cache_ext: Customizing the Page Cache with eBPF

Tal Zussman* [1] Ioannis Zarkadas* [1] Jeremy Carin [1] Andrew Cheng [1] Hubertus Franke [2] Jonas Pfefferle [2] Asaf Cidon [1]

[1]Columbia University [2]IBM Research

Problem Statement

- The OS page cache is widely used by a large class of applications
- It significantly affects application performance by reducing excessive access to storage
- However, its LRU-like eviction policy is inflexible and performs poorly with specific workloads
- Result: performance left on the table!

Background: Page Cache

- Linux's default eviction policy is an LRU approximation algorithm with two FIFO lists: the *active* and *inactive* lists
  - Pages are added to the tail of the inactive list
  - Repeatedly accessed pages will eventually be promoted to the active list
  - Pages are evicted from the head of the inactive list, and demoted from the active list to the inactive list
- Each cgroup maintains its own pair of active and inactive lists
- The page cache manages pages as *folios*

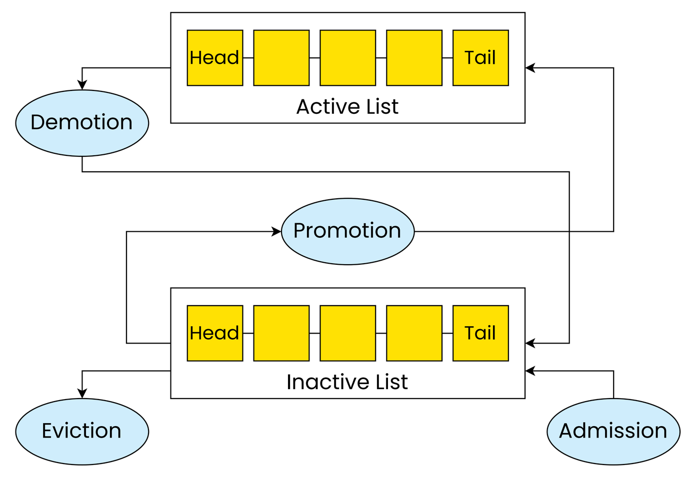


Figure 1. Default Linux page cache policy.

Existing Approaches

Can we modify the page cache policy?

- Very difficult to modify: hardcoded and requires extensive kernel knowledge
  - Took Google years to upstream its proposed Multi-Generational LRU (MGLRU) policy
- Any changes would modify the policy globally, affecting all applications

Can existing customization interfaces solve this?

- `fadvise()` and other interfaces provide hints which may be ignored
- Depending on the workload, may not have a performance impact
- Still subject to the kernel's inflexible LRU-like policy

What about userspace-based caching?

- Userspace caches require significant effort to implement, are typically not dynamically sized, and are tough to share across processes
  - In contrast to the page cache, they can lead to duplicated memory and lower resource utilization
- Applications with userspace caches may still use the page cache as a second-tier cache

Our Approach: eBPF-Based Policies

We want to enable applications to run **custom page cache policies** within Linux.

eBPF

- eBPF allows userspace functions to run within the Linux kernel in a safe and controlled manner by verifying them in advance
- eBPF has been used for observability, security, networking, scheduling, and more in Linux

Challenges

⚡ Scalability
- Modern SSDs support millions of IOPS, so the page cache must handle millions of events per second
- Any policies and changes to the page cache must be low overhead

🧩 Flexibility
- There exists a wide variety of caching algorithms, many of which require custom data structures
- An interface for custom policies must be flexible enough to accommodate the diversity of caching algorithms

🚧 Isolation
- One application's policy should not interfere with those of other applications
- However, pages should still be able to be shared between applications

🛡 Security
- Custom policies must not lead to unsafe memory references or kernel crashes

cache_ext: Design and Implementation

Interface

- cache_ext allows users to run custom page cache *policy functions*, which are implemented as eBPF programs
- Page cache events trigger policy functions
  - Policy initialization
  - Folio admission
  - Folio access
  - Folio removal
  - Request for eviction
- Policy functions operate on variable-sized *eviction lists*
  - Many complex eviction policies can be implemented exactly or approximately using lists
  - Eviction lists store pointers to folios. cache_ext operates on lists using eBPF *kfuncs*.
- On eviction requests, a policy proposes a batch of folios for the kernel to evict
  - The policy chooses folios by iterating over eviction lists and selecting candidates for eviction based on policy metadata (e.g., recency, frequency, etc.)

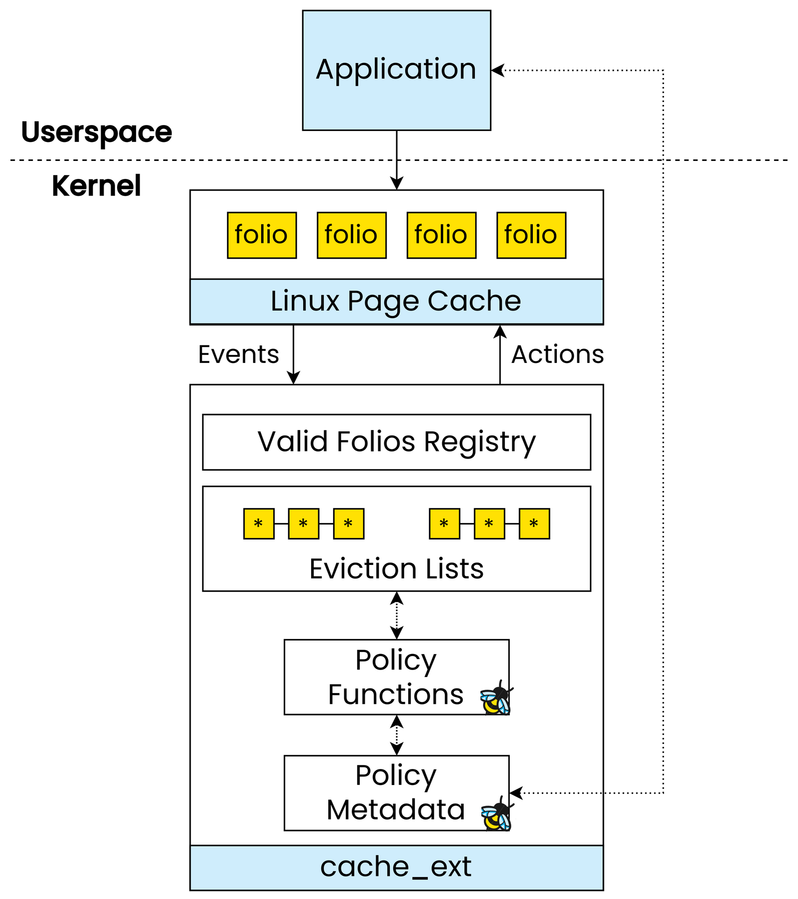


Figure 2. Overview of cache_ext.

Eviction list API
`u64 list_create(struct mem_cgroup *memcg)`
`int list_add(u64 list, struct folio *f, bool tail)`
`int list_move(u64 list, struct folio *f, bool tail)`
`int list_del(struct folio *f)`
`int list_iterate(struct mem_cgroup *memcg, u64 list, s64(*iter_fn)(int id, struct folio *f), struct iter_opts *opts, struct eviction_ctx *ctx)`

Table 1. cache_ext eviction list API.

Isolation

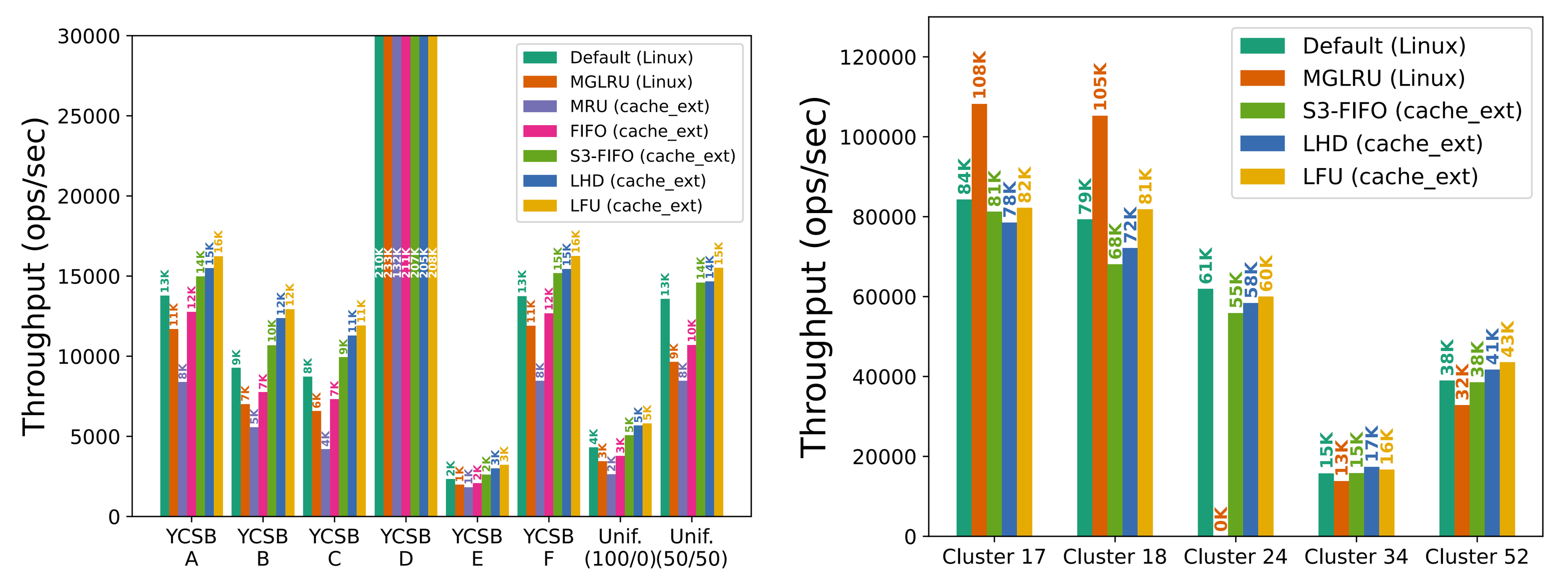
- We implement policies **per-cgroup**. Within a cgroup, processes share an eviction policy. Different cgroups can use their own policies.
  - Matches modern deployment practices: isolate applications in their own memory cgroup
  - Preserves memory sharing: processes from cgroup A can still access pages managed by cgroup B

Security

- **Memory safety**: eBPF policy functions must return valid folio pointers to the kernel
  - We implement a *valid folios registry* in-kernel. When a folio is inserted, it is added to the registry, and removed on eviction. The kernel uses the registry to verify that eviction candidates are valid folios.
- **Eviction fallback**: We protect against adversarial policy behavior with in-kernel fallbacks
  - For example, if a policy does not provide sufficient eviction candidates, the kernel uses the default policy

Real-World Workloads

We run several generic cache_ext policies on YCSB and Twitter trace workloads. cache_ext's LFU policy achieved up to 37% higher throughput on YCSB, while no single policy dominated the Twitter traces. **In general, there is no one-size-fits-all policy that performs best for all workloads. cache_ext enables the experimentation necessary to maximize performance.**

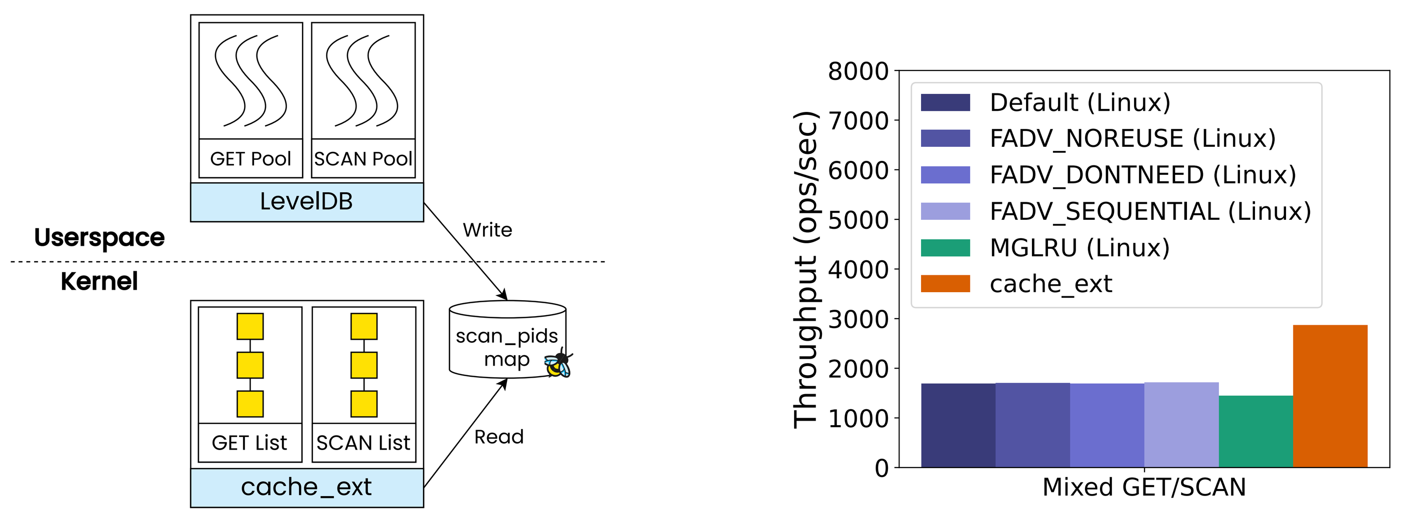


(a) YCSB workloads. (b) Twitter trace workloads.

Figure 3. Comparison of Linux and cache_ext policies on YCSB and Twitter cache trace workloads.

Application-Informed Policies

- Example: a database serving both small GET queries (e.g., transactions) and larger, slower SCAN-like queries (e.g., analytics).
  - SCAN data pollutes the page cache. Prioritize GETs using two eviction lists: GET data and SCAN data.
  - Maintain an eBPF map with SCAN thread PIDs. On folio insertion, choose list based on PID of current task.
  - Evict folios primarily from the SCAN list.



(a) Mixed SCAN/GET workload policy implementation. (b) Mixed SCAN/GET: GET throughput.

Figure 4. Mixed SCAN/GET policy.

- Application-aware policies can significantly improve performance
  - 70% higher GET throughput and 58% lower tail latency than baseline
- Existing Linux page cache customization interfaces (`fadvise()`) are insufficient

Multi-Tenancy

- Example: Two cgroups, one running a LevelDB YCSB C workload, and the other running a file search workload with ripgrep.
  - Configs: both default, both LFU, both MRU, and "tailored": YCSB with LFU and file search with MRU.
- Tailored is best: 50% and 79% increase for YCSB and file search, respectively
- **cache_ext with per-cgroup policies enables fine-grained control and better performance**

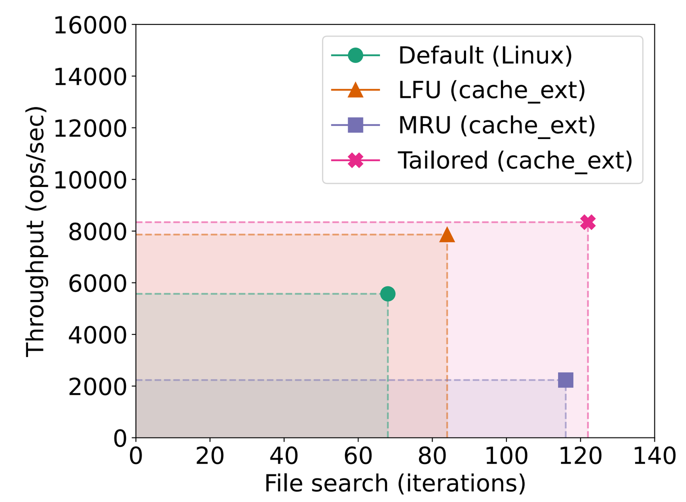


Figure 5. Multi-tenancy results: a file search workload (number of searches executed) and a YCSB C workload (throughput) running concurrently in two cgroups.

Implementation Complexity

Policy	eBPF LoC	Userspace LoC
Admission filter	35	262
FIFO	56	131
MRU	101	101
LFU	215	110
S3-FIFO	287	157
GET-SCAN	324	112
LHD	367	165
MGLRU	689	105

Table 2. Lines of code in cache_ext policies.

- Overall implementation complexity for cache_ext policies is modest, even for sophisticated policies.
- cache_ext and policies are open-source. We expect most developers will use and experiment with pre-existing policies.
- Can implement new policies for advanced use cases, policy innovations.

Conclusion

cache_ext enables safe and performant customization of Linux page cache policies, allowing applications to choose a policy matching their needs. We believe cache_ext opens the door to exploring new dynamic page cache policies and experimenting with policy innovations in a realistic setting.

Full paper and code at `github.com/cache-ext/cache_ext`