Kernel-Level Data Deduplication Mechanism

Operating Systems Project – Topic 15

Linux 5.15.154+ Kernel Prototype

Author:	Haoyi Wang
Date:	April 2025
Platform:	QEMU ARM64 VM, Ubuntu 22.04
Core Method:	4KB inline deduplication with SHA-256 and `memcmp`

This report follows a specification-style academic structure with a title page, contents, scoped requirements, implementation-defined behavior, evaluation tables, and ordered references, inspired by the organization style of the OpenMP API Specification [5].

Contents

Abstract

This report presents a kernel-space data deduplication prototype for CSC5031 Topic 15. The objective is to implement inline block deduplication at the Linux page-cache or file-system level by modifying the kernel write path, computing block fingerprints, and remapping duplicate logical writes to shared physical content. The prototype is built on Linux 5.15.154+ and uses fixed 4KB block granularity, SHA-256 fingerprints, byte-level verification with `memcmp`, logical-to-physical metadata, reference counting, read remapping, discard cleanup, and `debugfs` observability. Beyond the basic requirement, the project evaluates backend write-elision in a controlled loop-device path, metadata persistence and recovery validation, and the CPU–I/O trade-off introduced by deduplication. Correctness tests cover duplicate hits, overwrite/refcount safety, concurrent duplicate writes, write-elision, persistence roundtrip, and crash-recovery remapping. Fio evaluation shows that duplicate-heavy workloads such as zero-fill and VM-image-like streams can improve bandwidth and latency, while low-redundancy random writes pay hashing and metadata overhead. The implementation is a working research and course prototype, not a production-ready general block-device deduplication file system.

1 Introduction

Modern storage systems frequently store repeated content. Virtual machine images may contain multiple copies of the same operating-system binaries and libraries. Backup snapshots can rewrite unchanged regions. Zero-filled files, structured application data, and generated artifacts often contain repeated block-sized regions. Without deduplication, the storage stack treats each logical write as a separate physical write, even when the payload is identical. This wastes capacity, increases write amplification, and can reduce SSD endurance.

Data deduplication addresses this problem by storing content once and representing repeated logical blocks through metadata. The general idea appears in content-addressable and archival storage systems such as Venti [6], wide-area and low-bandwidth file systems such as LBFS [4], and duplicate-file reclamation systems [2]. This project explores the same principle inside the Linux kernel write path: before duplicate data reaches the storage device, the kernel computes a fingerprint, checks whether the content already exists, verifies correctness, and updates deduplication metadata.

The project focuses on an inline kernel prototype rather than an offline user-space deduplication tool. User-space scripts are used only for workload generation, validation, and result collection; the core deduplication logic lives in kernel space. This choice directly matches the Topic 15 objective: implement data deduplication at the kernel page-cache or file-system level by extending `fs/buffer.c` to detect duplicate blocks using hashing and implement basic deduplication logic.

2 Project Requirements and Scope

2.1 Topic Requirements

The topic requires a kernel-level deduplication mechanism. Table 1 maps the project requirements to the implemented evidence.

Table 1: Requirement-to-implementation mapping.

Requirement	Implementation evidence
Modify or extend `fs/buffer.c`	Write hook at `fs/buffer.c:3073` invokes `dedupe15_maybe_handle_write(bio)`; read remap at `fs/buffer.c:3063` invokes `dedupe15_map_read(...)`.
Detect duplicate blocks using hashing	Each 4KB block is fingerprinted using SHA-256 in kernel space; the digest indexes candidate physical blocks.
Implement basic deduplication logic	The core path performs hash lookup, candidate verification with `memcmp`, logical-to-physical remapping, and reference-count updates.
Maintain kernel-space metadata	The `dedupe15` subsystem maintains hash index entries, L2P mappings, refcounts, and runtime counters.
Evaluate advanced options	The project evaluates CPU–I/O trade-offs, backend write-elision, and metadata persistence/recovery validation.

2.2 Scope and Non-goals

The implemented system is a controlled Linux kernel prototype. It is intentionally scoped to a dedicated loop-device test path to reduce failure blast radius and to make experiments reproducible. The prototype demonstrates the mechanism and its trade-offs; it does not claim to be a production file system or a production-ready general block-device deduplication layer. In particular, production-grade crash consistency, large-scale persistent metadata journaling, long-running memory pressure behavior, and cross-device deduplication remain future work.

2.3 Problem Statement

The concrete systems problem is not simply detecting equal byte strings. A kernel deduplication mechanism must preserve the semantics of the block interface while changing the relationship between logical and physical placement. The implementation must answer four questions for every 4KB write: whether the incoming bytes already exist, whether a candidate is truly equal, how logical reads will later reach the shared content, and when shared content can be safely released. A user-space post-processing tool can answer these questions after data is already written, but this project targets inline interception, where the decision is made during the kernel I/O path.

This makes the problem stricter than an ordinary hash-table exercise. A wrong hit can corrupt data, a wrong reference count can free data still in use, and a wrong L2P mapping can redirect reads to an unrelated sector. Therefore, the design prioritizes conservative correctness over maximum deduplication ratio. If any required operation fails, the system should fall back to the normal write path rather than risk accepting an unsafe duplicate.

2.4 Design Principles

The prototype follows four design principles.

1. **Minimal kernel hooks:** hook points in `fs/buffer.c` and `block/blk-lib.c` should remain small, so the invasive patch surface is limited.

2. **Hash for speed, bytes for safety:** SHA-256 narrows the candidate set; `memcmp` is the authority for equality.

3. **Metadata is part of correctness:** hash index, L2P mapping, and refcount must be updated as one logical transition.

4. **Observable prototype behavior:** counters and state interfaces are exposed through `debugfs` so tests can inspect kernel-internal behavior, not only external throughput.

3 Background

3.1 Block-level Deduplication

Deduplication can be applied at several granularities. File-level deduplication is simple but coarse: a small file modification may break sharing for the entire file, and repeated regions within a file are missed. Byte-level deduplication can maximize storage savings, but it creates high CPU cost, high metadata overhead, and substantial kernel implementation complexity. This project uses fixed-size 4KB block-level deduplication, balancing granularity, metadata cost, and implementation feasibility. The 4KB unit also aligns with the Linux page size and common file-system block sizes.

3.2 Content-addressable Storage

Content-addressable storage identifies data by content rather than location. Venti, for example, uses a hash of block contents as the block identifier, allowing duplicate block copies to be coalesced [6]. Similarity and chunk-based deduplication techniques have also been studied for backup workloads [1]. In this project, SHA-256 digests serve as lookup keys, but the digest alone is not treated as proof of equality.

3.3 Hash Collision and Verification Policy

A hash match is necessary but not sufficient for accepting a duplicate. Even though SHA-256 collisions are practically unlikely, a storage system cannot allow a false duplicate because it would cause data corruption. Therefore, the prototype always performs byte-level candidate verification with `memcmp` before remapping a logical block to existing physical content. In short, hashing accelerates lookup, while `memcmp` provides the correctness guarantee.

4 System Design

4.1 Architecture Overview

The design separates interception, core processing, metadata, verification, and observability. Kernel hooks are kept small and targeted; most logic is implemented in the `kernel/dedupe15/` subsystem. The relevant Linux block and kernel APIs are documented by the Linux Kernel documentation [3].

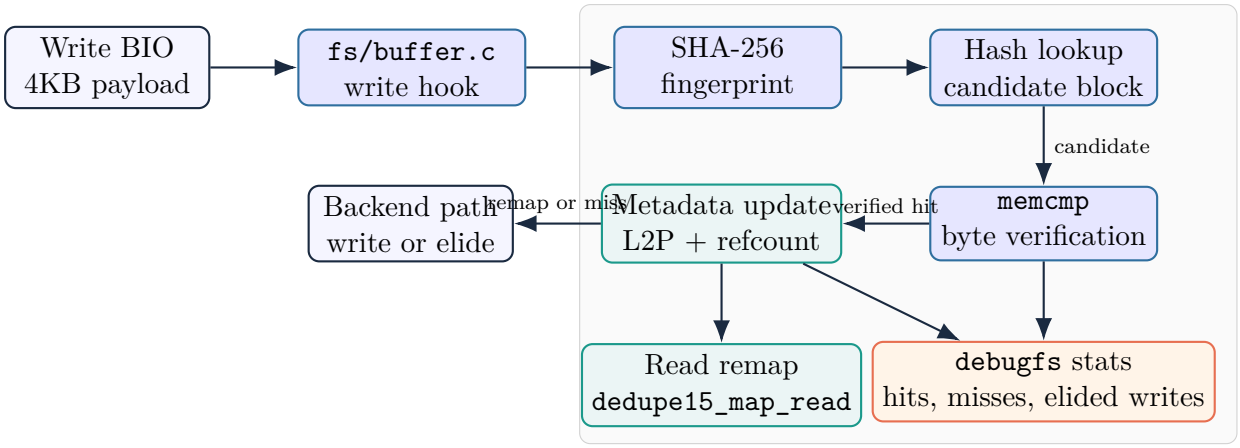


Figure 1: High-level architecture of the inline deduplication path. Small hooks intercept I/O events, while the `dedupe15` subsystem performs hashing, verification, metadata management, read remapping, and observability.

Table 2: Main kernel components modified or analyzed.

Component	Role
`fs/buffer.c:3063`	Read remap through `dedupe15_map_read(...)`.
`fs/buffer.c:3073`	Write hook through `dedupe15_maybe_handle_write(bio)`.
`block/blk-lib.c:149`	Discard hook through `dedupe15_handle_discard(...)`.
`kernel/dedupe15/core.c:259`	Core write processing through `dedupe15_process_write()`.
`kernel/dedupe15/core.c:380`	Discard range cleanup through `dedupe15_discard_range(...)`.
`debugfs`	Exposes runtime stats plus state/recovery interfaces.

4.2 Subsystem Modules

The `dedupe15` subsystem is structured as follows:

- **Core:** orchestrates write processing, hit/miss handling, write-elision decisions, and metadata transitions.
- **Hash:** computes SHA-256 fingerprints for 4KB blocks.
- **Verify:** performs `memcmp`-based candidate validation before reuse.
- **Metadata:** maintains the hash index, logical-to-physical map, and refcount state.
- **Read:** maps logical reads to remapped physical sectors.
- **Discard:** decrements references and removes stale entries on discard.
- **Stats:** exposes counters such as hits, misses, hash operations, bytes saved, fallback writes, and elided writes through `debugfs`.
- **Extension paths:** async hashing and compression are represented as extension modules, but the evaluated advanced results emphasize CPU–I/O analysis, write-elision, and recovery.

4.3 Metadata Model

The prototype tracks three main metadata structures:

1. **Hash index:** maps SHA-256 digests to candidate physical block entries.
2. **L2P map:** maps logical sectors to physical sectors, enabling read remap after deduplication.
3. **Reference count:** tracks how many logical sectors share a physical content block.

Reference counting is necessary because overwrite and discard operations must not release a physical content block that is still referenced by another logical sector. L2P mapping is necessary because the logical location of a block may no longer match the physical location that stores its shared content.

Table 3: Core metadata responsibilities.

Structure	Logical role	Correctness responsibility
Hash entry	`digest -> phys_sector`	Provides candidate lookup for duplicate content; must not be trusted without verification.
L2P entry	`logical_sector -> phys_sector`	Preserves read-after-write behavior after a logical write is remapped.
Reference count	`phys_sector -> refs`	Prevents discard or overwrite from recycling content still referenced by another logical sector.
Stats counters	`hits`, `misses`, `hash_ops`	Makes kernel-internal behavior observable during validation and live demonstration.
State snapshot	hash/L2P serialized state	Supports prototype-level persistence roundtrip and crash-recovery remap validation.

4.4 Concurrency and Consistency Model

Concurrent writes to identical data can race in several places: hash lookup, physical block allocation, L2P updates, and reference-count transitions. The implementation strategy is to treat deduplication metadata as shared kernel state and protect transitions with bucket-level synchronization. The design plan fixes lock acquisition order as L2P bucket first, hash bucket second, and allocator state last. This prevents cycles where one path holds a hash bucket and waits for L2P while another path holds L2P and waits for the same hash bucket.

Reference counts are treated as the final guard before recycling content. The important invariant is:

> A physical content block may be removed from the hash index or recycled only after all logical mappings that point to it have been removed and its reference count reaches zero.

The concurrency tests do not prove production-level scalability, but they validate the key safety property for the project: multiple writers producing the same data should not create inconsistent refcounts, double-free an entry, or break the shared read path.

4.5 Failure Policy

The failure policy is conservative. If hashing fails, candidate verification cannot read candidate data, metadata allocation fails, or a transition cannot be completed safely, the system should prefer the baseline write path. This may reduce the deduplication ratio, but it preserves correctness. In other words, deduplication is an optimization layer; it must not become a reason for returning incorrect data.

5 Implementation Details

5.1 Write Path Workflow

The write path uses a hit/miss state machine. Listing 1 summarizes the behavior.

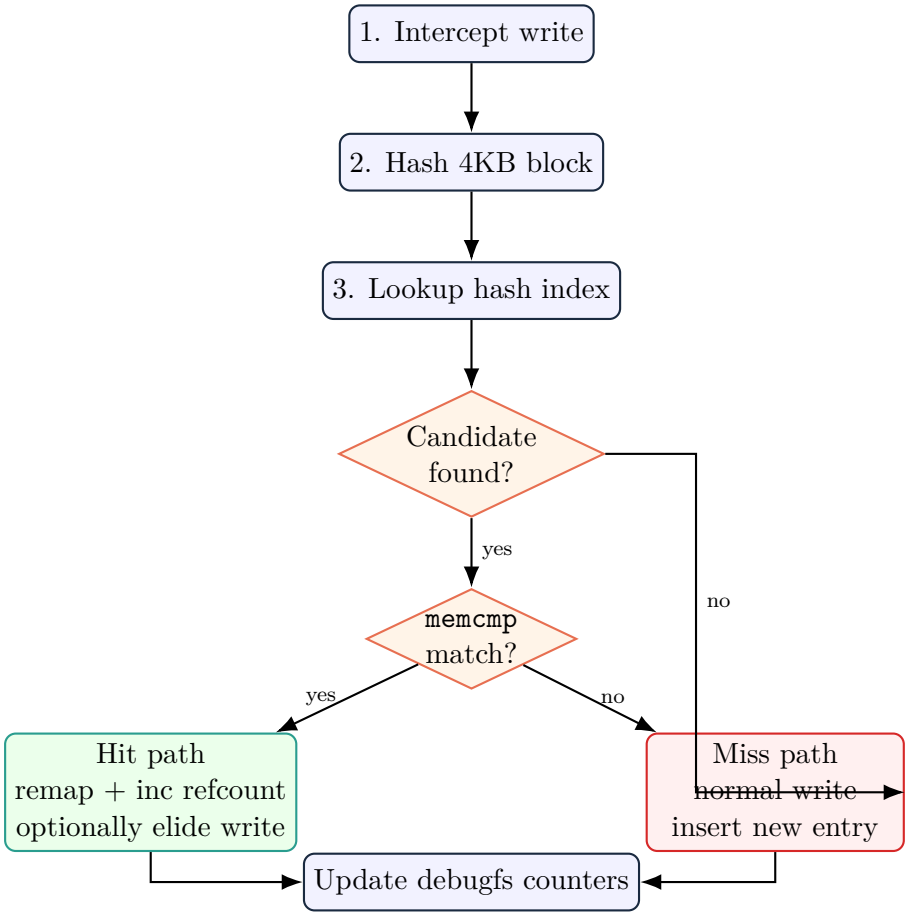


Figure 2: Write path state machine. A SHA-256 match is only a candidate; the hit path is accepted only after byte-level verification.

Listing 1: Simplified write path workflow.

```
function dedupe15_process_write(bio):
    block = extract_4KB_payload(bio)
    digest = SHA256(block)
    candidate = hash_index.lookup(digest)

    if candidate exists and memcmp(block, candidate.data) == 0:
        old_phys = l2p.lookup(bio.logical_sector)
        l2p.update(bio.logical_sector, candidate.phys_sector)
```

```
        refcount.inc(candidate.phys_sector)
        refcount.dec_and_recycle_if_zero(old_phys)
        stats.hits++
        stats.bytes_saved += 4096
        if write_elision_enabled:
            stats.elided_writes++
            return DEDUPE_HANDLED
        return REMAPPED

    new_phys = allocate_or_use_baseline_target()
    submit_or_allow_normal_write(bio, new_phys)
    hash_index.insert(digest, new_phys)
    l2p.update(bio.logical_sector, new_phys)
    stats.misses++
    return NORMAL_WRITE
```

On a hit, the digest lookup finds a candidate and `memcmp` confirms byte equality. The logical block is remapped, the candidate's reference count is incremented, and the old mapping is decremented. On a miss, the write proceeds normally and a new hash entry is inserted after the content is accepted as unique.

5.2 Hit Path versus Miss Path Semantics

The hit path and miss path differ not only in performance but also in metadata semantics. On a hit, the incoming write becomes a metadata operation: the logical sector is redirected to an already verified physical content block. This requires incrementing the candidate reference count before releasing the old mapping, so the system never creates a window where shared content is unreferenced prematurely. On a miss, the system must preserve baseline behavior by allowing the data to be written normally and then inserting a new digest-to-physical mapping.

The prototype also distinguishes duplicate detection from backend write-elision. A basic deduplication hit can update metadata and report a hit, but still allow normal backend write behavior. The advanced write-elision path goes further: after byte verification, it can avoid redundant backend write work in the controlled test path. This distinction is important because it prevents the report from overclaiming: basic duplicate detection and physical write elimination are related but not identical achievements.

5.3 Read Remap

The read path checks the L2P map before reading. If the logical sector is mapped, the read is redirected to the physical sector that stores the shared content. This preserves read-after-write correctness after a logical block has been deduplicated.

5.4 Discard Cleanup

Discard operations decrement reference counts for affected logical sectors. If a physical block's reference count reaches zero, the corresponding hash entry can be removed and the block can be recycled. This prevents stale metadata and ensures that logical deletion does not corrupt data still referenced by other logical sectors.

5.5 Complexity Analysis

For each 4KB write, the prototype performs one SHA-256 computation and one hash-table lookup. If the digest is absent, the write follows the miss path and pays the hash and metadata insertion cost. If the digest is present, the write may additionally pay one candidate verification cost, including reading or accessing candidate content and running `memcmp` over 4KB. With a

well-distributed hash table, lookup is expected $O(1)$, while the per-block hashing and verification cost is $O(B)$ for block size $B = 4096$ bytes. Metadata space grows with the number of unique physical content blocks and active logical mappings.

This complexity explains the experimental trend. Low-redundancy workloads still pay the $O(B)$ hashing cost for most writes, but rarely enter a useful hit path. High-redundancy workloads amortize this cost because repeated blocks produce hits and can reduce backend write work.

5.6 debugfs Interfaces

The prototype uses `debugfs` as an observability and validation interface. The main stats file is:

```
/sys/kernel/debug/dedupe15/stats
```

It exposes counters including `hits`, `misses`, `hash_ops`, `bytes_saved`, `fallback_writes`, and `elided_writes`. State and recovery interfaces are also exposed for persistence validation:

```
STATE_PATH=/sys/kernel/debug/dedupe15/state
RECOVER_PATH=/sys/kernel/debug/dedupe15/recover
```

`debugfs` is not treated as a production API. It is used because this is a kernel prototype, and direct observation of internal kernel counters is more convincing than external benchmark results alone.

6 Correctness Validation

Correctness is more important than performance in a deduplication system. A false duplicate or incorrect metadata transition can return wrong data to the read path. Table 4 summarizes the validation suite.

Table 4: Correctness and advanced validation tests.

Test	Purpose	Result
`smoke_hit_test.sh`	Duplicate write hit path; validates that identical writes increase hit counters.	pass
`overwrite_refcount_test.sh`	Overwrite/refcount safety; verifies that shared content is not freed while still referenced.	pass
`concurrency_same_data_test.sh`	Concurrent duplicate writes; validates synchronization under multiple writers.	pass
`write_elision_noop_test.sh`	Backend write-elision path for verified duplicate writes.	pass
`write_elision_hit_remap_test.sh`	Hit remap plus write-elision behavior.	pass
`persistence_roundtrip_test.sh`	State snapshot roundtrip through debugfs interfaces.	`H:6 L:8`
`crash_recovery_remap_test.sh`	Recovery of remap semantics after restoring metadata state.	pass

Representative pass lines include `phase1-correctness-pass`, `write-elision-pass elided_writes:0->4 hits:0->4`, `persistence-roundtrip-pass H:6 L:8`, and `crash-recovery-remap-pass`. Counter values can vary across repeated live runs depending on prior state, but pass/fail semantics and remap correctness are the relevant validation outputs.

6.1 Validation Invariants

The tests are organized around invariants rather than only script names. The first invariant is **duplicate safety**: identical writes should produce dedupe hits only after verification. The second invariant is **overwrite safety**: changing a logical block must not invalidate another logical block that still shares the old physical content. The third invariant is **concurrency safety**: simultaneous writers should not create duplicate metadata entries that violate refcount accounting. The fourth invariant is **lifecycle safety**: discard and recovery paths must not leave stale L2P entries or unusable remap state.

Table 5: Validation invariants and corresponding evidence.

Invariant	Risk if violated	Evidence
Duplicate safety	False duplicate causes data corruption.	SHA-256 plus `memcmp`; smoke hit test.
Overwrite safety	Shared content may be freed too early.	Overwrite refcount test.
Concurrency safety	Double allocation or inconsistent refcount.	Concurrent same-data test.
Write-elision safety	Backend write skipped for unsafe data.	Write-elision and hit-remap tests.
Recovery safety	Restored L2P map points to wrong content.	Persistence roundtrip and crash-recovery remap tests.

7 Evaluation Methodology

7.1 Environment

The evaluation runs inside a reproducible virtualized environment. Table 6 lists the configuration.

Table 6: Experiment environment.

Item	Value
Host	macOS
Virtual machine	QEMU ARM64
Guest OS	Ubuntu 22.04
Kernel	Linux 5.15.154+
CPU	4 vCPU
Memory	8GB
Project path	`/home/ubuntu/topic15`
Main runner	`/home/ubuntu/topic15/submission/reproduce.sh`

7.2 Reproducibility Workflow

The experiment is packaged as a host-to-guest workflow. On the host, VM artifact checks ensure that the disk image, firmware, seed image, and SSH key exist. The QEMU ARM64 VM is then launched and polled until SSH becomes reachable. Inside the guest, the command `bash /home/ubuntu/topic15/submission/reproduce.sh` runs the full pipeline. The pipeline executes short correctness tests first, then advanced tests, then fio workloads, then collection and rendering scripts.

Listing 2: Reproduction workflow used for the live demonstration.

```
cd /Users/barryw/Downloads/operatingsystetmcuhksz
bash scripts/vm/tests/vm_artifacts_test.sh
bash scripts/vm/launch_vm.sh
bash scripts/vm/wait_for_ssh.sh 240
ssh -i artifacts/ssh/topic15_ed25519 -p 2222 ubuntu@127.0.0.1
cd /home/ubuntu/topic15
bash /home/ubuntu/topic15/submission/reproduce.sh
sudo cat /sys/kernel/debug/dedupe15/stats
cat experiments/results/summary/final-matrix.csv
cat experiments/results/summary/baseline-vs-dedupe.csv
```

This workflow matters because it demonstrates more than a static patch. It shows that the patched kernel boots, accepts SSH access, runs the validation suite, executes fio workloads, and generates the same classes of result artifacts used in this report.

7.3 fio Workloads

The fio configuration is fixed across workloads to isolate workload-dependent redundancy effects. Table 7 lists the common configuration and workload meanings.

Table 7: fio configuration and workloads.

Parameter	Value
`bs`	`4k`
`size`	`128M`
`runtime`	20 seconds
`time_based`	`1`
`ioengine`	`libaio`
`direct`	`1`
`numjobs`	`1`

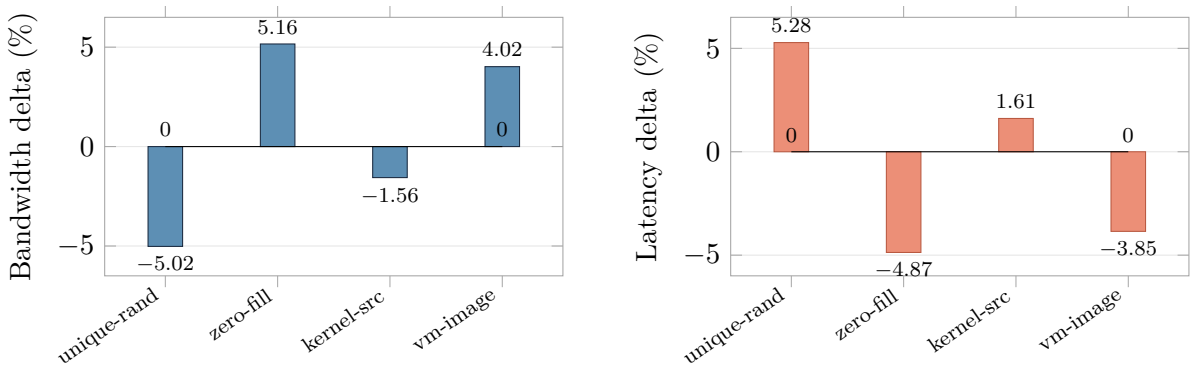
Table 8: Workload definitions.

Workload	Pattern	Meaning
`unique-rand`	random	Low redundancy; measures hash overhead.
`zero-fill`	all zero	High duplicate; approximates best-case dedupe benefit.
`kernel-src-replay`	kernel source-like	Semi-structured repeated data.
`vm-image-dup`	VM-image-like	Duplicate stream similar to VM image reuse.

8 Results and Analysis

8.1 Baseline versus Dedupe

Table 9 compares baseline and dedupe runs. Positive bandwidth delta and negative latency delta indicate improvement; negative bandwidth delta and positive latency delta indicate overhead.



(a) Bandwidth impact.

(b) Latency impact.

Figure 3: External performance effect of deduplication. The sign pattern is workload-dependent: high-redundancy workloads improve, while low-redundancy random writes pay overhead.

Table 9: Baseline versus dedupe comparison.

Workload	BW Delta	Lat Delta	CPU Delta	Saved Bytes	Interpretation
`unique-rand`	-5.02%	+5.28%	-0.70%	8192	Low redundancy pays hash overl
`zero-fill`	+5.16%	-4.87%	-11.72%	57344	Duplicate-heavy workload benefi
`kernel-src-replay`	-1.56%	+1.61%	-8.89%	49152	Mixed workload has mixed resul
`vm-image-dup`	+4.02%	-3.85%	-9.65%	4096	VM-like duplicate stream benefit

The results show that deduplication is workload-dependent. For `unique-rand`, the duplicate rate is low, so the system pays SHA-256 and metadata lookup overhead without saving much I/O. For `zero-fill`, repeated content produces many duplicate candidates, so I/O savings dominate. The `vm-image-dup` workload also benefits, supporting the original motivation that VM images often contain repeated content. The `kernel-src-replay` workload is mixed: it contains some repeated structure but not enough to overcome all overhead.

8.2 Per-workload Interpretation

`unique-rand`. This workload is the expected worst case for inline deduplication. Most blocks are unique, so the hash table produces few useful hits. The measured bandwidth drops by 5.02% and latency increases by 5.28%. This is an important negative result: it shows that deduplication should not be treated as an always-on performance optimization.

`zero-fill`. This workload is the best case because repeated all-zero blocks maximize duplicate opportunities. Bandwidth improves by 5.16% and latency decreases by 4.87%. The result confirms that the prototype can convert repeated data into measurable I/O benefit when duplicate density is high.

`kernel-src-replay`. This workload is intentionally more realistic than zero-fill. Source-tree-like data contains repeated structure but also many unique regions. Its mixed result shows that partial redundancy may not be enough to overcome hashing and metadata overhead in every run.

`vm-image-dup`. This workload reflects the project motivation: VM-like images often contain repeated operating-system blocks. The measured improvement of 4.02% bandwidth and 3.85% latency supports the claim that kernel-level deduplication is useful for duplicate-heavy system images.

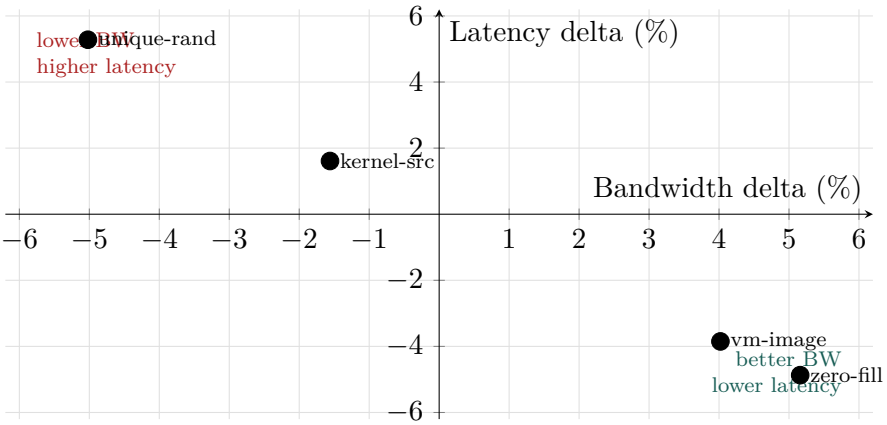


Figure 4: CPU–I/O trade-off as a quadrant view. The best region is positive bandwidth delta and negative latency delta. This visualization makes the workload-dependent nature of inline deduplication explicit.

8.3 Final Dedupe-mode Matrix

The final matrix in Table 10 reports dedupe-mode throughput, IOPS, and mean latency for the prototype run.

Table 10: Final dedupe-mode performance matrix.

Workload	IOPS	BW (MiB/s)	Mean Latency (ns)
`unique-rand`	64.6k	252.27	492834.61
`zero-fill`	117.1k	457.27	106617.13
`kernel-src-replay`	68.3k	266.63	191537.31
`vm-image-dup`	120.3k	469.80	119840.64

A later live reproduction run produced slightly different raw throughput due to VM runtime state, which is expected for fio inside a VM. The stable analytical conclusion remains the same: duplicate-heavy workloads benefit, while low-redundancy workloads can pay overhead.

9 Advanced Options

9.1 CPU–I/O Trade-off Analysis

The strongest advanced option is the CPU–I/O trade-off analysis. Deduplication adds CPU work: SHA-256 hashing, hash-index lookup, optional candidate reads, `memcmp` verification, and metadata updates. In return, it may reduce backend I/O by remapping duplicate writes. The experiments demonstrate that this is not a universal win. A practical system should be workload-aware: it could sample recent writes, estimate hit rate, and bypass deduplication when misses dominate.

9.2 Backend Write-elision

Backend write-elision allows a verified duplicate write to skip the backend write path in the controlled loop-device setup. The relevant evidence is `write-elision-pass elided_writes:0->4 hits:0->4`. This goes beyond basic duplicate detection: the kernel not only observes a duplicate but also avoids redundant backend work after verification. The report intentionally scopes this claim to the controlled test path.

9.3 Persistence and Recovery Validation

The prototype provides debugfs state and recovery interfaces for metadata roundtrip validation. The pass line `persistence-roundtrip-pass H:6 L:8` indicates restored hash and L2P state, while `crash-recovery-remap-pass` validates remap behavior after recovery. This is prototype-level recovery validation, not a full production journaling design.

9.4 Compression and Async Extension Paths

The project structure includes async hashing and compression-related extension modules. These are presented as extension paths, not as the main evaluated results. A production design could integrate LZ4 or zlib for unique blocks and use workqueues or per-CPU hashing contexts to reduce write-path blocking. However, this report avoids overclaiming compression performance because the most complete evidence is for write-elision, recovery validation, and CPU–I/O analysis.

10 Discussion

10.1 Why Inline Kernel Deduplication is Difficult

Inline kernel deduplication is difficult because it sits on the boundary between optimization and correctness. The optimization goal is to avoid repeated physical writes. The correctness obligation is to preserve the block abstraction exactly as if every logical write had its own independent physical copy. The implementation therefore has to coordinate multiple subsystems: hashing, candidate verification, metadata mapping, reference accounting, read redirection, discard cleanup, and recovery state.

The most subtle part is that correctness spans time. A write may be accepted today, but the mapping must still be correct after a later read, overwrite, discard, or recovery operation. This is why the report emphasizes lifecycle behavior instead of only the first duplicate hit.

10.2 Why debugfs is Used

`debugfs` is used because the project is a kernel prototype. A production system would expose stable policy and control interfaces through mechanisms such as `sysfs`, `ioctl`, or file-system-specific tools. For this project, however, the primary need is observability. External fio numbers can show bandwidth and latency changes, but they cannot prove that the dedupe path was triggered internally. Counters such as `hits`, `misses`, `hash_ops`, and `elided_writes` provide direct kernel-side evidence.

10.3 Threats to Validity

The evaluation has several threats to validity. First, the VM environment introduces noise; repeated fio runs can produce slightly different raw throughput. Second, the workload set is intentionally small and synthetic, so it cannot represent all file-system workloads. Third, the controlled loop-device path limits generality. Fourth, CPU percentages in virtualized environments should be interpreted as comparative indicators rather than absolute hardware costs. These limitations do not invalidate the mechanism, but they constrain how broadly the performance numbers should be generalized.

11 Limitations

- **Controlled device scope:** The prototype is validated on a controlled loop-device path, not arbitrary production block devices.

- **Crash consistency:** Metadata persistence/recovery tests validate roundtrip behavior, but general block-device crash consistency needs journaling, atomic update ordering, and failure-model analysis.
- **Memory and scalability:** In-memory hash and L2P metadata are sufficient for the course prototype but would need capacity management and eviction policies in production.
- **Workload sensitivity:** Low-redundancy workloads such as `unique-rand` can pay hashing overhead without enough I/O savings.
- **Not a production file system:** The implementation demonstrates a kernel mechanism; it is not a general-purpose deduplication file system.

12 Future Work

Future work should focus on three directions. First, a persistent metadata journal should be implemented to provide stronger crash consistency. Second, hashing and metadata lookup should be optimized using async workqueues, per-CPU state, or faster lookup structures. Third, a workload-aware policy should be added so that the kernel can dynamically reduce or bypass deduplication when recent hit rate is low. Compression can also be revisited once the core deduplication path has stable production-style metadata semantics.

13 Conclusion

This project implements and evaluates a Linux kernel-level 4KB inline deduplication prototype. It satisfies the basic Topic 15 requirements by extending `fs/buffer.c`, computing SHA-256 hashes, verifying duplicate candidates with `memcmp`, and maintaining kernel-space deduplication metadata. It also demonstrates advanced work through backend write-elision, persistence/recovery validation, debugfs observability, and fio-based CPU–I/O trade-off analysis. The central finding is that deduplication is beneficial for duplicate-heavy workloads but can hurt low-redundancy random writes. Correctness, not raw performance, is the primary engineering constraint: false duplicate acceptance, incorrect refcounting, or incorrect L2P remapping would cause data corruption. The prototype therefore favors conservative verification and explicit observability over aggressive production claims.

References

[1] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *2009 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 1–9, 2009. doi:10.1109/MASCOT.2009.5366623.

[2] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pp. 617–624, 2002.

[3] The Linux Kernel Community. The Linux Kernel Documentation: Core API and Block Layer Documentation. Available at https://docs.kernel.org/.

[4] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 174–187, 2001.

[5] OpenMP Architecture Review Board. OpenMP Application Programming Interface, Version 5.0. November 2018. Available at https://www.openmp.org/specifications/.

[6] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *Conference on File and Storage Technologies (FAST 02)*, pp. 89–101, 2002.