Kernel-Level Buffer Cache Instrumentation and LFU-Based Replacement Optimization

Wang Peizhu and He Haodong
{225040176, 225040162}@link.cuhk.edu.cn

***Abstract*—This project studies kernel-level buffer cache instrumentation and a frequency-aware optimization for Linux buffer management. The implementation modifies the Linux buffer-cache path in `fs/buffer.c`, extends `struct buffer_head` with an atomic access counter, records cache hits and misses using low-overhead per-CPU counters, and exports runtime statistics through `/proc/cache_stats`. On top of this observability layer, the project implements a bounded Least Frequently Used (LFU) enhancement with linear decay so that frequently accessed metadata buffers can survive short-term memory pressure while stale hot buffers gradually lose protection. The evaluation uses a custom Linux 5.15.202 kernel on Ubuntu 22.04 LTS, a 1.5GB ext2 loop device, controlled memory pressure, and `fio`-based sequential, random-write, and Zipfian workloads. The results show that LFU improves cold random-write and Zipfian database-like workloads substantially, while its benefit is small for sequential scans and can become slightly negative in already warm workloads. The project therefore supports a bounded conclusion: frequency-sensitive buffer retention is useful for skewed and metadata-intensive access patterns under pressure, but it is not a universal replacement for recency-based behavior.**

***Index Terms*—Linux kernel, buffer cache, buffer_head, LFU, LRU, cache instrumentation, per-CPU counters, `/proc/cache_stats`, linear decay, fio benchmark.**

I. INTRODUCTION

MODERN operating systems rely heavily on memory-based caching to reduce the performance gap between main memory and persistent storage. Although storage devices have become faster over time, accessing data from disk or SSD is still much more expensive than serving the same data from RAM. For this reason, the Linux kernel aggressively caches file-system data and metadata in memory so that repeated accesses can be satisfied without issuing unnecessary block I/O requests [1], [5], [9].

In the Linux storage stack, the buffer cache plays an important role in managing block-level metadata. While the page cache is mainly responsible for caching file contents at page granularity, buffer heads are still used to describe the relationship between memory pages and physical disk blocks. Such metadata is especially important for file-system structures such as superblocks, inode tables, allocation bitmaps, and directory blocks. Therefore, improving the behavior of the buffer cache can directly reduce the cost of metadata-intensive operations such as directory traversal, file creation, random updates, and database-like access patterns [4], [11].

This project focuses on kernel-level buffer cache instrumentation and frequency-aware optimization. Specifically, we modify the Linux buffer cache path to measure cache hits and misses, expose these measurements to user space, and evaluate whether a bounded Least Frequently Used (LFU) enhancement can better preserve frequently accessed metadata buffers under memory pressure. The implementation is developed on Ubuntu 22.04 LTS with Linux 5.15.202 and targets the core buffer-cache-related files `fs/buffer.c` and `include/linux/buffer_head.h`.

The main contributions of this project are as follows. First, we add kernel-level instrumentation to record buffer cache hits and misses. Second, we use lockless per-CPU counters to reduce overhead on the hot lookup path. Third, we extend each buffer head with an atomic frequency counter and implement a bounded LFU policy. Finally, we introduce linear decay during buffer reclamation so that hot buffers survive short-term memory pressure, while stale buffers can eventually be reclaimed.

II. MOTIVATION

The main problem addressed in this project is that the default recency-oriented caching behavior does not explicitly distinguish between one-time scan blocks and repeatedly accessed metadata blocks. A Least Recently Used style policy works well when recent accesses predict future accesses, but it can suffer from scan pollution. For example, a large sequential scan may bring many blocks into memory even though they will not be reused, while pushing out metadata blocks that are frequently accessed but not necessarily the most recent. This behavior can increase cache misses and lead to additional disk I/O under memory pressure [3], [6], [7].

This issue is especially relevant for metadata-heavy workloads. In workloads such as random writes, directory traversal, and database-like skewed access, a small group of metadata blocks may be accessed repeatedly. These blocks may include inode tables, allocation bitmaps, directory blocks, and other filesystem management structures. If such hot metadata is evicted because of a short-term scan or broader memory pressure, later operations must reload or reconstruct the corresponding block-level state, increasing lookup latency and I/O cost.

This project, therefore, has two closely related goals. The first goal is observability: the kernel should provide a

CSC5031 Operating Systems Project – Topic 10. Wang Peizhu: Student ID 225040176. He Haodong: Student ID 225040162. May 2026.

lightweight mechanism for measuring buffer cache hit and miss behavior during real workloads. Without such instrumentation, it is difficult to evaluate whether a replacement policy is actually improving cache behavior. The second goal is optimization: frequently accessed metadata buffers should be protected more effectively when the system needs to reclaim memory.

To achieve these goals, we instrument the buffer lookup path, expose aggregated cache statistics to the user space, and implement a bounded LFU replacement enhancement with linear decay. The implementation extends the buffer metadata structure with an atomic access counter, updates the counter when buffers are touched, and modifies the buffer reclaim path so that high-frequency buffers are less likely to be immediately evicted. At the same time, the design avoids naive LFU's "ghost item" problem by bounding the counter and gradually decaying stale access counts. The implemented system, therefore, improves both visibility and replacement behavior while preserving the correctness of the original buffer I/O path [7], [8].

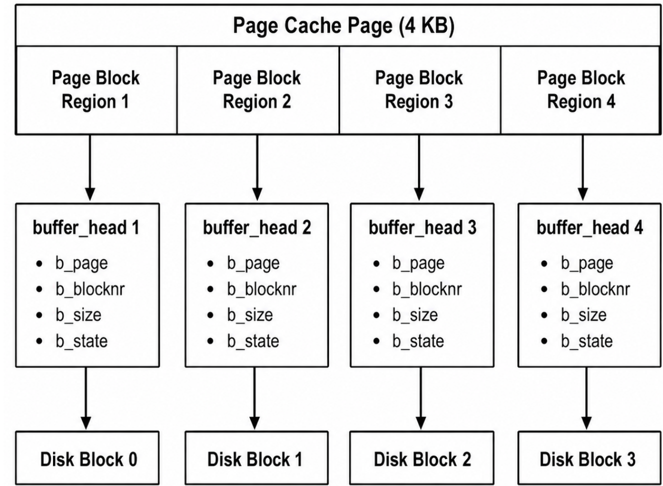

Fig. 1. Relationship between a 4KB page-cache page, its block-sized regions, and the corresponding `buffer_head` descriptors. Each `buffer_head` stores metadata such as `b_page`, `b_blocknr`, `b_size`, and `b_state`, thereby linking an in-memory page region to a disk block.

III. Background and Related Work

A. Linux Buffer Cache

The Linux buffer cache is a block-level caching mechanism used by file systems and block devices. Its basic unit is the `buffer_head`, which describes a block-sized region within a memory page and records its relationship with an underlying disk block. A buffer head stores state flags, block number, block size, the backing page, the associated block device, and a reference counter. These fields allow the kernel to determine whether a buffer is mapped, up to date, dirty, locked, or currently in use [4], [11].

Although file data is generally cached by the page cache, buffer heads are still used for block-level metadata management. A single page may contain multiple buffer heads, and each buffer head can represent a smaller block inside that page. This distinction is important because page-level replacement and buffer-level metadata management operate at different granularities. In this project, the buffer head is treated as the key object for instrumentation and frequency-aware replacement.

Fig. 1 illustrates the relationship between a page cache page and its associated buffer heads.

B. Buffer Lookup and Hit/Miss Semantics

A buffer cache lookup checks whether the requested block already has a corresponding buffer head in memory. If the buffer is found, the lookup returns a valid buffer head and the request can be served without issuing a new disk read. This event is counted as a cache hit. If the buffer is not found, the lookup returns `NULL`, and the caller must allocate or load the corresponding buffer. This event is counted as a cache miss.

The function `__find_get_block()` is the central lookup point used in this project. It first searches the per-CPU buffer-head LRU and then falls back to a slower page-cache based lookup. This path provides a natural instrumentation point because it directly reflects whether the kernel can locate a requested block from memory [11].

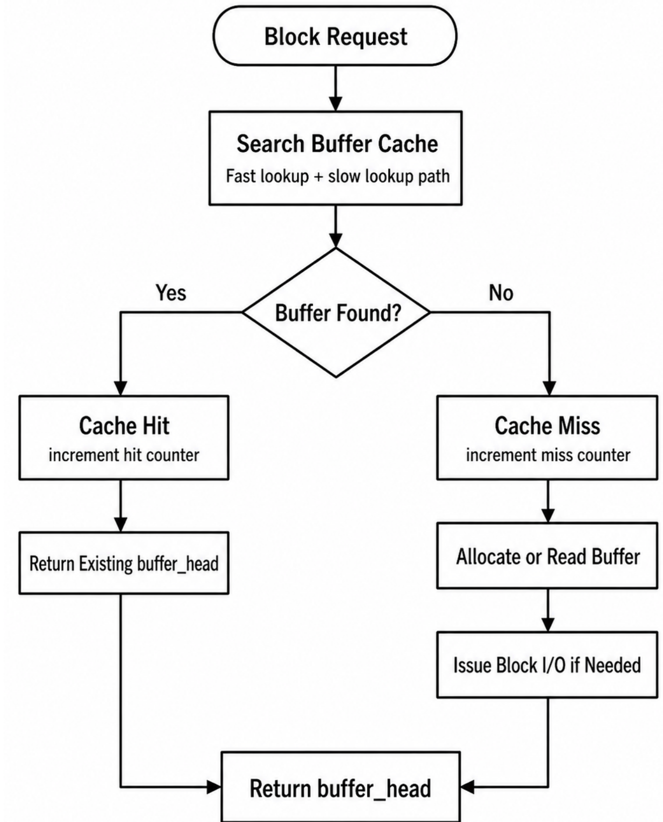

Fig. 2. Buffer lookup flow and hit/miss semantics. A block request enters the buffer-cache lookup path. If an existing `buffer_head` is found, the operation is counted as a cache hit and the existing descriptor is returned. Otherwise, the operation is counted as a cache miss, the kernel allocates or reads the buffer, and block I/O may be issued if needed.

Conceptually, the lookup path can be summarized as shown in Fig. 2.

C. Cache Replacement Policies

The default cache behavior in Linux is largely recency-oriented. Least Recently Used policies assume that recently accessed blocks are likely to be accessed again soon. This assumption is effective for many workloads with strong temporal locality. However, LRU-like policies can be vulnerable to scan pollution. A one-time sequential scan can replace useful cached metadata with blocks that will not be reused [3], [6].

Least Frequently Used replacement uses access frequency rather than access recency. It assumes that blocks accessed many times in the past are more likely to remain valuable. This makes LFU attractive for metadata-heavy workloads, where a small set of metadata blocks, such as bitmaps, inode tables, or directory blocks, may be accessed repeatedly. However, naive LFU may keep old “ghost” items forever if their historical access counts remain high [8].

The design in this project, therefore, adopts a practical variant of LFU. Instead of allowing access counts to grow without limit, each buffer has a bounded counter. In addition, the reclaim path applies linear decay: a hot buffer loses one count when it resists eviction. This design keeps the benefit of frequency awareness while allowing stale buffers to cool down over time. Similar concerns about scan resistance and adaptive cache replacement have also motivated prior replacement algorithms such as ARC and LRU-K [7], [8].

D. Design Challenges

There are three main challenges in adding LFU behavior to the kernel buffer cache. First, the lookup path is performance-sensitive. Adding heavy locks or expensive operations to every lookup could reduce system performance and distort the measured results. Second, buffer heads are accessed concurrently on multi-core systems, so frequency counters must be safe under concurrent updates. Third, cache replacement must not violate the correctness of buffer I/O. Dirty, locked, or referenced buffers should not be incorrectly freed.

To address these challenges, the implementation uses per-CPU counters for global statistics, atomic counters for per-buffer access frequency, and conservative changes to the existing buffer reclaim path. In this way, the design changes eviction preference without changing the fundamental safety rules of the Linux buffer cache.

IV. Design and Implementation

A. Overall Architecture

The implementation consists of three layers: instrumentation, statistics reporting, and frequency-aware replacement. The instrumentation layer records hit and miss events in the buffer lookup path. The statistics layer aggregates per-CPU counters and exposes the results to the user space. The replacement layer extends buffer metadata with an access counter and modifies the reclaim path to protect frequently accessed buffers.

This architecture is shown in Fig. 3. The user-space workload triggers file-system operations through the VFS layer. The buffer cache layer then performs lookup, touch, and reclaim

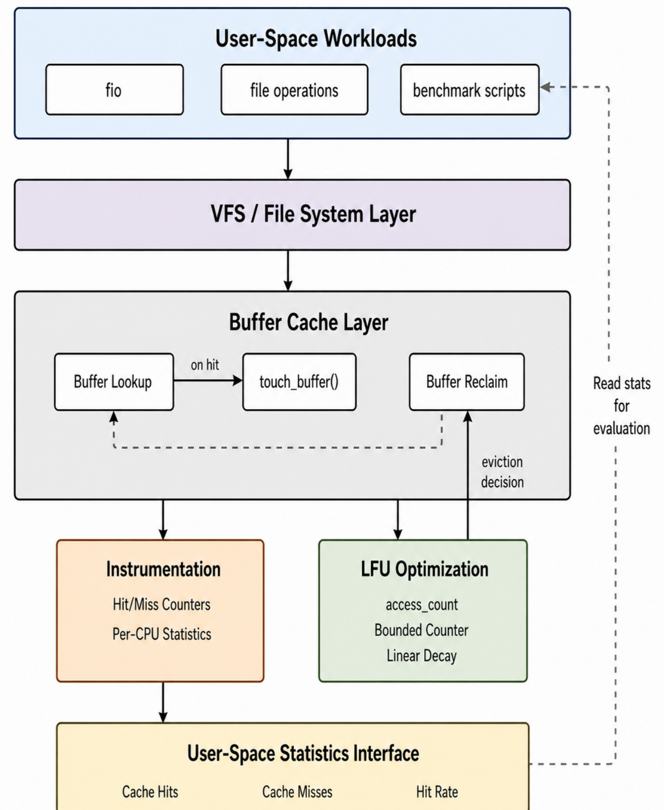

Fig. 3. Overall architecture of buffer cache instrumentation and LFU-based replacement. User-space workloads such as `fio`, file operations, and benchmark scripts pass through the VFS/file-system layer into the buffer-cache layer. The instrumentation component records hit/miss counters with per-CPU statistics, while the LFU optimization component uses `access_count`, a bounded counter, and linear decay to influence reclaim decisions. The aggregated statistics are exposed to the user space for evaluation.

operations. Hit/miss counters provide runtime observability, while the LFU access counter influences replacement decisions.

B. Cache Hit and Miss Instrumentation

The first modification records whether a buffer lookup succeeds or fails. The implementation instruments `__find_get_block()`. When the buffer is found in the per-CPU LRU, the event is treated as a hit. When the fast lookup fails, the kernel invokes the slower lookup path; in this implementation, that case is recorded as a miss. The modified function also calls `touch_buffer()` on cache hits so that the LFU access counter is updated together with the hit event [11].

The key logic is shown in Code Listing 1. This design keeps the instrumentation close to the actual lookup decision. It does not require external tracing tools to infer cache behavior. Instead, the kernel directly records lookup outcomes at the point where they occur.

```
1 struct buffer_head *
2 __find_get_block(struct block_device *bdev,
3                  sector_t block,
4                  unsigned size)
5 {
```

```
6      struct buffer_head *bh;
7
8      bh = lookup_bh_lru(bdev, block, size);
9
10     if (bh == NULL) {
11         bh = __find_get_block_slow(bdev, block);
12         if (bh)
13             bh_lru_install(bh);
14
15         this_cpu_inc(cpu_cache_misses);
16     } else {
17         touch_buffer(bh);
18         this_cpu_inc(cpu_cache_hits);
19     }
20
21     return bh;
22 }
```

Code Listing 1. Hit and miss instrumentation in the buffer lookup path.

C. Lockless Per-CPU Statistics

A simple global counter protected by a spinlock would be easy to implement, but it would be unsuitable for a hot kernel path. Buffer lookups can happen frequently, and a global lock would create unnecessary contention on multi-core systems. To avoid this problem, the implementation uses per-CPU counters.

Each CPU updates its own hit and miss counters independently. When the user space reads the statistics interface, the kernel aggregates all per-CPU values and computes the hit rate. This design trades slightly more complex reading logic for much cheaper updates on the performance-critical path. The aggregation logic is shown in Code Listing 2.

```
1  static DEFINE_PER_CPU(unsigned long,
2                        cpu_cache_hits) = 0;
3  static DEFINE_PER_CPU(unsigned long,
4                        cpu_cache_misses) = 0;
5
6  static int my_stats_show(struct seq_file *m,
7                           void *v)
8  {
9      unsigned long hits = 0;
10     unsigned long misses = 0;
11     unsigned long hit_rate;
12     int cpu;
13
14     for_each_possible_cpu(cpu) {
15         hits += per_cpu(cpu_cache_hits, cpu);
16         misses += per_cpu(cpu_cache_misses, cpu);
17     }
18
19     hit_rate = (hits * 100) / (hits + misses + 1);
20
21     seq_printf(m, "Buffer Cache Hits: %lu\n",
22                hits);
23     seq_printf(m, "Buffer Cache Misses: %lu\n",
24                misses);
25     seq_printf(m, "Hit Rate: %lu%%\n",
26                hit_rate);
27
28     return 0;
29 }
```

Code Listing 2. Per-CPU counters and statistics aggregation.

The `+1` in the denominator prevents division by zero when the statistics file is read before any lookup events have been recorded. Although this slightly affects the displayed rate when the total count is extremely small, it has no meaningful impact once real workloads are running.

D. User-Space Statistics Interface

The instrumentation is useful only if the measured values can be observed from the user space. Therefore, the implementation adds a lightweight read-only statistics interface. This interface reports the total number of buffer cache hits, the total number of misses, and the computed hit rate. It allows benchmark scripts to sample the counters before and after each workload and compute per-workload deltas.

This design is intentionally simple. Instead of requiring `perf`, `ftrace`, or custom kernel logging, the benchmark only needs to read the exported statistics file. This makes the evaluation workflow repeatable and easy to automate.

E. Extending Buffer Metadata

To implement LFU-style replacement, the kernel must remember how frequently each buffer has been accessed. For this purpose, `struct buffer_head` is extended with an atomic access counter. The modified header file adds `atomic_t access_count` after the original synchronization-related fields, as shown in Code Listing 3 [4], [11].

```
1  struct buffer_head {
2      unsigned long b_state;
3      struct buffer_head *b_this_page;
4      struct page *b_page;
5
6      sector_t b_blocknr;
7      size_t b_size;
8      char *b_data;
9
10     struct block_device *b_bdev;
11     bh_end_io_t *b_end_io;
12     void *b_private;
13
14     struct list_head b_assoc_buffers;
15     struct address_space *b_assoc_map;
16
17     atomic_t b_count;
18     spinlock_t b_uptodate_lock;
19
20     /* Custom field for LFU replacement */
21     atomic_t access_count;
22 };
```

Code Listing 3. Extending struct buffer_head with an access counter.

Using an atomic counter is important because the buffer cache is accessed concurrently. A normal integer could suffer from lost updates, while adding a spinlock to every access would increase overhead. The atomic counter provides a lightweight and concurrency-safe mechanism for recording per-buffer access frequency.

F. Bounded LFU Counter Update

The access counter is updated when a buffer is touched. The implementation modifies `touch_buffer()` so that it increments `access_count` whenever the buffer is accessed, but only up to a fixed upper bound. The current implementation caps the counter at 50. This bound prevents old hot buffers from accumulating extremely large counts and becoming practically unevictable.

Code Listing 4 shows the bounded counter update in `touch_buffer()`.

```
inline void touch_buffer(struct buffer_head *bh)
{
    trace_block_touch_buffer(bh);
    mark_page_accessed(bh->b_page);

    if (atomic_read(&bh->access_count) < 50) {
        atomic_inc(&bh->access_count);
    }
}
```

Code Listing 4. Bounded LFU counter update in `touch_buffer()`.

This bounded design directly addresses the weakness of naive LFU. Without a bound, a buffer that was frequently accessed in the past could remain protected long after it stops being useful. With a bound, the system still recognizes hot buffers, but the counter remains controlled.

G. Counter Initialization

Whenever a buffer is initialized for a new block mapping, its access counter must be reset. Otherwise, a buffer head reused for a different block might carry stale frequency information from its previous lifetime. The implementation resets `access_count` during buffer initialization, preventing frequency state leakage between unrelated block mappings.

Code Listing 5 shows the counter reset logic.

```
if (!buffer_mapped(bh)) {
    bh->b_end_io = NULL;
    bh->b_private = NULL;
    bh->b_bdev = bdev;
    bh->b_blocknr = block;

    atomic_set(&bh->access_count, 0);

    if (uptodate)
        set_buffer_uptodate(bh);

    if (block < end_block)
        set_buffer_mapped(bh);
}
```

Code Listing 5. Resetting the access counter during buffer initialization.

This step is essential for correctness. LFU should measure the access frequency of the current logical block, not the historical frequency of a reused buffer-head object.

H. Linear Decay in Buffer Reclamation

The final part of the LFU implementation modifies the buffer reclaim path. In the original logic, a clean and unused buffer can be detached and freed. The modified logic first checks the access counter. If the buffer has an access count greater than one, the count is decreased by one, and the eviction attempt is skipped. This mechanism is called linear decay.

Code Listing 6 shows the key logic of linear decay during buffer reclamation.

```
static int drop_buffers(struct page *page,
                        struct buffer_head **
    ↪ buffers_to_free)
{
    struct buffer_head *head = page_buffers(page);
    struct buffer_head *bh;

    bh = head;
    do {
        int current_count;

        if (buffer_busy(bh))
            goto failed;

        current_count =
            atomic_read(&bh->access_count);

        if (current_count > 1) {
            atomic_set(&bh->access_count,
                       current_count - 1);
            goto failed;
        }

        bh = bh->b_this_page;
    } while (bh != head);

    /* Existing buffer detach logic follows. */

failed:
    return 0;
}
```

Code Listing 6. Linear decay during buffer reclamation.

Linear decay gives hot buffers temporary protection under memory pressure. A frequently accessed buffer will not be removed immediately; instead, its count cools down gradually. If the buffer continues to be accessed, the counter will be refreshed. If it is no longer used, repeated reclaim attempts will eventually reduce the counter enough for the buffer to be reclaimed.

I. Correctness and Stability Considerations

The implementation is designed to preserve the correctness of the original buffer cache. The LFU logic is inserted only after the existing `buffer_busy()` check. Therefore, buffers that are dirty, locked, or still referenced are not incorrectly freed.

The design also minimizes synchronization overhead. Per-CPU counters make hit/miss recording cheap, while atomic access counters avoid data races without introducing additional locks. As a result, the implementation improves observability and replacement preference without fundamentally changing the I/O semantics of the buffer cache.

V. Experimental Design

The experimental design follows the same implementation scope as the preceding kernel modifications. The measured quantity is buffer lookup behavior in the buffer-cache path, not application-level throughput alone, and not a generic page-cache hit rate. Therefore, the workloads are designed to exercise block lookup, metadata updates, and memory-pressure-driven eviction rather than only large sequential file reads.

A. Environment

The experiment uses a custom-compiled Linux kernel with the LFU modifications. The software and storage environment are summarized in Table I.

The ext2 loop-device setup is used because it gives a controlled block-oriented test environment. The benchmark file is large enough to exceed the forced available memory budget, which helps trigger eviction behavior.

TABLE I
EXPERIMENTAL ENVIRONMENT.

Item	Configuration
Operating system	Ubuntu 22.04 LTS
Kernel	Linux 5.15.202, custom compiled
Filesystem	ext2 on loop device
Loop device size	1.5 GB
Mount point	`/mnt/ext2_final`
Test payload	1 GB `fio_test.dat`
Benchmark tool	`fio`
Statistics interface	`/proc/cache_stats`

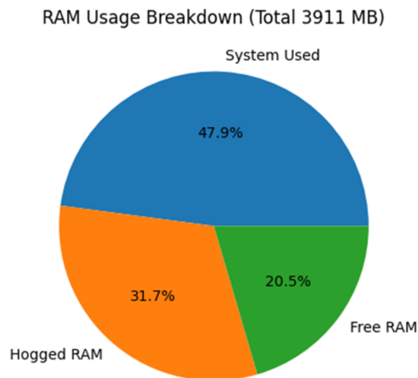

Fig. 4. Forced memory-pressure design. The benchmark data is larger than the remaining free memory, causing the kernel to trigger buffer reclamation.

TABLE II
BENCHMARK WORKLOAD DEFINITIONS.

Workload	Purpose
Sequential read	Models a linear scan. It tests whether one-time scan traffic pollutes the cache and removes useful blocks.
Random write	Exercises dirty metadata behavior, such as allocation bitmap and block-mapping updates. It tests whether frequently modified metadata is protected.
Zipfian DB simulation	Models skewed database-index access with Zipfian distribution ($\alpha = 1.2$). It tests whether LFU protects hot blocks under heavy access skew.

B. Forcing Memory Pressure

To evaluate replacement behavior, the experiment must force the kernel to reclaim cached buffers. The project disables swap using the command `swapoff -a`, mounts a temporary memory-backed filesystem, and uses it as a memory hog so that only about 800 MB of RAM remains free. Since the benchmark file is approximately 1 GB, the system cannot keep the entire workload resident and must actively reclaim buffers.

C. Workloads

The evaluation uses three main `fio` workloads: sequential read, random write, and Zipfian database-like access. Each workload is tested under cold or warm conditions where applicable.

TABLE III
LRU AND LFU HIT-RATE COMPARISON.

Workload	Stage	LRU (%)	LFU (%)	Improvement(%)
Sequential Read	Warm	83	84	+1
Sequential Read	Cold	13	16	+3
Random Write	Cold	53	69	+16
Random Write	Warm	97	96	-1
DB Simulation (Zipfian)	Cold	52	67	+15
DB Simulation (Zipfian)	Warm	52	66	+14

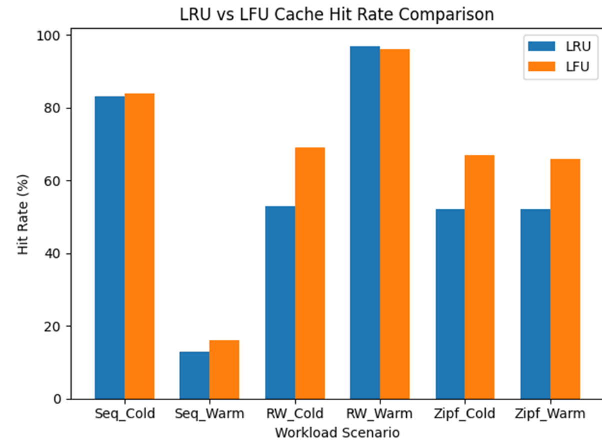

Fig. 5. Hit-rate comparison between LRU and LFU across sequential, random-write, and Zipfian workloads.

D. Measurement Procedure

For each workload, the benchmark reads `/proc/cache_stats` before and after execution. The difference between the two snapshots gives workload-specific hit and miss counts. The hit rate is computed as

$$\text{HitRate} = \frac{\Delta\text{Hits}}{\Delta\text{Hits} + \Delta\text{Misses}} \times 100\%. \quad (1)$$

The same workload scripts are executed once under the baseline LRU kernel and once under the LFU-modified kernel. This keeps workload definitions consistent and makes hit-rate differences attributable to the cache policy rather than to different benchmark inputs.

VI. RESULTS

A. Hit Rate Results

The main benchmark results are shown in Table III. The LFU policy improves sequential cold access slightly, improves cold random-write access substantially, and improves Zipfian database-like workloads strongly. However, it slightly underperforms LRU for warm random writes.

B. Interpretation of Sequential Read

Sequential reads show only a small improvement: +1% in warm mode and +3% in cold mode. This is expected because sequential scans often have weak long-term reuse. LFU cannot create reuse where the access pattern itself is mostly one-time. The bounded counter and decay logic may reduce some scan pollution, but the benefit is naturally limited.

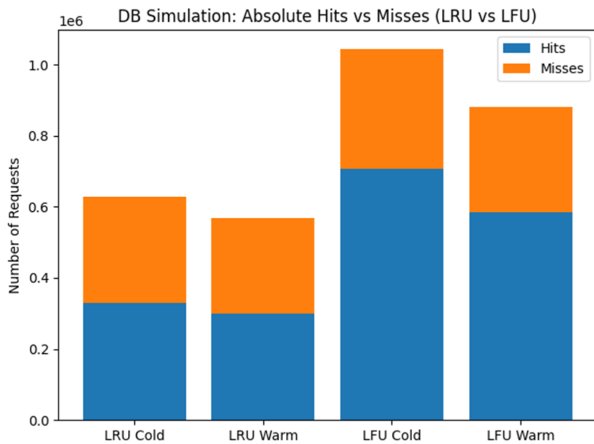

Fig. 6. Absolute hit and miss volume comparison. LFU should show substantially more hits than LRU in the cold Zipfian database simulation.

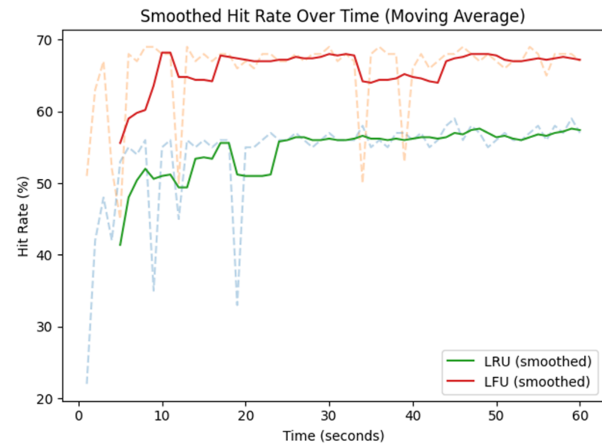

Fig. 7. Zipfian hit rate over time under memory pressure. LFU is expected to converge faster and maintain a more stable hit-rate plateau.

C. Interpretation of Random Write

Cold random write improves from 53% under LRU to 69% under LFU. This is one of the strongest results in the experiment. Random writes tend to repeatedly touch filesystem metadata, including allocation bitmaps and block-mapping structures. Under LRU, these metadata blocks may be displaced by recent but less important blocks. Under LFU, frequently modified metadata accumulates higher `access_count` values and is therefore more likely to survive memory pressure.

The warm random-write workload shows a slight regression from 97% to 96%. This does not contradict the LFU advantage under cold pressure. In warm conditions, most useful structures may already be resident, and LRU can perform nearly optimally. LFU's additional retention preference may then provide little benefit and can occasionally reduce adaptability.

D. Interpretation of Zipfian Database Simulation

The Zipfian workload is the clearest success case. In the cold stage, LFU improves hit rate from 52% to 67%. In the warm stage, LFU improves hit rate from 52% to 66%. This pattern is consistent with the theory of LFU: when a small fraction of blocks receives a large fraction of accesses, frequency becomes a strong predictor of future reuse.

LFU serves 706,392 hits during the cold database simulation, compared with 329,378 hits under LRU. This absolute-hit difference supports the hit-rate result: LFU is not merely changing the denominator but is preserving hot blocks more effectively under pressure.

E. Zipfian Hit Rate Over Time

The time-series result indicates that LRU is less stable under memory pressure, with early hit-rate drops and slower convergence. LFU rises quickly and maintains a more stable plateau. This behavior is consistent with bounded LFU with decay: hot blocks are retained long enough to support repeated accesses, while stale blocks gradually lose protection.

VII. Discussion

A. Why LFU Helps Skewed Workloads

LFU helps when past frequency is a reliable signal of future access. This condition holds in database-like Zipfian workloads, where a small number of blocks are accessed repeatedly. It also holds in random-write workloads when filesystem metadata blocks are updated many times. In these cases, the `access_count` field captures useful information that LRU ignores.

B. Why LFU Does Not Always Win

LFU is not universally better than LRU. In sequential workloads, frequency information is weak because most blocks are not reused. In warm workloads, the working set may already be resident, leaving little room for improvement. In dynamic workloads, old hot blocks may remain protected slightly longer than ideal, which can reduce adaptability. The bounded counter and linear decay reduce this problem, but do not eliminate it completely.

C. Systems Perspective

The main value of the project is not merely the numerical improvement. The stronger systems contribution is the full measurement-and-evaluation pipeline: identifying a kernel path, adding low-overhead instrumentation, exposing statistics through a simple user-space interface, inducing controlled memory pressure, and evaluating the policy under multiple workload distributions. This is a more reliable approach than relying only on application-level runtime measurements.

VIII. Limitations and Future Work

A. Limitations

The first limitation is scope. The project modifies the buffer-cache path and buffer reclamation behavior, but it is not a complete redesign of the Linux page cache or virtual-memory subsystem. Therefore, the conclusions should be limited to

workloads that exercise buffer-head lookup and block metadata behavior.

The second limitation is workload coverage. The evaluation includes sequential read, random write, and Zipfian database-like access. These are useful stress cases, but they do not cover all filesystem workloads. Real applications may mix data reads, metadata updates, directory traversal, journaling, and background writeback in more complex ways.

The third limitation is policy simplicity. Bounded LFU with linear decay is intentionally lightweight, but it does not include adaptive switching, workload classification, or multi-queue balancing. It may still retain stale hot entries longer than ideal.

B. Future Work

Future work can extend the project in several directions. First, the kernel instrumentation can distinguish fast `bh_lru` hits from successful slow-path lookups. Second, the replacement policy can use adaptive aging, such as periodic counter halving, to reduce stale-frequency effects. Third, the system can implement dynamic policy switching between LRU, LFU, and adaptive replacement strategies depending on observed workload skew. Fourth, the benchmark can be extended to ext4, XFS, and database workloads such as SQLite or RocksDB. Finally, debugfs or tracepoint support can provide more detailed hotspot visualization beyond aggregate hit/miss counts.

IX. Conclusion

This project implements kernel-level buffer-cache instrumentation and an LFU-based replacement enhancement. The system adds hit/miss counters to the buffer lookup path, exposes runtime metrics through `/proc/cache_stats`, extends `struct buffer_head` with an atomic frequency counter, applies bounded LFU updates, and uses linear decay to prevent permanently protected ghost blocks.

The experimental results show a clear pattern. LFU provides limited gains for sequential reads, strong gains for cold random writes, and strong gains for Zipfian database-like workloads. Its slight regression in warm random writes confirms that frequency-based replacement is not universally superior. Overall, the project supports the conclusion that buffer-cache metadata workloads with stable hot blocks benefit from frequency-aware retention, especially under memory pressure, while LRU remains competitive for simple temporal-locality and already-warm cases.

Division of Work

TABLE IV
Team member contributions.

Member	Contribution
Wang Peizhu	Extended `buffer_head` with atomic counters; implemented lockless per-CPU statistics; hooked `fs/buffer.c` paths for global hit/miss tracking; implemented bounded LFU and linear decay.
He Haodong	Exposed real-time metrics through `/proc/cache_stats`; designed memory-pressure tests using tmpfs memory hogging; automated `fio` workloads; verified kernel stability under sustained heavy load; visualized and analyzed LRU/LFU benchmark results.

References

[1] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2005.

[2] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2005.

[3] P. J. Denning, "The working set model for program behavior," *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, May 1968.

[4] Linux Kernel Developers, "Buffer heads," *The Linux Kernel Documentation*. [Online]. Available: https://docs.kernel.org/filesystems/buffer.html. Accessed: Apr. 2026.

[5] R. Love, *Linux Kernel Development*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2010.

[6] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[7] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, USA, 2003, pp. 115–130.

[8] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. ACM SIGMOD International Conference on Management of Data*, Washington, DC, USA, 1993, pp. 297–306.

[9] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[10] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Boston, MA, USA: Pearson, 2015.

[11] L. Torvalds and Linux Kernel Contributors, "fs/buffer.c," *Linux Kernel Source Tree*. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/buffer.c. Accessed: Apr. 2026.