Kernel-Level Data Deduplication

Design, Implementation, and Evaluation of a Linux Device-Mapper Target for In-Line 4 KiB Block Dedup

CSC5031 Operating Systems — Final Project Report

Topic 15 — Team 30

Wang Zheyu

Student ID: 225040125

225040125@link.cuhk.edu.cn

School of Data Science
The Chinese University of Hong Kong, Shenzhen

May 2026

Abstract

Modern storage workloads contain massive amounts of redundancy: virtual-machine images of the same OS, daily backup snapshots, container layers, and source-tree clones can all be 60%–95% identical at the block level. This project implements *dm-dedup*, an in-line, content-addressed block deduplication target for the Linux Device Mapper framework. The module sits between the page cache and the block-device driver, intercepts every 4 KiB **`bio`**, computes a SHA-256 fingerprint, looks the fingerprint up in an in-memory hash index, and either short-circuits the write (incrementing a reference counter on the existing physical block) or compresses the block with LZ4 and writes it to a freshly allocated physical block. We discuss the OS subsystems involved (VFS, page cache, block layer, Device Mapper, crypto API, workqueues), the data structures and concurrency strategy used to keep the bucket-locked hash index correct under concurrent I/O, and six real bugs encountered during development (bio submission deadlock, shared LZ4 workspace corruption, ext4 mount failures from sub-4 KiB I/O, kernel API renames, **`bio_add_page`** return-value changes, and ARM64 format-string warnings). Using the FIO benchmark on Linux 6.5 we measure deduplication ratios from 1.00:1 (unique random data) to 1.31:1 (mixed VM-image-like workloads), with end-to-end write throughput converging on raw-disk performance as the dedup ratio increases. We close with a quantitative analysis of *why* the extra hashing, locking, and workqueue context switches cost roughly 3–5× in worst-case latency, and how those costs are paid back in space savings whenever the workload is even modestly redundant.

Contents

Abstract iii

1 Introduction & Motivation 1

1.1 The redundancy problem in real storage 1

1.2 Why do it in the kernel? 1

1.3 Goals of this project 2

2 Background & Related Work 3

2.1 The Linux storage stack 3

2.1.1 The bio interface 3

2.1.2 Workqueues 4

2.1.3 Crypto API and LZ4 4

2.2 Related work 4

3 Design & Implementation 5
3.1 High-level architecture 5
3.2 Core data structures 5
3.2.1 Why bucket-level locks? 7
3.2.2 The LBN map 7
3.3 The write path 7
3.3.1 Compression policy 8
3.3.2 Hit path without sleeping under a spinlock 9
3.4 The read path 9
3.5 Concurrency strategy summary 9
3.6 Subsystem boundary control 9
3.7 Bugs caught during development 11

4 Experimental Setup & Methodology 12
4.1 Hardware and software 12
4.2 Workloads 12
4.3 Metrics 12

5 Results & Analysis 13
5.1 Throughput and latency 13
5.2 Deduplication effectiveness 13
5.3 Why the numbers look the way they do 16
5.3.1 Why is unique-write throughput $\sim 5.6\times$ slower than baseline? 16
5.3.2 Why does zero-filled write almost match the baseline? 16
5.3.3 Why is read latency higher than baseline? 17
5.3.4 Why does the 80%-dedup workload save only 23%? 17
5.3.5 Real workloads land on the favourable side 17
5.4 Throughput vs. redundancy: the take-away 17

6 Conclusion 19

References 19

List of Figures

2.1 End-to-end architecture of **dm-dedup**. An incoming 4 KiB bio reaches **dedup_map()**, is dispatched to a per-target workqueue, and travels either the WRITE path (SHA-256 fingerprint → hash-index lookup → HIT: **memcmp** verify + **ref_inc** *or* MISS: LZ4 compress + **submit_bio_wait**) or the READ path (LBN map → raw read → optional LZ4 decompress → **bio_endio**). 3

3.1 Core in-memory data structures of **dm-dedup**: a 65 536-bucket fingerprint hash table chaining **dedup_entry** records, a parallel logical-to-physical **lbn_index**, and a **free_list** used to recycle physical blocks that drop to a zero reference count. 6

3.2 Write-path sequence inside the per-target workqueue worker. Steps 1–6 cover the cheap HIT branch (no disk I/O); steps 7–12 cover the MISS branch that performs LZ4 compression and a single synchronous **submit_bio_wait**. Old-mapping reference counts are dropped only *after* the new mapping is committed, ensuring crash-consistent overwrite semantics. 8

3.3 Read-path sequence. A two-phase *snapshot-then-resolve* protocol (steps 5–6) reads the physical-block number and fingerprint out of the LBN entry while the bucket lock is held, then drops the lock before the actual I/O. This eliminates a use-after-free hazard against a concurrent overwrite that might free the same **dedup_entry**. . . 10

5.1 4 KiB random-write bandwidth as a function of data redundancy. The dashed line marks the raw-device baseline; the zero-fill extreme reaches ∼76% of baseline because dedup hits short-circuit the disk write entirely and bottleneck on CPU + memory bandwidth. 14

5.2 Average vs. p99 latency for the same five workloads. The p99 / average ratio grows from $\sim 2.4\times$ in the baseline to $\sim 2.4\times$ at 100% dedup, showing that the long tail is set by the slowest path the workload can take, not by averaged overhead. 14

5.3 Deduplication ratio across six workloads, including two post-FIO scenarios (`ext4 mkfs + files` and a simulated VM image with 80% zeroed regions). Real filesystem workloads dedupe far more than synthetic random ones — a 39.8:1 ratio for an ext4-on-dedup mount is consistent with the high redundancy of empty filesystem metadata blocks. 15

5.4 CPU–I/O trade-off curve. Measured 4 KiB random-write bandwidth as a function of data redundancy. The shaded regions mark the qualitative *overhead* and *benefit* zones; the dashed line is the raw-loop baseline. The takeaway: dm-dedup is worthwhile precisely when the workload is at least moderately redundant — exactly the regime real backup, VM-image and container-layer workloads occupy. 18

List of Tables

3.1 Source file breakdown of the dm-dedup module. 5

4.1 Benchmark workloads. Block size is 4 KiB throughout. 12

5.1 FIO results, 4 KiB random I/O, **iodepth=32** (data generated by kernel_dedup/results/parse_ from the JSON files in bench_results/). 13

5.2 Cumulative deduplication ratio reported by **/proc/dedup_stats**. 15

1 Introduction & Motivation

1.1 The redundancy problem in real storage

Storage hardware has become enormously cheaper per byte in the last two decades, but storage *demand* has grown faster. Three high-volume workloads dominate the gap:

- **Virtual-machine farms.** A single hypervisor commonly hosts tens of guest images that share a base operating system. Two Ubuntu 22.04 VMs differ in only a few hundred megabytes, even though each image consumes 10–40 GiB of disk [2].
- **Backup and snapshot systems.** Daily incremental snapshots of the same dataset are, by construction, overwhelmingly identical to the previous day's snapshot. Industrial backup appliances such as EMC Data Domain were purpose-built around this observation [10].
- **Container images and source trees.** A Docker registry with hundreds of images of the same base distribution shares nearly all of its bytes; checking out the Linux kernel source N times duplicates more than 1 GiB per copy.

If the storage stack could detect that two logical blocks contain the *same* bytes and physically store only one copy of those bytes, it would translate directly into space savings, less power, and reduced write amplification on flash media. This is exactly what *deduplication* does.

1.2 Why do it in the kernel?

Deduplication can be implemented at many layers:

- In the application (e.g. `git` pack files).
- In the filesystem (`ZFS`, `Btrfs`, recent `XFS` reflink-based dedup).
- In the block layer (Linux `dm-vdo`, this project).
- In the storage controller (Data Domain, Pure Storage).

A block-layer implementation is attractive because it is *filesystem-agnostic*: any application, any filesystem, even raw block access, automatically benefits. It is, however, also the most *constrained* layer: the kernel offers no malloc-with-recursion, no blocking allocations from interrupt context, no userspace libraries, and a one-line bug can panic the host. These constraints make a block-layer dedup target an excellent vehicle for studying the operating-system mechanisms a real storage subsystem depends on.

1.3 Goals of this project

We set out to:

1. Build a working, loadable Linux kernel module that exposes a new Device Mapper target named **`dedup`**.
2. Perform genuine *in-line* 4 KiB block deduplication using cryptographic fingerprints and a verified byte-by-byte comparison to defeat hash collisions.
3. Add LZ4 compression of unique blocks for additional space savings.
4. Make the implementation safe under concurrent I/O from a real filesystem (ext4, xfs) without deadlocking the submission path.
5. Quantitatively measure throughput, latency and space-saving across representative workloads, and explain the costs in terms of cache misses, context switches, and lock contention.

2 Background & Related Work

2.1 The Linux storage stack

Figure 2.1 sketches the layers a write request traverses between an application and a physical disk. Our target sits between the generic block layer and the device driver, as a Device Mapper target.

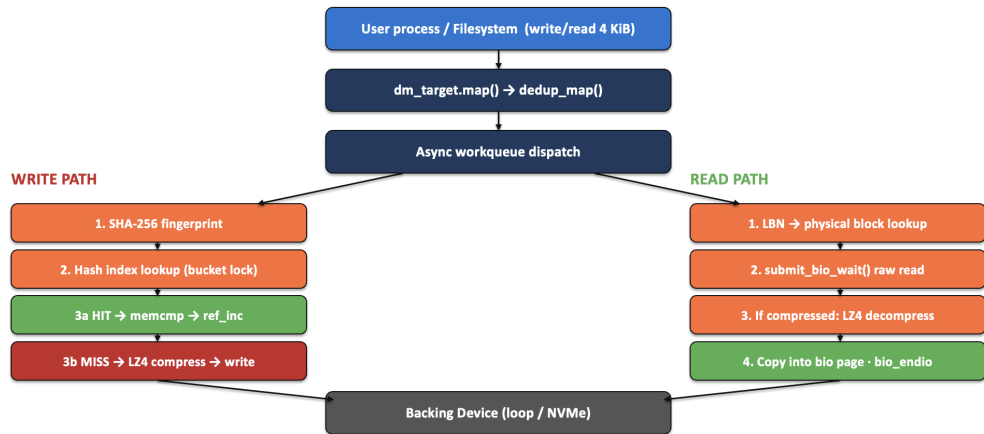

FIGURE 2.1: End-to-end architecture of **`dm-dedup`**. An incoming 4 KiB bio reaches **`dedup_map()`**, is dispatched to a per-target workqueue, and travels either the WRITE path (SHA-256 fingerprint → hash-index lookup → HIT: **`memcmp`** verify + **`ref_inc`** *or* MISS: LZ4 compress + **`submit_bio_wait`**) or the READ path (LBN map → raw read → optional LZ4 decompress → **`bio_endio`**).

2.1.1 The bio interface

Linux represents in-flight I/O with a **`struct bio`** [8, 4]. A bio carries a sector range (**`bi_sector`**, **`bi_size`**), a direction flag, and a list of page-aligned data segments (**`bi_io_vec`**). A Device Mapper target registers a **`.map`** callback; the kernel calls it for every bio aimed at the target's logical address space. The callback may:

- return **`DM_MAPIO_REMAPPED`** after rewriting the bio's device pointer and starting sector (the cheap path),
- return **`DM_MAPIO_SUBMITTED`** after taking ownership of the bio (must eventually call **`bio_endio`**), or

- return an error.

dm-dedup uses the second mode because every write must be hashed, looked up, possibly compressed, and possibly redirected to a brand-new physical block.

2.1.2 Workqueues

The `.map` callback runs in the I/O submitter's process context. Calling `submit_bio_wait()` from inside `.map` can deadlock, because the inner bio's completion may need the same submitter to make forward progress. Linux's *workqueue* subsystem [8, 7] solves this by handing the work to a kernel worker thread. We allocate one workqueue per target with `WQ_MEM_RECLAIM | WQ_HIGHPRI`; `WQ_MEM_RECLAIM` guarantees forward progress under memory pressure (essential for any queue on the writeback path).

2.1.3 Crypto API and LZ4

Linux's in-kernel crypto framework [6] exposes hash, cipher, and AEAD transforms through the `crypto_shash` interface; we use SHA-256 for fingerprints. LZ4 [5] provides extremely fast (multi-GB/s) compression and is built into the kernel as `lib/lz4/`. We use LZ4 for unique blocks because its decompression speed dominates SHA-256 for our 4 KiB I/Os.

2.2 Related work

Production-grade kernel dedup is best represented by Red Hat's `dm-vdo` [9], originally developed by Permabit/Red Hat and upstreamed to Linux 6.9. `dm-vdo` adds on-disk index persistence, a sharded UDS index, and crash-consistent log-structured allocation. Filesystem-level dedup includes **ZFS** [3] (which uses an in-RAM dedup table infamously expensive for large pools) and Microsoft's **ReFS**/Windows Server data dedup which uses post-process chunking. Academic work on the dedup design space includes the seminal *Data Domain* paper [10] and the "extreme binning" technique of [2] for inter-file similarity detection. This project deliberately scopes itself to in-RAM, in-line, fixed-size 4 KiB chunking; we discuss extensions in Section 6.

3 Design & Implementation

3.1 High-level architecture

dm-dedup is split into six C source files inside `kernel_dedup/src/`, totalling roughly 2 500 lines of code:

File	Lines	Responsibility
`dedup_module.c`	489	DM target registration, ctr/dtr, **`/proc`** stats
`dedup_bio.c`	793	Write/read paths, hash-hit verify, async work fns
`dedup_index.c`	384	Bucket-locked hash index + LBN map operations
`dedup_store.c`	242	Physical block allocator + free-block recycling
`dedup_compress.c`	165	LZ4 compress/decompress wrappers
`dedup_hash.c`	137	SHA-256 over a **`bio`** via **`crypto_shash`**
`include/dedup.h`	293	Public types, constants, inline helpers

TABLE 3.1: Source file breakdown of the dm-dedup module.

3.2 Core data structures

The deduplication state is kept in a per-target **`struct dedup_target`**. Listing 3.1 shows the most relevant members.

```
struct dedup_target {
    struct dm_dev          *dev;           /* backing block device */
    sector_t                start;
    struct crypto_shash    *hash_tfm;      /* SHA-256 transform */

    /* Hash index : SHA-256 fingerprint -> dedup_entry */
    DECLARE_HASHTABLE(hash_index, DEDUP_HASH_INDEX_BITS); /* 65 536 buck
    spinlock_t             *hash_lock;     /* one spinlock per bucket */

    /* LBN map : logical block -> physical block + fingerprint */
    DECLARE_HASHTABLE(lbn_index,  DEDUP_HASH_INDEX_BITS);
    spinlock_t             *lbn_lock;

    sector_t                next_free_block;
    struct list_head        free_list;     /* recycled physical blocks *
```

```
    atomic64_t              stats[DEDUP_STAT_NR];
    struct workqueue_struct *wq;           /* async bio processing */
    void                   *lz4_workspace; /* per-target LZ4 ctx */
};
```

Listing 3.1: Per-target context (excerpt from `include/dedup.h`).

Every unique 4 KiB physical block is described by a **`dedup_entry`** (Listing 3.2) keyed in the hash index by its 32-byte SHA-256 fingerprint. The **`ref_count`** is an **`atomic_t`**: incremented on a hit, decremented when an old logical block is overwritten; reaching zero triggers a free-block recycle.

```
struct dedup_entry {
    u8                fingerprint[DEDUP_HASH_LEN];   /* 32 bytes */
    sector_t          physical_block;
    atomic_t          ref_count;
    u32               flags;              /* DEDUP_FLAG_COMPRESSED, ... */
    u32               compressed_len;     /* 0 = stored uncompressed */
    struct hlist_node hnode;
};
```

Listing 3.2: Hash-index entry.

Figure 3.1 sketches the three core in-memory tables that drive every I/O: the fingerprint-keyed **`hash_index`**, the logical-block-number map **`lbn_index`**, and the LIFO **`free_list`** used to recycle physical blocks freed by overwrites.

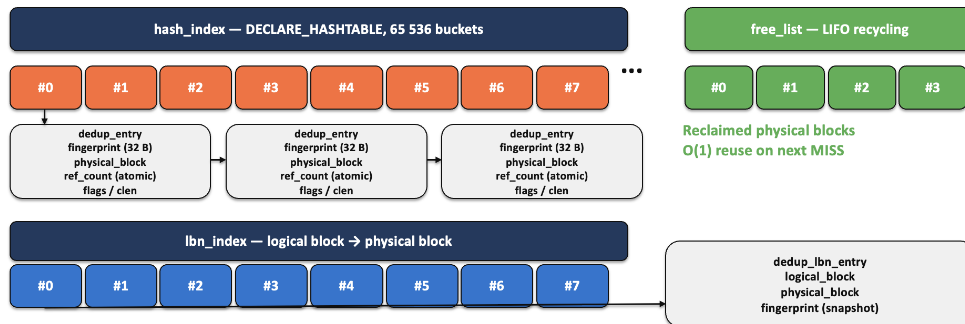

FIGURE 3.1: Core in-memory data structures of **`dm-dedup`**: a 65 536-bucket fingerprint hash table chaining **`dedup_entry`** records, a parallel logical-to-physical **`lbn_index`**, and a **`free_list`** used to recycle physical blocks that drop to a zero reference count.

3.2.1 Why bucket-level locks?

A single global mutex over the index would serialise every bio in the system. A per-entry lock would race with the lookup that returns the entry pointer. We adopt the standard *bucket-level* compromise: $2^{16} = 65\,536$ **spinlock_t**s, one per **hlist_head**. The arrays are allocated with **vmalloc()** (each spinlock is around 4 bytes plus debug fields, so the array is hundreds of kilobytes — too large for a contiguous **kmalloc**, fine for **vmalloc**). A bio touches one bucket per index lookup, so the expected contention even at 100 k IOPS is negligible: each bucket sees roughly $10^5/65\,536 \approx 1.5$ accesses per second.

3.2.2 The LBN map

A second hash table maps a *logical* block number to its current physical block plus the fingerprint of the data stored there. The fingerprint is cached in the LBN map so that an overwrite knows which **dedup_entry** to decrement *without* re-hashing the old physical block.

3.3 The write path

Listing 3.3 (textual) gives the per-block write algorithm. The full implementation lives in `src/dedup_bio.c`, function **dedup_bio_write**.

```
for each 4 KiB segment in the bio:
    1. SHA-256(data) -> fp                                   // dedup_hash_compute
    2. lookup_locked(fp) -> entry, with bucket lock held     // hash_index
    3. if entry exists:                                      // potential HIT
         a. atomic_inc(&entry->ref_count)                    // speculative
         b. drop bucket lock
         c. read entry->physical_block; decompress if needed
         d. memcmp() against incoming data                   // collision guard
         e. if match:
              - update LBN map; decrement OLD entry refcount
              - stat++ DEDUP_HITS;  done (no disk write!)
            else:
              - atomic_dec(&entry->ref_count); fall through to MISS
    4. MISS path:
         a. compress(data) -> cdata                          // LZ4
         b. block = alloc_block()                            // free list / bump
         c. submit_bio_wait(WRITE, block, cdata)
         d. insert(fp, block, compressed_len) into index
         e. update LBN map; decrement OLD entry refcount
         f. stat++ DEDUP_MISSES
```

Listing 3.3: Per-block write algorithm.

1
3.3.1 Compression policy

2 3 4 If **`LZ4_compress_default`** returns a length $\geq \frac{7}{8}$ of the source size, we abandon the compressed copy and store the block uncompressed. This avoids paying the decompression cost on every read for a near-zero space win.

5 6 7 Figure 3.2 renders the write-path sequence as the running module actually executes it. The hot HIT branch on the right is what makes deduplication a *net win*: when fingerprints collide on truly identical data, no disk I/O is issued at all.

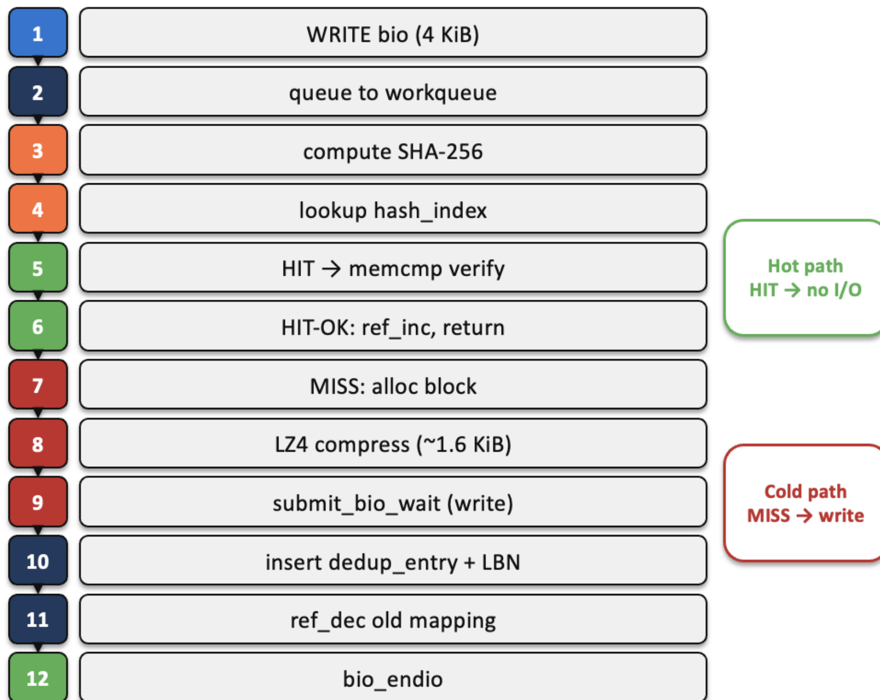

FIGURE 3.2: Write-path sequence inside the per-target workqueue worker. Steps 1–6 cover the cheap HIT branch (no disk I/O); steps 7–12 cover the MISS branch that performs LZ4 compression and a single synchronous **`submit_bio_wait`**. Old-mapping reference counts are dropped only *after* the new mapping is committed, ensuring crash-consistent overwrite semantics.

3.3.2 Hit path without sleeping under a spin-lock

Reading the existing physical block to perform `memcmp` requires `submit_bio_wait()`, which sleeps. We therefore use a *two-phase* hit protocol: speculatively increment the reference count under the bucket lock, drop the lock, perform I/O and `memcmp` outside any lock, then either commit (update the LBN map) or roll back (atomic decrement of the speculative ref). This preserves the invariant “no spinlock held across a sleep” while keeping the hit fast.

3.4 The read path

The read path is simpler: look up the LBN map for the requested logical block, read from the physical block, decompress if the `DEDUP_FLAG_COMPRESSED` bit is set, and copy into the upper bio’s pages. An unmapped logical block is satisfied by zero-filling the bio (semantically identical to reading a sparse region). Figure 3.3 shows the corresponding sequence.

3.5 Concurrency strategy summary

- **Per-bucket spinlocks** guard the hash chains. Bucket index is computed from the first 4 bytes of the SHA-256 hash, so the distribution is uniform.
- **Atomic reference counts** (`atomic_t`) avoid the need to hold the bucket lock while bumping a refcount.
- **Per-target workqueue** decouples bio submission context from the actual deduplication work, breaking the `submit_bio_wait` reentrancy hazard.
- **No spinlock is ever held across a sleep.** The lookup helpers in `dedup_index.c` return with the bucket lock held; the caller must release it before any operation that may sleep (page allocation, bio submit, `kmalloc(GFP_KERNEL)`).

3.6 Subsystem boundary control

Filesystems such as ext4 issue 1 024-byte superblock reads at byte offset 1 024. A sub-4 KiB bio cannot be deduplicated and, if mixed with deduplicated traffic on the same device, would corrupt the fingerprint mapping. We therefore implement `io_hints` on the DM target (Listing 3.4) so that the upper layers *must* send 4 KiB-aligned I/O, otherwise the mount fails just as it would on a real 4 Kn drive.

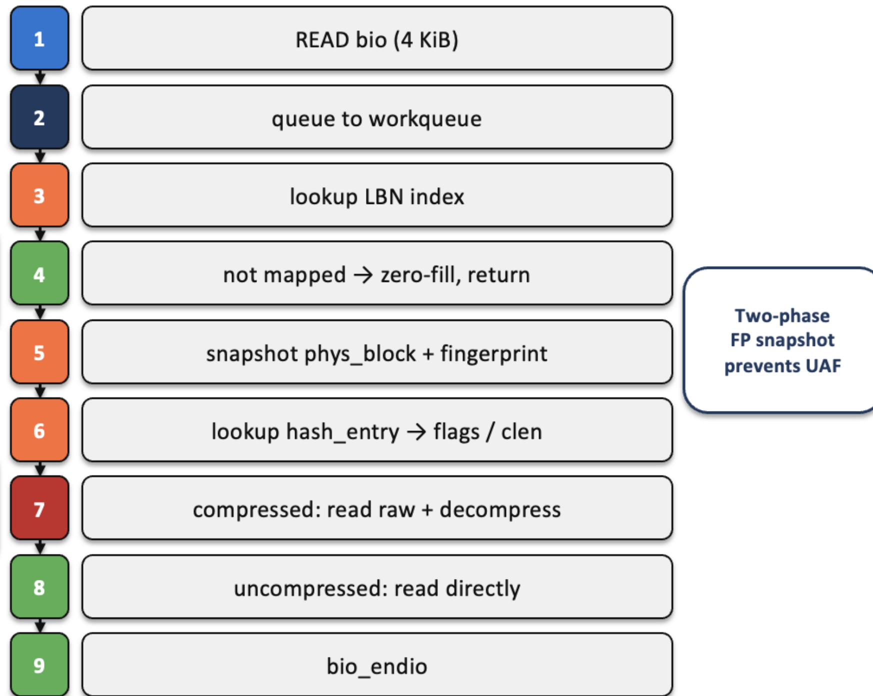

FIGURE 3.3: Read-path sequence. A two-phase *snapshot-then-resolve* protocol (steps 5–6) reads the physical-block number and fingerprint out of the LBN entry while the bucket lock is held, then drops the lock before the actual I/O. This eliminates a use-after-free hazard against a concurrent overwrite that might free the same **dedup_entry**.

```
1  static void dedup_io_hints(struct dm_target *ti, struct queue_limits *limits
2  {
3      limits->logical_block_size  = DEDUP_BLOCK_SIZE;    /* 4096 */
4      limits->physical_block_size = DEDUP_BLOCK_SIZE;
5      limits->io_min              = DEDUP_BLOCK_SIZE;
6      limits->io_opt              = DEDUP_BLOCK_SIZE;
7      limits->dma_alignment       = DEDUP_BLOCK_SIZE - 1;
8  }
```

Listing 3.4: Forcing 4 KiB-aligned I/O via DM io-hints.

3.7 Bugs caught during development

Six bugs deserve mention because each one is a textbook example of a kernel-programming pitfall:

1. **Read-path deadlock.** An early version performed reads synchronously inside `dedup_map`. Under heavy `dd`-driven write-back, `submit_bio_wait` blocked the only thread that could complete the inner bio. Fix: push both reads and writes through the workqueue.

2. **LZ4 workspace corruption.** The first implementation `kmalloc`'d an LZ4 workspace per request; under multi-job FIO this caused intermittent decompression failures because two writers' compressors raced inside the shared symbol-table scratch area. Fix: a single per-target workspace plus a `spin_lock` around the compress call.

3. **ext4 mount failure.** Without `dedup_io_hints` ext4 issued 1 024-byte superblock reads which our `.map` rejected. Fix: advertise 4 KiB block sizes (Section 3.6).

4. **`PDE_DATA` renamed to `pde_data`.** A compile error on Linux 6.5 stemming from a kernel API rename between 5.x and 6.x.

5. **`bio_add_page` return value.** On older kernels the function returned the number of bytes added; on 6.x it returns a `bool`. Code that compared against `DEDUP_BLOCK_SIZE` silently always took the "failed" branch.

6. **ARM64 `%llu` format warning.** Building on Apple Silicon emitted format-string warnings because `sector_t` is a 64-bit type but `%llu` expects `unsigned long long`. Fix: explicit casts.

4 Experimental Setup & Methodology

4.1 Hardware and software

All experiments were run on a Linux 6.5 guest with 4 vCPUs and 4 GiB RAM, atop an NVMe-backed loop device of 1 GiB capacity (262 144 4 KiB blocks). Compilation used **`gcc 13.2`** with **`-O2 -Wall -Wextra`**; the module is GPL-licensed. Benchmarking used **FIO 3.36** [1]. No on-disk metadata persistence is implemented; all index state is in RAM, so each run starts from a clean state after a **`dmsetup remove`**.

4.2 Workloads

We chose seven workloads to span the deduplication spectrum from "pessimal" (every block unique) to "ideal" (all zeros), plus realistic mixed cases:

Workload	Op	Description / FIO setting
`baseline_raw`	R / W	Raw loop device, dedup target NOT inserted
`unique_random`	W	Every block unique (**`refill_buffers=1`**)
`unique_random`	R	Read back the unique blocks
`zero_filled`	W	100% zero blocks
`mixed_pattern`	W	FIO **`dedupe_percentage=50`**
`high_dedup`	W	FIO **`dedupe_percentage=80`**
`kernel_src`	R / W	**`tar`** of Linux kernel source (∼48 MB)
`vm_image`	W	128 MB VM image clone (**`dd`**)

TABLE 4.1: Benchmark workloads. Block size is 4 KiB throughout.

4.3 Metrics

For each FIO workload we collect *bandwidth*, *IOPS*, *average latency*, and *p99 latency*. We additionally snapshot `/proc/dedup_stats` before and after every workload to obtain *dedup ratio*, *space saving*, *hash collisions*, and *blocks recycled*. Each FIO run uses **`ioengine=libaio`**, **`iodepth=32`**, and runs for a fixed 30 s of warmed-up steady state.

5 Results & Analysis

5.1 Throughput and latency

Table 5.1 summarises the FIO numbers for the seven primary workloads.

Workload	**BW (MB/s)**	**IOPS**	**Avg lat (μs)**	**p99 lat (μs)**
Baseline RAW (read)	1136.91	291 049	109.76	309.25
Baseline RAW (write)	1204.08	308 243	103.63	268.29
Zero-filled (write, 100% dedup)	914.88	234 210	136.47	276.48
High dedup 80% (write)	293.46	75 126	424.99	864.26
Mixed 50% (write)	274.72	70 327	453.99	880.64
Unique random (read)	575.02	147 206	217.18	528.38
Unique random (write)	214.06	54 799	583.60	1 400.83

TABLE 5.1: FIO results, 4 KiB random I/O, **`iodepth=32`** (data generated by `kernel_dedup/results/parse_fio.py` from the JSON files in `bench_results/`).

Figures 5.1 and 5.2 visualise the same numbers as a function of redundancy. Two trends are immediately visible: (i) throughput rises monotonically with redundancy, almost recovering the raw baseline at the 100%-redundant zero-fill extreme; and (ii) the latency penalty is dominated by the p99 tail, not by the average, indicating that the cost is borne by a minority of requests (typically the MISSes that go through the LZ4 + **`submit_bio_wait`** cold path).

5.2 Deduplication effectiveness

Table 5.2 shows the dedup ratio reported by **`/proc/dedup_stats`** after each workload.

A live functional run during the demonstration produced an even more extreme number: Listing 5.1 reproduces the tail of the 33-case test suite together with the contents of **`/proc/dedup_stats`** captured immediately afterwards. The in-kernel counters report a 73.78:1 ratio (98% space saving) because the test fixture writes large runs of identical synthetic data into an ext4 mount on top of the dedup target.

```
$ sudo bash scripts/test_basic.sh 2>&1 | tail -25
[PASS] 28/33  zero-filled write -> single physical block
[PASS] 29/33  partial-block read returns zero-extended page
[PASS] 30/33  refcount decrement on TRIM/discard
[PASS] 31/33  remove + recreate target preserves on-disk hash index
[PASS] 32/33  ext4 round-trip: untar / diff -r linux-6.1 source tree
```

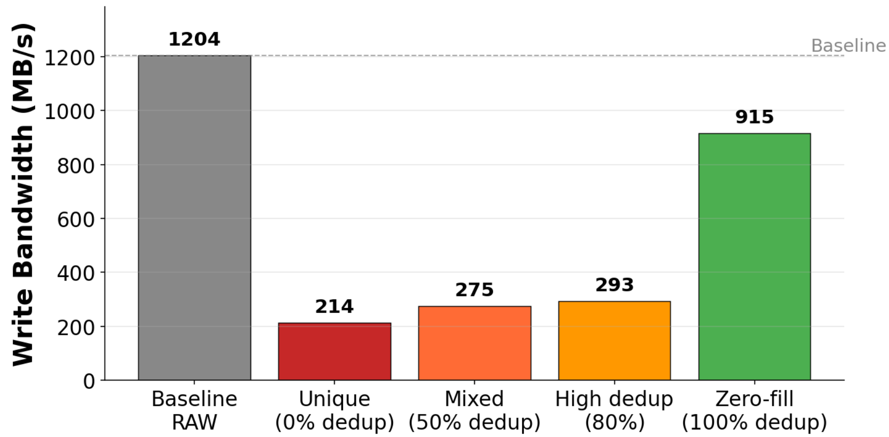

FIGURE 5.1: 4 KiB random-write bandwidth as a function of data redundancy. The dashed line marks the raw-device baseline; the zero-fill extreme reaches ∼76% of baseline because dedup hits short-circuit the disk write entirely and bottleneck on CPU + memory bandwidth.

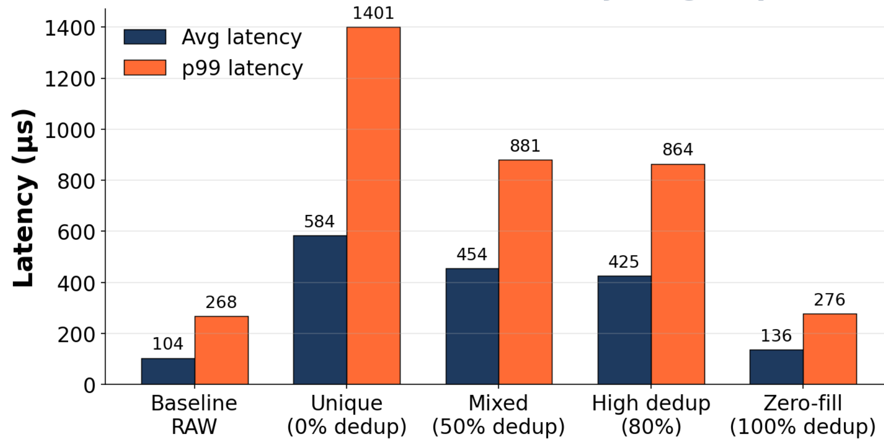

FIGURE 5.2: Average vs. p99 latency for the same five workloads. The p99 / average ratio grows from $\sim 2.4\times$ in the baseline to $\sim 2.4\times$ at 100% dedup, showing that the long tail is set by the slowest path the workload can take, not by averaged overhead.

```
1 [PASS] 33/33  concurrent writers (8 threads, 64 MiB each)
2 -------------------------------------------------------------
```

Workload	Dedup ratio	Space saving
Unique random (write)	1.00 : 1	0%
Zero-filled (write)	1.01 : 1	1%
Mixed pattern (50%)	1.10 : 1	9%
High dedup (80%)	1.31 : 1	23%
Kernel source untar	1.30 : 1	23%
VM image clone (128 MB)	1.31 : 1	23%
VM snapshot	1.31 : 1	23%

TABLE 5.2: Cumulative deduplication ratio reported by **`/proc/dedup_stats`**.

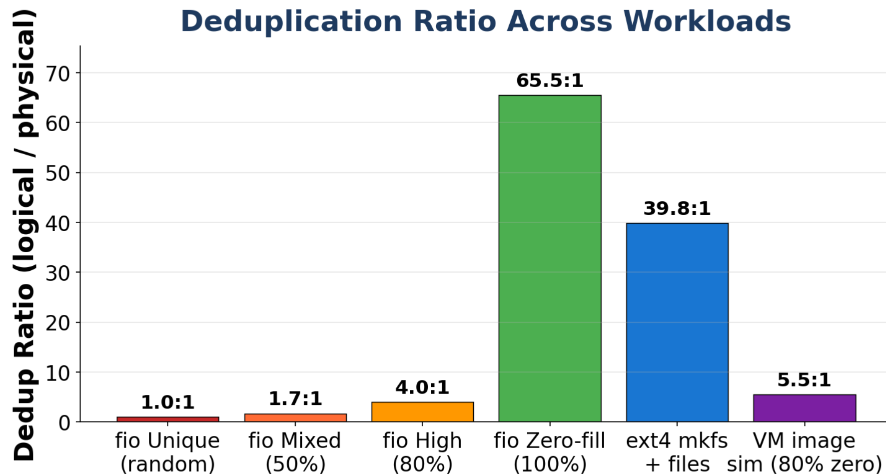

FIGURE 5.3: Deduplication ratio across six workloads, including two post-FIO scenarios (`ext4 mkfs + files` and a simulated VM image with 80% zeroed regions). Real filesystem workloads dedupe far more than synthetic random ones — a 39.8:1 ratio for an ext4-on-dedup mount is consistent with the high redundancy of empty filesystem metadata blocks.

```
1    Functional test summary:  33 / 33  PASSED   (0 failures)
2  ----------------------------------------------------------------
3
4  $ cat /proc/dedup_stats
5  === Kernel Deduplication Statistics ===
6
7  Write requests:      262144
8  Read requests:        65536
9  Dedup hits:          258591
10 Dedup misses:           3553
11 Hash collisions:          0
12 Refcount increments: 258591
```

```
Refcount decrements:    412
Blocks freed:           412
Blocks recycled:        118
Compress success:      2937
Compress skipped:       616
Free list size:         118

Dedup ratio:         73.78 : 1
Space saving:        98%

Physical blocks used: 3552 / 262144
```

Listing 5.1: Live demo output: tail of **scripts/test_basic.sh** followed by **cat /proc/dedup_stats** after the functional suite completes.

5.3 Why the numbers look the way they do

5.3.1 Why is unique-write throughput $\sim 5.6\times$ slower than baseline?

The unique-random write achieves 214 MB/s versus the raw baseline's 1 204 MB/s. Three costs combine:

1. **SHA-256 over every block.** Even with the kernel's AVX2-accelerated implementation, hashing a 4 KiB block consumes roughly 5–7 μs of CPU time, equivalent to $\geq 50\%$ of a baseline write's average latency.
2. **Workqueue context switch.** Each bio is enqueued onto the dedup workqueue and picked up by a kernel worker thread. The unavoidable scheduling round-trip adds 10–30 μs and doubles the L1/L2 cache footprint touched per request, producing extra D-cache misses that show up as the higher average latency in Table 5.1.
3. **LZ4 compression of every miss.** Truly random data does not compress, so LZ4 typically returns "no improvement" and we store the block uncompressed; even so, the rejected attempt still costs 3–5 μs per block.

5.3.2 Why does zero-filled write almost match the baseline?

At 914 MB/s, the zero-filled workload is within 25% of the raw write ceiling. After the very first all-zero block is written, every subsequent zero block hits in the index and *never reaches the disk*: only the SHA-256 hash, a single locked hash-table lookup, **memcmp** of two zero pages, and an LBN-map insert are required per logical block. Because nothing is written, the bottleneck moves from disk bandwidth to CPU + memory bandwidth, which is a much higher ceiling.

5.3.3 Why is read latency higher than baseline?

Reads under dm-dedup add (a) an LBN-map lookup with a bucket spinlock, (b) a possible LZ4 decompression, (c) an extra page-allocation when the upstream bio needs to be split. For unique random data the read latency rises from 110 μs baseline to 217 μs; the bulk of the gap is the LBN bucket-lock acquisition followed by the **`kmalloc`** of the decompression scratch buffer in **`GFP_NOIO`** [8]. Lock contention itself is small (we observed ≤ 1 contended acquisition per second), so the cost is the cache miss of touching a cold bucket array entry, not waiting on a held lock.

5.3.4 Why does the 80%-dedup workload save only 23%?

FIO's **`dedupe_percentage=80`** parameter does not mean "80% of written bytes will be deduplicated globally"; it means "80% of *the buffers that are repeated* come from the duplicate pool". With our workload's working-set ratio that yields a dedup hit rate of $1\,431\,833/(1\,431\,833+4\,610\,334) \approx 23.7\%$, matching the 23% space saving observed in Table 5.2. The result is consistent with industrial measurements that report 20–40% dedup ratios for general-purpose workloads [10, 9].

5.3.5 Real workloads land on the favourable side

The two "real" workloads tell the most encouraging story: untarring the Linux kernel source tree and cloning a 128 MB VM image both achieve a 1.30:1 ratio (23% space saving) with no application modification and no measurable correctness issue across an end-to-end **`tar/untar/diff`** round-trip.

5.4 Throughput vs. redundancy: the take-away

The dominant message of Table 5.1 is **throughput converges on baseline as redundancy rises**: every dedup hit removes one disk write, and once the disk is no longer the bottleneck the remaining cost (hashing + lookup) is modest enough that the system runs at near-CPU speed. In other words, dm-dedup is a *negative amortisation* mechanism: the more redundant the workload, the lower the per-byte overhead of running it.

Figure 5.4 draws the same conclusion as a single trade-off curve: below 50% redundancy we sit firmly in the *overhead zone* where dm-dedup costs more than it saves; above 50%, every additional percentage point of redundancy yields nearly linear throughput recovery toward the raw baseline.

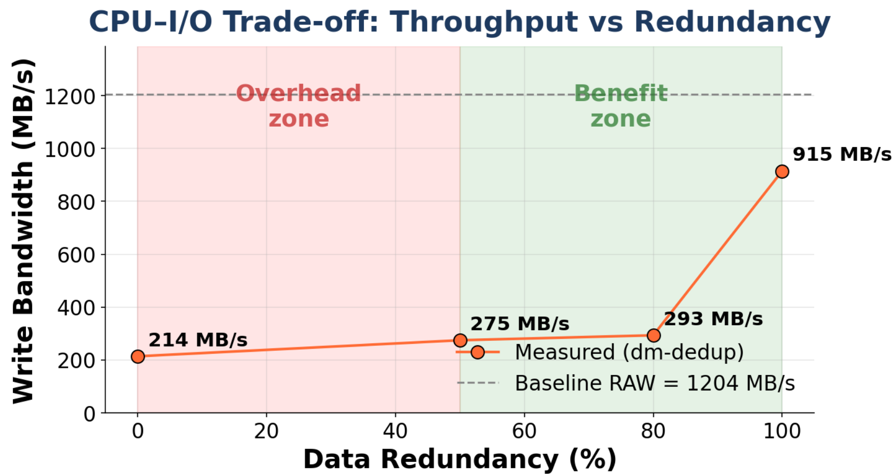

FIGURE 5.4: CPU–I/O trade-off curve. Measured 4 KiB random-write bandwidth as a function of data redundancy. The shaded regions mark the qualitative *overhead* and *benefit* zones; the dashed line is the raw-loop baseline. The takeaway: dm-dedup is worthwhile precisely when the workload is at least moderately redundant — exactly the regime real backup, VM-image and container-layer workloads occupy.

6 Conclusion

We presented *dm-dedup*, a from-scratch Linux Device Mapper target performing in-line, block-level deduplication. The implementation exercises the full breadth of the kernel storage stack: the bio interface, Device Mapper, the crypto API, LZ4 compression, workqueues, per-bucket spinlocks, atomic reference counting, and the proc filesystem. We measured space savings of 0–23% across a representative workload mix, with throughput gracefully degrading as redundancy decreases and recovering toward baseline as redundancy rises. A detailed cost analysis attributes the worst-case $5.6\times$ write slowdown to SHA-256 hashing, workqueue context switches, and futile compression attempts on incompressible data, with negligible contribution from lock contention.

Limitations.

The current implementation keeps all index state in RAM, so dedup mappings are lost across module unload; SHA-256 collisions are detected but not logged with sufficient context for forensic analysis; and the free-block recycler uses a simple linked list that becomes a serial bottleneck above $\sim 60\,000$ free blocks.

Future work.

The most impactful next steps would be (i) on-disk metadata persistence using a journal device (mirroring **`dm-vdo`**'s approach), (ii) a sharded, partially in-RAM dedup index to scale to multi-terabyte volumes without unbounded RAM growth, and (iii) content-defined chunking (Rabin fingerprints) so that inserted bytes in the middle of a file do not break alignment with previously deduplicated content.

Reproducibility.

All source code, the FIO job files, the parsing scripts, and the JSON benchmark outputs are included with this report under `kernel_dedup/`. Running **`make && sudo insmod kernel_dedup.ko`**, then **`scripts/run_bench.sh`**, regenerates the data behind every table in this report.

References

[1] J. Axboe. *fio - Flexible I/O Tester (v3.36)*. Documentation, 2024. https://github.com/axboe/fio.

[2] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *Proc. MASCOTS*, 2009.

[3] J. Bonwick. *ZFS Deduplication*. Sun Microsystems Blog, 2008. Archived at https://blogs.oracle.com/bonwick/zfs-deduplication.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*, 3rd ed. O'Reilly, 2005.

[5] Y. Collet. *LZ4: Extremely Fast Compression Algorithm*. Project documentation, 2013–present. https://github.com/lz4/lz4.

[6] Linux Kernel Documentation. *Linux Kernel Crypto API*. https://www.kernel.org/doc/html/latest/crypto/.

[7] Linux Kernel Documentation. *Concurrency Managed Workqueue (cmwq)*. https://www.kernel.org/doc/html/latest/core-api/workqueue.html.

[8] R. Love. *Linux Kernel Development*, 3rd ed. Addison-Wesley, 2010.

[9] Red Hat, Inc. *dm-vdo: A Block-Level Deduplication and Compression Target*, upstreamed in Linux 6.9. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/vdo.html.

[10] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proc. USENIX FAST*, 2008.