Operating Systems Project: Topic 15

Kernel-Level Data Deduplication Mechanism

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.2.5

Outline

1. Part 1: Requirements & Scope
2. Part 2: Theoretical Foundations
3. Part 3: Kernel Architecture
4. Part 4: Implementation Details
5. Part 5: Evaluation Plan

Outline

1. Part 1: Requirements & Scope
2. Part 2: Theoretical Foundations
3. Part 3: Kernel Architecture
4. Part 4: Implementation Details
5. Part 5: Evaluation Plan

1.1 Core Requirements (Topic 15)

Objective: Implement an *In-line* Deduplication mechanism within the Linux Kernel.

Mandatory Tasks (The "Must-Haves")

1. **Interception Point:** Modify `fs/buffer.c` or the Block Layer to intercept `submit_bio()` or buffer head operations.
2. **Fingerprinting:** Utilize the Kernel Crypto API to compute hashes (e.g., SHA-256) for data blocks.
3. **Mapping Logic:** Maintain a `Hash` → `Physical Block` index to map multiple logical writes to a single physical sector.

Constraint

This must be done in **Kernel Space**. User-space solutions (like FUSE) are not allowed for the core logic.

1.2 Advanced Scope & Goals

Beyond basic functionality, we aim for robustness and efficiency.

A. Optimization (Performance)

- **Compression:** Integrate `LZ4` or `ZLIB` to compress unique blocks before writing.
- **Async Processing:** Move hashing to a workqueue to avoid blocking the main write thread.

B. Robustness (Safety)

- **Collision Handling:** What happens if two different data blocks produce the same hash? (Hash Collision).
- **Reference Counting:** Ensuring a block is never freed while a file still references it.

Outline

1 Part 1: Requirements & Scope

2 Part 2: Theoretical Foundations

3 Part 3: Kernel Architecture

4 Part 4: Implementation Details

5 Part 5: Evaluation Plan

2.1 The Problem: Data Redundancy

Modern storage systems suffer from massive redundancy.

- **Scenario 1: Virtual Machines.** Running 10 Ubuntu VMs results in 10 copies of the exact same OS binaries (/bin/bash, kernel, libraries).
- **Scenario 2: Backups.** Incremental backups often store unchanged data blocks repeatedly.
- **The Cost:**
  - **Space:** Wasted $$$ on SSDs/HDDs.
  - **Endurance:** SSD Flash cells wear out faster due to *Write Amplification*.

2.2 Choosing the Right Granularity

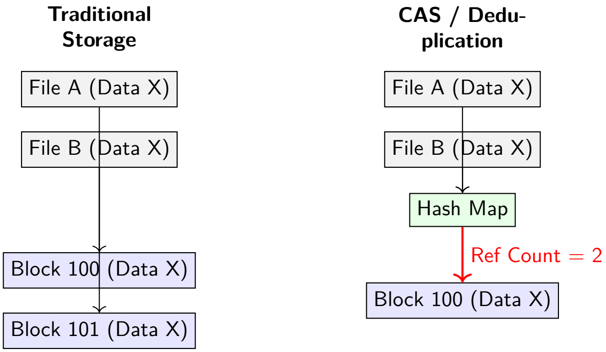
At what level should we deduplicate?

Level	Pros	Cons
File Level	Easy to implement.	Low savings. Changing 1 byte → new file.
Block Level	**High savings.** Balanced CPU cost.	**Complex mapping table.**
Byte Level	Max savings.	Extremely high CPU & RAM overhead.

Our Choice: Fixed-Size Block Level (4KB). This matches the Linux Page Size and File System Block Size.

2.3 Concept: Content-Addressable Storage (CAS)

We shift from *"Where should I write this?"* to *"Do I already have this?"*



2.4 Hashing Strategy & Collisions

The Fingerprint: A unique summary of the data block.

Algorithm Selection

- **CRC32:** Very fast, but high collision risk. (Not suitable for primary key).
- **SHA-256:** Slower, but cryptographically secure. Collision chance is negligible (1 in 10^{77}).

The "Birthday Paradox" (Collision)

Even with SHA-256, strictly speaking, collisions represent data corruption.

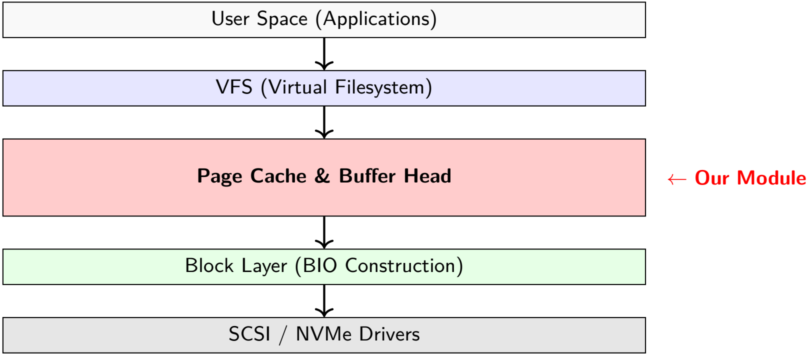
Safeguard Strategy: If Hash matches $\rightarrow$ Perform a **byte-by-byte comparison** (memcmp) to verify data is truly identical before discarding the write.

Outline

1. Part 1: Requirements & Scope
2. Part 2: Theoretical Foundations
3. Part 3: Kernel Architecture
4. Part 4: Implementation Details
5. Part 5: Evaluation Plan

3.1 Where in the Kernel?

Linux IO Stack is layered. We target the **Buffer Cache / Page Cache** boundary.



3.2 Kernel Data Structures (Conceptual)

We need persistent structures to track duplicates.

The Deduplication Entry Node

Every unique block in memory has a descriptor:

- **Fingerprint (32 bytes):** The SHA-256 Hash.
- **Physical Address (8 bytes):** Sector number on disk.
- **Reference Counter (4 bytes):** Atomic integer. How many files claim this block?
- **List Pointer:** For handling hash bucket chains.

Global Hash Index

- A Hash Map in Kernel RAM.
- Protected by **Spinlocks** to allow concurrent access by multiple CPUs.

3.3 The "Write" Workflow

What happens when `write()` is called?

1. **Intercept:** Catch the dirty page in `fs/buffer.c`.
2. **Calculate:** Run SHA-256 on the 4KB data page.
3. **Lookup:** Check `dedupe_index` for this hash.
4. **Decision:**
   - *Found (Hit):*
     - Increment `ref_count` of existing block.
     - Update Inode mapping.
     - **Mark page clean (Skip disk write).**
   - *Not Found (Miss):*
     - Allocate new `phys_block`.
     - Write data to disk.
     - Insert new entry into `dedupe_index`.

Outline

1. Part 1: Requirements & Scope
2. Part 2: Theoretical Foundations
3. Part 3: Kernel Architecture
4. Part 4: Implementation Details
5. Part 5: Evaluation Plan

4.1 Using the Kernel Crypto API

We leverage the existing Linux Crypto Subsystem (not writing SHA-256 from scratch).

1. Allocate Transform (`sha256`) → 2. Init Request → 3. Digest Data

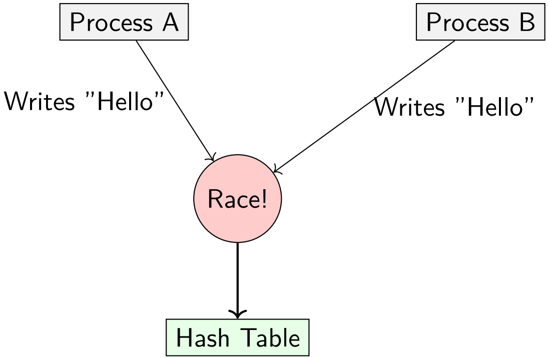
Input: 4KB Page Buffer
Output: 32-Byte Hash

Key Implementation Logic:

- Use `kmalloc` for temporary buffer allocation.
- Use `crypto_shash_digest()` for the actual calculation.
- Ensure `kfree` is called to prevent memory leaks.

4.2 Concurrency & Race Conditions

The Kernel is highly multi-threaded. Two processes might write the same data simultaneously.



Solution: Fine-grained locking.

- Instead of locking the whole table, we use **Bucket Locking**.
- Lock only the specific hash bucket relevant to the data, allowing parallel processing for other data.

Outline

1. Part 1: Requirements & Scope
2. Part 2: Theoretical Foundations
3. Part 3: Kernel Architecture
4. Part 4: Implementation Details
5. Part 5: Evaluation Plan

5.1 Evaluation Metrics

1. Storage Efficiency

$$\text{Dedupe Ratio} = \frac{\text{Logical Data Size}}{\text{Physical Disk Usage}}$$

- Target: $10:1$ for VM Backups.
- Target: $1:1$ for Encrypted Data (Worst case).

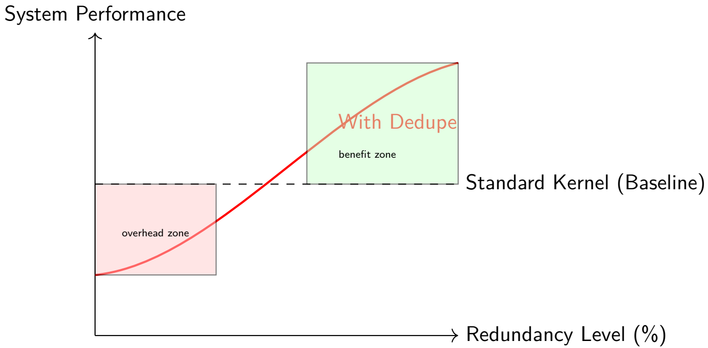
2. Performance Overhead

- **Latency:** Time per 4KB write (ms).
- **Throughput:** MB/s under high load.
- **CPU Usage:** % of CPU spent in SHA-256 calculation.

5.2 Experimental Workloads

We will use **FIO** and **Real Data** for testing.

Test	Data Type	Hypothesis
Baseline	Unique Random	Overhead only. Performance drops.
Ideal	Zero-filled	100% Dedupe. Extremely fast write (no I/O).
Real World 1	Linux Kernel Src	5-10% Dedupe (Code reuse).
Real World 2	VM Disk Images	>50% Dedupe (Same OS files).



At low redundancy, hashing cost > I/O savings. At high redundancy, eliminating I/O > hashing cost.

Summary

1. **Goal:** Kernel-level In-line Deduplication.
2. **Mechanism:** Intercept `buffer_head`, Hash Content, Map to Physical Block.
3. **Impact:** Reduce Disk Usage & Write Amplification.
4. **Challenge:** Balancing CPU Overhead vs. Storage Savings.

Q & A