. This directive specifies the intervals at which snapshots should be taken. For example:

```
save 900 1
save 300 10
save 60 10000
```

These lines indicate that a snapshot will be taken if:

- At least one key has changed in 900 seconds (15 minutes).
- At least 10 keys have changed in 300 seconds (5 minutes).
- At least 10,000 keys have changed in 60 seconds (1 minute).

In the default Redis configuration, RDB snapshots are triggered based on the number of key changes within specified intervals. By default, Redis will save the database under the following conditions:

- **After 3600 seconds (1 hour) if at least 1 change has been performed.**
- **After 300 seconds (5 minutes) if at least 100 changes have been performed.**
- **After 60 seconds if at least 10,000 changes have been performed.**

2.2 AOF

AOF (Append Only File) is another persistence mechanism in Redis that logs every *write* operation received by the server to an AOF file. Similar to a Write-Ahead Log (WAL), AOF ensures that the database state can be reconstructed by replaying the logged commands from a log file. However, unlike WAL, data is set in memory first and then written to the disk. AOF offers several advantages that make it suitable for specific use cases (8):

- **High Durability:** By recording every *write* operation, AOF ensures a higher level of data durability, resulting in a lower risk of data loss compared to RDB.
- **Flexibility:** The `appendfsync` configuration options allow users to choose between performance and durability based on their specific requirements.
- **Data Consistency:** Since AOF logs every operation, it ensures that the database remains consistent, which is not achieved by RDB, even after unexpected shutdowns.

However, AOF falls short regarding the following aspects (8):

- **Performance Impact:** Logging every *write* operation can impact performance, especially when `appendfsync` is set to `always`, as it requires frequent disk I/O.
- **Larger File Size:** AOF files are typically larger than RDB snapshots because they log every *write* command (such as SET and INCR), leading to increased disk space usage. Although the growing file is reconstructed by the AOF rewrite process, as explained in Section 2.2.2, it usually remains larger than snapshot files.
- **Startup Time:** Loading the AOF file during Redis startup takes longer compared to RDB snapshots, as Redis needs to replay all logged commands to reconstruct the database state.

2. BACKGROUND ON REDIS AND IO_URING

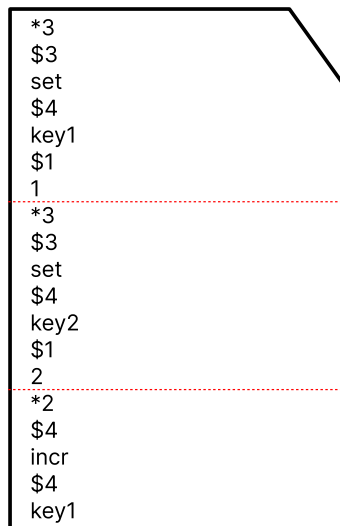
AOF persistence is configured using the **appendonly** directive in the Redis configuration file (`redis.conf`):

appendonly yes

This setting enables AOF persistence, and Redis will start logging all *write* operations to the AOF file.

AOF is typically used in scenarios where data durability and consistency are crucial, and performance can be optimized through configuration. For example, it is ideal for environments where *write*-heavy operations are common and data loss is unacceptable.

2.2.1 AOF Log



```
*3
$3
set
$4
key1
$1
1
-----
*3
$3
set
$4
key2
$1
2
-----
*2
$4
incr
$4
key1
```

Figure 2.1: AOF log structure

Figure 2.1 illustrates the structure of data within the AOF log. Each red dashed line in the figure indicates the separation of commands issued by the client. The AOF log in the image shows three Redis commands, separated by dashed lines. The AOF file maintains the order of issued commands: the first command is **SET key1 1**, which appears first in the file; the second command is **SET key2 2**, followed by the third command, **INCR key1**.

Each command is prefixed with an asterisk (*) indicating the number of arguments, followed by the arguments themselves, each prefixed by a dollar sign (\$) denoting their length. For example, in the first command, the length is specified as 3 because **SET** has three characters (9).

2.2.2 AOF Rewrite

The AOF rewrite process in Redis is implemented to address the inefficiencies associated with the growth of the AOF log over time. As the AOF log expands due to the continuous logging of write operations, it can lead to excessive log sizes and inefficiencies in both storage and recovery times. The rewrite process optimizes the AOF log by consolidating only the essential commands necessary to reconstruct the current state of the dataset, thereby eliminating redundant operations and reducing the overall log size (?).

For instance, consider a sequence of commands such as `SET key1 "value1"` followed by `SET key2 "value2"`, `DEL key1`, and subsequently `SET key2 "value2_modified"`. In this case, the initial `SET key1 "value1"` command becomes redundant once `DEL key1` is executed, and the original `SET key2 "value2"` command is overridden by `SET key2 "value2_modified"`. During an AOF rewrite, these commands would be optimized to include only `DEL key1` and `SET key2 "value2_modified"`, effectively reducing the AOF log's size while preserving the integrity of the current dataset.

By default, Redis includes an RDB snapshot in its rewrite process, adding it to the AOF manifest to serve as a preamble that optimizes data recovery. This approach combines the benefits of RDB snapshots with AOF logging. The rewrite process begins with Redis forking a child process to create a new AOF log. This child process generates a binary RDB snapshot of the current in-memory dataset, which is more compact and faster to load than replaying the entire command log. Concurrently, the main Redis process continues to log any new write operations, ensuring they are recorded in both the existing AOF log and a buffer. Once the new AOF log, containing the RDB snapshot followed by any additional commands, is complete, it atomically replaces the old AOF log.

2.2.3 Loading Data in AOF

When Redis operates in AOF persistence mode and encounters both an RDB snapshot and an AOF log within the same directory (result of an AOF rewrite), it follows a structured procedure to accurately reconstruct the dataset.

Redis begins by referencing the AOF manifest within the AOF directory. This manifest contains metadata about the contents, including whether the directory holds an RDB snapshot, AOF logs, or both. The AOF manifest guides Redis in determining the correct sequence for loading the data.

If the AOF manifest indicates the presence of an RDB snapshot, Redis will load this snapshot first. The RDB snapshot is a representation of the database at a specific point in

2. BACKGROUND ON REDIS AND IO_URING

time, allowing Redis to rapidly restore the in-memory data structures to their state at the time the snapshot was taken. This process is significantly faster than replaying a series of commands from the AOF log.

After successfully loading the RDB snapshot, Redis checks the AOF manifest for any subsequent AOF logs that need to be applied. These logs contain the incremental changes made to the dataset after the RDB snapshot was taken. Redis replays these logs in sequence, ensuring that the dataset is updated to reflect the most recent state.

2.2.4 Redis AOF Flow of Execution

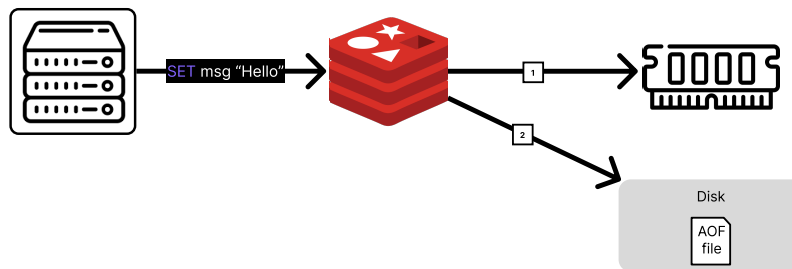


Figure 2.2: AOF Flow of Execution

Figure 2.2 illustrates the progression of a command through Redis, with the numbered arrows indicating the actual sequence of operations. When a server issues a command, the Redis server first updates the key in memory. Following this, the command is written to disk, involving several steps. Initially, the command is saved into a buffer that can contain multiple operations. Redis accumulates these operations in the buffer until it decides to write them to disk, typically every millisecond. The timing of an *fsync* operation depends on the appendfsync configuration.

Buffer
or .

As depicted in Figure 2.3, Redis flushes the kernel buffer differently based on the `appendfsync` configuration (8):

- **`appendfsync always`:** Ensures that every write operation is immediately flushed to the AOF file, providing maximum data durability at the cost of performance.
- **`appendfsync everysec`:** Flushes the AOF buffer to disk every second. This setting offers a good balance between performance and durability and is the default configuration. In this mode, Redis writes commands to the AOF file and schedules an **`fsync`** operation in a background thread every second. If an **`fsync`** operation is still in progress, Redis may delay subsequent writes to batch multiple operations together,

✓

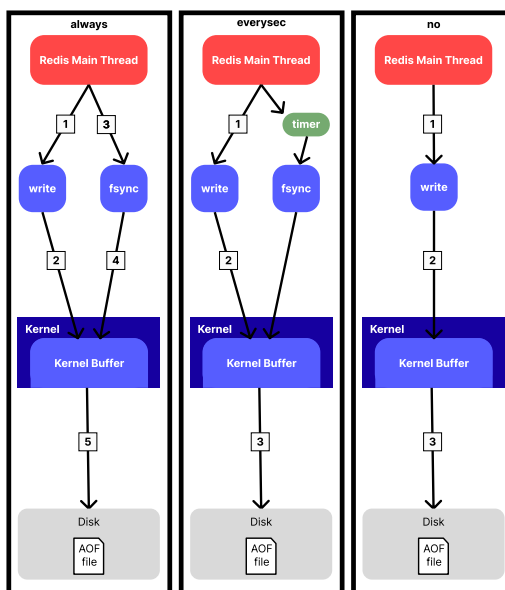


Figure 2.3: Illustration of the AOF persistence mechanism depending on `appendfsync`

which reduces the total number of write operations. This batching is especially beneficial when disk I/O is under heavy load, as it allows Redis to group multiple commands into a single write, improving efficiency and reducing I/O overhead.

- **`appendfsync no`:** Relies on the operating system to flush the AOF buffer to disk, providing the best performance but with a higher risk of data loss. In this mode, every command is written to the AOF file immediately without waiting for an `fsync` or any batching mechanism, leading to more frequent write operations.

When the AOF log grows too large, Redis initiates an AOF rewrite as explained in section 2.2.2.

2.3 io_uring

`io_uring` is an advanced I/O interface in the Linux kernel that aims to provide highly efficient and low-latency asynchronous I/O operations (10).

Figure 2.4 illustrates the architecture of `io_uring`. The numbered arrows do not strictly indicate the order of execution. The figure depicts two primary components: the Submission Queue (SQ) and the Completion Queue (CQ).

The Submission Queue (SQ) is a circular buffer where user space applications submit I/O requests. Each request is encapsulated in a Submission Queue Entry (SQE). This

2. BACKGROUND ON REDIS AND IO_URING

architecture allows multiple SQEs to be batched before notifying the kernel, which significantly reduces the overhead associated with system calls. The Completion Queue (CQ) is a circular buffer where the results of the processed I/O requests are stored. Each result is represented by a Completion Queue Entry (CQE) (11).

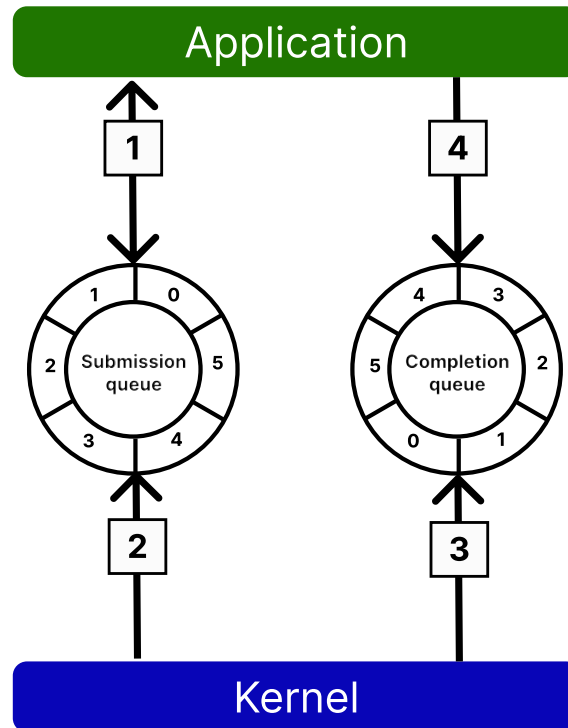


Figure 2.4: System Architecture io_uring

The steps illustrated in Figure 2.4 are as follows:

1. **Application Submits an SQE:** The application obtains an SQE from the Submission Queue (SQ) and configures it for a specific system call, such as *write* or *fsync*. This involves setting up the SQE with the necessary parameters, including file descriptors, buffer locations, and offsets. By batching multiple SQEs before submission, the application minimizes the frequency of system calls. ✓
2. **Kernel Processes the SQE:** Once the SQE is added to the Submission Queue, the kernel reads the SQE and processes the corresponding I/O operation. The kernel's role is to execute the requested operation, such as writing data to a file or syncing the file system.
3. **Kernel Produces a CQE:** After processing the SQE, the kernel generates a Completion Queue Entry (CQE) and places it in the Completion Queue (CQ). The CQE ✓

contains the results of the I/O operation, such as the number of bytes successfully written or an error code if the operation failed.

4. **Application Reads and Handles the CQE:** The application reads the CQE from the Completion Queue to retrieve the outcome of the I/O operation. This step involves interpreting the results, handling any errors, and performing subsequent actions based on the completion status.

io_uring advantages over POSIX I/O:

- **Reduced System Call Overhead:** io_uring significantly reduces the overhead associated with system calls by batching multiple I/O operations into a single system call, thus enhancing efficiency.
- **Asynchronous I/O:** io_uring supports asynchronous I/O operations, allowing applications to perform non-blocking I/O operations without waiting for the completion of each operation.


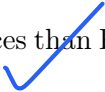
Additionally, io_uring provides two different polling modes, unlike native Linux asynchronous I/O (AIO):

- **SQ_POLL:** Enabled by setting `IORING_SETUP_SQPOLL` when creating the ring. This creates a thread that runs in kernel space, polling the submission queue (SQ) ring for new completions to submit. This eliminates the submission overhead from the application and allows it to perform I/O operations without invoking a syscall.
- **IO_POLL:** Enabled by setting `IORING_SETUP_IOPOLL` when creating the ring. This enables I/O to files or block devices without triggering interrupts. The application, when performing a wait-for-events `io_uring_enter` syscall, will actively poll for completions on the target device. This reduces overhead for high IOPS (Input/Output Operations Per Second) applications and decreases latency in general.

The primary disadvantages of io_uring include:


- **Compatibility:** io_uring is a relatively recent API, and thus, not all systems or applications are compatible with it. Its functionality requires a Linux kernel version of 5.1 or higher.
- **Limited Documentation:** As an emerging technology, io_uring currently has fewer resources, documentation, and community support compared to more established I/O interfaces.

2. BACKGROUND ON REDIS AND IO_URING

- **Higher Resource Consumption:** `io_uring` consumes more resources than POSIX I/O. *can*  

2.4 Liburing

Using `io_uring` directly can be complex and error-prone due to the low-level details involved. Liburing is a library that provides a simplified interface to the Linux kernel's `io_uring` subsystem. It abstracts the complexities of `io_uring` and offers a more user-friendly API, making it easier to harness the performance benefits of `io_uring` without managing the intricate details of the underlying data structures (12).

A great example of how liburing facilitates `io_uring` development is through its preparation functions, such as `io_uring_prep_write` (13), and `io_uring_prep_fsync` (14). These functions take the arguments of the classic system calls they are associated with, along with an additional pointer to a submission queue entry (SQE). They then configure the SQE to the desired system call with all the necessary arguments we would traditionally use in POSIX I/O. 

3

Design of AOFUring

In this chapter, we present the integration design of `io_uring` with Redis AOF persistence. The primary objective is to deliver a thorough analysis of the design, highlighting its fundamental components.

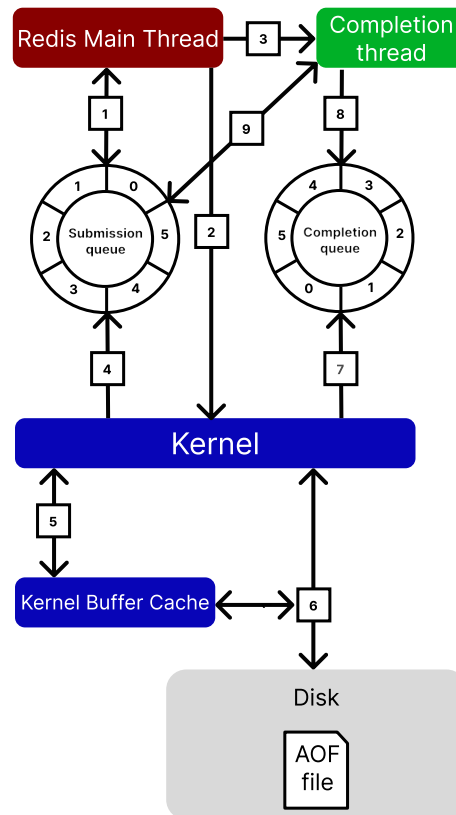


Figure 3.1: High-level overview of AOFUring

Figure 3.1 illustrates the design of AOFUring. Arrows annotated with sequential num-

3. DESIGN OF AOFURING

bers represent essential interactions within the system that do not necessarily follow a strict order of execution.

3.1 Submission of SQEs

The submission process, orchestrated by the **main Redis thread**, is pivotal in enhancing data durability. The prompt submission of *write* and *fsync* SQEs is essential for minimizing the risk of data loss. The earlier these SQEs are placed into the Submission Queue, the sooner the **Kernel** can process them. Although this approach to maximizing data durability does not guarantee that the design is more durable than the original implementation, it significantly improves the durability in case of a failure compared to batching and submitting these SQEs all at once.

is this
the
approach
of the
thesis

In Figure 3.1, the **main Redis thread** first prepares a *write* SQE, followed by an *fsync* SQE, which are then submitted to the Submission Queue (1). Furthermore, the **Kernel** thread is notified (2) that SQEs are pending in the queue, as detailed in Section 2.3. Upon notifying the **Kernel** (2) and the successful submission of SQEs, two important actions occur: the **Completion Thread** is spawned (3), and the **Kernel** begins processing the SQEs (4). Both of these processes are explained in detail in the following sections 3.2 and 3.3.

A critical limitation of this design is that asynchronous *fsyncs* are impractical. Consider a scenario where two Redis **SET** commands (15) are issued, both with the same key but with different values, say 1 and 2. Due to the asynchronous nature of *fsyncs*, there is no guarantee that the data flushed to the journal (AOF log) will appear in the correct order, thereby compromising the system's data integrity. The following subsection 3.1.1 addresses this issue.

✓ good.

3.1.1 SQE Setup and Linking

io_uring provides a mechanism for linking SQEs through the `IOSQE_IO_LINK` flag. This flag ensures that an SQE is processed in the order it was submitted by linking it to the subsequent SQE (16). Consequently, the **Kernel** waits for the *fsync* of a previous *write* to complete before processing the next *write* and its corresponding *fsync*. By creating a chain of SQEs, the **Kernel** processes them sequentially, ensuring that they are written to the journal in the correct order.

However, linking SQEs introduces certain drawbacks. This approach reduces the performance of the **Kernel** thread responsible for processing these SQEs, as the *fsync* operation,

Single
header
No,
3.1.2

✓
✓

which previously blocked the **main thread**, now blocks the **Kernel** thread. Additionally, a significant challenge with linking is that if an **SQE** fails (i.e., the system call returns an error), all subsequent **SQEs** in the chain will be discarded. This scenario requires re-queuing the remaining **SQEs**, potentially leading to a slowdown in the system's overall performance. ✓

3.2 Processing of SQEs

Although this aspect is not directly part of the AOFUring design, it is essential to understand how **io_uring** and the **Kernel** operate to fully grasp the overall system architecture. As detailed in Section 3.1, upon the successful submission of **SQEs**, the **Kernel** begins processing these entries ([4]). For *write* **SQEs**, the **Kernel** handles them in the same manner as standard *write* system calls, transferring the data to the **Kernel** buffer cache ([5]). Afterward, the **Kernel** generates a Completion Queue Entry (CQE) ([7]) containing the result code of the operation. Likewise, *fsync* **SQEs** are processed as regular *fsync* system calls, triggering a flush of the **Kernel** buffer cache to the disk ([6]) and into the AOF log. The **Kernel** then produces a CQE with the result code for the *fsync* operation and places it in the Completion Queue ([7]).

3.3 Completion of CQEs

Once the **Kernel** is notified ([2]) and the **SQEs** are successfully submitted, as outlined in Section 3.1, an additional detached thread is spawned to manage the CQEs and their corresponding results. It is important to note that only a single Completion Thread exists throughout the lifetime of Redis; the main Redis thread will not spawn additional completion threads while one is already active.

The Completion Thread operates in a loop, consuming CQEs in batches ([8]). For each CQE, the Completion Thread determines whether it originated from an *fsync* or *write* operation and processes the associated result codes accordingly. In the case of a successful *write*, the Completion Thread simply frees the memory buffer associated with that operation. Since the kernel is configured not to generate CQEs for successful *fsync* SQEs, as discussed in Section 3.1.1, there is no need to handle those cases.

A failed *write* or *fsync* typically occurs due to a closed file descriptor during the Redis rewrite process (as detailed in section 2.2.2). As discussed previously in section 3.1.1, the submission queue is essentially a chain of linked operations. If one of these operations fails, it results in the entire chain being dropped. If the AOF log increment changed, the

3.1.1 doesn't discuss this -

?? where ?

what's this ?

3. DESIGN OF AOFURING

2o **Completion Thread** does not requeue these operations because, as explained in Section 2.2.2, Redis has already created a snapshot of the in-memory state. However, if the **AOF log increment** has not changed, the operations will be requeued. When this happens, the **Completion Thread** detects the failure through the error **CQEs** and subsequently requeues the entire chain of operations ([9]) to the newly opened AOF log file. This requeueing process generates a new chain of linked **SQEs** that mirrors the original sequence of operations. Although this approach might seem excessive, as it involves reprocessing potentially a large number of operations, it is necessary to maintain the integrity of the data. ✓

Additionally, unlike the default AOF, AOFUring utilizes a file descriptor without the `O_APPEND` flag. Instead, each write CQE includes a specific write offset, for handling partial writes (17). Although partial writes are rare, as discussed in (17), this functionality was retained in AOFUring to faithfully replicate the original AOF implementation. Using a file descriptor without the `O_APPEND` flag enables control over where the remaining data is written by specifying the desired offset. This allows us to write the unwritten portion of the buffer to a specific location within the log.

As the process continues, if the Completion Thread does not detect any **CQEs**, it enters a busy-wait state for approximately 10 seconds. During this interval, should any **CQEs** be detected in the queue, the timer is reset, and the thread resumes its processing duties. However, if the timer elapses without the appearance of new **CQEs**, the thread terminates and signals the main Redis thread that it is no longer active. Thus, upon the subsequent successful submission of **SQEs** (as discussed in Section 3.1), the Completion Thread will be reactivated. ✓

3.4 Data Durability of AOFUring

Data durability tests are inherently challenging to conduct, particularly when it comes to accurately simulating power failures and other system interruptions. Therefore, we hypothesize a power failure scenario and analyze how **AOFUring** is expected to behave based on its design.

In the event of a power failure, the primary concern is the extent of potential data loss. Unlike **AOF (always)**, which synchronizes every write to disk immediately, **AOFUring** does not block *fsync* calls, meaning data loss is expected. Any writes still in the Submission Queue at the time of failure will be lost. The actual amount of data lost is difficult to estimate without a proper test and depends heavily on the system load. ✓ ok.

3.4 Data Durability of AOFUring

A specific durability issue arises from the asynchronous nature of **AOFUring**. By utilizing `io_uring` for non-blocking writes, there is a possibility that the server might acknowledge a write to the client before it is persisted. In such a case, if a crash occurs while the kernel is still handling the queued *writes* and *fsyncs*, the application could mistakenly assume that the data has been safely persisted, when in reality, it may be lost. This potential issue is not unique to **AOFUring**; **AOF (everysec)** could also encounter a similar problem due to the delayed synchronization of data to disk.

Then what's the
durability guarantee of
AOFUring?
Not clear.
But Fine.

4

Implementation of AOFUring

In this section, we explore the implementation of the AOFUring design as presented in Chapter 3. This chapter details the tools, technologies, and methodologies used to bring the design to life, while also addressing the challenges encountered during the development process. The discussion highlights crucial aspects such as the choice of programming language, thread management, I/O operations, and the specific complexities of integrating `io_uring` with Redis AOF persistence. ✓

4.1 Development Environment and Tools

The implementation of AOFUring was conducted in C, utilizing a forked version of the Redis project. Compilation was handled using the GNU Compiler Collection (GCC) (18), with support for `io_uring` integrated via the `Liburing 2.6` library. This library is included within the Redis `deps/` directory and is compiled alongside the project. A kernel version of at least 5.1 is required to ensure compatibility with the necessary `io_uring` system calls. ✓

The implementation utilized the ext4 filesystem. Research has shown that different filesystems can significantly impact performance, particularly in I/O-intensive applications like Redis. For instance, a conference paper (19) highlights that ext4 may introduce substantial I/O amplification due to its metadata handling, which can degrade performance under heavy workloads. While Redis is not LSM-based, similar concerns regarding filesystem I/O patterns and fragmentation should be considered. ✓

Debugging and troubleshooting were performed using the GNU Project Debugger (GDB) (20), which was instrumental in identifying and resolving issues during implementation. For performance evaluation, the built-in benchmarking tool, `redis-benchmark` (21), was employed. This tool generated workloads and provided key performance metrics, including ✓

requests per second (RPS) and latency, for assessing the performance of different Redis persistence mechanisms, including AOFUring.

4.2 AOFUring Configuration

In the implementation of AOFUring, specific configuration options were employed to tailor the behavior of `liburing` to the system's needs. These configurations can be found in the `redis.conf` file in the root of the project.

1. **appendonly-liburing:** This option enables the use of `liburing` for AOF operations. When set to 'yes', Redis utilizes the `io_uring` interface for handling AOF tasks. ✓

2. **liburing-queue-depth:** This setting controls the depth of the `liburing` queue, determining how many I/O operations can be queued simultaneously. It allows users to specify different levels, such as 'xs', 's', 'm', 'l', 'xl', and 'xxl', corresponding to 1024, 2048, 4096, 8192, 16384, and 32768 operations, respectively. In the development and testing phases, the 'xl' queue depth is utilized. ✓

3. **liburing-retry-count:** This configuration sets the number of retries for obtaining a Submission Queue Entry (SQE) in `liburing`. The retry count can be adjusted with levels 'xs', 's', 'm', 'l', 'xl', and 'xxl', representing 3, 10, 50, 100, 500, and 1000 retries, respectively. Under high concurrent loads (e.g., 1,000,000 requests over 100 concurrent connections), the queue depth might not suffice, potentially causing dropped SQEs. This parameter enhances robustness by permitting multiple attempts to secure an SQE. During development and testing, the 'xl' retry count is utilized. ✓

4. **liburing-sqpoll:** This option enables the SQPOLL mode in `liburing` (22), where the kernel continuously polls the Submission Queue. In this mode, the kernel spawns an additional thread dedicated to polling the Submission Queue. When utilizing this option, files are registered using the `io_uring_register` function provided by `liburing` (23). Fixing and registering files or user buffers allows the kernel to maintain long-term references to internal data structures or establish long-term mappings of application memory, significantly reducing the overhead for each I/O operation (23). This configuration did not yield significant performance improvements; throughput and latency remained unchanged, while CPU usage increased due to the additional kernel thread. It was primarily an experimental setup used during implementation. 2

5. **correct-test:** This option is set to 'no' by default. When enabled (setting it to 'yes'), it triggers a log entry when the final *fsync* completes. Since the Redis server may still be persisting requests even after all of them have been committed to memory. This

what is the final fsync?

4. IMPLEMENTATION OF AOFURING

entry is then used in the correctness test to verify that all data has been safely persisted before the server is shut down.

6. correct-test-reqnum: The correctness test uses this option to set the number of requests issued during the test, therefor defining what is the completing *fsync*.

4.3 AOFUring Ring Initialization

The `io_uring` ring is initialized at startup if the `appendonly-liburing` configuration option (Section 4.2) in `redis.conf` is set to `yes`. This configuration ensures that Redis utilizes `liburing` for its append-only file (AOF) operations.

During the initialization process, the ring is set up with the specified queue depth, as defined by the `liburing-queue-depth` configuration option. The initialization is also integrated into the server state so that the `io_uring` ring can be shared across various components of the Redis server. Optionally, by using the `liburing-sqpoll` flag, the ring can be initialized in SQPOL mode.

4.4 Execution Flow of AOFUring

When a command is received by the Redis server, it first enters the event loop, where the server updates the in-memory dataset as described in Section 2.2.4. The server then prepares to persist this command to disk using the Append-Only File (AOF) mechanism.

This is accomplished by adding the command data to the `aof_buf` buffer. ✓

Simultaneously, the Redis server's `serverCron` function, which manages scheduled tasks, triggers the `flushAppendOnly` function every millisecond. The `flushAppendOnly` function checks whether `aof_buf` contains data. If data is present, it calls the `aofWriteUring` and `aofFsyncUring` functions (outlined in Sections 4.5.1 and 4.5.2), replacing the traditional `write()` and `fdatsync()` system calls with their `io_uring`-based counterparts. These functions configure the necessary SQE parameters for the `write` and `fdatsync` operations.

Once the Write and `Fsync` Submission Queue Entries (SQEs) are prepared, they are submitted to ensure the kernel processes these SQEs promptly. This submission process also initiates a detached completion thread, `process_completions`, which manages Completion Queue Entries (CQEs) and handles any errors, as further described in Section 4.6.

What mode is this?
Then this is for =no flag?

4.5 Submitting Submission Queue Entries

Whenever the Redis `flushAppendOnly` function is invoked, as mentioned in the previous section, it prepares the *write* and *fsync* SQEs using custom functions that replace the traditional `write()` and `fdatasync()` system calls. These SQEs are then submitted using the `io_uring_submit` function. The specifics of these custom functions will be further discussed in Sections 4.5.1 and 4.5.2. Additionally, the amount of data written is added to the accumulating write offset. If the AOF log increments due to an AOF rewrite (as described in section 2.2.2), the accumulated write offset is reset to 0.

4.5.1 AOFUring Write Function

The `aofWriteUring` function is a key component of the AOFUring implementation, designed to handle write operations using the `io_uring` interface. Unlike traditional *write* functions, `aofWriteUring` requires additional parameters beyond the standard file descriptor and data buffer. These parameters include a pointer to the `io_uring` ring structure, the file descriptor for the target file, the maximum number of retries as specified by the `liburing-retry-count` configuration, the offset in the AOF log, the current AOF log increment, and the `sqpoll` flag. Unlike the traditional AOF implementation, `aofWriteUring` uses a file descriptor without the `O_APPEND` flag, making the write offset necessary. This offset determines where data should be written, particularly in the event of a partial write (as discussed in section 3.3).

The function begins with a setup phase, where necessary variables are initialized. This involves copying the data into a temporary buffer and setting up an `OperationData` structure. The `OperationData` structure encapsulates key details of the operation, including the operation type (set to `WRITE_URING` for write operations), the data length (`len`), the buffer pointer (`buf_ptr`), the write offset (`write_offset`), and the current AOF log increment. The function then attempts to retrieve an available Submission Queue Entry (SQE) from the `io_uring` submission queue. If the function fails to secure an SQE after the number of retries specified by the `liburing-retry-count` configuration in Section 4.2, it returns an error, indicating that the operation could not proceed and the SQE is dropped.

Once an SQE is obtained, the function prepares it for the *write* operation by specifying the target file descriptor, the data buffer, and the file offset. Notably, the offset is used only in cases of partial writes, as the file descriptor is typically opened with the `O_APPEND` flag, rendering the offset largely irrelevant (as described in Section 3.3). Following this, appropriate flags are set on the SQE based on the provided configuration arguments. For

Contradict
??

4. IMPLEMENTATION OF AOFURING

example, if the `liburing-sqpoll` configuration is active, the file descriptor is fixed using `IOSQE_FIXED_FILE`. Additionally, a link flag (`IOSQE_IO_LINK`) is set to chain the current SQE to the subsequent *fsync* SQE, ensuring that the *write* operation is linked to the next operation.

After setting the necessary flags, the `OperationData` structure is associated with the SQE through the `user_data` field, which essentially takes in a pointer to the allocated structure. This association allows the completion thread to access and process the operation details once the *write* is completed. Finally, the function returns the length of the data written, indicating the success of the operation and completing the write process.

4.5.2 AOFUring Fsync Function

The `aofFsyncUring` function is designed to execute *fsync* operations using the `io_uring` interface. This function requires several parameters to manage the *fsync* operation effectively. These parameters include a pointer to the `io_uring` ring structure, the file descriptor to which the *fsync* operation will be applied, the maximum number of retries specified by the `liburing-retry-count` configuration (detailed in Section 4.2), the `sqpoll` flag, and the current AOF log increment. ✓

The execution of the `aofFsyncUring` function begins with a setup phase, where the necessary variables are initialized. This setup involves preparing an `OperationData` structure, which stores essential details about the *fsync* operation. The `OperationData` structure contains the following fields: the operation type (set to `FSYNC_URING` to distinguish it from write operations), the data length (set to 0 for *fsync* operations), the current AOF log increment, and the buffer pointer (set to `NULL` as no data buffer is needed for *fsync*). Unlike the write function discussed in Section 4.5.1, the *fsync* function only sets the type of operation to `FSYNC_URING` so that the completion thread can differentiate between *write* and *fsync* CQEs. Following the setup, the function attempts to retrieve an available Submission Queue Entry (SQE) from the `io_uring` submission queue. If the function fails to obtain an SQE after the specified number of retries, it returns an error, signaling that the *fsync* operation could not proceed.

Upon successfully retrieving an SQE, the function prepares it for the *fsync* operation. This preparation involves specifying the target file descriptor and setting the `IORING_FSYNC_DATASYNC` flag, effectively making the *fsync* operation an *fdatsync*. The key difference between *fsync* and *fdatsync* is that while *fsync* flushes both the file data and metadata to disk, *fdatsync* only ensures that the file data and minimal metadata required to retrieve that data are written, making it generally faster (24). The function then sets the appropriate flags on the

4.6 Processing Completion Queue Entries

SQE based on the configuration arguments. If the `sqpoll` flag is enabled, the file descriptor is fixed using `IOSQE_FIXED_FILE`. Additionally, a link flag (`IOSQE_IO_LINK`) is set to chain the current SQE to the subsequent *write* SQE, creating a chain of operations (as discussed in section 3.1.1).

The next step involves associating the `OperationData` structure with the SQE through the `user_data` field. This association is crucial as it allows the completion thread to access the necessary information once the *fsync* operation is completed. The function concludes by returning a value of 0, indicating the successful execution of the *fsync* operation.

4.6 Processing Completion Queue Entries

The `process_completions` function serves as a dedicated detached thread within the AOFUring implementation, tasked with handling Completion Queue Entries (CQEs) that the kernel generates after processing Submission Queue Entries (SQEs). This function manages cases of errors and frees memory associated with buffer and passed structures. ✓

Upon initialization, the function receives a `CompletionThreadArgs` structure, which encapsulates all the necessary components for its operation. This structure includes a pointer to the `io_uring` ring structure, the batch size of CQEs to be processed (calculated as one-tenth of the queue depth set in the configuration, as detailed in Section 4.2), a pointer to the current file descriptors open for the AOF log, a pointer to an integer tracking the AOF log file increment, a `running` flag that controls the thread's execution, and a pointer to a logging function for error reporting in the main Redis thread. Additionally, it includes a pointer to a mutex lock, which is used to secure the `running` flag before any modifications. ✓

The thread is initiated based on the state of the `running` flag, which is checked after each successful submission of *write* and *fsync* SQEs (as discussed in Section 4.5). A mutex lock is employed to ensure proper synchronization, particularly when the thread is already active. The state of the `running` flag is jointly managed by the main Redis thread and the completion thread. Initially set to `false`, the flag is updated to `true` when the completion thread starts, signaling that the thread is now running.

The `process_completions` function operates in a continuous loop, attempting to retrieve a batch of CQEs using `io_uring_peek_batch_cqe`. When CQEs are available, the function processes each entry according to the operation type specified in the associated `OperationData` structure. ✓

For `WRITE_URING` operations, if a *write* operation fails or is only partially completed, and the AOF log increment at the time the command was issued matches the current log

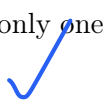
4. IMPLEMENTATION OF AOFURING

increment, the function logs the error and retries the operation with the original arguments by invoking the method detailed in Section 4.5.1. However, if the AOF log increment has changed, indicating that an AOF rewrite has occurred (as discussed in Section 2.2.2), the operation is not requeued, since the in-memory database has already been persisted in a snapshot.

In the event of a partial write, the function adjusts the offset to account for the successfully written data before requeuing the operation. This ensures that only the remaining data is written to the correct location in the file. However, if an AOF rewrite has occurred, the operation is not requeued because the old AOF log file has already been replaced by a snapshot of the in-memory state (discussed in section 2.2.2). Since the snapshot reflects the complete and accurate state of the database at the point of the rewrite, any partial writes to the old log file are rendered irrelevant, preventing any possibility of file corruption.

Regarding `FSYNC_URING` operations, the function assesses the result; if the *fsync* is successful, no further action is taken. However, if the *fsync* fails and the AOF log increment remains unchanged, the operation is requeued using the `fsync` function detailed in section 4.5.2. If the increment has changed, indicating that an AOF rewrite has occurred, the operation is not requeued.

After processing each `CQE`, the function deallocates the associated `OperationData` structure and marks the `CQE` as completed using `io_uring_cqe_seen`. This loop continues as long as there are `CQEs` available in the Completion Queue. If the queue becomes empty, the `process_completions` function enters a busy-wait loop for approximately 10 seconds, periodically checking for new `CQEs`. If the timer expires without new `CQEs`, the thread acquires a mutex lock, sets the `running` flag to `false`, releases the lock, and exits. The thread will restart when new submissions are made. This procedure ensures that only one thread is active at any time.



5

Evaluation

This chapter evaluates our Redis AOF implementation with `io_uring` alongside other Redis persistence modes. The evaluation involves benchmarking to measure the performance of these persistence modes, as well as a data correctness test to ensure proper data persistence. The benchmark primarily focuses on Redis write commands, such as `SET`, which store or modify data in the database. Focusing on write commands is crucial for assessing the underlying performance of writing to the AOF file or creating snapshots. In contrast, the correctness test combines write commands (to persist the data) with read commands (to verify that the data is correctly persisted), such as `GET` (25), which retrieve data from the database.

5.1 Benchmark Environment

The benchmarks were conducted on a fresh AWS Virtual Private Server (VPS) running Ubuntu 24.04 with an ext4 filesystem. The hardware configuration includes an AWS c5a.8xlarge instance featuring an AMD EPYC™ 7R32 processor, 32 vCPUs, and 64GB of DDR4 memory (26). Storage is provided by an Amazon EBS Provisioned General Purpose SSD (gp3) (27). The instance is equipped with two EBS volumes: a 10GB volume used for the root and boot partitions, and a 50GB volume dedicated to running the benchmarks. The underlying filesystem for all tests is ext4, as discussed in section 4.1.

5.2 Benchmarking Process

The benchmarking process is designed to evaluate various Redis persistence modes, including Redis AOF with three `fsync` configurations (always), Redis RDB, and AOFUring.

During the benchmarking process, the following metrics are collected:

5. EVALUATION

- **Requests per second (RPS):** Assessed using the `redis-benchmark` tool, which provides a quantitative measure of the throughput achieved by the Redis server.
- **System calls:** Monitored through `strace`, capturing and analyzing the system calls executed by the Redis process to provide insights into the operational overhead.
- **CPU load:** Gathered using the Python `psutil` library, offering detailed information on CPU utilization throughout the benchmarking period.
- **Memory usage:** Also monitored via the Python `psutil` library, which records the memory consumption associated with the Redis process.
- **Latency:** Measured by the `redis-benchmark` tool, which includes the average, minimum, maximum, p50, and p99 latency.

The RPS (Requests Per Second) and latency metrics are selected to demonstrate the efficiency of different modes in handling commands. RPS serves as the primary performance metric, while latency, which generally correlates with RPS, provides additional context to explain variations in throughput between modes. CPU and memory usage metrics are included to assess the resource intensity of each mode, helping to understand the trade-offs between performance and resource consumption. System call analysis is conducted to further explain the performance differences, offering a deeper understanding of why some modes are faster or slower than others.

The `redis-benchmark` tool serves as the primary workload generator in our benchmarking framework, simulating a high volume of requests to the Redis server. We selected the `SET` (15), `HSET` (28), `LPUSH`(29), and `INCR` (30) commands because they represent commonly used Redis write operations that require logging to the AOF log.

The `SET` command stores a value at a specified key, representing basic key-value storage. `HSET` sets a field in a hash, allowing structured data storage under a single key. `LPUSH` adds an element to the head of a list, useful for implementing queues or stacks. `INCR` atomically increments the value of a key, essential for operations like counters. These commands cover a broad spectrum of typical Redis operations, enabling us to evaluate Redis’s performance in handling various types of workloads. The sizes of the requests for each command, generated by `redis-benchmark`, are as follows: `SET` is 45 bytes, `HSET` is 65 bytes, `LPUSH` is 36 bytes, and `INCR` is 41 bytes.

Each persistence mode is evaluated by running the `redis-benchmark` tool three times. The first run captures the raw performance metrics directly from the `redis-benchmark`

output. The second run focuses on resource utilization, using the `psutil` library in a separate thread to monitor CPU load and memory usage of the `redis-server` process.

Lastly, the third run uses `strace` to capture system calls, including `write`, `fdatasync`, and `io_uring_enter`. For each type of system call, `strace` records the average time per call, the total time taken by each set of system calls (e.g., summing the time for 50 `write` calls), and the overall time spent on system calls by the process and any subprocesses it spawns.

All persistence modes discussed in the following sections were tested using a workload of 4,000,000 requests, distributed across the previously mentioned commands. These requests were executed by 50 concurrent clients, adhering to the default configuration of the `redis-benchmark` tool. Each benchmark was repeated three times, resulting in a total running time of approximately 250 minutes. To ensure consistency, the testing partition was unmounted, reformatted to `ext4`, and remounted after each persistence mode was tested.

5.3 Performance Analysis

This section presents an analysis of throughput and latency across different persistence modes. The focus here is on the performance metrics rather than the data durability.

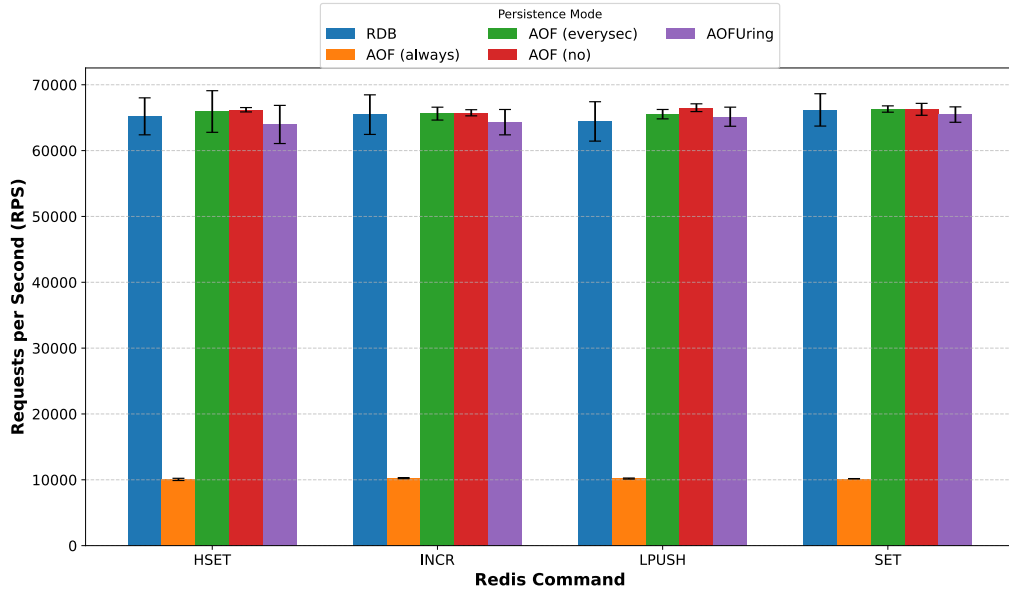


Figure 5.1: Throughput Comparison across Redis Persistence Modes for Various Commands

In Figure 5.1, the x-axis displays the various Redis commands executed during the benchmark, while the y-axis indicates the number of requests per second (RPS) that Redis can

5. EVALUATION

process. This figure compares the throughput performance across different Redis persistence modes for each command. The results represent the average RPS across all benchmark runs, with error bars indicating the corresponding standard deviation.

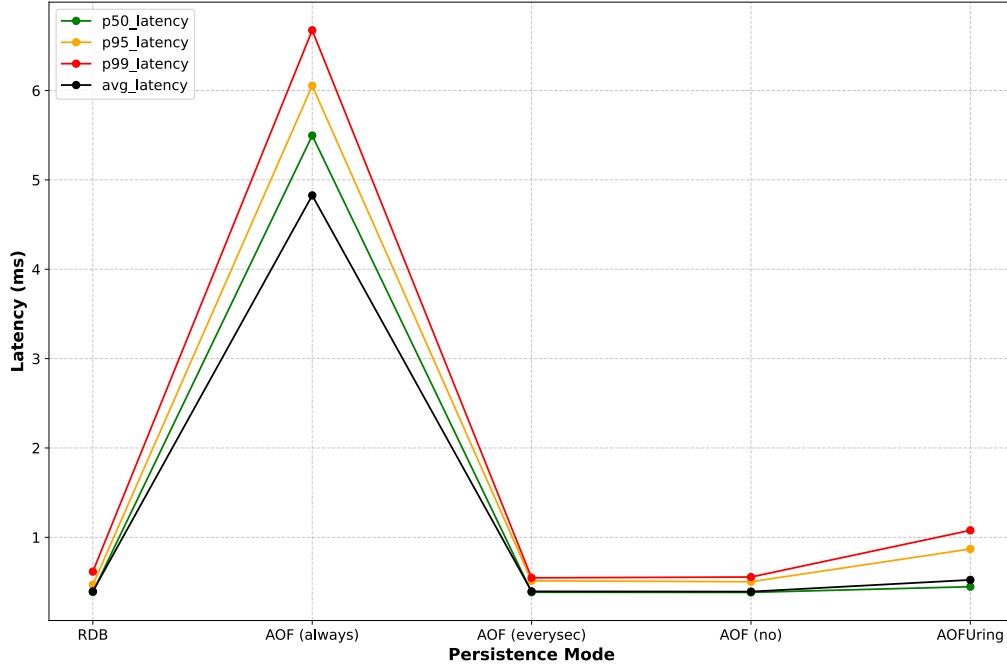


Figure 5.2: Latency Comparison across Redis Persistence Modes

Similarly, in the latency figure 5.2, the x-axis represents the different Redis persistence modes, and the y-axis shows the latency in milliseconds. This figure provides a comparison of latency statistics across the various persistence modes.

The throughput figure 5.1 reveals that the performance of **RDB**, **AOF (everysec)**, and **AOF (no)** is remarkably similar across various tests. The most significant deviation is observed with the **LPUSH** command, where **AOF (everysec)** exhibits approximately 1.5% lower throughput compared to **AOF (no)**. Additionally, **RDB** demonstrates a slight decrease in performance for the same test compared to **AOF (no)**, with a difference of about 3%. When analyzing the latency figure 5.2, it becomes evident that the p99 latency percentiles are nearly identical, with **RDB** recording the highest at 0.616 ms.

The main reason why **AOF (no)** only slightly outperforms **AOF (everysec)**, which is unexpected given that **AOF (everysec)** still makes *fsync* systemcalls, is because **AOF (everysec)** avoids the blocking nature of *fsync*. Instead, it performs an *fsync* operation every second in the background using a separate **BIO (Background I/O)** thread (discussed in section 2.2).

5.3 Performance Analysis

Further inspecting figure 5.1, we can observe that AOFUring is not far behind in terms of throughput. It performs best in the SET test, where it is only 1.1% slower than RDB. However, in the HSET test, it is 3.4% slower compared to AOF (no). Moreover, the latency figure 5.2 indicates that AOFUring performs marginally worse than the other persistence modes at the p99 percentile, with a latency of 1.078 ms. This is likely due to the frequent memory allocations for the write buffer, as discussed in section 4.5.1. ✓

Lastly, it is evident that AOF (always) significantly lags in throughput compared to other persistence modes, being more than 6 times slower than the others. As discussed in section 2.2, AOF (always) performs an *fsync* after every write, which greatly reduces its throughput. When examining latencies in figure 5.2, a noticeable spike is observed across all latency metrics. FIX

Persistence Mode	write	fdatasync	io_uring_enter	total
AOF (always)	80006	80098	0	160104
AOF (everysec)	79971	1349	0	81320
AOF (no)	79979	56	0	80035
RDB	0	20	0	20
AOFUring	0	48	79977	80025

for what time?
per sec?
duration of the whole test?

Table 5.1: System Call Counts by Persistence Mode

Persistence Mode	Syscall	Single Time (ms)	Total Time (s)	Overall Time (s)
RDB	write	0	0	0.015
	fdatasync	1.315	0.001	
	io_uring_enter	0	0	
AOF (always)	write	0.045	3.608	220.085
	fdatasync	2.702	216.477	
	io_uring_enter	0	0	
AOF (everysec)	write	0.045	3.613	6.537
	fdatasync	2.17	2.924	
	io_uring_enter	0	0	
AOF (no)	write	0.04	3.241	3.501
	fdatasync	13.117	0.26	
	io_uring_enter	0	0	
AOFUring	write	0	0	3.156
	fdatasync	2.789	0.1	
	io_uring_enter	0.038	3.059	

Table 5.2: System Call Times by Persistence Mode

5. EVALUATION

Table 5.1 summarizes the total number of system calls traced by `strace` across all subprocesses of the main Redis server process, as well as the main process itself. Each row corresponds to a different persistence mode, and the columns represent the counts for the three specific system calls. The "total" column is the sum of all these system calls for each persistence mode.

Table 5.2 provides insights into the time-related metrics for system calls associated with different persistence modes. The "Single Time (ms)" column indicates the average time it took for a single invocation of each system call, measured in milliseconds. This value represents the average duration of one system call across all processes and subprocesses. The "Total Time (s)" column represents the cumulative time spent on each system call type during the whole test, expressed in seconds. The "Overall Time (s)" column shows the total time, in seconds, spent on all system calls for a given persistence mode.

AOF (always) mode generates the highest total number of system calls, with 160,104 calls, nearly equally split between *write* and *fdatsync*. As shown in Table 5.2, this results in approximately 220 seconds of blocking time, primarily due to the 216.5 seconds spent on *fdatsync* calls. This extensive blocking is the primary reason for the significantly lower throughput and higher latency observed in this mode, compared to others. Despite making the same number of *write* and *fdatsync* calls, the *write* operations block for far less time, averaging only 0.045 ms per call, while *fdatsync* blocks for 2.17 ms per call.

In the **AOF (everysec)** mode, the *fdatsync* operations are handled in a separate process, so the blocking time shown in Table 5.2 is the combined time for both the main and the BIO processes. Because all *fdatsync* calls occur in this separate process, the main event loop is less affected, resulting in throughput and latency similar to the **AOF (no)** mode. This design reduces the blocking impact on the main process, explaining the almost identical performance between these two modes. The **AOF (no)** mode has a similar total number of system calls (80,035), almost all of which are *write* operations. The blocking time is even lower at about 3.5 seconds, with *fdatsync* contributing only 0.26 seconds.

Finally, the **AOFuring** mode is similar, with 80,025 system calls, the majority being *io_uring_enter* calls. The overall blocking time is about 3.16 seconds, with *fdatsync* contributing only 0.1 seconds. The *io_uring_enter* system calls have the lowest single call time at 0.038 ms, as these calls merely notify the kernel.

FIXME:
by what?
How is it
measured?

FIXME
out of
what
runtime?

✓

✓

5.4 Resource Consumption

The resource consumption across different Redis persistence modes varies significantly, particularly in CPU and memory usage. These differences stem from how each mode handles write operations and disk I/O synchronization.

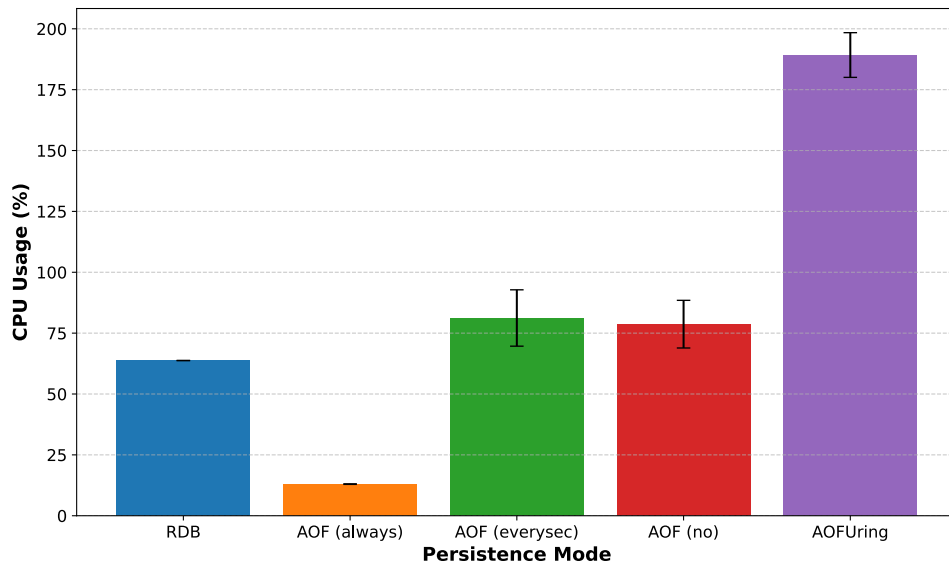


Figure 5.3: CPU Usage Comparison across Different Redis Persistence Modes

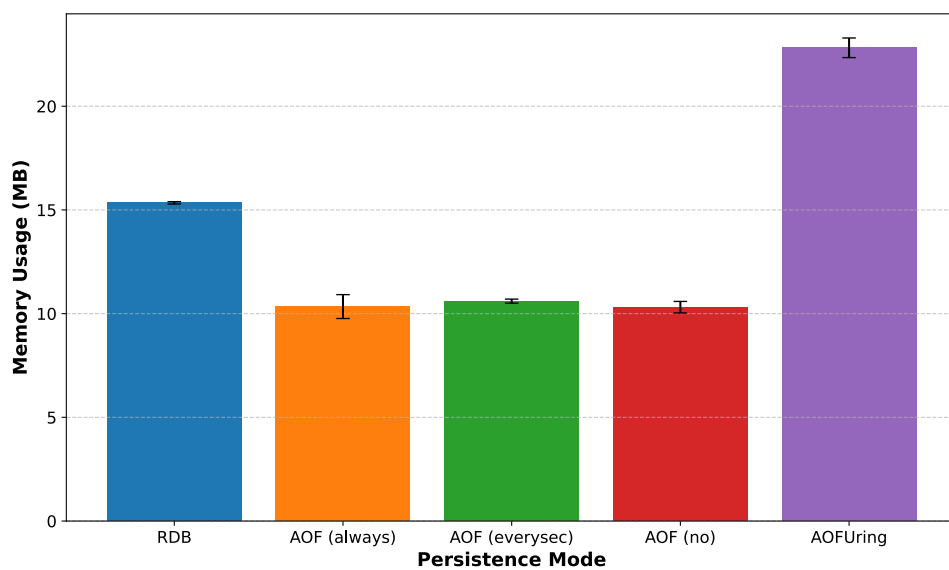


Figure 5.4: Memory Usage Comparison across Different Redis Persistence Modes

5. EVALUATION

Figures 5.3 and 5.4 illustrate the average CPU and memory usage across all benchmark runs, with error bars indicating the corresponding standard deviation. In both figures, the x-axis represents the different Redis persistence modes, while the y-axis shows the corresponding resource usage: CPU usage as a percentage in Figure 5.3 and memory usage in megabytes in Figure 5.4. It's important to note that the CPU idle usage was consistently at 0% before each benchmark run, ensuring a fair comparison of resource consumption.

AOF (always) demonstrates the lowest CPU usage at 14.18%, largely due to its blocking *fsync* calls that cause the main thread to wait for disk synchronization, resulting in reduced overall CPU activity. In contrast, **RDB** shows a relatively high CPU usage at 63.75%. **AOF (everysec)** and **AOF (no)** exhibit similar CPU usage levels at 81.22% and 78.69%, respectively. In **AOF (everysec)**, *fsync* is performed once per second in the background, minimizing its impact on the main processing thread. Similarly, **AOF (no)** relies on the operating system to manage disk writes, allowing the CPU to continue processing client requests efficiently without blocking.

Notably, **AOFUring** exhibits the highest average CPU usage at 189.23%, which can be attributed to the large number of `io_uring` worker threads spawned to handle unbounded tasks such as *fsync*. As a result, all 32 cores of the EC2 instance are fully active. `io_uring` is designed to optimize asynchronous I/O operations by allowing the submission and completion of I/O tasks without requiring the main thread to block. However, when dealing with operations like *fsync*, which can take an indeterminate amount of time to complete, `io_uring` classifies these as unbounded work. As highlighted in the Cloudflare blog (31), `io_uring` addresses unbounded tasks by spawning worker threads at the application level, rather than blocking the submission queue. This approach prevents the main thread from stalling but at the cost of significant CPU resources, as each *fsync* could trigger the creation of multiple worker threads.

Memory Usage, shown in Figure 5.4, also varies across the modes. **AOFUring** consumes the most memory on average at 22.81 MB, primarily due to extra allocations for the write buffer, as explained in section 4.5.1. **RDB** uses 15.34 MB of memory, which is possibly attributed to the overhead of managing the snapshot files. The different AOF configurations **AOF (always)**, **AOF (everysec)**, and **AOF (no)**, have very similar memory usages. This is because they are all essentially utilize the same implementation under the hood with small variance regarding the *fsync* configurations.

How these numbers gathered?

100% = 1 core or Full CPU?

Hqstypo
= Ry 31

5.5 AOFUring Data Correctness Test

In our AOFUring implementation, ensuring data correctness is essential to validating that the system consistently maintains accurate data under various operational conditions. The testing framework includes three specific correctness tests, which can be configured with a required argument specifying the number of requests to issue and an optional argument to disable AOF rewrites.

The first test sets a sequence of keys with incrementing names (e.g., `key_1`, `key_2`, etc.) and corresponding values, continuing until the specified request count is reached. This test evaluates whether AOFUring can reliably store and retrieve these key-value pairs. After setting the keys, the system verifies that each key exists and holds the correct value.

The second test focuses on incrementing a single key repeatedly until the specified request count is reached. This scenario is designed to assess how AOFUring handles frequent updates to the same piece of data. The test concludes by verifying that the final value of the key matches the number of increments specified by the request count.

The third test examines the process of overwriting the same key with incrementing values until the specified request count is reached. In this case, a single key is repeatedly updated with values ranging from 1 to the specified request count. The final verification ensures that the key contains the last value set.

If AOF rewrites are not disabled by the given argument, the tests randomly issue a `BGREWRITEAOF` command, which forces a rewrite of the AOF file. Additionally, Redis may automatically trigger a rewrite on its own during the test execution.

These tests set the `correct-test` and `correct-test-reqnum` configurations (discussed in Section 4.2) to instruct Redis to log when the final `fsync` occurs. This is particularly important in the AOFUring implementation, where data might still be persisting after the request workload has completed. Once the tests capture this output, the Redis server can be restarted to force the in-memory dataset to rebuild. Subsequently, we can verify whether the keys contain the correct values or if they were persisted at all.

In our benchmarking environment, we run these tests with 2,000,000 requests each. The tests pass successfully in our implementation. The test that sets different keys with incrementing values confirms that all keys are persisted correctly and hold the expected values, as evidenced by the CSV file generated by the correctness test, which logs all key-value pairs. Additionally, both the incrementing test and the test that overwrites the same key with different values verify that the final key holds the value 2,000,000, indicating that the data has been correctly persisted.

✓
custom
tests
OK.

6

Related Work

Reducing latency and increasing throughput in database systems have long been critical challenges. Various studies have proposed innovative solutions, including optimized logging mechanisms, advanced storage technologies, and involving asynchronous I/O APIs like `io_uring`, to address these issues.

There have been numerous studies aimed at enhancing performance in various journal modes such as **Write-Ahead Logging (WAL)**. In traditional **WAL** (32), database systems such as SQLite *write* entire pages of data to the **WAL** file to ensure atomicity and durability of transactions. This approach, while ensuring data integrity, results in significant *write* amplification and I/O overhead, negatively impacting performance. Park et al. propose an innovative solution called **SQLite/SSL**, which addresses these performance issues by logging only the SQL statements of transactions instead of the entire modified pages (33). This SQL statement logging method significantly reduces the amount of data written to disk, thereby decreasing I/O overhead and *write* amplification.

There are works that optimize logging for novel storage technologies, such as **WALTZ**, which leverages the `zone append` command for **ZNS** SSDs to enhance performance. Lee et al. (34) address performance issues in LSM tree key-value stores like RocksDB, Cassandra, and LevelDB with **WALTZ**. This system improves storage efficiency and performance by utilizing **WAL** zone replacement, reservation mechanisms, and lazy metadata management, ensuring continuous *write* operations with minimal latency spikes.

Recent advancements in key-value store optimization have leveraged modern asynchronous I/O technologies to significantly enhance performance. **Prism**, a key-value store introduced in a recent conference (35), first writes data to **NVM** (36) to ensure immediate persistence and then employs `io_uring` to asynchronously *write* data to SSDs. This approach allows **Prism** to achieve high throughput and low latency by maximizing SSD

unknown
ref n

bandwidth, all while maintaining strong guarantees for data durability.

Another `io_uring` example is the **Walack** algorithm (37), which uses `io_uring` to perform asynchronous `fsync` operations during checkpoints in **WAL** systems. In **WAL**, a checkpoint is the process of transferring changes from the **WAL** file to the main database file, ensuring consistency and preventing the **WAL** file from growing too large (32). By leveraging asynchronous `fsync`, **Walack** reduces latency spikes associated with this process, while dynamically adjusting checkpoint timing based on workload.

While `io_uring` has garnered significant attention for its high performance, it is not the only Asynchronous I/O (AIO) API available. Recent research by Didona et al. (38) systematically compares AIO APIs, such as `libaio`, `io_uring`, and **SPDK**. Their study reveals that while `io_uring` can achieve performance close to that of **SPDK**, it requires tuning with `SQ-POLLING` enabled and a sufficient number of CPU cores to match **SPDK**'s efficiency. Notably, **SPDK** consistently outperforms the other APIs, offering the fastest performance, particularly in high IOPS and low-latency scenarios. This work highlights the trade-offs between ease of use and raw performance among modern storage APIs, offering valuable insights for developers of I/O-intensive applications.

Other databases have adopted asynchronous I/O to enhance performance. For example, **PostgreSQL** uses asynchronous commit to reduce the wait time for transactions. Normally, **PostgreSQL** waits for **WAL** records to be written to disk before confirming a transaction. With asynchronous commit, transactions can be confirmed without waiting for the **WAL** records to be flushed to disk. This approach reduces the latency of transaction commits, improving overall throughput. However, in the event of a crash, some recently committed transactions might be lost because their **WAL** records were not yet written to disk (39).

Additionally, **InnoDB**, the storage engine for **MySQL**, uses Linux native asynchronous I/O (**AIO**) (40) to improve performance by reducing wait times for disk I/O operations. Asynchronous I/O allows **InnoDB** to process other tasks while waiting for I/O operations to complete, enhancing overall throughput and responsiveness (41).

Conclusion

7.1 Answering Research Questions

Format δ or \underline{R}

- (RQ1) *How does the performance of AOFUring compare to traditional Redis persistence modes?*

AOFU~~Ring~~ exhibits performance that is closely aligned with that of RDB, AOF (everysec), and AOF (no), demonstrating comparable throughput and latency. In the conducted benchmark experiment, AOFU~~Ring~~ was found to be at most 3.4% slower across various Redis commands, while outperforming AOF (always) by more than 6 times. Despite its competitive standing among other high-throughput persistence modes, AOFU~~Ring~~ incurs a significantly greater cost in CPU usage, with an average CPU usage of 189%.

- (RQ2) *What impact does AOFUring have on data correctness and durability?*

Data is correctly stored in AOFU~~Ring~~ and contains accurate values, as verified by our correctness tests conducted with 2,000,000 requests per test. On the other hand, durability is harder to assess because no specific test for it was developed. We can assume that it ~~likely~~ performs worse in this regard compared to AOF (always), since this mode guarantees durability. The asynchronous nature of io_uring introduces the risk that data might be acknowledged as written before it is fully committed to disk, increasing the likelihood of data loss if a failure occurs.

Note: As I understand uring implementation asynchronously issues more fsync with each write, but doesn't wait for completion.

7.2 Limitations and Future Work

This study examined the performance and durability of various Redis persistence modes, including **AOFUring**. While the benchmarks yielded valuable insights, certain limitations must be considered. First, the use of **redis-benchmark** as the primary testing tool, rather than more widely recognized benchmarks like YCSB (42). Although **redis-benchmark** is designed specifically for Redis, it may not fully capture the diversity of workloads or accurately reflect real-world usage patterns across different environments. Additionally, the durability analysis was largely theoretical, as reliably simulating real-world power failures and system crashes posed significant challenges.

Future research should address these limitations by incorporating a broader range of benchmarking tools, such as YCSB, to enable a more comprehensive evaluation of performance. Expanding the scope of testing to include various hardware configurations and workload types would also help in providing a more accurate performance profile. Moreover, developing more advanced methods to simulate and measure the impact of power failures on data integrity would enhance the understanding of durability of **AOFUring**. Finally, optimizing **AOFUring** to reduce its high CPU usage.

↓
check and
make persistent.

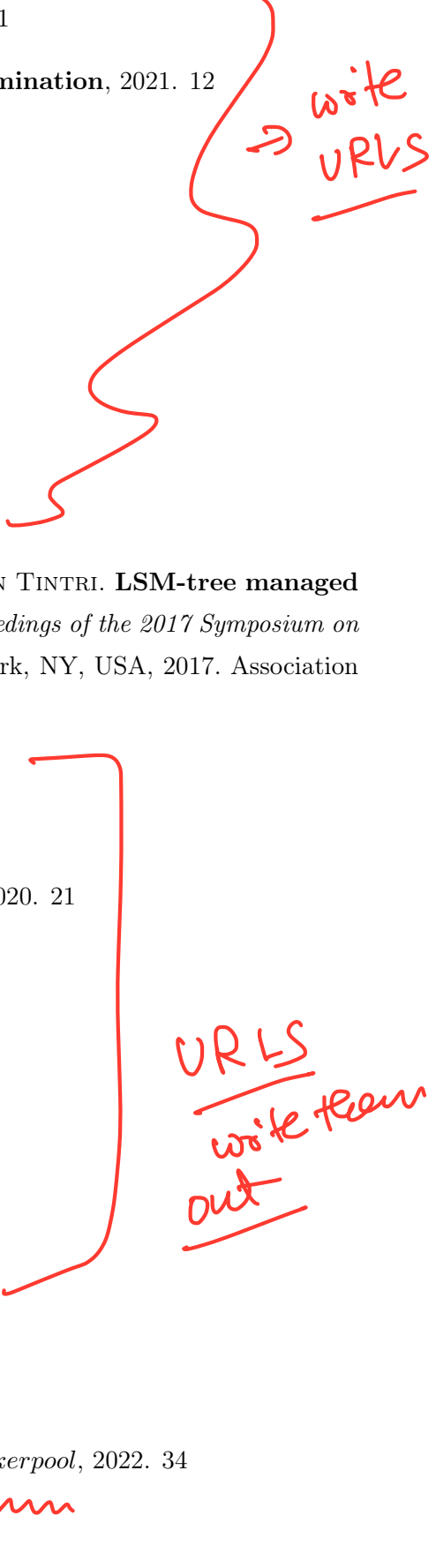


References

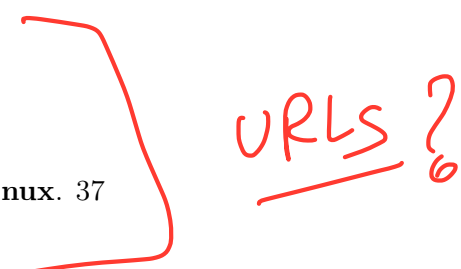
please write the
URL in the
BIB

- [1] SIG FUTURE COMPUTER ICT RESEARCH PLATFORM NETHERLANDS AND NETWORK SYSTEMS. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**. 1
- [2] AMAZON WEB SERVICES. **Amazon ElastiCache for Redis**, 2023. Accessed: 2024-07-18. 1 -URL
- [3] **Redis Use Case Examples for Developers**. 1 -2 URL
- [4] STEPHEN J. BIGELOW, BRIEN POSEY, AND ERIN SULLIVAN. **What Is a Storage Snapshot?**, 2024. 1
- [5] FASTERCAPITAL. **Persistence Strategies: Journaling File Systems - Journaling File Systems: A Diary for Data Persistence**, 2023. 1
- [6] YONATAN GOTTESMAN, JOEL NIDER, RONEN KAT, YARON WEINSBERG, AND MICHAEL FACTOR. **Using Storage Class Memory Efficiently for an In-memory Database**. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, New York, NY, USA, 2016. Association for Computing Machinery. 1, 2
- [7] ZEBIN REN AND ANIMESH TRIVEDI. **Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring**. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '23, page 35–45, New York, NY, USA, 2023. Association for Computing Machinery. 2
- [8] REDIS. **Redis Persistence**. 5, 6, 7, 10 -URL
- [9] RAPHAEL DE LIO. **Understanding Persistence in Redis: AOF RDB on Docker**, 2022. 8

REFERENCES

- [10] SHUVEB HUSSAIN. **Lord of the io_uring**, 2020. 11
 - [11] NIELS DE WAAL. **io_uring: A quantitative examination**, 2021. 12
 - [12] SHUVEB HUSSAIN. **Liburing**, 2020. 14
 - [13] LINUX. **io_uring_prep_write**. 14
 - [14] LINUX. **io_uring_prep_fsync**. 14
 - [15] REDIS. **SET Command**. 16, 28
 - [16] SHUVEB HUSSAIN. **Linking requests**, 2020. 16
 - [17] **Parital Write**. 18
 - [18] **GCC, the GNU Compiler Collection**. 20
 - [19] FEI MEI, QIANG CAO, HONG JIANG, AND LEI TIAN TINTRI. **LSM-tree managed storage for large-scale key-value store**. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17*, page 142–156, New York, NY, USA, 2017. Association for Computing Machinery. 20
 - [20] **GDB: The GNU Project Debugger**. 20
 - [21] REDIS. **Redis benchmark**. 20
 - [22] SHUVEB HUSSAIN. **Submission Queue Polling**, 2020. 21
 - [23] SHUVEB HUSSAIN. **io_uring_register**, 2020. 21
 - [24] **fdatasync(2) - Linux man page**. 24
 - [25] **GET**. 27
 - [26] **Amazon EC2 C5 Instances**. 27
 - [27] **Amazon EBS General Purpose Volumes**. 27
 - [28] **HSET**. 28
 - [29] **lpush**. 28
 - [30] REDIS. **INCR Command**. 28
 - G[31] JAKUB SITNICKI. **Missing Manuals - io_uringworkerpool**, 2022. 34
- 

REFERENCES

- [32] WIKIPEDIA. **Write-ahead logging**. 36, 37
- [33] SANG-WON LEE JONG-HYEOK PARK, GIHWAN OH. **SQL Statement Logging for Making SQLite Truly Lite**, 2017. 36
- [34] JONGSUNG LEE, DONGUK KIM, AND JAE W. LEE. **WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD**. *Proc. VLDB Endow.*, **16**(11):2884–2896, jul 2023. 36
- [35] YONGJU SONG, WOOK-HEE KIM, SUMIT KUMAR MONGA, CHANGWOO MIN, AND YOUNG IK EOM. **Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices**. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 588–602, New York, NY, USA, 2023. Association for Computing Machinery. 36
- [36] **NVM**. 36 ?
- [37] LI ZHU, YANPENG HU, AND CHUNDONG WANG. **Asynchronous and Adaptive Checkpoint for WAL-based Data Storage Systems**. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1238–1245, 2023. 37
- [38] DIEGO DIDONA, JONAS PFEFFERLE, NIKOLAS IOANNOU, BERNARD METZLER, AND ANIMESH TRIVEDI. **Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring**. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 120–127, New York, NY, USA, 2022. Association for Computing Machinery. 37
- [39] POSTGRESQL. **Asynchronous Commit**. 37
- [40] LINUX. **aio**. 37
- [41] MYSQL. **17.8.6 Using Asynchronous I/O on Linux**. 37
- [42] YCSB. 39
- 

Appendix A

Reproducibility

A.1 Abstract

Appendix A provides an overview of how to access the Redis `io_uring` artifact and benchmarks, how to run the artifact, and how to reproduce results obtained in the evaluation section.

A.2 Artifact Check-list (Meta-information)

- **IMPORTANT!** Kernel Requirement: `>= 6.x`
- **Program:** Redis `io_uring`
- **Compilation:** Make, Python3
- **Run-time environment:** C standard library, Python3
- **Metrics:** RPS, Latency Statistics, Memory Usage, CPU Usage, System Calls numbers and times
- **Output:** CSVs, Graphs
- **Experiments:** Comparisons
- **Preparation Time:** Approximately 10 minutes
- **Experiment Completion Time:** Depends on benchmark arguments and hardware; for 100,000 requests, around 30 minutes
- **Publicly Available:** Yes

A. REPRODUCIBILITY

A.3 Description

A.3.1 How to Access

To access the complete framework, including Redis `io_uring` and benchmarks, find it [here](#).
For just the Redis `io_uring` implementation, find it [here](#).

A.3.2 Software Dependencies

IMPORTANT! Kernel `>= 6.x`

- `make`
- `gcc`
- `python3`
- `strace`
- Python libraries:
 - `pandas`
 - `psutil`
 - `matplotlib`
 - `redis`
 - `redis.asyncio`

A.4 Installation

A.4.1 Cloning the Repository

Pull the repository using:

```
git clone --recurse-submodules https://github.com/daraccrafter/Thesis-Redis-IO_Uring
```

A.4.2 Ubuntu & Debian-based

Run the `./setup.sh` script. This script will:

1. Pull both `redis` and `redis-io_uring` git submodules, if not already pulled.
2. Install all necessary dependencies.

3. Build both `redis` and `redis-io_uring`.

Execute the following commands in your terminal:

```
chmod +x setup.sh
./setup.sh
```

A.4.3 Manual Dependency Installation

If you prefer to install dependencies manually, ensure the following are installed on your system:

- `make`
- `gcc`
- `python3`
- `strace`
- Python libraries:
 - `pandas`
 - `psutil`
 - `matplotlib`
 - `redis`

Follow these steps:

1. Pull Git Submodules:

```
git submodule update --init --recursive
```

2. Build Redis and Redis-io_uring:

```
make -C redis
make -C redis-io_uring
```

3. Copy Redis Tools:

A. REPRODUCIBILITY

```
cp redis/src/redis-benchmark scripts/  
cp redis/src/redis-check-aof scripts/
```

To run the AOFUring implementation itself:

```
cd redis-io_uring  
# Start the Redis server with the specified configuration file (redis.conf)  
src/redis-server redis.conf
```

To issue commands in a separate terminal:

```
cd redis-io_uring  
# Issue a SET command to the Redis server, setting 'key' to '1'  
src/redis-cli SET key 1
```

To run redis-benchmark:

```
cd redis-io_uring  
# Run the Redis benchmark tool to test the performance of the 'SET' command  
# -t set: Specifies the 'SET' command as the operation to benchmark  
# -n 100000: Executes the 'SET' command 100,000 times during the benchmark  
src/redis-benchmark -t set -n 100000
```

A.5 Evaluation and Expected Results

In this section, we will demonstrate how to reproduce our results.

A.5.1 Benchmarks

First, navigate to the `scripts` directory by executing:

```
cd scripts
```

Ensure that the directory contains the necessary executables:

```
redis-benchmark  
redis-check-aof
```

To run the benchmark without reformatting the filesystem, execute the following command:

```
sudo python3 run_benchmarks.py --requests 4000000
```

A.5 Evaluation and Expected Results

To run the benchmark with reformatting between each test, follow these steps:

```
cp -r data/RDB-<timestamp> <root-partition>
sudo umount /mnt/ext4
sudo mkfs -t ext4 /dev/<drive>
sudo mount /mnt/ext4
# REPEAT INSTALLATION
sudo python3 run_benchmarks.py --benchmark AOF --requests 4000000
cp -r data/AOF-all-<timestamp> <root-partition>
sudo umount /mnt/ext4
sudo mkfs -t ext4 /dev/<drive>
sudo mount /mnt/ext4
# REPEAT INSTALLATION
sudo python3 run_benchmarks.py --benchmark URING_AOF --requests 4000000
```

Repeat this sequence three times to achieve results comparable to those presented in the evaluation.

IMPORTANT! Execute the benchmarks with elevated privileges because **strace** requires these privileges to function properly.

Benchmark Data: Each Redis configuration directory stores its respective benchmark data, typically located in **benchmarks/<config>/data**.

Arguments:

- **-benchmark:** Specifies which benchmark to run. Options include:
 - **AOF:** Runs the Append-Only File benchmark.
 - **RDB:** Runs the Redis Database benchmark.
 - **URING_AOF:** Runs the benchmark using **io_uring** with AOF.

If no benchmark is specified, the script will run all three benchmarks by default.

- **-requests:** Specifies the number of requests to be sent during the benchmark. The default is 100,000, but for a more extensive test, you can increase this number as shown in the example (4,000,000 requests).
- **-fsync:** Defines the **fsync** mode for the AOF benchmark. Available options include:
 - **always:** Ensures that data is written to disk immediately after each write operation.
 - **everysec:** Synchronizes data to disk every second.

A. REPRODUCIBILITY

- **no**: Disables synchronization after write operations.
- **all**: Runs the benchmark for all **fsync** modes.

The default setting is **all**.

- **-no-strace**: When this flag is set, the benchmark runs without invoking **strace**, which can reduce overhead and improve performance during the tests. By default, **strace** is used.

A.5.2 Data Correctness Test

Navigate to the **scripts** directory:

```
cd scripts
```

To verify the correctness of the data, and run the 3 mentioned tests, you can execute the following command in the terminal:

```
sudo python3 correctness-test.py
```

Arguments:

- **-requests**: Specifies the number of requests to be used in the benchmark. The default value is 100,000, but you can adjust this number depending on the scope of your testing.
- **-no-bgrewriteaof**: This flag, if set, disables the triggering of the **BGREWRITEAOF** command during the test. The **BGREWRITEAOF** command is typically used to rewrite the AOF (Append Only File) to reduce its size and optimize its structure. By default, this feature is enabled, but you can disable it with this flag to test scenarios without AOF rewriting.

A.5.3 Plotting

To generate plots navigate to the **scripts** directory:

```
cd scripts
```

And execute the script:

```
sudo python3 plot.py --dir_rdb <path> --dir_aof <path> --dir_uring <path>  
--dir <output-dir> --type all
```


A.5 Evaluation and Expected Results

The directories for each persistence mode can contain many sub directories (for each run) generated by the benchmark. For example if u run the benchmark without reformatting u can just execute:

```
sudo python3 plot.py --dir_rdb benchmark/RDB/data --dir_aof benchmark/AOF/data
--dir_uring benchmark/URING_AOF/data --dir ./output --type all
```

This will generate all the graphs.

Arguments:

- **-dir_rdb**: Specifies the directory containing the CSV files for the **RDB** persistence mode. This argument is mandatory.
- **-dir_aof**: Specifies the directory containing the CSV files for the **AOF** persistence mode. This argument is mandatory.
- **-dir_uring**: Specifies the directory containing the CSV files for the **URING_AOF** persistence mode. This argument is mandatory.
- **-dir**: Defines the directory where the generated graphs will be saved. The default is the current directory.
- **-type**: Specifies the type of graph to plot. The following options are available:
 - **rps**: Generates a graph comparing the requests per second (RPS) across the different persistence modes.
 - **cpu**: Generates a graph comparing CPU usage across the different persistence modes.
 - **memory**: Generates a graph comparing memory usage across the different persistence modes.
 - **latency**: Generates a graph comparing latency statistics across the different persistence modes.
 - **all**: Generates all of the above graphs.