SIFT: A High-Performance and CPU-Efficient Data Persistence Architecture for User-Space File Systems

by

Peizhe Liu

Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Adviser:

Associate Professor Tianyin Xu

Abstract

As high-speed storage hardware continues to evolve, device-side latency has been steadily compressed. To fully exploit this hardware potential, high-performance user-space file systems increasingly rely on polling-based I/O and aggressive concurrency. In this setting, background persistence work becomes increasingly visible: rather than remaining an invisible maintenance task, it consumes host CPU budget and interferes with foreground progress.

Computational storage provides a practical basis for moving this heavy background persistence work away from the host. However, directly pushing the background persistence path toward the device would still leave foreground `fsync` exposed to background delay. This thesis therefore proposes SIFT, a persistence architecture that reorganizes persistence into two layers: a split persistence path that gives foreground requests a short, self-contained durable step while background persistence work proceeds separately, and a device-side checkpoint path that coalesces replay across transactions so that convergence remains efficient on a resource-constrained processor. A synchronization protocol ties these layers together by defining durability, visibility, replay eligibility, and reclamation boundaries.

Evaluation shows that, at a representative online operating point, the offloaded path already releases more than 50% of the raw host-side CPU burden relative to the host-journaling baseline. Under sustained online checkpoint pressure, the fully coalesced design achieves more than 35× higher foreground throughput. In a synthetic high-overlap backlog, it eliminates duplicated replay across transactions, reducing raw block replay by about 98.7% and raw inode replay by about 99.9%; under a sequential no-reuse workload, it still retains about 1.15× speedup.

Acknowledgements

I would like to sincerely thank my advisor, Prof. Tianyin Xu. Tianyin approaches research with genuine passion. Working with him has shown me what serious and disciplined research should look like. Throughout my master's studies, he provided invaluable guidance, resources, opportunities, and support. More importantly, his advising helped me take my first real steps into research, develop my interests, and identify the direction I hope to pursue in the future.

I am also deeply grateful to my thesis co-advisor Dr. Jongyul Kim for his guidance and support. Jongyul is deeply knowledgeable, thoughtful, and exceptionally grounded as a researcher. Throughout our collaboration, he provided clear direction and extensive help with both the overall direction and the technical details of the project. He has been one of the most important mentors in my research journey. Without him, I could not have completed this thesis.

I would also like to thank Jiyuan Zhang. Jiyuan is an outstanding researcher with deep insight into systems research. He offered me thoughtful guidance, encouragement, and help. He has also been one of the most important mentors in my research journey.

I am also thankful to Prof. Weiwei Jia and Dr. Chloe Alverti for their guidance and support. I would like to thank Siyuan Chai, Cathy Cai, and all members of xLab as well. I have benefited greatly from working with and learning from them throughout my master's studies.

The work described in this thesis also relied on collaboration with colleagues from KAIST. I would like to sincerely thank Prof. Youngjin Kwon, Jaehwan Lee, Inhoe Koo, and Junho Ahn. I feel very fortunate to have had the opportunity to work with them and to contribute to this collaboration. This valuable experience also made it possible for me to complete this master's thesis.

To my family, for their unwavering support throughout my academic journey.

Contents

Chapter 1 Introduction

1.1 Background and Motivation

In recent years, a new generation of storage devices, including NVMe SSDs, persistent memory, and CXL-attached storage, has evolved rapidly. Both throughput and latency have improved by a large margin [1, 5, 22]. As storage devices continue to get faster, the traditional kernel-centric storage stack is increasingly becoming a bottleneck [3]. To better exploit the I/O potential of modern hardware while also retaining flexibility and fast development, user-space file systems have become an important direction in both academia and industry [10, 13, 15, 30, 33].

The appeal of user-space file systems is clear. They are easier to customize for specific applications and hardware, and they are much easier to iterate on than in-kernel file systems. But this flexibility is not free. To achieve microsecond-scale response time and higher throughput, modern high-performance user-space file systems and user-space storage stacks often rely on polling-based I/O and aggressive concurrency [7, 34]. These choices can improve I/O performance, but they also significantly increase host-side CPU consumption. As workload grows together with hardware throughput, the bottleneck often appears first on the host CPU rather than on the storage devices themselves.

Once host CPU becomes scarce, the next question is which part of the file system consumes that budget most heavily. In journaling-based systems, a substantial portion of persistence cost comes from heavy background persistence work rather than from the short foreground completion path itself [14, 25, 31]. Under write-intensive and `fsync`-heavy workloads, this background work continuously consumes host CPU and competes with applications for the same local resources.

Recent advances in computational storage provide a practical basis to address this problem [2, 27]. Offloading helps not because the device processor is stronger than the host CPU, but because this heavy background persistence work currently competes with

the foreground path and co-located applications for the same host resources, making it a natural target for device-side execution.

1.2 Challenges

As soon as heavy background persistence work is pushed toward the device, three challenges emerge.

The first challenge is that, in conventional journaling, persistence proceeds through a durable commit phase followed by a deferred checkpointing phase. If this longer background path is simply pushed toward the device, foreground `fsync` remains tied to background commit progress, and moving work to the device only changes where the delay is paid.

The second challenge is that checkpointing remains expensive even after background persistence is offloaded. Once checkpointing becomes the convergence and space-reclamation stage of the device-side background path, a weak device processor still cannot afford to mechanically replay the full raw history of overwrites and metadata updates if replay semantics remain unchanged. In addition, delayed space reclamation can still propagate pressure back to the foreground path.

The third challenge is correctness. Once foreground `fsync` and background persistence are processed along separate but coordinated paths across host and device, the system must define when an update becomes durable, when it becomes visible to background processing, when the device is allowed to replay it on the checkpoint path, and when space can safely be reclaimed. Without explicit boundaries, better performance would come at the cost of weaker semantics.

1.3 Design Overview

At a high level, this thesis presents SIFT, a persistence architecture built around two layers and a synchronization protocol to address the three challenges above.

The first layer is a split persistence path. To keep `fsync` on a short host-side path, the design reorganizes persistence into a short, self-contained foreground durable step and a longer background path that proceeds separately on the device. We call the three stages foreground staging, background journaling, and checkpointing. Foreground staging establishes the foreground durable step, background journaling records wider durable state, and checkpointing converges the final state and reclaims space. This addresses the first challenge.

The second layer is cross-transaction replay reduction for device-side checkpointing. The design logically folds repeated updates before physical I/O occurs. We call this approach coalescing. Rather than relying on naive pairwise overlap elimination at batch scale, the coalescing incrementally maintains the final visible state in a range tree, reducing the dominant cost from quadratic pairwise elimination to roughly logarithmic incremental maintenance. This addresses the second challenge.

Finally, a synchronization protocol ties these two layers together by establishing explicit boundaries for durability, visibility, replay eligibility, and reclamation. It ensures that asynchronous split persistence across host and device does not compromise persistence guarantees. This addresses the third challenge.

1.4 System Configurations

To study the design space, we evaluate three configurations that share the same split persistence organization. In all three, foreground requests first reach a short, self-contained foreground durable step in the staging area, while longer background persistence work proceeds separately through background journaling and checkpointing. The configurations differ in where this longer background path executes and whether checkpoint replay is reduced before final convergence.

The first configuration, *vanilla*, is the host-journaling baseline. It preserves the same split persistence organization as the rest of the system, but keeps the longer background

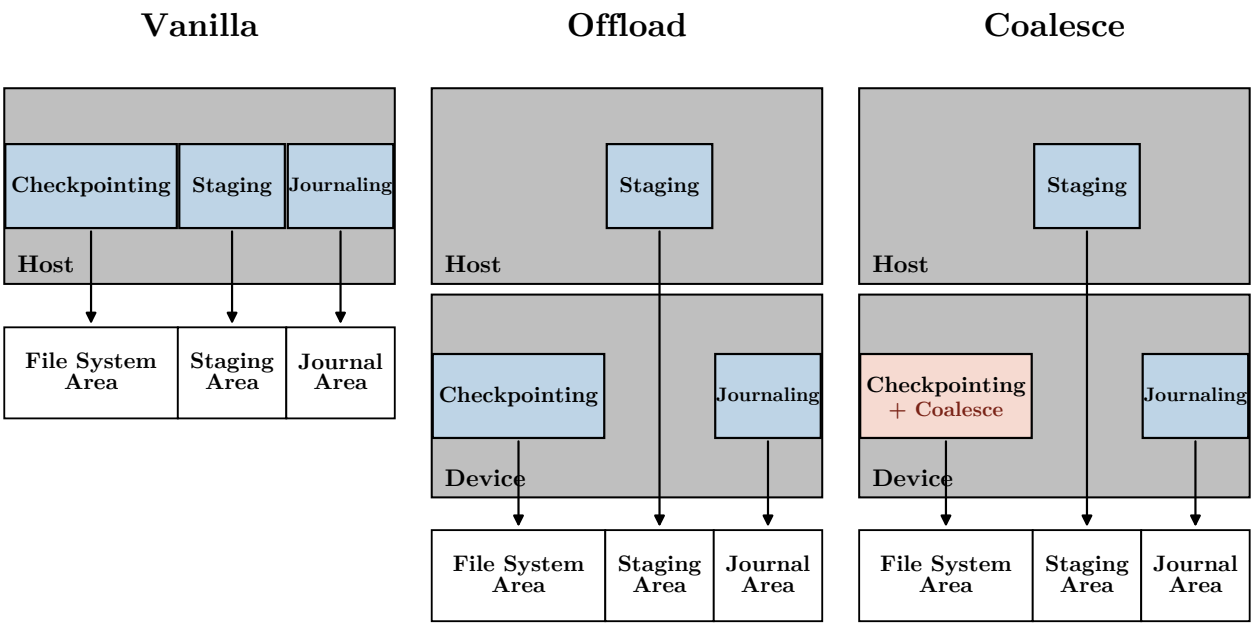

Figure 1: System configurations and persistence path. The three layouts contrast where the longer background path executes and where replay reduction is applied.

path on the host. Foreground execution, background transaction assembly, background journaling, checkpointing, and space reclamation therefore continue to share the same local CPU, cache hierarchy, memory bandwidth, and storage path.

The second configuration, *offload*, keeps the same split persistence organization while moving the longer background path to the device side. Foreground requests still complete after the short foreground durable step in the staging area, but background journaling, checkpointing, and space reclamation proceed on the device.

The third configuration, *coalesce*, retains the offloaded placement and further changes the replay semantics of device-side checkpointing. Rather than replaying the full raw history of durable updates, the checkpoint engine constructs the final visible state of a committed batch via coalescing, and replays only that state.

Figure 1 summarizes these three configurations and their persistence path.

1.5 Contributions

The main contributions of this thesis are summarized as follows.

- We design and implement SIFT, a split persistence architecture that reorganizes persistence into a short, self-contained foreground durable step and a longer background path for later convergence and space reclamation, thereby keeping `fsync` on a short host-side path while enabling device-side execution of background persistence work.

- We design and implement a coalesced checkpointing mechanism for device-side background convergence. Built on a range-tree-based replay-reduction engine together with inode deduplication and a windowed replay pipeline, this mechanism reconstructs the final visible state across transactions instead of mechanically replaying the full raw durable history. Under a high-overlap backlog, it removes about 98.7% of raw block replay and about 99.9% of raw inode replay.

- We define a synchronization protocol that ties the split persistence path and device-side checkpointing together. The protocol specifies durability, visibility, replay eligibility, and reclamation boundaries so that asynchronous execution across host and device does not weaken correctness.

- We evaluate three configurations, *vanilla*, *offload*, and *coalesce*, and show that, at a representative online operating point, the offloaded path already releases more than 50% of the raw host-side CPU burden compared to the host-journaling baseline, while, under sustained online checkpoint pressure, the fully coalesced design sustains more than 35× higher foreground throughput. On a high-overlap manual backlog, it also shortens checkpoint time by about 3.7×. Even under a sequential no-reuse workload, it still retains about 1.15× speedup over the non-coalesced offloaded path.

1.6 Thesis Organization

The rest of this thesis is organized as follows.

- Chapter 2 introduces the background needed for the rest of the thesis. It explains why fast devices make host CPU scarce, why background persistence is the right offload target, and what the baseline means.
- Chapter 3 presents the design of the system, including the split persistence path, the coalescing mechanism, and the synchronization protocol.
- Chapter 4 describes the prototype implementation.
- Chapter 5 evaluates the three configurations, explains the performance regimes they enter under online checkpoint pressure, quantifies replay work reduction, and studies boundary workloads.
- Chapter 6 discusses related work.
- Chapter 7 concludes the thesis and outlines future directions.

Chapter 2 Background and Motivation

2.1 Fast Devices and Visible Software Cost

For a long time, file-system design assumed a simple fact: storage devices were slow, and software overhead was relatively cheap. In the era of magnetic hard disks, this assumption was mostly valid. Millisecond-scale seek and rotational delays were far larger than the CPU time spent inside the file system. As a result, system designers focused more on data layout, access locality, and concurrency control than on instruction-level efficiency in the software stack.

This assumption no longer holds. With the rise of NVMe SSDs, persistent memory, and CXL-related storage technologies, device throughput and latency have changed significantly [1, 5, 22]. Device time has been compressed, while the fixed costs in the software stack have not disappeared. As a result, many modern storage systems are no longer limited first by the storage media itself. Instead, they are increasingly limited by whether the host CPU can drive the device efficiently enough.

2.2 User-space File Systems and Explicit Host-side Persistence

2.2.1 Flexibility and Performance

User-space file systems have regained attention under this new hardware trend [10, 13, 15, 30, 33]. Compared with in-kernel file systems, they are easier to customize for specific applications and hardware. They are also easier to evolve and much easier to iterate on. This flexibility is one of the main reasons why user-space file systems have become attractive in both academia and industry.

However, this flexibility comes at a cost. To approach device-level performance, many high-performance user-space file systems and user-space storage stacks rely on polling-based I/O and aggressive concurrency [7, 34]. These choices can improve throughput and latency, but they also consume substantial host CPU cycles. As

workload scales together with device performance, the bottleneck often moves away from the storage media and appears first on the host CPU.

2.2.2 Explicit Host-side Persistence Cost

User-space file systems make host-side persistence overhead more explicit. Work that already exists in conventional journaling systems, such as transaction construction, background commit processing, checkpointing, and space reclamation, does not disappear; instead, it is often managed more directly by user-space runtime threads rather than being absorbed by kernel subsystems [10, 13, 31].

The background persistence path must assemble durable state, process background commits, revisit metadata, emit final writes, and determine when temporary space can be safely reused. Under write-intensive or `fsync`-heavy workloads, foreground applications and the background persistence path can therefore contend for the same host CPU and memory-system resources [14, 23]. This interference becomes increasingly visible as throughput scales. Once host CPU becomes scarce, this background persistence path becomes a natural target for device-side offloading.

2.3 The True Cost of Checkpointing

Among all explicit background tasks, checkpointing is particularly expensive.

Crash consistency in file systems is typically provided by journaling [6, 25]. Persistence therefore usually proceeds in two stages: updates are first recorded in a journal as a durable transaction, and later replayed into the main file-system area while the corresponding journal space is reclaimed. This second stage is checkpointing.

Checkpointing is a CPU-intensive and I/O-intensive process: it must identify what needs to be replayed, locate the target state, update metadata in order, and finally emit writes to the file-system area. On the host side, this cost mainly shows up in two forms.

The first is data round-trip tax. Checkpointing involves many read-modify-write

operations on metadata blocks, especially inode blocks. If checkpointing runs on the host, the host CPU must still fetch replay inputs and target metadata from the device into host memory and cache, interpret and merge them, and then write the results back to the device, even though checkpointing mainly operates on state that has already been durably written to the device.

The second is host cache pollution. Background replay touches a large amount of cold data and repeatedly revisits metadata state. This can heavily disturb the host cache hierarchy and directly interfere with foreground applications [14].

This cost does not show up as a uniform tax. At a representative online operating point examined later in Chapter 5, the host-journaling baseline still keeps median `fsync` latency below a quarter millisecond, yet its p99.99 latency reaches the multi-second range, and the checkpoint episode coincides with a host CPU peak of about 12% on a 32-core machine. Host-side checkpointing therefore behaves as a bursty background service, periodically blocking foreground progress and consuming multiple cores' worth of local compute budget.

Figure 2 illustrates this point directly: the common path remains short, but rare checkpoint episodes create the long tail and the corresponding CPU burst that matter most to the foreground.

2.4 Conventional Persistence Path and the Limitation of Direct Offloading

In conventional journaling, persistence proceeds through a durable commit phase followed by a deferred checkpointing phase [25, 31]. The commit phase establishes durable journal state, while the checkpointing phase converges that state to the main file-system area and reclaims temporary space. Together, they form the longer background persistence path that follows foreground `fsync`.

This structure has direct consequences for offloading. If the longer background persistence path is simply pushed toward the device, foreground `fsync` still remains tied

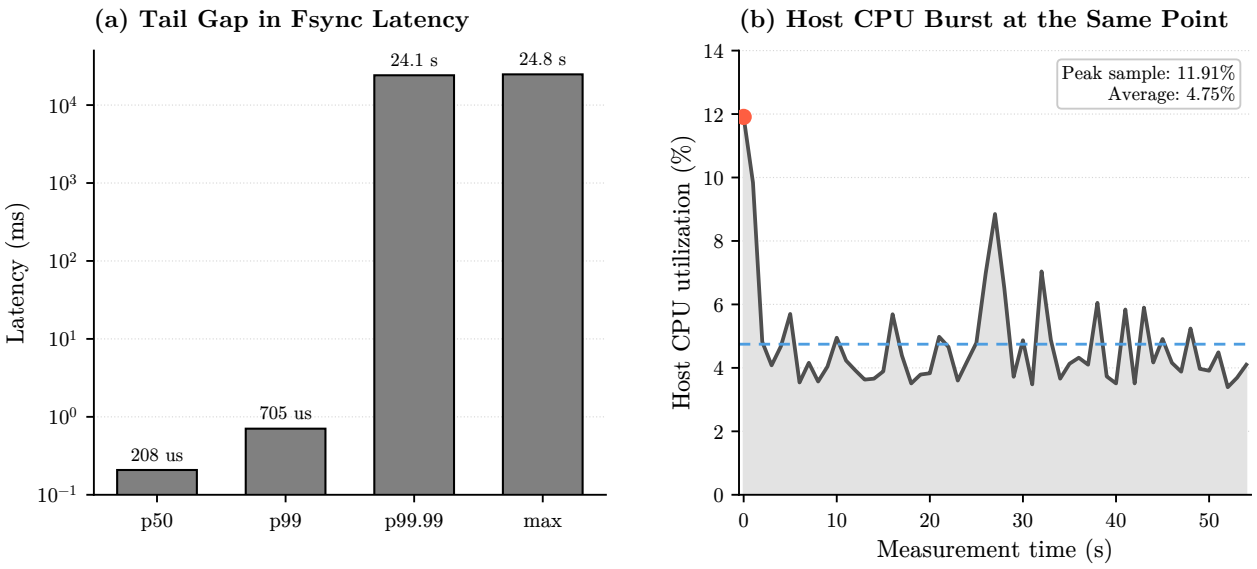

Figure 2: Representative host-side checkpoint interference at one online operating point. Rare checkpoint episodes dominate the extreme tail and coincide with a visible host CPU burst.

to background commit progress, and moving work to the device only changes where the delay is paid. Offloading alone therefore does not separate foreground durability from the longer persistence path.

A split persistence path addresses this gap: foreground `fsync` obtains a short, self-contained durable step in the foreground, while background journaling and checkpointing proceed separately.

This reorganization yields three stages in the design studied in this thesis: staging for the foreground durable step, background journaling, and checkpointing.

2.5 Baseline, Hardware Model, and Failure Assumptions

Throughout this thesis, *vanilla* denotes the host-journaling baseline. It preserves the same split persistence organization used by the rest of the system, including staging, background journaling, and checkpointing, but keeps the longer background persistence path on the host. Foreground requests, background transaction assembly, background journaling, checkpointing, and space reclamation therefore continue to compete for the

same local CPU, cache hierarchy, memory bandwidth, and storage path. *vanilla* is not a generic state-of-the-art host file system, but the host-side counterpart of the same persistence organization, used to separate the cost of keeping the longer background persistence path on the host from the effects of device-side placement and replay reduction.

The hardware model assumed in this thesis is deliberately asymmetric [2, 17]. The host provides a strong multicore CPU, large DRAM, and the software environment in which foreground applications and the host-side file runtime execute. The device provides a weaker general-purpose processor and limited local memory, but it can execute persistence logic close to durable state in the shared storage namespace.

This asymmetry explains both the opportunity and the remaining design task of offloading. Device-side execution is attractive because it can isolate heavy background work from the host. At the same time, limited device resources make replay reduction necessary once checkpointing is placed on the device-side background path.

The shared storage namespace itself is divided into three persistent regions: a staging area, a journal area, and the main file-system area. Foreground updates first become durable in staging, background journal transactions record wider durable state in the journal, and checkpointing finally applies the surviving state to the file-system area while reclaiming staging and journal space.

This thesis assumes a fail-stop crash model. A crash may interrupt execution while a foreground staging transaction is being staged, while a background journal transaction is becoming durable, or while ordered checkpointing is replaying durable state into the main file-system area. At the moment of failure, durable state may therefore exist in any combination of the staging area, the journal area, and the file-system area. These failure assumptions matter because the rest of the thesis will rely on explicit boundaries for durability, visibility, replay eligibility, and safe reclamation.

2.6 Summary

The background above narrows the design space. The host is no longer abundant, heavy background persistence work is too expensive to remain an invisible detail, and the device is too weak to replay durable history mechanically. At the same time, the fail-stop model rules out any design that leaves durability, visibility, and reclamation implicit.

The next chapter turns these constraints into an explicit design. It first reorganizes persistence into a short foreground durable step and a longer background path that proceeds separately, and then reduces replay work across transactions so that the device-side checkpoint path remains effective under limited device resources.

Chapter 3 Design

3.1 Design Goals and Overview

As discussed in Chapter 2, heavy background persistence work is a natural target for device-side offloading in a high-performance user-space file system. The design of SIFT must therefore satisfy three requirements already motivated there: it must preserve a short foreground `fsync` path, keep checkpointing tractable once the longer background path runs on a resource-constrained device, and make durability, visibility, replay eligibility, and reclamation boundaries explicit.

This chapter presents SIFT in three parts. It first introduces a split persistence path that gives foreground requests a short, self-contained durable step while allowing the longer background path to proceed separately. It then presents a cross-transaction checkpoint coalescing mechanism that reduces the replay cost of checkpointing on the device-side background path. Finally, it defines a synchronization protocol that ties these two layers together and preserves the required ordering and crash-consistency semantics.

Table 1 fixes the terminology used in the rest of this chapter and in the implementation and evaluation chapters. Throughout the rest of this thesis, *conventional journaling* refers to the journal-based crash-consistency mechanism in prior systems, whereas *background journaling* denotes the background commit stage within the split persistence path. Likewise, *foreground staging transaction* and *background journal transaction* are the canonical names of the two durable objects in the design, while *staged transaction* is used as the protocol-level shorthand for a foreground staging transaction once its staging identity has been fixed.

Category	Canonical Terms	Role
Stages	foreground staging; background journaling; checkpointing	The three stages of the split persistence path.
Objects	foreground staging transaction; background journal transaction	The foreground per-file durable object and the wider background durable object.
Boundaries	durability; visibility; replay eligibility; reclamation	The semantic boundaries enforced by the synchronization protocol.
Rules	*publish-after-fix*; *late-write-wins*; *wait-not-skip*	The protocol and replay rules that constrain publication, reduction, and replay.
Configurations	*vanilla*; *offload*; *coalesce*	The host-journaling baseline, offloaded path, and coalesced path used throughout evaluation.

Table 1: Canonical terminology used throughout the design, implementation, and evaluation chapters. The table fixes the stage, object, rule, and configuration names referenced later.

3.2 Split Persistence Path

3.2.1 Foreground Durability and Background Completion

To preserve a short foreground `fsync` path under device-side offloading, the system must first reconstruct the persistence path itself. The core idea is to split persistence into two parts with different roles and execution tempos. One is a short foreground path that exists only to establish a self-contained foreground durable step for the current `fsync`. The other is a longer background path that performs background journaling, checkpointing, and space reclamation.

At a high level, the reconstructed path has three stages. Foreground staging provides a short foreground durable step for the current file. Background journaling organizes wider durable state in the background. Ordered checkpointing later applies the surviving state to the main file-system area and releases temporary space. This decomposition keeps foreground `fsync` off the longer background persistence path.

3.2.2 Foreground Staging and the Foreground Durable Point

In this reconstructed path, a foreground `fsync` does not wait for progress on the longer background path. Instead, it produces a foreground staging transaction, a file-level durable object, and writes that transaction into the staging area. This transaction contains the inode and data updates relevant to the current file. Once the foreground staging transaction becomes durable, the foreground request can return.

Staging changes the dependency structure of the system. In a traditional design, the foreground request cannot complete independently because its persistence semantics remain entangled with background commit progress. Under staging, the foreground path obtains a clear durable point, and the remaining work is delegated to the background.

Foreground staging therefore establishes a short, explicit, and predictable durability boundary for foreground requests rather than merely buffering temporary data.

3.2.3 Self-Contained Foreground Transactions

A foreground staging transaction must remain semantically complete with respect to in-flight background commit progress. Otherwise, a fast return would merely expose an incomplete state earlier. To fully remove the dependency of `fsync` on background commit progress, the foreground staging transaction must be self-contained.

Concretely, if part of the current file's state has already been included in an in-flight background journal transaction that has not yet become durable, then those relevant updates must be captured again into the current foreground staging transaction. Only then does the foreground durable point become semantically independent. It no longer relies on some external background state to complete its meaning later.

This self-contained property is the foundation that allows the foreground path to return quickly while the host and device continue making progress in parallel.

3.2.4 Background Journaling

Once foreground durability is handled by staging, the background can organize work at a much broader granularity. The system introduces a background journal transaction, a compound durable object for the background path. Instead of driving the entire persistence chain around a single `fsync`, background threads gather updates from multiple files together with global metadata and package them into a single durable unit.

The primary change is in object granularity. The foreground staging transaction is file-scoped and tied to one `fsync`; the background journal transaction is wider in scope and represents a larger set of system state over time.

This object is also chosen to match device-side execution. The device does not need to interpret high-level file-system semantics. It only needs to process durable block-level state and the references that connect journaled state to earlier staged state. Background journaling thus translates host-side file-system state into a durable form that is suitable for device-side checkpointing.

Background journaling thus realizes the commit stage of the longer background path, leaving checkpointing to perform final convergence and space reclamation.

3.2.5 Ordered Checkpointing and Space Reclamation

After a background journal transaction becomes durable, the system still needs to apply its state to the main file-system area. This is the role of ordered checkpointing. For each committed background journal transaction, the persistence service must first process the staged state that the transaction depends on, then apply the journaled updates carried by that transaction, and finally release the corresponding staging and journal space.

Checkpointing plays two indispensable roles: it turns durable intermediate state into final file-system state, and it reclaims temporary space. Without it, staging and journal space would eventually be exhausted, and the background problem would immediately return to the foreground path. Checkpointing is therefore the convergence point of the

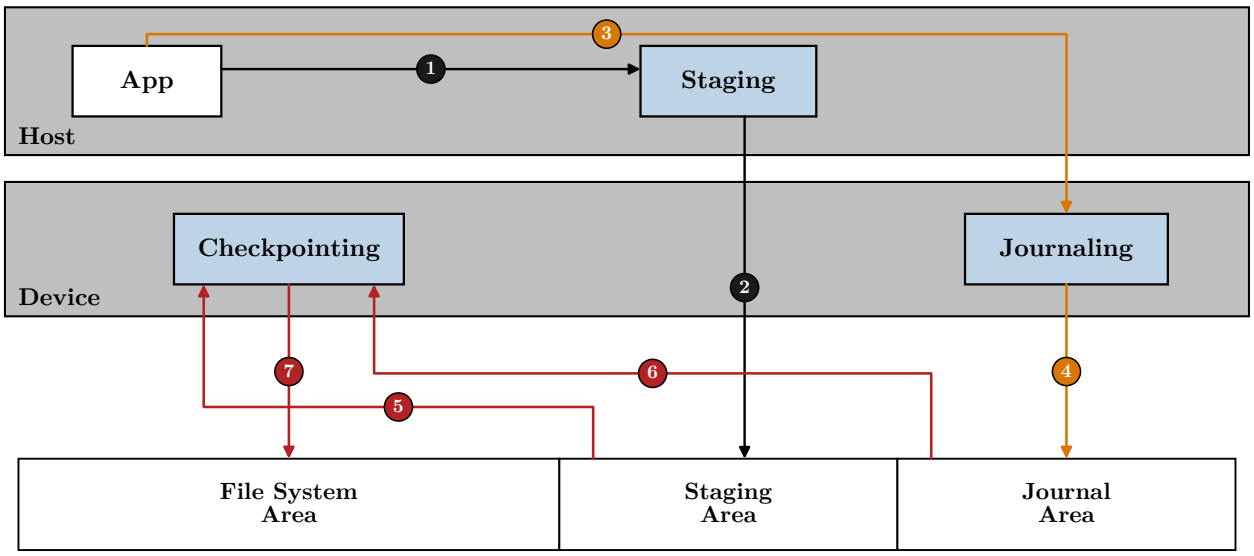

Figure 3: Split persistence path across host, device, and the shared namespace. The foreground `fsync` path (steps 1–2), background commit path (steps 3–4), and device-side checkpointing path (steps 5–7) are color-coded in the figure.

entire persistence pipeline, not merely a replay step. Figure 3 places this end-to-end pipeline onto a single host / device / shared-namespace view.

3.2.6 Trigger Policy

The split persistence path is only useful if checkpointing is triggered in a way that keeps reclaim off the foreground path whenever possible. The system therefore uses a two-level trigger policy.

The normal path is a low-watermark asynchronous trigger. When free staging or journal space falls below a configured threshold, the background path issues an asynchronous checkpoint request so that space can be reclaimed before the foreground path runs out of room. The intent is to keep checkpointing in the background and to replenish credits before foreground staging becomes blocked.

The second path is an emergency synchronous trigger. If a foreground reservation cannot be satisfied with the currently available space, the system issues a synchronous checkpoint request and waits for enough credits to be returned. This fallback exists to

preserve liveness, not to improve steady-state behavior. Ideally, it should remain rare; when it becomes frequent, the system has entered a pressure regime in which replenishment is no longer keeping up with demand.

3.2.7 Snapshot-Style Handoff

The split persistence path also introduces an important engineering requirement: the host must hand a durable background journal transaction to the persistence service without stalling foreground updates for too long. To address this issue, the system adopts a snapshot-style handoff mechanism.

Instead of transferring scattered pages or blocks one by one, the host gathers the metadata and dirty data associated with the current background journal transaction into a contiguous buffer, and hands this buffer to the persistence service as a whole. This extra memory copy serves a dual purpose: it prepares transmission and simultaneously acts as a snapshot. After a short copy interval, the host can release the relevant pages and locks, while the persistence service continues processing the captured snapshot independently.

This mechanism significantly reduces interference between foreground writes and background commit processing, and it raises the granularity of the background durable object from scattered pages to a structured transaction-level form.

3.3 Cross-Transaction Replay Reduction

Once the split persistence path has been established and the longer background path has been offloaded, checkpoint replay remains the main residual cost on the device side. A straightforward ordered replay engine would still process stale versions, repeated inode updates, and fragmented writes, merely relocating the replay burden from the host to the weaker device processor. The design therefore reduces checkpoint replay itself by reconstructing only the final visible state across transactions, rather than mechanically

replaying raw durable history.

3.3.1 Batch-Scoped Visible State

The first step toward reducing checkpoint replay work is to change the observation window of checkpointing. As long as the system only replays one background journal transaction at a time, it cannot detect overwrite patterns across transactions. In contrast, if the system processes a batch of consecutive committed background journal transactions together, it can identify which states remain relevant and which ones have already been overwritten by later transactions.

Therefore, the checkpoint engine no longer takes a single transaction as input. Instead, it consumes what we call a *ready-prefix batch*: the longest prefix of the commit queue such that every transaction in the prefix has become durable, read in commit order. This batch boundary is both ordered and safe. It preserves the original commit order and creates a well-defined scope for replay reduction.

3.3.2 Ordered Replay Semantics

Batch-oriented processing does not allow the system to violate internal causality. In the split persistence path, each background journal transaction is associated with two classes of state: staged state carried over from foreground staging, and journaled updates carried by the background journal transaction itself. In the implementation, the former is replayed from the staging area and the latter from the journal area.

To preserve the final visible state, the system must enforce the following rule: staged state is processed before the journaled updates of the same transaction. The optimization scope may become larger, but the meaning of the original persistence order must remain unchanged.

Late-Write-Wins. Once the batch scope and ordering constraints are fixed, the engine applies the core reduction rule: *late-write-wins*. Within one batch, only the last version of each logical block is allowed to remain in the final replay set. If an earlier version has already been overwritten by a later transaction, then the earlier version, although durable, no longer belongs to the final state that must be replayed into the file-system area.

This rule fundamentally changes checkpointing: instead of replaying every durable update, the engine constructs the final visible state for the batch and replays only that state.

3.3.3 Inode Deduplication

The replay overhead of checkpointing does not come only from data blocks. Inode-related metadata is often an equally serious source of amplification. Across a batch of committed background journal transactions, the same inode block may be updated many times in small increments. If the device-side engine replays every such update separately, it ends up performing a large number of fine-grained read-modify-write operations on metadata.

To address this problem, the system applies the same keep-only-the-final-state idea to inode updates. Within one batch, all updates targeting the same inode block are collapsed into the last one. The result is that a long chain of small inode modifications is reduced to a single final version that needs to be replayed.

This step removes a major source of metadata amplification and ensures that replay reduction is not limited to data blocks alone.

3.3.4 Range Tree Coalescing Engine

The coalescing path is implemented with a range tree that maintains the current final visible state of the batch. Each new interval is inserted into this structure incrementally. As it arrives, the engine identifies stale intervals, trims residual fragments when

necessary, and merges adjacent survivors so that the tree always represents the replay set that still belongs to the final file-system state.

If a batch contains N interval updates and the total number of overlap-handling events is M, the overall work of the engine can be expressed as $O(N \log N + M)$. The tree avoids the naive global pairwise elimination step, whose worst-case behavior would be $O(N^2)$, and instead reduces the dominant cost to logarithmic search plus bounded local restructuring.

This structure serves two purposes. It turns the reduction rule into a continuously maintained execution structure that can directly feed the replay pipeline, and it avoids rebuilding the replay set from scratch after all updates have been collected. In practice, this makes the engine suitable for a resource-constrained device.

3.3.5 Windowed Replay Pipeline

Even after the system extracts a reduced final visible state, it still cannot blindly emit all replay operations at once. On a constrained device, the cost of replay depends not only on how many blocks remain, but also on how these blocks are organized and issued.

The system therefore introduces a windowed replay pipeline. It first constructs the final replay set, then partitions that set into windows. Within each window, reads are reorganized to improve source locality, writes are regrouped to improve destination locality, and temporary buffers are reused aggressively. Requests that can be merged locally are merged before they are issued.

This mechanism does not change replay semantics; it changes the physical shape of replay by reducing the number of I/O requests, improving locality on both the source and destination sides, and keeping the device working set bounded.

Figure 4 summarizes these checkpoint sub-processes end-to-end: each background journal transaction in the ready-prefix batch draws two sources from the shared namespace, staged state from the Staging Area and journaled updates from the Journal

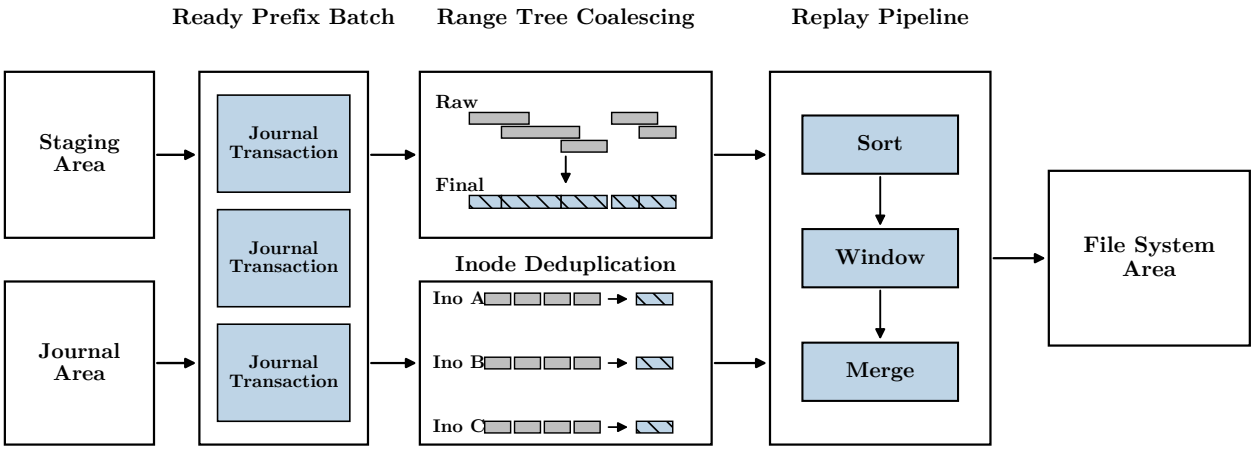

Figure 4: Device-side coalescing pipeline. Each journal transaction in the ready-prefix batch references two sources (staged state in the Staging Area, journaled updates in the Journal Area); after range-tree-based coalescing and inode deduplication, the surviving records feed the replay pipeline, whose output returns to the File System Area.

Area; range-tree-based coalescing and inode deduplication then compact the replay set in parallel, and the windowed replay pipeline finally converges the surviving state on the File System Area.

3.3.6 Worst-Case Mitigation

Even when user-level rewrite overlap disappears, the tree does not become pure overhead. Replay reduction is not driven only by user-level overwrite patterns: the split persistence path still introduces two representations of the same durable update, one in staged state and one in journaled state, and repeated inode updates to the same files remain highly compressible even under no-reuse writes. The engine therefore continues to remove protocol-level duplication and metadata redundancy in that regime.

The windowed replay pipeline also remains useful here. Even when the logical replay set is close to the raw user data footprint, the physical I/O pattern can still be improved substantially by regrouping reads and writes into larger, more regular requests.

None of this implies that the tree is free. When user-level overlap disappears, the tree's own maintenance cost becomes much more visible; it only explains why the

coalesced path may still retain a positive net benefit even under workloads that remove user-level rewrite overlap. Chapter 5 returns to this point explicitly.

3.4 Synchronization Protocol

Once foreground staging, background journaling, and checkpointing proceed asynchronously across host and device, the system faces a concrete coordination problem: the same staged update may already be durable enough for the foreground `fsync` to return, yet still be unsafe for background observation, unsafe for replay, or unsafe to reclaim. A naive implementation could therefore return too early, replay an incompletely published object, or reuse space that still belongs to recovery. This section defines the synchronization protocol that solves exactly this problem under the fail-stop model.

Protocol invariants. The rest of the thesis relies on four invariants.

1. Foreground completion implies that a foreground staging transaction has reached a durably self-contained foreground durable step.

2. Background visibility implies *publish-after-fix*: no staged transaction may be observed before its physical identity is fixed.

3. Replay eligibility implies descriptor-and-commit dual readiness for the referenced foreground staging transaction.

4. Reclamation implies that the corresponding staged and journaled state no longer participates in recovery or replay.

3.4.1 Transaction States and Failure Boundaries

Under the fail-stop model introduced in Chapter 2, a foreground update may be interrupted before it is staged, after it has been staged, after it has been referenced by a durable background journal transaction, or while checkpoint replay is still in progress.

The protocol therefore treats the update as a finite-state object rather than as an undifferentiated durable payload.

A staged update moves through five states:

Unstaged The foreground change exists only in transient host state.

Staged The foreground staging transaction has obtained a valid staging address, and the minimum metadata required for safe interpretation has become durable.

Journaled A background journal transaction has durably recorded a reference to that staged object.

Replay-Eligible The persistence service can validate both the staged descriptor and the corresponding commit block.

Reclaimed Ordered checkpointing has applied the surviving state to the main file-system area, and the temporary space has been released.

Foreground final completion is related to these states but not identical to them. From the perspective of the foreground request, completion is reached when the staging transaction crosses its final commit fence. From the perspective of the persistence service, however, replay eligibility is a later and stricter condition. This distinction allows the system to remain both asynchronous and correct.

Figure 5 projects the five states onto a shared lifetime axis and pins, directly below it, the three questions the protocol must answer: *May the background assembler observe this update?*, *May the foreground* `fsync` *return?*, and *May device-side checkpointing replay this update?* Each observer is *unblocked* at a different event within the same lifetime, so the figure shows exactly where the three boundaries sit, and why collapsing any two of them would either over-serialize the system or expose a not-yet-valid update.

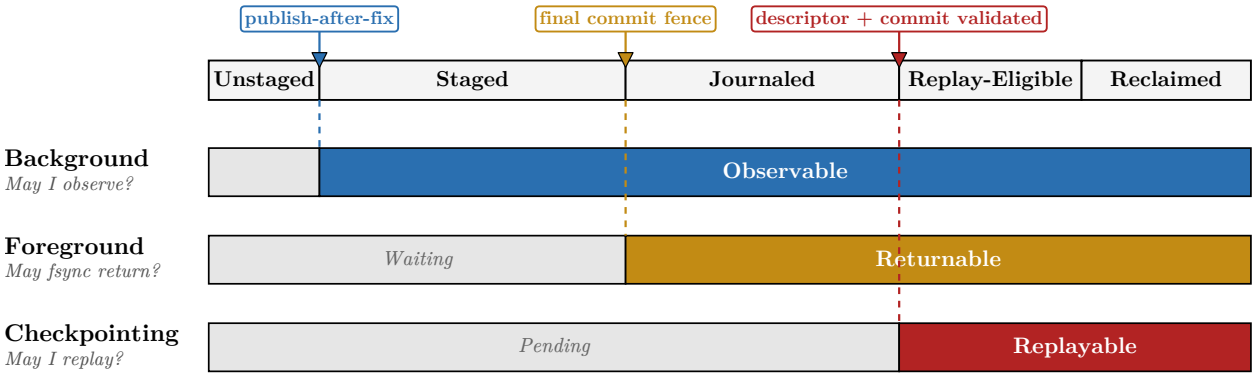

Figure 5: Three protocol boundaries on a staged update's lifetime. The top bar shows the state progression of a single update; three swimlanes below read the same lifetime from the perspective of the background assembler (visibility at *publish-after-fix*), the foreground `fsync` caller (durability at the *final commit fence*), and device-side checkpointing (replay eligibility at *descriptor + commit validated*).

3.4.2 Durability, Visibility, and Replay Eligibility

The protocol separates three boundaries that would otherwise be easy to conflate.

The first is the foreground durability boundary. A foreground `fsync` completes once its staging transaction becomes durably self-contained. The second is the background visibility boundary. A staged transaction may become visible to background assembly before the final file-system state has been produced, but not before the transaction has obtained a valid physical identity in the staging area. The third is replay eligibility. The persistence service may replay a staged transaction only after it can validate both the descriptor and the commit block associated with that transaction.

These three boundaries correspond to three different questions: whether the foreground request is durable enough to return, whether the background path may legally observe the transaction, and whether the persistence service may replay it. Treating them as identical would either over-serialize the system or weaken correctness, and the protocol keeps them separate for exactly this reason.

3.4.3 Publication, Ordering, and Safe Assembly

The host must not publish a staged transaction before its physical identity has been fixed. If publication occurred before address assignment, the background path could capture an object whose durable location was still undefined. The protocol therefore requires *publish-after-fix*: a staged transaction becomes visible to background assembly only after its staging address and minimum durable metadata are in place.

Once published, the transaction must also remain ordered with respect to later journaled updates. Background journaling may widen the scope of durable state, but it must not change the final visible state that a fully ordered execution would have produced. Optimization may remove redundant intermediate work, but it may not change the result.

Wait-Not-Skip. The protocol must also specify what happens when the persistence service reaches a staged transaction that is visible but not yet replay-eligible. The rule is *wait-not-skip*: it waits and revalidates; it does not skip.

Skipping would remove an object that still belongs to the current replay order and would therefore violate the semantics of ordered checkpointing. The persistence service instead repeatedly refreshes the descriptor and revalidates both the descriptor and the commit block until the transaction becomes replay-eligible. This rule is what allows host-side publication to remain decoupled from full replay readiness without turning temporary non-readiness into silent data loss.

3.4.4 Checkpoint Credits and Safe Reclamation

Finally, the protocol must define how space reclaimed by checkpointing is observed by the host. Reclamation determines when the foreground path may safely consume the returned staging and journal credits, and is therefore more than a bookkeeping event.

The system therefore treats reclaimed space as an ordered resource. Credits are

produced monotonically by checkpoint progress and consumed monotonically by later foreground reservations. The low-watermark asynchronous trigger attempts to replenish these credits before the foreground path notices pressure, while the emergency synchronous fallback exists to guarantee forward progress when replenishment falls behind. In both cases, the key requirement is the same: space must not be reused until the protocol can guarantee that the corresponding staged and journaled state no longer participates in recovery or replay.

3.5 Summary

This chapter redesigned persistence along three axes. The split persistence path preserves a short foreground `fsync` path by reorganizing persistence into a short, self-contained durable step and a separate longer background path. Cross-transaction replay reduction keeps the device-side checkpoint path tractable, so that offloaded checkpointing does not simply reintroduce the host-side burden on a weaker processor. The synchronization protocol ties the two layers together with explicit boundaries for durability, visibility, replay eligibility, and reclamation; without them, better performance would come at the cost of weaker correctness.

The next chapter describes how these design choices are realized in the prototype.

Chapter 4 Implementation

4.1 Prototype Overview

This chapter describes the prototype implementation used to realize SIFT. The emphasis now shifts from protocol meaning to object layout, execution structure, and engineering trade-offs under real hardware constraints.

The prototype is written entirely in C. The full system consists of roughly 53 KLOC, but it is not written from scratch for this thesis; it is built by extending an internal pre-existing research codebase[1] for a user-space file system that already provided the basic file runtime, namespace management, block allocation, and journaling substrate needed for end-to-end execution. Of the full 53 KLOC, about 13.3 KLOC correspond to the mechanisms introduced or substantially reworked for this thesis.

From the software-stack perspective, the prototype does not rely on the traditional kernel block layer for its critical persistence path. Instead, it is built on a fully user-space NVMe stack [12, 29]. Both the host side and the device side make explicit use of DMA buffers and interact directly with the underlying NVMe submission and completion queues through a custom user-space storage engine built on top of SPDK. The host-side file runtime directly serves application I/O; the host-side transaction service constructs foreground staging transactions and background journal transactions; and the persistence service executes the longer background path, including background journaling, checkpointing, and space reclamation.

The prototype is deployed on a BlueField-2-DPU-based CSD emulation platform [19]. In this platform, the device-side processor is provided by a BlueField-2 DPU, while the host and device share the persistent namespace exported by an NVMe SSD that supports SR-IOV and namespace sharing [20, 24]. The prototype also uses RDMA- and NVMe-oF-based paths where device-side access requires host-device communication [18,

[1] This codebase is an internal research project that has not yet been publicly released.

21]. This platform preserves the two physical properties that matter most for this thesis: the device-side processor remains significantly weaker than the host CPU, and host and device must coordinate around a shared persistent namespace.

On top of this prototype, the system exposes three configurations: *vanilla*, *offload*, and *coalesce*. *vanilla*, the host-journaling baseline, keeps the longer background path on the host. *offload* moves that path to the device without changing replay semantics. *coalesce* retains the offloaded placement and adds cross-transaction replay reduction, including range-tree-based state maintenance, inode deduplication, and a windowed replay pipeline.

4.2 Host-Side Persistence Path

4.2.1 Persistent Layout and Staging Ring

The shared persistent namespace is divided into three physically distinct regions: the file-system area, the staging area, and the journal area. The file-system area stores the final state. The staging area stores foreground staging transactions. The journal area stores background journal transactions.

The staging area is not a software buffer in host DRAM. It is a persistent circular region on the device. In the prototype, this ring is managed with a head, a tail, and an explicit gap marker used during wrap-around. The ring preserves a single sentinel block at all times, so that `head == tail` denotes the empty state only and never the full state. Allocation therefore maintains the invariant that the requested extent and any wrap-around gap must remain strictly smaller than the available free space, while reclamation clamps over-release so that the tail can never advance beyond the head.

This refinement is not cosmetic. It removes the ambiguity under which a full ring could be mistaken for an empty one and prevents newly allocated staging space from overlapping unreclaimed state.

Figure 6 unrolls the staging area into a single horizontal strip that makes these

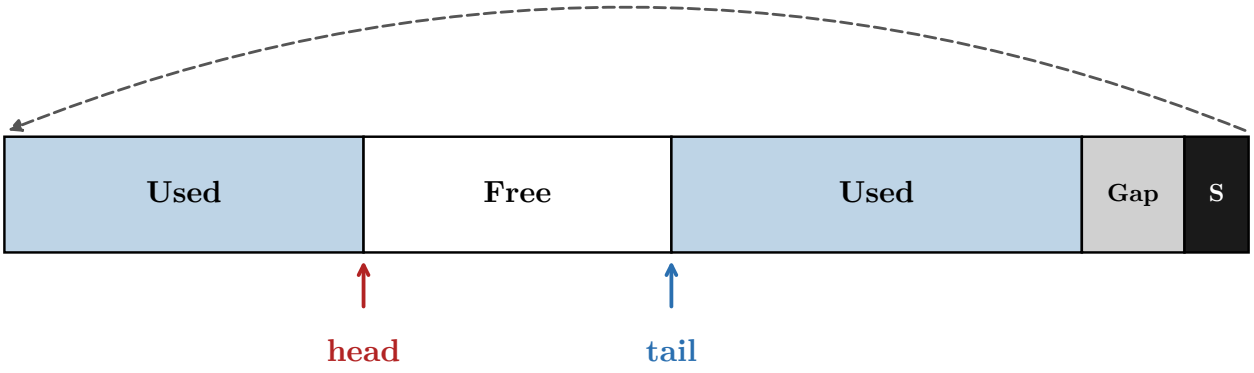

Figure 6: The staging area on the device, unrolled as a ring buffer. head and tail mark the next write and next reclaim positions; a persistent sentinel (S) keeps `head == tail` unambiguously meaning empty; a gap (Gap) appears when the next write does not fit in the remaining end space.

invariants visible at a glance: the two Used segments are the same logical region seen through the wrap, Free is the reclaimable space between them, the small Gap covers the slots that were too short to hold the next write, and S is the sentinel block that keeps `head == tail` from meaning both empty and full.

4.2.2 Foreground Staging Transactions

Whenever an `fsync` is issued, the host constructs a foreground staging transaction for the target file. This object carries not only the data updates of the file, but also the corresponding inode state and transaction metadata. In memory, the prototype assigns separate descriptor and commit buffers to each transaction, each represented as an independent 4 KiB DMA buffer. This physical separation directly supports the protocol boundaries described in Chapter 3.

The prototype follows a *publish-after-fix* discipline. A staged transaction becomes publishable only after its physical staging location has been fixed. The host first establishes a valid staging address and the minimum durable metadata required for safe interpretation. The transaction may then be published into the per-inode staged list even while body I/O is still in progress, but publication never precedes the point at which the transaction has a fixed physical identity. Final completion remains defined by the final

commit fence and the completion flag. In this way, publication can precede full replay readiness without exposing an ill-defined transaction to the background path.

4.2.3 Self-Contained Assembly and Shared State

The self-contained property described in Chapter 3 is realized during transaction assembly. When generating a foreground staging transaction, the host does not only collect newly dirtied data. It also re-captures waiting dirty state that has already been claimed by background assembly but has not yet completed background journaling. As a result, the durable point exposed to foreground `fsync` does not rely on unfinished background work to complete its semantics.

Foreground and background therefore do not communicate through a simple queue. They share journal state, per-file staged state, and the metadata needed to reconnect newly staged transactions with the latest committed background state.

4.2.4 Snapshot Buffers and Background Handoff

When background journal transactions are handed to the persistence service, the prototype does not rely on page-by-page locking or copy-on-write. Instead, it uses a snapshot-style handoff based on contiguous DMA buffers. Background assembly employs separate cursors to pack metadata and dirty data into contiguous buffers. The persistence service then proceeds on top of this captured snapshot rather than on the original host pages.

This additional copy defines a physical snapshot. The host locks the current state only briefly, copies the relevant contents, and then releases the original pages and locks, allowing foreground writers to continue updating the file. The persistence service, in turn, uses the captured snapshot to complete background journaling. Checkpointing then replays durable state from the staging and journal areas rather than from the transient snapshot itself.

4.2.5 Trigger Policy and Reclaim Control

The trigger policy described in Chapter 3 is realized on the host side through two distinct reclaim paths. The first is a low-watermark asynchronous path. When free staging space drops below a configured threshold, the host issues an asynchronous checkpoint request so that credits can be replenished before the foreground path notices pressure. The second is a synchronous fallback path. If a foreground reservation cannot be satisfied, the host issues a synchronous reclaim request and waits for enough credits to be returned.

The prototype keeps separate in-flight gates for these two paths. This separation prevents completion on the synchronous path from accidentally reopening the asynchronous reclaim window. It also ensures that the host can distinguish background replenishment from emergency space recovery.

4.3 Persistence Service Backend

4.3.1 Journal and Checkpoint State

On the longer background path, the persistence backend maintains journal metadata, checkpoint progress, and the queue of committed journal entries that are eligible for checkpointing.

Once a background journal transaction becomes durable, its state is recorded for checkpointing. In *coalesce*, these committed entries are further organized into a ready-prefix batch over the current durable prefix of the background-journal commit queue, so that replay reduction operates over an ordered batch boundary.

For each committed background journal transaction selected for checkpointing, the backend first resolves the referenced foreground staging transactions and reads their staged descriptors and bodies from the staging area. It then reads the journaled updates carried by the background journal transaction from the journal area. These two classes of durable input form the raw replay set for checkpointing, and they are combined under

the staged-before-journaled ordering rule before any replay reduction is applied.

4.3.2 Range Tree Coalescing Engine

The coalescing engine is built around a range tree that maintains the final visible intervals of the current batch. Each node stores the logical interval boundaries together with the source offset, the source type, and the augmented metadata needed to accelerate overlap search. The node therefore represents not just an interval, but a replayable fragment of final visible state.

As new intervals arrive, the engine updates this tree incrementally. It invalidates stale intervals, trims residual fragments, and merges adjacent survivors in place. This avoids the need to reconstruct the replay set from scratch after the batch has been fully collected.

The tree is not implemented as a concurrent data structure. Within a checkpoint batch, updates to the tree are serialized by a single checkpoint execution context on the device. This choice is deliberate. The device-side processor is weak, and the prototype is designed to avoid lock contention inside the coalescing engine itself.

4.3.3 Inode Deduplication Table

The second key device-side structure is the inode deduplication table. This component performs batch-scoped deduplication for inode-related blocks. The prototype uses a composite key based on block address and inode-block identity, resolves collisions through open addressing with linear probing, and treats repeated insertions of the same key as overwrite operations. The result is that only the final version of a given inode block survives within the batch.

Without this structure, the system could reduce stale data replay yet remain bottlenecked by repeated inode-side read-modify-write operations; it is a core component rather than an auxiliary optimization.

4.3.4 Windowed Replay Pipeline

Even after the device has computed the final visible state, it does not immediately emit replay I/O one by one. Instead, the prototype uses a windowed replay pipeline. It first sorts replay operations according to source-side locality, then partitions the working set into windows of bounded size, and finally reorganizes reads and writes within each window before dispatching them.

The current prototype uses a 64 MiB read window. This number is large enough to expose read-side locality and support meaningful I/O regrouping, but small enough to keep the device working set bounded under the memory constraints of the platform. Within each window, the engine merges locally compatible operations before issuing them to storage.

This pipeline does not change replay semantics; it changes the physical shape of replay. By reducing fragmented requests, improving both source-side and destination-side locality, and keeping intermediate state bounded, the pipeline turns a logically reduced replay set into a physically executable sequence of I/O requests. The end-to-end realization of this path corresponds directly to Figure 4 from Chapter 3.

4.3.5 Memory Management and Working-Set Control

The prototype does not employ a dedicated memory pool for tree nodes. Node allocation is handled through standard dynamic allocation. This choice would be dangerous if the engine attempted to materialize arbitrarily large replay state at once. Instead, working-set control is achieved through three other mechanisms: the ready-prefix batch boundary, the shrinkage of visible state during coalescing, and the later partitioning of replay into windows.

This is not the most aggressively optimized allocator design, but it reflects the actual shape of the prototype. Scalability on the device comes not from a single implementation trick but from bounding the live replay state throughout the entire checkpoint pipeline.

4.4 Synchronization and Completion

4.4.1 Descriptor-and-Commit Dual Readiness

The protocol introduced in Chapter 3 is realized in the prototype through explicit readiness checks. A staged transaction does not become replayable merely because the host has published it. The persistence service must validate both the descriptor and the corresponding commit block before replay is allowed to proceed.

This dual-readiness condition is what separates background visibility from replay eligibility in the implementation. It also allows temporary non-readiness to be distinguished from actual corruption.

4.4.2 Host-Side Locking and Shared-State Coordination

The host-side transaction service does not consume staged state without synchronization. The prototype coordinates through a combination of shared-state publication rules and per-file locking. The purpose of these locks is not to serialize the entire persistence path, but to ensure that background assembly never cuts a staged list whose publication boundary has not yet been reached.

Host-side correctness therefore comes not from one global lock, but from a combination of *publish-after-fix*, self-contained transaction assembly, and localized synchronization on shared file state.

4.4.3 Asynchronous and Synchronous Reclaim Gates

As described earlier, the prototype distinguishes asynchronous low-watermark reclaim from synchronous emergency reclaim. This distinction also appears at the synchronization level. The host maintains separate in-flight gates for the two paths, and completion on one path does not satisfy the other.

This prevents false reopening of the asynchronous reclaim window and ensures that the host never interprets synchronous fallback progress as successful background

replenishment.

4.4.4 Request-Level Completion Barrier

At the I/O completion boundary, synchronous completion is defined more strictly than raw completion counting. The upper layers do not return merely because a matching number of lower-level completion events has been observed. They return only after the corresponding requests have been marked complete. This avoids early return caused by completion overcounting or mixed completion events and ensures that upper-layer durability checks do not observe partially finished replay I/O.

***Wait-Not-Skip* in the Prototype.** When the persistence service encounters a staged transaction that is visible but not yet replay-eligible, it does not skip the entry. Instead, each waiting round re-reads the staged descriptor and re-validates both the descriptor and the commit block. In the prototype, this behavior is implemented with a 1 ms short-sleep polling loop rather than a busy spin or a complex event-driven wait. This is an engineering compromise, but it faithfully realizes the *wait-not-skip* rule defined in Chapter 3.

4.5 Implementation Boundaries and Trade-offs

The prototype includes substantial runtime statistics and debug support. During checkpoint execution, the persistence service records the amount of raw and surviving state, inode overwrite counts, batch size, window size, and the final I/O shape. These statistics are used directly in Chapter 5 and also make the internal behavior of the checkpoint engine observable.

At the same time, the implementation has clear boundaries. Tree nodes are dynamically allocated rather than coming from a dedicated pool. Tree maintenance is serialized within a checkpoint batch rather than implemented as a concurrent data

structure. Host-side reclaim still includes an emergency synchronous fallback path, which becomes visible under pressure. None of these choices invalidate the design claims of Chapter 3, but they do define the actual engineering shape of the prototype and therefore matter for the interpretation of the results.

4.6 Summary

This chapter has turned the architecture of Chapter 3 into an executable prototype. It described the host-side staging area, foreground staging transactions, snapshot-style background handoff, and the trigger policy that governs reclaim. It then showed how the persistence service maintains journal and checkpoint state, how the range-tree-based coalescing engine and inode deduplication table reduce replay work, and how the windowed replay pipeline turns a reduced replay set into executable I/O. Finally, it explained how the synchronization protocol is realized in code through *publish-after-fix*, descriptor-and-commit dual readiness, separate reclaim gates, and request-level completion barriers.

The next chapter evaluates these three configurations and examines the operating regimes they enter under different workloads.

Chapter 5 Evaluation

5.1 Evaluation Goals and Experimental Setup

5.1.1 Evaluation Questions

This chapter addresses three questions.

1. Under online checkpoint pressure, what operating regimes do *vanilla*, *offload*, and *coalesce* enter? This question speaks directly to the main systems claim of the thesis: both checkpoint placement and replay semantics shape foreground throughput, `fsync` latency, and host-side resource consumption.

2. How much replay work does *coalesce* actually remove? The design chapters have already argued that the value of coalescing is not that it executes the same replay path slightly faster, but that it changes the amount of replay work the device must perform in the first place. Here, block-level and inode-level statistics are used to quantify that reduction directly.

3. What remains when user-level rewrite overlap disappears? A convincing evaluation cannot stop at the most favorable workload. It must also explain what happens in a more demanding regime, where user data no longer rewrites the same logical blocks and coalescing must rely on residual protocol-level redundancy rather than on heavy overwrite locality alone.

5.1.2 Platform and System Configurations

All experiments were conducted on one fixed host/device platform summarized in Table 2. The setup combines a dual-socket Intel Xeon Gold 5218 host, a BlueField-2 DPU as the device-side processor, a shared Samsung PM1735 namespace, and an SPDK-based user-space NVMe path [8, 17, 26]. These components preserve the asymmetric host/device split and the shared persistent namespace that matter most to this thesis.

Platform Item	Value
Host CPU model / sockets	2× Intel Xeon Gold 5218 @ 2.30 GHz
Total host cores / threads	32 physical cores / 64 hardware threads
Host DRAM	128 GiB
SSD model / capacity	Samsung PM1735 NVMe SSD, 3.2 TB
SR-IOV / namespace sharing	Supported and enabled for the shared namespace
Vendor peak SSD bandwidth	8.0 GB s^{-1} read, 3.8 GB s^{-1} write
DPU model	NVIDIA BlueField-2 DPU
DPU cores	8 Armv8-A72 cores
DPU memory	16 GiB DRAM
DPU eMMC capacity	64 GiB
Host–device network bandwidth	100 Gbit s^{-1}
Storage-engine I/O workers	8 threads for 8 NVMe VF queues

Table 2: Evaluation platform. The table collects the fixed host, device, and storage parameters shared by all experiments.

This chapter compares three configurations: *vanilla*, *offload*, and *coalesce*. As introduced in Section 1.4, all three share the same split persistence organization: foreground `fsync` first reaches a foreground staging transaction and returns after that short durable step, while background journaling and checkpointing proceed separately. Their counterpart relationship is summarized in Table 3. *vanilla* is the host-journaling baseline for this organization and keeps the longer background path on the host. *offload* moves that path to the device while leaving replay semantics untouched. *coalesce* builds on *offload* and adds cross-transaction replay reduction for device-side checkpointing. The differences observed here can therefore be attributed directly to background-path placement and replay semantics rather than to unrelated protocol changes.

Configuration	Foreground durable step	Longer background path	Checkpoint semantics	Replay reduction
vanilla	Foreground staging transaction; `fsync` returns after staging durability	Host	Ordered checkpointing over raw durable staged and journaled history	None
offload	Same as *vanilla*	Device	Same ordered checkpointing as *vanilla*	None
coalesce	Same as *offload*	Device	Ordered checkpointing over coalesced final visible state	Range-tree coalescing and inode deduplication

Table 3: Evaluation configurations and their counterpart relationship. All three share the same split persistence organization, staging and journal layout, trigger policy, and synchronization protocol; they differ only in where the longer background path executes and whether checkpoint replay is reduced before final convergence.

5.1.3 Workloads and Namespace Profiles

Two classes of benchmark are used in this chapter, and their exact file sizes, hotsets, write granularities, and shared-namespace profiles are summarized in Table 4.

The first is an online pressure benchmark. One foreground thread issues 4 KiB writes followed immediately by `fsync` within a small hot region, while 2 background writers issue a background `fsync` after every $i \in \{32, 64, 128, 256\}$ writes. This background `fsync` interval corresponds to durable burst sizes of 128 KiB, 256 KiB, 512 KiB, and 1 MiB and, together with the small online namespace profile in Table 4, makes reclaim visible on the critical path.

The second is a manual checkpoint benchmark. It first constructs a durable backlog under controlled conditions and then triggers an explicit manual checkpoint, isolating the checkpoint engine from the online foreground path.

For the high-overlap workload, 4 threads operate on hot regions that repeatedly overwrite the same logical range, creating a 20 GiB durable backlog whose replay set is highly compressible.

For the sequential no-reuse workload, the aggregate write volume and `fsync` interval

Parameter	Online	High-Overlap	Sequential No-Reuse
Foreground file size	64 MiB	—	—
Foreground hotset	8 MiB	—	—
Background file size	512 MiB per BG thread	—	—
Background hotset	64 MiB per BG thread	—	—
Manual file size	—	2 GiB per thread	6 GiB per thread, pre-allocated
Manual active region	—	64 MiB per thread	5 GiB per thread
Threads	1 foreground + 2 background	4 threads	4 threads
Write size	4 KiB	4 KiB	4 KiB
Access pattern	Uniform random within hotsets	Uniform random within hotsets	Sequential, no logical-block reuse
Writes per `fsync`	32, 64, 128 and 256 writes	256 writes	256 writes
Warmup	5 s	0	0
Run length	30 s measurement window	20 GiB durable backlog + manual checkpoint	20 GiB durable backlog + manual checkpoint
Namespace profile	FS 8 GiB; staging 2 GiB; journal 2 GiB	FS 60 GiB; staging 60 GiB; journal 60 GiB	FS 60 GiB; staging 60 GiB; journal 60 GiB

Table 4: Workloads and namespace profiles. The table consolidates the benchmark parameters and shared-namespace sizes referenced throughout the evaluation.

remain the same, but each thread advances sequentially through a pre-allocated file without reusing logical blocks. In this workload, user-level rewrite overlap disappears, while protocol-level duplication and repeated inode updates remain.

5.1.4 Metrics and Reporting Method

This chapter reports four classes of metric.

1. The first is foreground throughput, which captures whether the system continues to make progress online.

2. The second is the distribution of `fsync` latency, with emphasis on p50, p99, p99.99, and max, so that common-case behavior can be separated from rare but destructive checkpoint stalls.

3. The third is replay-work statistics, including block-level and inode-level raw work and surviving work, which quantify the actual reduction achieved by coalescing. The prototype also reports the number of logical update records, which appear in the logs as `tags`. These are internal per-update records used by the checkpoint engine; they are not storage-controller I/O tags.

4. The fourth is host CPU cost. We report both raw CPU utilization and a throughput-normalized CPU metric.

Unless otherwise noted, all benchmarks were run five times, and the median run is reported.

For CPU efficiency, we define normalized host CPU cost as

$$\text{CPU Cost} = \frac{\text{Average host CPU utilization (\% of the whole machine)}}{\text{Foreground throughput (Kops/s)}} \tag{1}$$

and report it in units of CPU% per Kops/s. This metric expresses how many percentage points of whole-machine CPU are consumed for each Kops/s of foreground throughput.

5.2 End-to-End Online Performance

5.2.1 Throughput, Tail Latency, and Host CPU Cost

The online evaluation is organized as a sweep over background `fsync` interval i. We examine $i = 32, 64, 128, 256$, corresponding to background durable bursts of 128 KiB, 256 KiB, 512 KiB, and 1 MiB. Among these, $i = 128$ serves as the representative operating point for detailed discussion because it lies in the middle of the sweep and all three configurations complete the run while still exposing clear behavioral differences. The conclusions in this section, however, are not tied to a single point. They are supported by the shape and relative ordering of the full sweep.

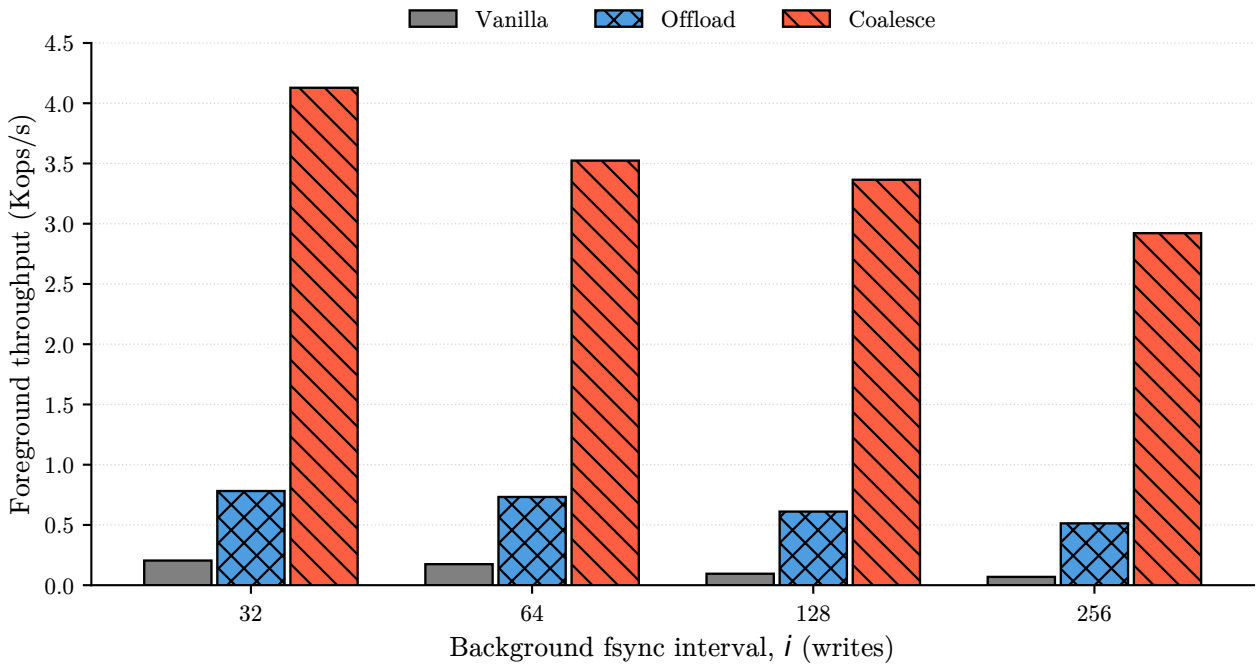

Figure 7: Foreground throughput across background `fsync` interval i. *coalesce* remains in the multi-Kops/s regime while *offload* and *vanilla* remain substantially lower throughout the sweep.

Foreground Throughput. Figure 7 shows foreground throughput across the online sweep. Across all four points, *coalesce* remains in the 2.9–4.1 Kops/s range, *offload* stays in the 0.51–0.78 Kops/s range, and *vanilla* drops further to 0.07–0.20 Kops/s. The separation is stable throughout the sweep.

At the representative point $i = 128$, *vanilla* completes only about 2,850 foreground operations within the 30 s measurement window, corresponding to roughly 0.095 Kops/s. *offload* raises that figure to about 0.611 Kops/s, and *coalesce* raises it further to about 3.365 Kops/s. Relative to *vanilla*, *offload* already improves throughput by about 6.4×. Relative to *offload*, *coalesce* provides a further improvement of about 5.5×. Compared directly against the host-journaling baseline, *coalesce* sustains more than 35× the foreground throughput.

These absolute numbers should not be read as a measure of the PM1735's standalone device capability. The PM1735's vendor-rated 8.0 GB/s read and 3.8 GB/s write reflect

ideal large-block, deep-queue, sequential behavior. The workload here is intentionally different. The foreground is a single-threaded stream of 4 KiB writes followed immediately by `fsync`, while two background threads continually generate durable bursts in the same shared namespace and on the same device queues. The online profile further compresses staging and journal space to 2 GiB each, making reclaim visible on the critical path. In *vanilla*, this pressure is absorbed directly by the host-side longer background path; in *offload* and *coalesce*, the same pressure is shifted onto the device-side background path, including replay and replenishment. The resulting throughput therefore reflects not the device's peak bandwidth in isolation, but the amount of foreground progress that remains possible when a latency-sensitive `fsync` stream is exposed to sustained background persistence pressure.

`fsync` Tail Latency. Throughput shows whether the system continues to advance; `fsync` latency shows how and when it stalls. Figure 8 reports p99, p99.99, and max across the sweep. The key point is that ordinary percentiles do not fully explain the behavior of the system under checkpoint pressure.

At $i = 128$, the p50 values of *offload* and *coalesce* are almost identical, about 53 μs and 54 μs, respectively. Even *vanilla*, at roughly 208 μs, is not slow enough at the median to explain its very low overall throughput. The decisive difference lies in the extreme tail. At the same operating point, the p99.99 latency reaches about 24.1 s in *vanilla*, 8.11 s in *offload*, and only about 0.675 s in *coalesce*. The corresponding max values follow the same pattern.

Host CPU Cost. Figure 9 reports host CPU cost across the online sweep. At $i = 128$, the average host CPU utilization is about 4.75% for *vanilla*, 3.72% for *coalesce*, and only 1.40% for *offload*. On its own, this might suggest that *offload* is the most economical configuration.

The normalized metric shows otherwise. At the same operating point, the host CPU

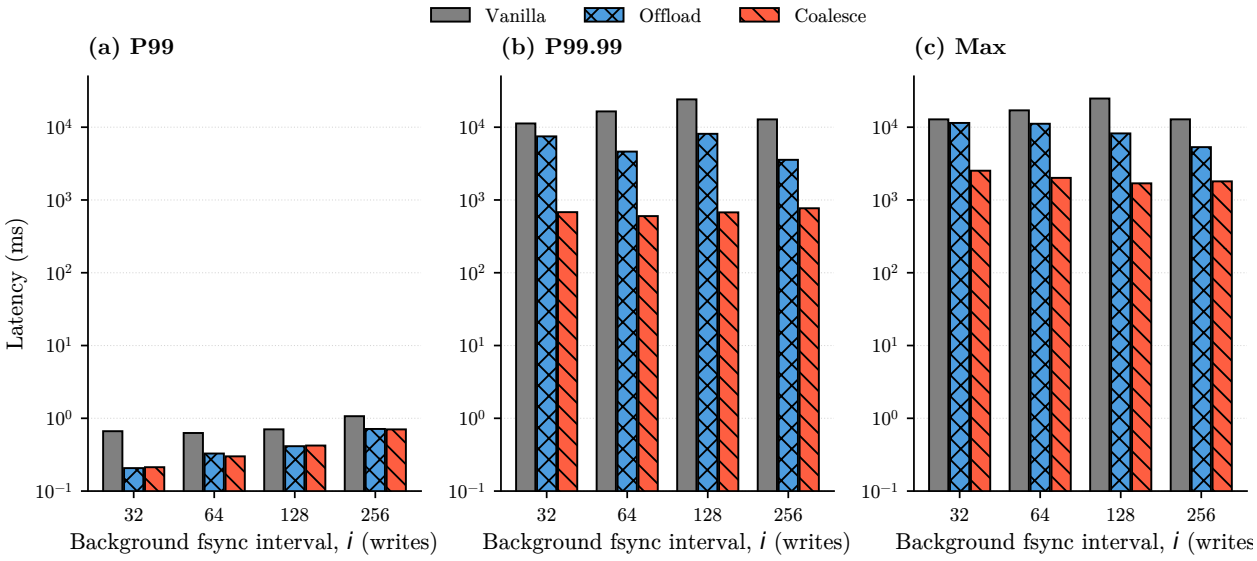

Figure 8: `fsync` latency tails across background `fsync` interval i. *coalesce* suppresses the extreme-tail stalls that dominate runs once checkpoint pressure becomes visible.

cost is about 49.96 CPU% per Kops/s for *vanilla*, 2.29 CPU% per Kops/s for *offload*, and 1.11 CPU% per Kops/s for *coalesce*. Across the full sweep, *coalesce* remains in the 0.93–1.32 CPU% per Kops/s range, while *offload* lies in 2.30–5.43 CPU% per Kops/s, and *vanilla* remains far above both.

5.2.2 Interpreting the Three Operating Regimes

Across the online sweep, *vanilla*, *offload*, and *coalesce* occupy three distinct operating regimes. At the representative operating point, median `fsync` latency still ranges only from tens to hundreds of microseconds, yet foreground throughput spans from 0.095 to 3.365 Kops/s. The key difference therefore lies not in the common-case cost of the short foreground durable step alone, but in how checkpoint pressure is absorbed after that step and how often it turns into long blocked episodes.

The first separation, between *vanilla* and *offload*, comes from background-path placement. Both configurations preserve the same split persistence organization and return after foreground staging durability (Table 3). What changes is where the longer background path is executed. In *vanilla*, background journaling, checkpointing, and

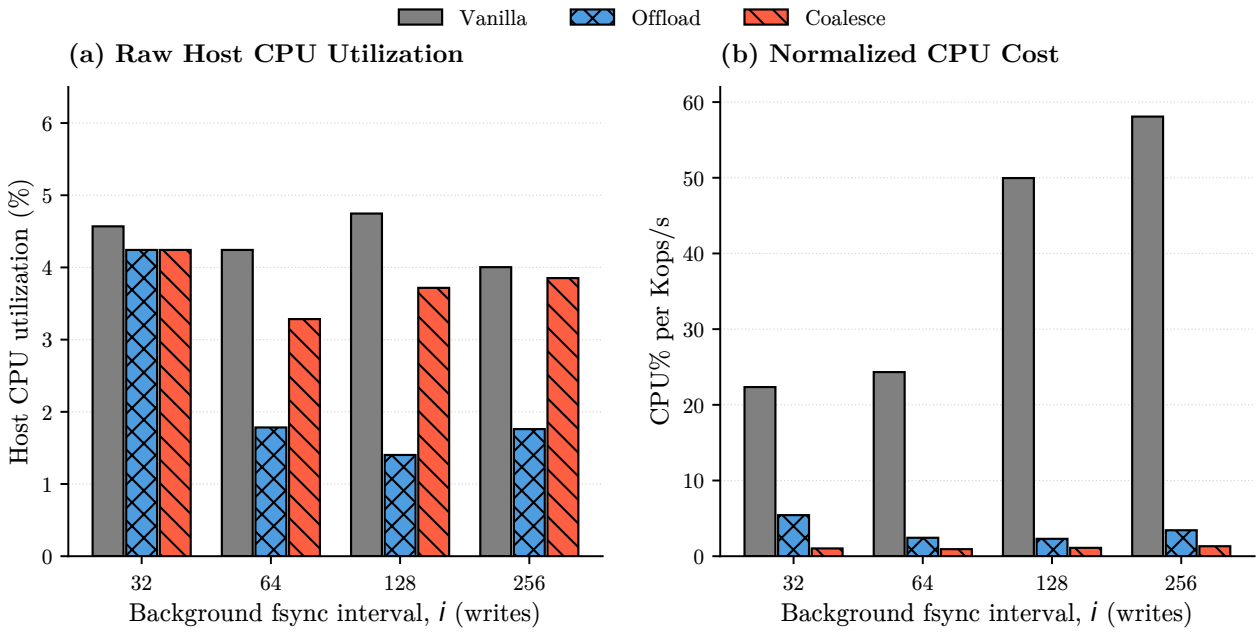

Figure 9: Host CPU cost across background `fsync` interval i. Offloading releases raw host CPU burden, while coalescing converts more of that released budget into useful foreground work.

credit replenishment remain on the host and continue to contend with the foreground for the same local CPU, cache hierarchy, memory bandwidth, and storage path. In *offload*, that same longer path is shifted to the device.

The second separation, between *offload* and *coalesce*, comes from replay semantics on the already-offloaded checkpoint path. *offload* still preserves raw ordered replay, whereas *coalesce* reduces the replay and reclaim work that survives into checkpointing. The separation therefore unfolds in two steps. Moving the longer background path off the host removes a large fraction of host-side checkpoint interference, and reducing replay on that already-offloaded path widens the gap further by shortening replay and reclaim on the device.

Beyond the four individual data points, the sweep also shows how the background `fsync` interval reshapes the time profile of persistence pressure. As the interval shrinks from 256 and 128 writes to 64 and 32, the corresponding background durable bursts shrink as well, but the system does not simply slow down in a uniform way. Instead,

persistence pressure becomes more bursty, and that burstiness makes the differences among the three configurations more visible.

What remains stable is the ordering of the regimes. *coalesce* stays in the Kops/s range throughout the sweep; *offload* remains well below that level but above the host-journaling baseline; *vanilla* stays in the most fragile regime and is the first to approach collapse as the pressure becomes more bursty. The background `fsync` interval is therefore an operating-regime control knob rather than an incidental benchmark parameter.

5.3 Actual Replay Reduction Under a High-Overlap Backlog

Constructing a High-Overlap Backlog. The online sweep suggests that the remaining *offload-coalesce* gap is largely shaped by replay work on the device-side checkpoint path. To isolate that replay-side effect, we return to the checkpoint engine itself and examine how much raw replay work survives once coalescing is applied.

For this purpose, we construct a high-overlap manual checkpoint backlog. Each of four threads operates on a 2 GiB file but writes within only a 64 MiB active hotset, using a uniform random access distribution. The aggregate write volume is 20 GiB, after which a single manual checkpoint is triggered. This workload intentionally creates heavy overwrite locality so that the potential gain from replay reduction is fully exposed.

Block-Level Reduction. Under this high-overlap backlog, raw block-level replay work consists of staged blocks from foreground staging transactions and journaled blocks from background journal transactions. In total, the raw block replay set contains more than 5.26 million block-level entries before reduction, while the surviving replay set contains only about 65.7 thousand blocks. This corresponds to roughly 98.7% block-level replay reduction.

Inode-Level Reduction. The same backlog reveals an even more dramatic reduction on the metadata path. Raw inode replay contains 20,492 inode-related updates, while the

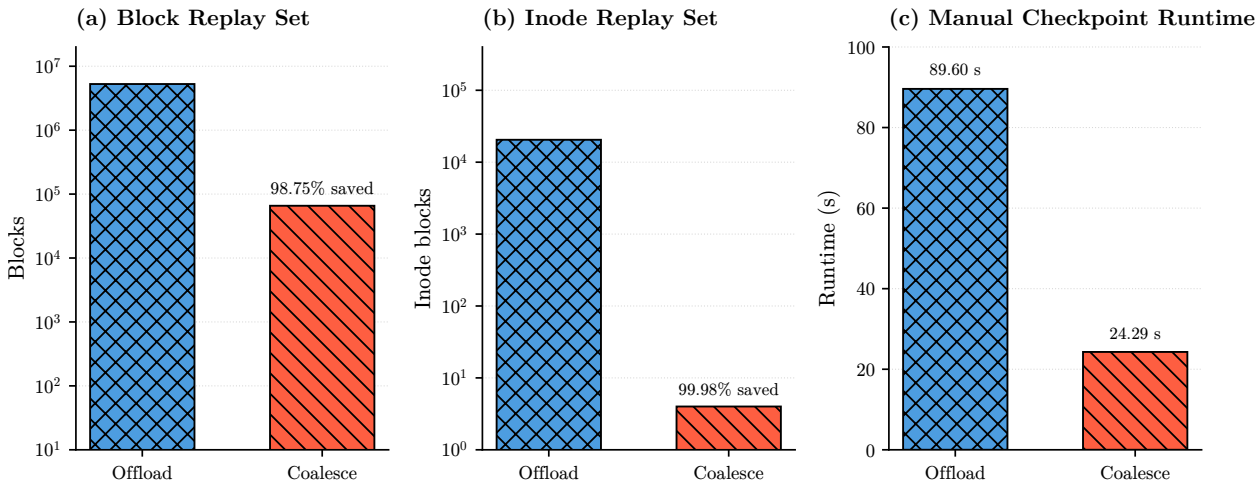

Figure 10: Replay work reduction under a high-overlap manual backlog. Most raw replay work disappears before device-side replay begins, and the reduced replay set translates directly into shorter manual checkpoint time.

surviving final inode state shrinks to only 4 entries, that is, one final inode state per file under this four-file workload. About 99.9% of inode replay work is therefore eliminated, and the surviving set reaches the physical minimum implied by the workload.

Checkpoint replay is not dominated by data blocks alone: repeated inode updates remain a major source of amplification, and without inode folding the system could reduce stale data replay yet remain bottlenecked by inode-side read-modify-write operations. Coalescing removes both.

Figure 10 summarizes these block-level and inode-level reductions together with their translation into manual checkpoint runtime.

Translating Work Reduction into Time. The reduction also translates directly into checkpoint time. On the same backlog, manual checkpoint runtime falls from 89.60 s in *offload* to 24.29 s in *coalesce*. Put differently, *coalesce* shortens manual checkpoint runtime by 72.9%, or equivalently, *coalesce* achieves a 3.69× speedup over *offload*.

5.4 Sequential No-Reuse Workload

What Remains When User-Level Rewrite Overlap Disappears. The high-overlap backlog shows the most favorable case for *coalesce*. It does not answer what happens once user-level rewrite overlap disappears. To isolate that regime, we construct a sequential no-reuse workload. Four threads each advance through 5 GiB of useful writes in a 5 GiB active region, with a pre-allocated 6 GiB file per thread. Aggregate durable writes remain 20 GiB, matching the previous section.

The clearest indicator that user-level overlap has been removed is the count of logical update records, reported by the prototype logs as `tags`. In the high-overlap case, the reduction in these update records is about 98.7%. In the sequential no-reuse workload, that number falls to roughly 6.1%. This confirms that user-level rewrite overlap has largely disappeared.

Residual Gains and Boundary Overhead. Even then, the gain does not fall to zero. The surviving block replay set still shrinks from roughly 10.49 million blocks to about 5.24 million, that is, by about half. Inode savings remain close to complete. The reason is that user-level rewrite overlap is not the only source of redundancy in the system. The split persistence path still creates staged and journaled representations of the same durable update, and repeated inode updates to the same files remain highly compressible.

This is also the point at which tree overhead becomes visible. Under the sequential no-reuse workload, manual checkpoint runtime is 80.35 s for *coalesce* and 92.81 s for *offload*. In absolute terms, *coalesce* shortens runtime by 13.4%; equivalently, *coalesce* maintains a 1.15× speedup over *offload*. This is much smaller than the gap seen in the high-overlap case, which makes the engineering cost of the range-tree-based engine much more apparent. But the result is still positive. Even without user-level rewrite overlap, protocol-level duplication and inode folding continue to justify the coalesced path.

Figure 11 collects this boundary case into one view, contrasting the

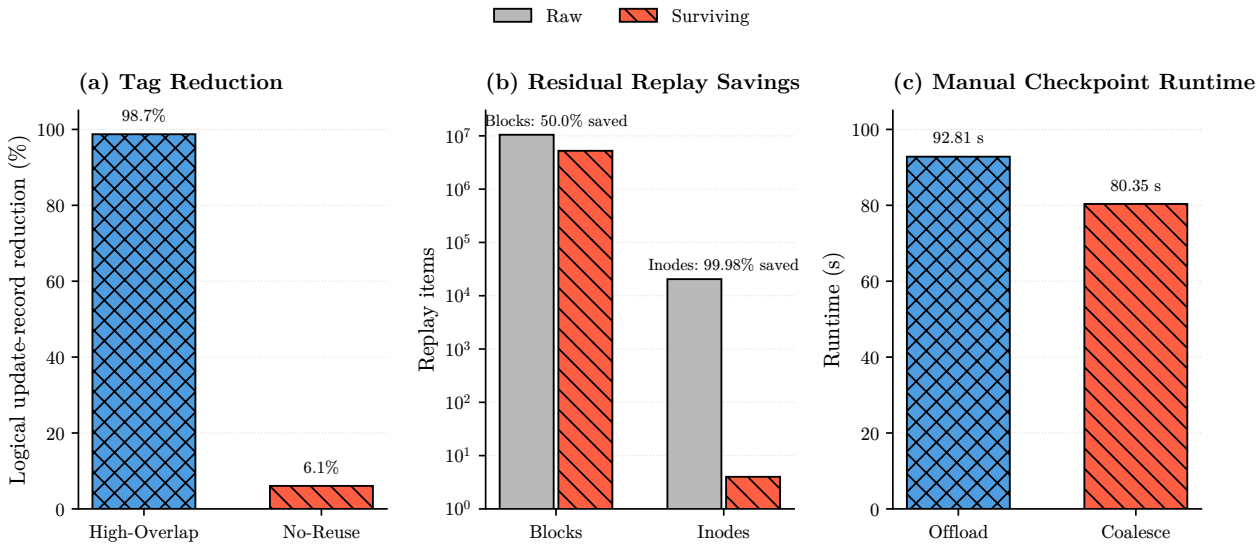

Figure 11: Sequential no-reuse workload. User-level overlap largely disappears, but protocol-level duplication and inode folding still leave a smaller positive gain for *coalesce*.

near-disappearance of user-level overlap with the residual replay savings and the smaller but still positive runtime gain.

5.5 Summary

This chapter has answered three questions.

First, across the online sweep over background `fsync` interval i, *vanilla*, *offload*, and *coalesce* occupy three clearly separated operating regimes. *coalesce* remains in the Kops/s regime, *offload* is confined to a lower-throughput regime, and *vanilla* remains in the most fragile host-side regime.

Second, under a high-overlap manual backlog, *coalesce* removes about 98.7% of raw block replay and about 99.9% of raw inode replay, while shortening manual checkpoint runtime by 72.9%.

Third, under a sequential no-reuse workload, the gain of coalescing contracts sharply but does not vanish. Once user-level rewrite overlap disappears, protocol-level duplication and inode folding still sustain a 13.4% net runtime advantage.

Taken together, these results make the main lesson of the thesis clear. Moving the

longer background path to the device already changes the interference profile by removing a large fraction of host-side checkpoint pressure, but it does not eliminate the replay burden; it only changes where that burden is paid. What lifts the system from the lower-throughput offloaded regime into the sustained Kops/s regime is the reduction of checkpoint replay itself.

Chapter 6 Related Work

6.1 User-space File Systems and Host-side Path Optimization

A large body of work has shown that user-space file systems and kernel-bypass storage paths can significantly reduce the fixed overheads of the traditional kernel I/O path [4, 10, 13, 15, 32, 33]. Early frameworks such as FUSE established the basic mechanism for moving file-system logic into user space, and later work studied its performance costs in detail. Subsequent systems improved kernel–user-space communication, as in RFUSE, or bypassed more of the conventional kernel path altogether, as in SplitFS, Strata, and BypassD. Collectively, these efforts pushed one clear direction: make the foreground path light enough that applications can exploit fast storage devices more directly.

However, these works do not directly address the core bottleneck studied in this thesis. Their main target is the foreground path: reducing read/write overhead, avoiding context switches, or enabling faster user-space access to shared storage. Even when they begin to touch file-grained transactions, operation logs, or background digest mechanisms, their primary concern remains host-side foreground efficiency or host-side persistence overhead. By contrast, this thesis asks a narrower question: once a high-performance user-space file system has already made the foreground path lightweight, can the longer background persistence path re-emerge as the dominant consumer of host CPU? If so, how should a short foreground durable step be preserved while later checkpoint replay is reduced on the device side?

From this perspective, we do not replace this line of research; rather, we build on it by shifting attention from the foreground path to the persistence bottleneck that remains behind it.

6.2 Computational Storage and Device-Side File-System Offloading

A second line of closely related work comes from computational storage and device-side file systems [9, 11, 16, 35–37]. DevFS embeds the file system inside the storage device and emphasizes direct access with firmware-level execution. FusionFS further introduces fused operations that combine multiple I/O and data-processing actions into larger device-side execution units. λ-IO raises the problem to the level of a unified host/device I/O stack, focusing on how computation and storage resources should be coordinated across the two sides. CETOFS studies host-server collaboration for remote storage and offloads selected file-system responsibilities toward the storage side. More broadly, computational storage and near-data processing systems have explored how storage-side computation can absorb part of the work that would otherwise remain on the host.

In that broad sense, this thesis clearly belongs to the same family of ideas. But the focus here is narrower and more specific. Existing device-side work typically treats offloading as a general near-data processing opportunity: the goal may be direct access, fused operations, collaborative caching, or host/device scheduling. This thesis instead focuses on the longer background persistence path and, within it, on checkpointing as the replay-heavy convergence stage. More importantly, it does not stop at placement alone. The key question is whether direct offloading preserves a short foreground path and whether replay semantics must also change once convergence runs on a weaker device-side processor.

We therefore differ from prior device-side work not by offloading per se, but by showing that direct offloading of the longer background persistence path is insufficient to preserve a short foreground path, and that device-side checkpoint replay must be reduced if offloaded execution is to remain effective.

6.3 Journaling, `fsync` Optimization, and Logical Reduction

A third line of work is especially close to this thesis: research on journaling, `fsync` latency, and write-amplification reduction [13, 23, 28]. iJournaling argues for file-level journaling in order to reduce `fsync` latency by committing only the file-relevant transaction rather than a larger compound transaction. Fast Commit and related ext4 work similarly pursue lighter-weight logical journaling paths, recording only the minimal metadata delta needed to reconstruct the final state. More broadly, Strata's operation log and digest mechanism already embodies the idea that persistence should not simply replay raw history blindly; instead, updates can be accumulated on a fast layer and later converged into a more efficient representation on a slower layer.

These works are not orthogonal to this thesis. On the contrary, they provide a major part of its intellectual starting point. Strata in particular deserves careful treatment here. Its digest mechanism already performs a form of cross-transaction aggregation to smooth the movement of updates from a fast layer to a slower one. That is not something we replace; it is something we extend. The difference is that Strata still assumes a host-resident digest path. Its purpose is to improve host-side persistence behavior in a cross-media file system, not to make checkpointing itself into an offloaded background service.

We extend this line in a specific direction. The contribution of this thesis is not the observation that repeated updates should be aggregated, but the observation that under a split persistence path with a weak device-side processor, replay reduction is no longer merely an optimization for reducing write amplification or shortening host-side commit latency: it becomes necessary for keeping device-side checkpoint convergence efficient once the longer background path has been offloaded. We therefore inherit the insight behind logical journaling and background convergence, and push it into the setting of split persistence and offloaded checkpoint replay.

6.4 Position of This Thesis

Taken together, prior work has answered three important but still separate questions. User-space file systems have shown how to make the foreground path lighter. Computational storage and device-side file systems have shown how to move some functionality closer to the device. Logical journaling and `fsync`-oriented work have shown how to reduce the cost of host-side commit and background convergence.

The contribution of SIFT lies precisely at the intersection of these three lines of work. It does not simply optimize user-space performance, nor does it merely offload arbitrary storage functionality to the device, nor does it only improve host-side journaling. Instead, SIFT connects these ideas along one specific persistence organization: it first separates a short foreground durable step from the longer background persistence path through split persistence; it then places that longer path on the device, where background journaling and checkpointing proceed and where checkpointing becomes the replay-heavy convergence stage; and finally, it reduces device-side checkpoint replay through cross-transaction coalescing while enforcing the required durability, visibility, replay eligibility, and reclamation boundaries through an explicit synchronization protocol.

Rather than extending any one of these directions in isolation, SIFT provides an integrated persistence design at their intersection.

Chapter 7 Conclusion and Future Work

7.1 Conclusion

This thesis has focused on a systems problem that is becoming increasingly difficult to ignore. As NVMe SSDs, persistent memory, and related high-speed storage hardware continue to improve, device-side latency keeps shrinking. At the same time, user-space file systems are often forced to rely on polling-based I/O and aggressive concurrency in order to exploit such hardware fully. Under these conditions, the host CPU is no longer merely a fast control plane; it becomes the scarcest resource in the system, and often the first one to be exhausted. The pressing question is therefore no longer whether the device is fast enough, but where host CPU time is actually being spent.

We argue that the longer background persistence path, and especially its checkpointing stage, becomes a dominant source of host-side interference under strict `fsync` semantics. It continuously consumes host CPU, and it also harms foreground latency and throughput through bursty background execution. For this reason, the longer background persistence path, and especially checkpointing as its replay-heavy convergence stage, becomes a natural target for offloading in a high-performance user-space file system. At the same time, we have shown that a naive offloading strategy is not viable. If checkpointing is simply moved to the device while its replay semantics remain unchanged, the bottleneck is not removed but merely transferred from the host to the device. On a device processor with weaker compute capability and tighter memory limits, the cost of mechanically replaying the full durable history quickly overwhelms the benefit of offloading.

To address this problem, we have proposed and implemented SIFT, a layered persistence architecture centered on checkpointing. At the architectural level, it introduces a split persistence path that separates a short foreground durable step from the longer background persistence path, so that foreground `fsync` completes through a

short, explicit, and self-contained durable step. At the algorithmic level, it equips the device-side checkpoint path with a cross-transaction coalescing engine that combines range-tree-based coalescing with inode deduplication, thereby changing checkpointing from "replay all durable updates" into "replay only the final visible state of a committed batch." At the protocol level, it defines explicit boundaries for durability, visibility, replay eligibility, and safe reclamation, so that the system preserves crash consistency under a fail-stop model even though foreground staging, background journaling, and checkpointing proceed asynchronously across host and device.

The evaluation demonstrates that SIFT changes the operating regime observed under checkpoint pressure, not just a few local metrics. Under sustained online checkpoint pressure, end-to-end results show that *coalesce* delivers more than 35× the foreground throughput of the host-journaling *vanilla* baseline, and, at the representative online operating point, it releases more than 50% of the raw host-side CPU burden relative to that baseline. Under a high-overlap manual checkpoint workload, the coalesced checkpoint path removes about 98.7% of raw block replay and about 99.9% of raw inode replay, shortening checkpoint runtime by about 72.9%. More importantly, even under a sequential no-reuse workload, where user-level rewrite overlap largely disappears, the system still retains a positive net gain: *coalesce* remains about 1.15× faster than *offload*. The benefit of this design therefore does not depend exclusively on the most favorable overwrite-heavy case; it also survives through protocol-level duplication removal, inode folding, and improved replay shape.

The main lesson is that, in a heterogeneous storage system, changing only where work executes is not enough. That merely transfers the bottleneck from one processor to another. Effective offloading requires reducing the work itself. In the case studied here, that means the background persistence path cannot simply be moved to the device. It must first be detached from the foreground path, and then reduced into a replay set that the device must actually sustain. Only under those conditions does offloading become a

meaningful systems answer rather than a relocation of the same problem.

7.2 Future Work

Although we have presented a complete solution around checkpointing, several natural directions remain open.

The first is adaptive offloading. The current design uses a static offloading structure. A natural next step is to make the placement of the longer background path, and in particular its checkpointing stage, responsive to device-side load, available memory, and background queue pressure. Offloading therefore need not remain a fixed binary choice; it may become an adaptive runtime policy.

The second is finer-grained coalescing. The current design already reduces replay work substantially, but it still relies on a unified tree-based coalescing path. Another direction is to vary the strength of coalescing based on workload structure. Under heavy overlap, full interval-level coalescing is clearly worthwhile. Under weaker overlap, a lighter-weight policy, such as retaining inode deduplication while avoiding more expensive interval maintenance, may further reduce boundary-case overhead.

More broadly, this thesis should be read as a starting point rather than the final word on checkpointing. In a storage system shaped by fast devices and heterogeneous execution, checkpointing is better treated as a first-class systems object to be designed and optimized explicitly, and used to reshape the operating regime of the entire file system.

References

[1] Baldassin, A., Barreto, J., Castro, D., and Romano, P. Persistent Memory: A Survey of Programming Support and Implementations. ACM Computing Surveys, Jul. 2021.

[2] Barbalace, A., and Do, J. Computational Storage: Where Are We Today? In Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR'21) (Jan. 2021).

[3] Bjørling, M., Axboe, J., Nellans, D., and Bonnet, P. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-core Systems. In Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13) (Jul. 2013).

[4] Cho, K.-J., Choi, J., Kwon, H., and Kim, J.-S. RFUSE: Modernizing Userspace Filesystem Framework through Scalable Kernel-Userspace Communication. In Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24) (Feb. 2024).

[5] Compute Express Link Consortium. Compute Express Link Specification Revision 2.0. `https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf`, Oct. 2020.

[6] ext4 General Information. `https://docs.kernel.org/admin-guide/ext4.html`.

[7] Hwang, J., Vuppalapati, M., Peter, S., and Agarwal, R. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21) (Jul. 2021).

[8] Intel. Intel® Xeon® Gold 5218 Processor Specifications. `https://www.intel.com/content/www/us/en/products/sku/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz/specifications.html`.

[9] Jia, W., Jiang, D., and Xiong, J. CETOFS: A High-Performance File System with Host-Server Collaboration for Remote Storage. In Proceedings of the 24th USENIX Conference on File and Storage Technologies (FAST'26) (Feb. 2026).

[10] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. SplitFS: reducing software overhead in file systems for persistent memory. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19) (Oct. 2019).

[11] Kannan, S., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Wang, Y., Xu, J., and Palani, G. Designing a True Direct-Access File System with DevFS. In Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18) (Feb. 2018).

[12] Kim, H.-J., Lee, Y.-S., and Kim, J.-S. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16) (Jun. 2016).

[13] Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., and Anderson, T. Strata: A Cross Media File System. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17) (Oct. 2017).

[14] Lee, E., Bahn, H., Jeong, M., Kim, S., Yeon, J., Yoo, S., Noh, S. H., and Shin, K. G. Reducing Journaling Harm on Virtualized I/O Systems. In Proceedings of the 9th ACM International Systems and Storage Conference (SYSTOR'16) (Jun. 2016).

[15] libfuse API documentation. `https://libfuse.github.io/doxygen/`.

[16] Lukken, C., and Trivedi, A. Past, Present and Future of Computational Storage: A Survey. arXiv preprint arXiv:2112.09691, Dec. 2021.

[17] NVIDIA. NVIDIA BlueField-2 DPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, Dec. 2021.

[18] NVIDIA. RDMA Aware Networks Programming User Manual. https://docs.nvidia.com/rdma-aware-networks-programming-user-manual-1-7.pdf, Sep. 2023.

[19] NVIDIA. NVIDIA BlueField-2 Ethernet DPU User Guide. https://docs.nvidia.com/nvidia-bluefield-2-ethernet-dpu-user-guide.pdf, May 2024.

[20] NVM Express, Inc. NVM Express Base Specification Revision 1.2a. https://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2a.pdf, Oct. 2015.

[21] NVM Express, Inc. NVM Express over Fabrics Revision 1.1a. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf, Jul. 2021.

[22] NVM Express, Inc. NVM Express Base Specification, Revision 2.1. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf, Aug. 2024.

[23] Park, D., and Shin, D. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17) (Jul. 2017).

[24] PCI-SIG. Single Root I/O Virtualization and Sharing Specification Revision 1.1. https://pcisig.com/PCIExpress/Specs/IOV/SingleRootIOVirtualizationandSharing_1.1, Jan. 2010.

[25] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14) (Oct. 2014).

[26] Samsung Electronics. HHHL PCIe SSD Specification (PM1735). `https://www.ssd.group/wp-content/uploads/2022/07/PM1735-HHHL-SSD-Datasheet_v1.0_for-China.pdf`, Feb. 2020.

[27] Santos, P. C., Carro, L., Kepe, T. R., Moreira, F. B., Cordeiro, A. S., Santos, S. R., and Alves, M. A. Z. Survey on Near-Data Processing: Applications and Architectures. Journal of Integrated Circuits and Systems, Aug. 2021.

[28] Shirwadkar, H., Kadekodi, S., and Tso, T. FastCommit: resource-efficient, performant and cost-effective file system journaling. In Proceedings of the 2024 USENIX Annual Technical Conference (ATC'24) (Jul. 2024).

[29] Storage Performance Development Kit (SPDK). `https://spdk.io/`.

[30] Tarasov, V., Gupta, A., Sourav, K., Trehan, S., and Zadok, E. Terra Incognita: On the Practicality of User-Space File Systems. In Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15) (Jul. 2015).

[31] Tweedie, S. C. Journaling the Linux ext2fs Filesystem. In Proceedings of the 4th Annual Linux Expo (LinuxExpo'98) (May 1998).

[32] Vangoor, B. K. R., Tarasov, V., and Zadok, E. To FUSE or Not to FUSE: Performance of User-Space File Systems. In Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17) (Feb. 2017).

[33] Yadalam, S., Alverti, C., Karakostas, V., Gandhi, J., and Swift, M. BypassD: Enabling fast userspace access to shared SSDs. In Proceedings of the 29th ACM

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24) (Apr. 2024).

[34] Yang, L., Zhou, Y., Zeng, G., Zhang, L., Zhang, S., Wu, R., Sun, C., Luo, S., Li, W., Niu, K., Zhang, X., Wu, J., Zhu, J., Wu, J., Barczak, M., Gao, W., Lu, R., Xu, E., and Xue, G. Here, There and Everywhere: The Past, the Present and the Future of Local Storage in Cloud. In Proceedings of the 24th USENIX Conference on File and Storage Technologies (FAST'26) (Feb. 2026).

[35] Yang, Z., Lu, Y., Liao, X., Chen, Y., Li, J., He, S., and Shu, J. λ-IO: A Unified IO Stack for Computational Storage. In Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23) (Feb. 2023).

[36] Zhang, J., Ren, Y., and Kannan, S. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. In Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22) (Feb. 2022).

[37] Zhang, J., Ren, Y., Nguyen, M., Min, C., and Kannan, S. OmniCache: Collaborative Caching for Near-storage Accelerators. In Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24) (Feb. 2024).