Topic 10 Final Report

Kernel-Level Buffer Cache Instrumentation and Evaluation of an LFU-like Buffer Lookup Policy

Authors

Team 19
Langye Li (Student ID: 225040134)
Rui Huang (Student ID: 225040524)

April 19, 2026

Contents

List of Figures

List of Tables

Abstract

This project studies the Linux kernel buffer cache lookup path rather than generic page-cache hit rate. More specifically, we instrument `find_get_block_common()` in `fs/buffer.c` to count buffer lookup hits and misses, export the statistics via `/proc/cache_stats`, and implement an LFU-like frequency-sensitive retention policy in the per-CPU buffer lookup cache. We extend `struct buffer_head` with an `access_count` field, update it along lookup-hit paths, and use it to retain higher-frequency entries in `bh_lru`. To ensure that the workload matches the actual instrumentation target, we redesign all scenarios around one shared pool of many small files and only vary cache state, access order, hotspot distribution, and traversal intensity. The results show that LFU is slightly better than LRU in `seq`, `rand`, and `zipf`, but not universally better; it underperforms in the heavier `meta` scenario, suggesting that frequency-sensitive retention may reduce adaptability under broader metadata working sets. Overall, the observed pattern is consistent with the true scope of the implementation: a lookup-cache optimization rather than a global page-cache replacement policy.

Keywords: Linux kernel; buffer cache; buffer_head; `find_get_block_common()`; LFU-like policy; metadata-heavy workload; `/proc/cache_stats`

1 Introduction

1.1 Background

Linux file access does not interact with a single cache. Instead, it traverses a layered cache hierarchy involving:

1. the **VFS layer**, including dentry cache and inode cache;
2. the **filesystem layer**, such as ext4;
3. the **buffer cache / buffer_head lookup layer** in `fs/buffer.c`;
4. the **block device and disk layer**.

This layered structure is exactly why kernel-cache experiments are easy to misinterpret. A common simplification is to talk about "cache hits" as if they always mean page-cache hits. In reality, different layers answer different questions:

- the dentry cache answers whether pathname resolution can be satisfied from memory;
- the inode cache answers whether inode metadata structures are already resident;
- the page cache answers whether file data pages are cached;
- the buffer cache / `buffer_head` path answers whether the block-related object already exists at the lower metadata/block layer.

Because of that, a meaningful Topic 10 project must first clarify **which layer is being observed** before it can interpret any result correctly.

1.2 Problem statement

This project addresses two concrete questions:

1. Can we instrument the unified buffer lookup path in Linux and expose reliable hit/miss statistics to user space?
2. Can we implement an LFU-like frequency-sensitive retention policy in the per-CPU buffer lookup cache and evaluate how it differs from the original LRU-like behavior?

To answer these questions, we completed:

- kernel-level hit/miss instrumentation;
- user-space export through `/proc/cache_stats`;
- `access_count`-based LFU-like retention in `bh_lru`;
- a unified small-file workload pipeline;
- automated comparison and plotting of LRU vs LFU results.

1.3 Structure of this report

The rest of the report covers:

- conceptual background and scope boundaries;
- implementation details;
- workload design and evaluation method;
- actual experimental results;
- deeper interpretation of the `cold` and `warm` phenomena;
- limitations and possible future work.

2 Background and Key Concepts

2.1 `buffer_head` and `fs/buffer.c`

Even though modern Linux heavily relies on the unified page cache, `buffer_head` is still relevant in block-level and metadata-related paths. As a result, `fs/buffer.c` remains a useful observation point for experiments involving filesystem metadata blocks and lower-level block lookup behavior.

2.2 Why `find_get_block_common()` is the right instrumentation point

If instrumentation is placed only in wrappers such as `__find_get_block()`, it may capture only a partial view of the lookup behavior. By instrumenting `find_get_block_common()`, we can observe the full decision flow:

- lookup in the per-CPU `bh_lru` cache;
- fallback to the slow path;
- final success or failure of the lookup.

This makes it the most appropriate single place to define hit and miss semantics consistently.

2.3 This is not a page-cache hit-rate experiment

This is the most important conceptual boundary of the whole project.

In this project:

- **hit** means that a `buffer_head` lookup succeeds;
- **miss** means that the lookup returns `NULL`.

This is not equivalent to asking whether a user-space read operation was served from the page cache. In other words, we are measuring:

- **buffer lookup hit/miss behavior**

rather than:

- **file-data page-cache hit/miss behavior**.

This distinction directly affects workload design. A large-file sequential read may be very meaningful for page-cache analysis, but it is not necessarily the best workload for a lower-level metadata-oriented buffer lookup experiment.

2.4 Multi-level caching in Linux

A useful mental model is to see Linux caching as a chain of filters. Consider a user-space call such as `stat()`:

1. **Dentry cache** may already know the pathname-to-inode mapping.
2. **Inode cache** may already contain the inode metadata structure.
3. Only if more lower-level information is needed does the request proceed into the filesystem implementation.
4. At that point, the filesystem may trigger a lower-level block lookup through `find_get_block_common()`.
5. Only if that lookup also fails is disk I/O eventually required.

This explains why user-visible work can continue while the buffer lookup statistics remain very small: some requests are simply satisfied by upper layers before they ever reach `fs/buffer.c`.

2.5 What per-CPU means, and why it matters

`per-CPU` does not mean "many copies of one global cache." It means that **each CPU core maintains its own small local instance of a data structure**, instead of all CPUs contending for one shared global structure.

If a lookup cache is globally shared, then:

- CPU0, CPU1, CPU2, and others all access the same structure;
- contention and sharing overhead become more likely;
- locality is weaker because multiple CPUs keep touching the same memory region.

If it is per-CPU instead, then:

- each CPU mainly consults its own local cache;
- local access is cheaper;
- it is better suited for keeping objects that this CPU just touched and may touch again soon;
- but each cache is usually much smaller, so the optimization scope is also more limited.

This is exactly the right background for understanding `bh_lru` in this project. We are not optimizing one large system-wide cache. We are optimizing **a small `bh_lru` lookup cache attached to each CPU**.

2.6 Direct connection between per-CPU `bh_lru` and this project

In this project, `find_get_block_common()` first checks the current CPU's `bh_lru`. That means:

- if the current CPU has recently touched certain `buffer_head` objects, they are more likely to be found in its own local cache;
- only when that local lookup misses does the code continue to slower paths;
- therefore, our LFU-like policy is really deciding **which objects deserve to remain in the current CPU's small local lookup cache**.

This is fundamentally different from a global page-cache replacement policy. A global replacement policy asks which pages should remain in memory across the whole system. Our project asks a more local question:

- inside the current CPU's small lookup cache,
- which `buffer_head` objects should be retained,
- so that later local lookups are more likely to hit.

This has two direct consequences for the interpretation of results:

1. **The gains should usually be local and modest.** The policy optimizes a small local lookup cache, not the entire caching system.
2. **Capacity limits matter more.** If stale hot entries stay too long in a tiny cache, they can crowd out currently relevant objects.

2.7 An intuitive example

Suppose one CPU's `bh_lru` has only a few slots and currently holds:

- A: access count 20
- B: access count 15
- C: access count 3
- D: access count 1

Now a new object E arrives with access count 10.

- Under a simple LRU-like idea, the decision would lean more on recency;
- under our LFU-like idea, the decision leans more on frequency;
- as a result, E is more likely to replace a lower-frequency entry like D, while higher-frequency entries such as A and B are more likely to stay in the current CPU's local cache.

This example shows that the goal of the policy is not global optimality. The goal is to keep **the current CPU's local hot objects** close at hand so that later lookups on that CPU are faster.

2.8 Spatial locality and block clustering

The second key concept is **spatial locality**. Suppose the filesystem block size is 4KB and a lower-level inode structure occupies around 256 bytes. Then a single block may contain multiple inode structures packed together. As a result:

- the first lookup for one file's metadata may miss and load the whole block;

- subsequent lookups for nearby files may hit immediately, because their metadata lives in the same loaded block.

This is why a "cold" experiment does not imply that the entire workload remains cold throughout its execution. It only means the cache is cold at the starting point. Once traversal begins, locality quickly creates hits.

3 Implementation

3.1 Hit/miss counters

We added the following global counters in `fs/buffer.c`:

- `atomic_long_t total_hits`
- `atomic_long_t total_misses`

These counters provide concurrency-safe accounting of lookup outcomes.

3.2 Counting logic in the unified entry

The hit/miss accounting is placed in `find_get_block_common()`. This means:

- a non-NULL `bh` is counted as a hit;
- a `NULL` return is counted as a miss.

This captures:

- per-CPU `bh_lru` hits;
- successful slow-path lookups;
- lookup failures.

3.3 Export through `/proc/cache_stats`

We implemented a `seq_file`-based proc interface: `/proc/cache_stats`. It reports: Buffer Cache Hits, Buffer Cache Misses, Total Lookups, and Hit Rate. This makes the experiment easy to automate with before/after snapshots.

3.4 `access_count` in `buffer_head`

We extended `struct buffer_head` with: `atomic_t access_count`. This field is initialized when the object is allocated and records the observed frequency of lookup success.

3.5 Frequency updates

The frequency signal is updated on two successful paths:

1. in `touch_buffer()`;
2. after a successful slow-path lookup in `find_get_block_common()`.

This design reduces bias and ensures that both fast-path and slow-path successful lookups contribute to the frequency estimate.

3.6 LFU-like retention in `bh_lru`

The custom policy is implemented in the per-CPU `bh_lru` lookup cache. Conceptually:

- if there is free space, insert directly;
- if the cache is full, compare the incoming object with lower-frequency resident entries;
- replace an existing entry only when the incoming object has a stronger frequency signal.

This is intentionally lightweight and local. It is **not** a global VM-level replacement policy.

The meaning of per-CPU also defines the practical boundary of this policy:

- it optimizes the current CPU's own small lookup cache;
- it does not optimize one system-wide shared cache;
- and it does not change the global Linux page-cache or reclaim policy.

Because of that, its most natural success cases are workloads where:

- the same CPU repeatedly touches related metadata objects over a short period;
- the same `buffer_head` objects are reused in local lookup patterns;
- frequency information helps retain genuine local hotspots.

Its limitations follow from the same design choice:

- each cache is small;
- the optimization is local rather than global;
- if stale hot entries stay too long, adaptability can drop quickly.

3.7 Why we did not modify global reclaim

Modifying `mm/vmscan.c` or implementing a full global replacement policy would make the project more invasive and much harder to interpret. A course project benefits from:

- a clearly defined scope;
- lower implementation risk;
- stronger interpretability of results.

From that perspective, implementing an LFU-like policy in `bh_lru` is a reasonable and disciplined design choice.

4 Experimental Design

4.1 Why we unified everything onto small files

One of the most important design corrections in the project was to unify all scenarios around the **same pool of many small files**. If we had kept a mixture of large-file `cat`, `fio`, and metadata traversal, then:

- some results would mainly reflect page-cache behavior;
- others would mainly reflect buffer lookup behavior;
- and the final comparison would be conceptually mixed.

The final design instead uses:

- one shared small-file pool;
- one shared manifest;
- identical object sets across scenarios.

Only the following dimensions change:

- cache state;
- access order;
- hotspot distribution;
- workload intensity.

4.2 Scenario definitions

The six final scenarios are:

1. **cold**: sequential metadata traversal from a cold state;
2. **warm**: repeat the exact same traversal after preheating;
3. **seq**: stable-order traversal;
4. **rand**: randomized traversal with a fixed seed;
5. **zipf**: hotspot-skewed metadata access;
6. **meta**: heavier metadata-intensive traversal.

4.3 Fairness controls

To make the LRU vs LFU comparison meaningful, we keep fixed: directory structure, file count, manifest, random seeds, zipf parameters, scenario definitions, and summary file format. This ensures that any difference is more likely to reflect a policy effect rather than workload drift.

4.4 Procedure

We run the same script on two kernels:

```
sudo bash scripts/topic10/run_experiments.sh --with-meta
```

once under the LRU kernel and once under the LFU kernel. Each run produces `_before.txt`, `_after.txt`, `.log`, and `summary.tsv`. The two runs are then compared with `plot_comparison.py`.

5 Experimental Results

Table 1: LRU Policy Results

Scenario	ΔHits	ΔMisses	ΔTotal	ΔHitRate
cold	4778	505	5283	90.44%
warm	23	2	25	92.00%
seq	4882	606	5488	88.96%
rand	4883	606	5489	88.96%
zipf	2852	613	3465	82.31%
meta	5051	622	5673	89.04%

Table 2: LFU Policy Results

Scenario	ΔHits	ΔMisses	ΔTotal	ΔHitRate
cold	4593	274	4867	94.37%
warm	24	6	30	80.00%
seq	4797	533	5330	90.00%
rand	4811	536	5347	89.98%
zipf	2743	535	3278	83.68%
meta	6624	1173	7797	84.96%

Table 3: LFU minus LRU hit-rate difference

Scenario	LFU - LRU
cold	+3.93%
warm	-12.00%
seq	+1.04%
rand	+1.02%
zipf	+1.37%
meta	-4.08%

To better visualize these differences, we plotted the comparison of hit rates, hits, and misses across all scenarios.

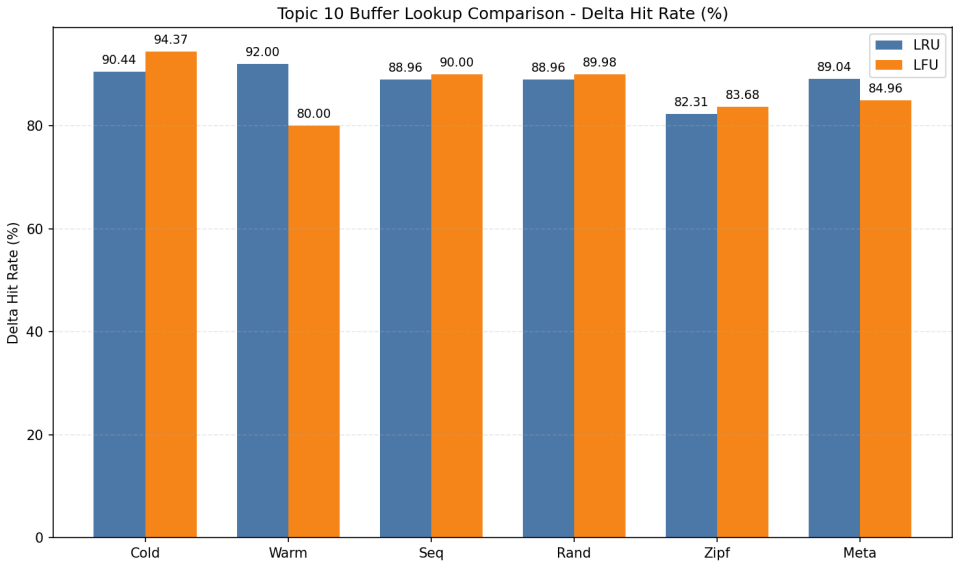

Figure 1: Comparison of Hit Rates between LRU and LFU Policies

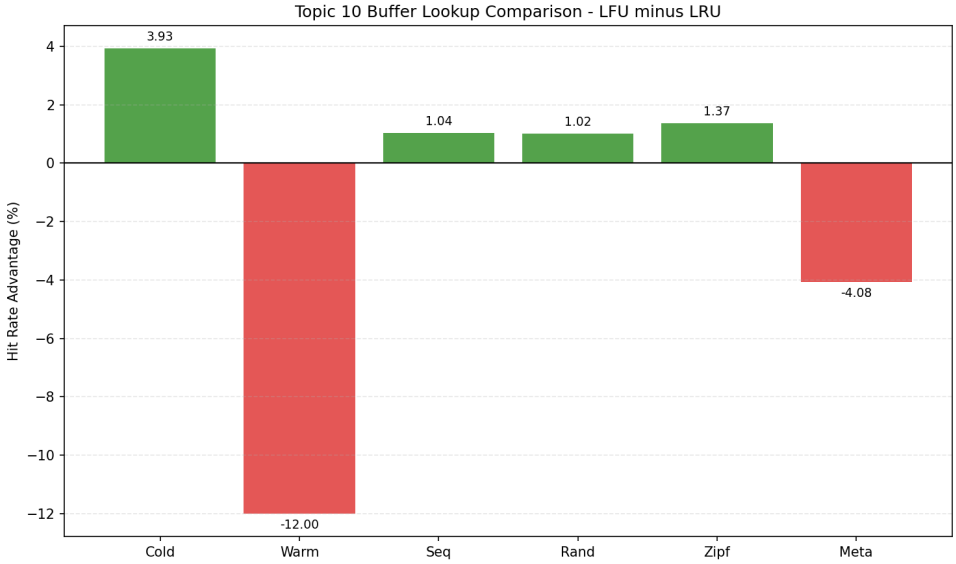

Figure 2: Hit Rate Advantage of LFU over LRU

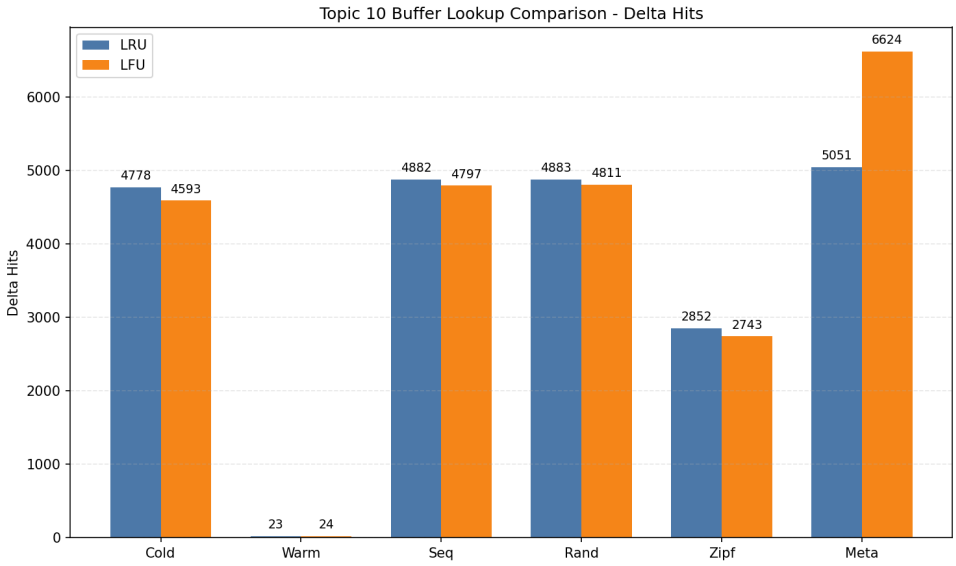

Figure 3: Comparison of Total Cache Hits (ΔHits)

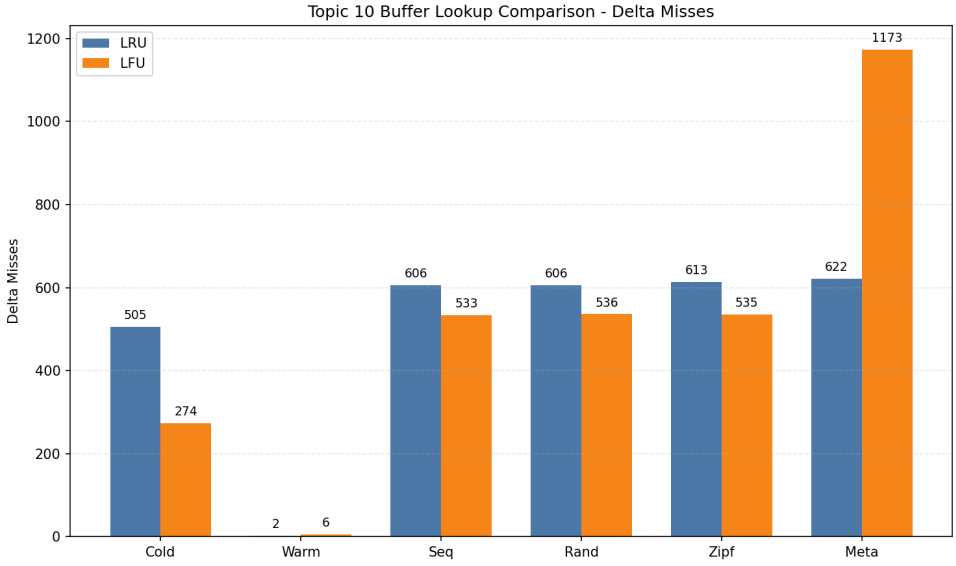

Figure 4: Comparison of Total Cache Misses (ΔMisses)

5.1 Immediate observations

At a glance, three patterns stand out:

1. LFU is slightly better in `seq`, `rand`, and `zipf`.
2. The `warm` scenario has an extremely small sample volume.
3. LFU is worse in the heavier `meta` traversal.

These three observations define the main interpretation of the entire experiment.

6 In-Depth Analysis

6.1 Overall assessment

The results appear **reasonable and internally consistent**. They do not look like broken instrumentation or a failed script run. Instead, they suggest that:

- the LFU-like policy does affect lookup-cache behavior;
- the effect is real but small;
- the gain depends on workload characteristics;
- and regressions are possible.

That is exactly what we would expect from a local optimization in a small per-CPU lookup cache.

6.2 Why the gains in `seq`, `rand`, and `zipf` are small

A naive expectation might be that LFU should dominate hotspot workloads by a large margin. But that would only be reasonable if we had implemented a global replacement policy. We did not. We modified:

- the per-CPU `bh_lru` lookup cache in `fs/buffer.c`

So the most realistic expectation is:

- measurable differences, but not dramatic ones;
- slight LFU gains in repeated or hotspot-oriented cases;
- no overwhelming win.

The data fits that expectation very well: `seq`: +1.04%, `rand`: +1.02%, `zipf`: +1.37%.

6.3 Why are both hits and misses so low in `warm`?

This is one of the deepest and most important questions in the whole experiment.

A natural intuition is: if the workload is warm, shouldn't we simply see many hits? Why is the **total number of lookups itself** so small?

The answer is: **because many requests are likely intercepted by higher-level caches before they ever reach the buffer lookup layer.**

Once the `cold` traversal has run, a large amount of metadata for the shared small-file pool is already resident in upper layers such as the **dentry cache** and the **inode cache**. When the `warm` scenario runs immediately afterwards, user-space operations such as `stat` may be satisfied at the VFS level, meaning there is no need to descend into lower-level block lookup for most accesses.

As a result:

- the user-space workload still runs;
- but very few operations reach `find_get_block_common()` again;
- the measured ΔTotal becomes tiny: 25 for LRU, 30 for LFU.

So the correct interpretation of `warm` is not that LFU is much worse, but rather that after preheating, upper-level VFS caches satisfy most requests.

6.4 Why is the `cold` hit rate still above 90%?

A common misconception is that after `drop_caches`, the hit rate should be very low. But `drop_caches` only guarantees that the workload starts from a cold point. It does **not** mean the system stays cold throughout the entire traversal. Once the workload begins, locality immediately starts to rebuild cache state.

Two mechanisms explain the high hit rate:

6.4.1 Spatial locality

Files stored close to each other often have metadata structures packed into nearby or identical blocks. The first access to one object may miss, but subsequent accesses to nearby objects may hit.

6.4.2 Reuse within the traversal itself

Even in a cold-start traversal, the workload repeatedly revisits related objects: parent directories, nearby inode blocks, neighboring metadata blocks. Therefore, the experiment starts cold, but the workload quickly rebuilds useful cache state.

6.5 Why does LFU underperform in `meta`?

The heavier `meta` scenario shows LRU at 89.04% and LFU at 84.96% (-4.08% difference).

This is not necessarily a flaw in the code. It is fully consistent with a classic LFU trade-off: LFU preserves entries that were hot in the past. Under a broader and more dynamic working set, those older hot entries may occupy a small cache longer than ideal, preventing newer metadata objects from being retained. This reduces adaptability, particularly in a small per-CPU cache.

6.6 Why are `seq` and `rand` so close?

The fact that `seq` and `rand` are numerically close is a good sign. In the final workload design, the object set and operation types are the same; only the access order changes. This suggests the experiment is performing a clean comparison of distribution/order effects.

6.7 Final evaluation of the LFU-like policy

Taken together, the most defensible evaluation is:

- the LFU-like policy is **not universally superior**;
- it helps repeated and hotspot-oriented metadata lookup patterns a little;
- it may reduce adaptability under broader metadata traversals;
- its effect size is naturally limited because it only optimizes a small per-CPU lookup cache.

7 Discussion

7.1 The main value of this project

The most important contribution is **not** proving a slogan like "LFU beats LRU." The real value is that the project:

1. identifies a meaningful kernel instrumentation point;

2. exports the measurement cleanly to user space;
3. aligns workload design with the true measurement target;
4. produces results that are interpretable in terms of the actual Linux cache hierarchy.

7.2 Why this is closer to systems research than an algorithm contest

In an algorithm contest, one might care mainly about who "wins" by a larger margin. In systems work, the more important questions are: did we instrument the correct layer? did the workload actually exercise the layer we intended to study? are the conclusions consistent with the architecture of the system? By that standard, the result is mature.

8 Limitations and Future Work

8.1 Limitations

1. **This is not a global page-cache or VM replacement study** – the policy only affects a per-CPU lookup cache.
2. **The results are most meaningful for metadata-heavy workloads** – they cannot be generalized to every file-I/O pattern.
3. **The `warm` scenario has too little signal** – it is better seen as a sanity check than as strong evidence.
4. **The LFU-like policy is intentionally lightweight** – it has no aging, decay, or global balancing mechanism.

8.2 Future work

Future extensions could include: distinguishing `bh_lru` hits from slow-path hits in more detail; adding aging or decay to reduce the impact of stale hot entries; improving debugfs snapshots and hotspot visualization; comparing behavior across multiple filesystems rather than only one setup.

9 Conclusion

This project successfully completes the two core requirements of Topic 10:

- hit/miss instrumentation for the buffer lookup path, exposed through `/proc/cache_stats`;
- an LFU-like retention policy for the per-CPU buffer lookup cache.

The results show that LFU has small benefits in `seq`, `rand`, and `zipf`; the `warm` scenario has too little lookup volume because upper-level caches likely intercept most requests; the high hit rate in `cold` comes from spatial locality and reuse within the traversal rather than from a failed cold start; LFU performs worse in `meta`, which indicates reduced adaptability under broader metadata working sets; the overall result pattern matches the true scope of the implementation: a lookup-cache optimization, not a global page-cache replacement policy.

Therefore, the project succeeds both as an implementation exercise and as a systems experiment with a defensible interpretation grounded in Linux's multi-level cache hierarchy.