Cache is King: Smart Page Eviction with eBPF

Tal Zussman*,1, Ioannis Zarkadas*,1, Jeremy Carin¹, Andrew Cheng¹, Hubertus Franke², Jonas Pfefferle², and Asaf Cidon¹

¹Columbia University, ²IBM, *denotes equal contribution

Abstract

The page cache is a central part of an OS. It reduces repeated accesses to storage by deciding which pages to retain in memory. As a result, the page cache has a significant impact on the performance of many applications. However, its one-size-fits-all eviction policy performs poorly in many workloads. While the systems community has experimented with a plethora of new and adaptive eviction policies in non-OS settings (e.g., key-value stores, CDNs), it is very difficult to implement such policies in the page cache, due to the complexity of modifying kernel code. To address these shortcomings, we design a novel eBPF-based framework for the Linux page cache, called cachebpf, that allows developers to customize the page cache without modifying the kernel. cachebpf enables applications to customize the page cache policy for their specific needs, while also ensuring that different applications' policies do not interfere with each other and preserving the page cache's ability to share memory across different processes. We demonstrate the flexibility of cachebpf's interface by using it to implement several eviction policies. Our evaluation shows that it is indeed beneficial for applications to customize the page cache to match their workloads' unique properties, and that they can achieve up to 70% higher throughput and 58% lower tail latency.

1 Introduction

In his seminal 1981 paper on OS support for database management, Michael Stonebraker described how existing OS buffer cache mechanisms were ill-suited for the needs of databases at the time [66]. He observed that the buffer cache's one-size-fits-all eviction policy, approximate least-recently used (LRU), cannot possibly address the heterogeneity of database workloads. Nevertheless, in the intervening decades, despite wide-ranging efforts to rethink the UNIX/Linux OS page cache [17, 31, 73], design customizable file systems [10, 11, 41, 52], and build clean-slate extensible kernels [8, 60, 62], applications by and large still contend with Linux's opaque and inflexible OS page cache policy.

At the same time, the diversity of applications and work-loads running on Linux has only increased, from enterprise file systems and large-scale distributed datacenter ML training, to multimedia rich applications running on an Android phone. All of these applications must use Linux's decades-old approximate LRU policy, despite the fact it is widely known

to be inadequate for many workloads and scenarios (e.g., large scans [40, 57, 59], multi-core applications [73]). For example, an application that searches through files in a codebase (a scan-based workload) would benefit from using a most-recently used (MRU) policy, while a key-value store running a fixed, skewed-distribution workload would improve under least-frequently used (LFU). However, both of these workloads currently run with the default Linux eviction policy.

The reasons applications are "stuck" with the same old eviction policy are twofold. First, modifying the Linux page cache is a hard task, requiring extensive kernel knowledge and attention to detail. Second, upstreaming changes to the page cache is difficult, because the changes have to work well for the wide range of applications that run on Linux, forcing a lowest common denominator. For instance, it took Google years to upstream its proposed Multi-Generational LRU (MGLRU) algorithm into the Linux kernel, and even after several years, it is still disabled by default in upstream [17, 19].

In this paper, we attempt to finally answer Stonebraker's plea for better OS support for buffer management, within Linux. To this end, we design a novel framework, cachebpf, which provides visibility and control of the OS page cache, without requiring the application to make kernel changes. cachebpf takes advantage of eBPF [26], a Linux (and Windows) supported runtime that allows safely running application code inside the kernel. We take a cue from sched_ext, an eBPF-based framework that allows applications to customize the OS scheduler [38,42] and has been adopted by Linux [15].

cachebpf's design is motivated by four main insights. First, modern storage devices are very fast and support millions of IOPS, so custom page cache policies must run with low overhead. Therefore, we design cachebpf so that its eBPF-based policies run in the kernel, avoiding expensive and frequent synchronization between the kernel and userspace. Second, caching algorithms are very diverse and may use complex data structures. To address this challenge, cachebpf exposes a simple yet flexible interface that allows applications to define one or more variable-sized lists of pages, and a set of policy functions (e.g., admission, eviction) that operate on these lists, which can be used to express a wide range of eviction policies. Third, in order for cachebpf to be useful in multi-tenant scenarios, it should allow each application to use its own policy without interfering with others. We identify cgroups as a natural isolation boundary. Thus, cachebpf allows each cgroup to implement its own eviction policy without interfering with other cgroups. Finally, custom policies determine which pages to evict and return page references to the kernel. However, these references may be invalid, which could lead to kernel crashes or security breaches. To solve this, cachebpf maintains a registry of valid page references, which is used to validate the page references returned by the user-defined policies.

We demonstrate cachebpf's utility and flexibility by implementing four custom eviction policies, which include both "classic" and recently-designed policies: most-recently used (MRU), least-frequently used (LFU), S3-FIFO [70], and least hit density (LHD) [5]. We also show how cachebpf enables application-informed policies with only minor policy changes, allowing applications to design policies that take into account application-level insights. For example, a database can implement a custom policy that prioritizes point queries over scans, yielding higher throughput for point queries. We compare these cachebpf policies with the kernel's default eviction policy and its different options (e.g., fadvise()), and with the recently-upstreamed MGLRU algorithm. We show that with cachebpf, developers can significantly improve their applications' performance far beyond the existing algorithms provided by the Linux page cache. In general, we find that there is no one-size-fits-all policy that improves all workloads – customization is necessary in order to maximize performance. In particular, applications can use cachebpf to improve throughput by up to 38% using "generic" policies, and achieve up to 70% higher throughput and 58% lower P99 latency with application-informed policies.

We will open source cachebpf and all our implemented policies upon publication. A key benefit of cachebpf is that any publicly available eviction policy can be used by other developers, lowering the barrier to using the system and experimenting with eviction policies on different workloads.

Our primary contributions are:

- cachebpf, a flexible, scalable, and safe eBPF-based framework for running custom eviction policies in the Linux kernel page cache.
- A suite of custom eviction policies and userspace libraries allowing developers to easily create new policies.
- An evaluation of cachebpf across various applications, demonstrating how they benefit from customized policies.

2 Background and Motivation

By default, the page cache buffers write and read operations to and from storage devices. In Linux, the page cache tracks pages and stores them in lists (see §2.1), on which it approximates the LRU algorithm. While this scheme works reasonably well for some workloads, it is inadequate for many others. The classic example is scan-heavy workloads, which perform poorly with LRU or its approximations [5, 50, 66]. While Linux provides some interfaces (e.g., fadvise() or sysctl)

through which the page cache behavior can be tweaked on a global or per-application basis, these interfaces are opaque and do not perform as intended, as we describe in §2.1 and evaluate in §5.1.5.

Therefore, to avoid compromising performance, some applications implement their own userspace-based caches [4, 29, 54, 56]. However, userspace-based caches are not a panacea. First, they require significant developer effort to implement. Second, they typically require the application to specify in advance how much memory will be allocated for the cache. However, the amount of memory available to an application may change over time (e.g., when multiple applications run on the same physical server). Third, application-specific caches are very hard to share across processes, due to security and compatibility issues. Ultimately, even applications that implement their own userspace-based cache often still rely on the page cache by default as a "second-tier" cache [4,29,54], allowing operators to fully utilize the server's memory and share memory across processes. As such, despite the page cache's limitations, it is still used extensively by storage-optimized workloads, such as key-value stores [33,54,65], databases [34,56], and ML inference and training systems [13,53].

Unfortunately, these factors yield a status quo where potential performance gains are left on the table. Properly customizing the page cache is not an easy task – it is deeply intertwined with other performance- and correctness-critical memory management and filesystem code paths. While work to modernize the page cache is ongoing, it does not yet seem to have achieved this goal. In particular, MGLRU, an alternative LRU implementation for the page cache, has still not been enabled by default in upstream Linux several years after it was introduced, and it does not provide customization interfaces [17,19]. Indeed, in §5 we show that MGLRU sometimes underperforms and sometimes outperforms the default LRU algorithm, and that in general there is no single eviction policy that performs best across a wide range of workloads.

We now provide a primer on the Linux page cache. We also describe the eBPF framework, which cachebpf uses to allow applications to write custom page cache policies.

2.1 Linux Page Cache

The page cache is a core component of the Linux kernel, responsible for accelerating access to storage. While anonymous memory pages are stored similarly to file-backed memory, in this paper we focus specifically on file-backed memory. The kernel's default eviction policy is an LRU approximation algorithm which uses two FIFO lists. As shown in Figure 1, when a page is first fetched from storage, it is added to the tail of the inactive list. If that page is accessed again, it will eventually be promoted to the active list. The goal of this policy is to use the inactive list as a preliminary filter and keep frequently accessed pages in the active list. When eviction is

¹The Linux page cache algorithm description is based on Linux v6.6.8.

triggered, pages are removed from the head of the inactive list. If necessary, the page cache will balance the lists by demoting pages from the head of the active list to the tail of the inactive list. Notably, during balancing or shrinking, pages in the active list that have been referenced are typically demoted to the inactive list, rather than being given another chance in the active list, as is typical for LRU or CLOCK algorithms.

Importantly, active and inactive lists are segmented by cgroup. cgroups are a Linux feature which isolate resource usage for groups of processes [35]. Each cgroup has its own set of page cache lists which count toward its memory allocation, allowing for cgroup-specific eviction when its memory threshold is reached. Processes in cgroup A can access a page "owned" by cgroup B – such an access will update the page's metadata (affecting its placement in cgroup B's lists), but will not count against cgroup A's memory limit. The combination of these per-cgroup lists make up the page cache as a whole.²

The page cache also keeps track of "shadow entries" in order to mitigate thrashing. These entries keep track of metadata enabling calculation of a page's refault distance (i.e. the time elapsed between eviction and the new request). If a page has been evicted and then fetched again recently enough, the kernel may decide to insert it directly into the active list instead of the inactive list. There are several additional edge cases and heuristics in the kernel's implementation, but these are the broad strokes of the existing policy.

Folios. Linux developers are in the process of converting various usages of struct page to *folios*, which represent either zero-order pages (a single page) or the head page of a compound page (a group of contiguous physical pages that can be treated as a single larger page) [16]. While the page cache now largely uses folios, we use the terms "folio" and "page" interchangeably, as in our workloads all folios represent a single page.

Userspace interfaces. While LRU is a commonly-used eviction policy that works well across many workloads, there are many applications that would benefit from a different policy for part or the entirety of their I/O requests. For example, LRU is notoriously bad for scan-like access patterns. This gap between applications and the kernel can be partially mitigated by the madvise() and fadvise() system calls. These interfaces allow userspace applications to give *hints* to the kernel about how to handle certain ranges of memory or files.

While these hints may help in simple cases, we show in our evaluation that they don't function as expected for some workloads. Additionally, while the hints may have a semantic meaning, their actual behavior is highly dependent on the kernel implementation, which is opaque, may change across versions, and can yield unexpected results [9, 49]. Advice values may also be ignored by the kernel for a range of reasons, or may have restrictions on what memory they can be applied

to. Most importantly, these hints are still subject to the basic inflexible structure of the kernel's approximate LRU policy.

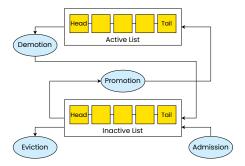


Figure 1: Overview of the current Linux page cache eviction policy.

2.2 eBPF

eBPF [26] is a sandboxing technology that enables userspace functions to run in the Linux kernel in a safe and controlled manner. eBPF has found many use cases, including observability [37], security [39,51], scheduling [15,38,42], and I/O acceleration [36,71,74–76]. eBPF programs are *verified* by the kernel before they can be run, ensuring, for example, that the programs don't contain illegal memory accesses, and that they will terminate within a fixed number of instructions.

3 Challenges

There are several challenges in allowing applications to customize the page cache using eBPF. We describe them below.

- Scalability. Modern SSDs support millions of IOPS [23, 63], requiring the page cache to efficiently handle millions of events a second. Any changes to the page cache in order to enable custom policies must incur a low overhead, and the policies themselves must also be efficient.
- Flexibility. Researchers have proposed many different caching algorithms for different use cases. These algorithms often require custom data structures. Any interface for custom policies must be flexible enough to accommodate the diversity of existing caching algorithms.
- 3. Isolation and sharing. The page cache is shared by many applications. Therefore, we must avoid a situation where one application's policy interferes with those of other applications, while still allowing applications to benefit from the shared nature of the page cache.
- 4. **Memory safety.** Custom eviction policies return page references to the kernel to indicate which pages to evict. This must not lead to unsafe memory references.

²Technically, each NUMA node has its own set of per-cgroup lists, but this does not affect our design.

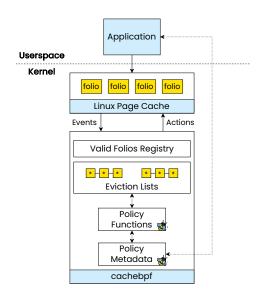


Figure 2: Overview of cachebpf. Eviction lists hold pointers to folios.

4 Design and Implementation

In this section, we present cachebpf's architecture and discuss how it addresses the challenges described in §3. Figure 2 shows a diagram of the system. At a high level, cachebpf allows users to run custom eviction policy functions, which are implemented as eBPF functions in the kernel. The policy functions are triggered by particular events (e.g., folio eviction, access, admission), and they operate on a user-specified number of variable-sized eviction lists, which store pointers to the folios managed by the policy. The policy functions decide which folios to admit or evict to and from the lists based on metadata (e.g., folio access frequency and recency, which thread accessed the folio), which is stored in eBPF maps. At eviction time, cachebpf runs a user-defined eviction function to propose a set of eviction candidates for the kernel to evict. We find that while this interface is relatively simple, it is quite flexible, and can support a very wide set of eviction policies from the literature either exactly, or approximately.

We now describe cachebpf's design in detail, starting with our design choice to implement cachebpf's eviction policies within the kernel, rather than in userspace, to ensure scalability (challenge 1 from §3).

4.1 Policies in Kernel or Userspace?

Our first key design decision is whether to run cachebpf's policies in the kernel or in userspace. While from a development standpoint it might be simpler to run the eviction policies in userspace, doing so would require notifying userspace about all page cache events. However, modern SSDs can service millions of IOPS, each of which may trigger a page cache event (e.g., folio access, insertion).

We run a set of experiments to estimate the "best-case"

| Workload | Baseline | Benchmark | % Degradation |
|------------------|----------------------------|----------------------------|------------------|
| YCSB A YCSB C | 82,808 op/s 76,166 op/s | 69,089 op/s 62,578 op/s | -16.6% -17.8% |
| Uniform | 44,618 op/s | 35,443 op/s | -20.6% |
| Search | 42.3s | 44.4s | -4.7% |

Table 1: Performance of workloads without and with userspace-dispatch.

overhead of such a userspace-offload architecture. We attach eBPF programs to existing kernel tracepoints (folio inserted, accessed, and evicted). The eBPF programs use a lockless ring buffer to notify userspace on each event [44]. Since no userspace logic actually processes these events, this provides an optimistic measure of this architecture's overhead.

We run two applications on a standard enterprise SSD to evaluate this architecture: YCSB workloads using RocksDB [65], a key-value store, and a file search workload using ripgrep [32], a parallelized grep-like tool, where we search the Linux kernel sources 10 times. The workloads are allocated 8 GiB and 1 GiB of memory, respectively. We run these applications on both the baseline system and with the eBPF benchmark programs. The results are presented in Table 1, with the benchmark yielding up to a 20.6% performance decrease, without even implementing a custom eviction policy.

Thus, based on the results of our experiments, we rule out implementing page cache policies in userspace, and instead opt to run the policies within the kernel as eBPF functions. We decide to use eBPF as it has already proven to match the kernel's performance, even in performance-critical domains such as networking [36] and storage [74]. While eBPF programs face many restrictions due to the verifier, we find that cachebpf can provide sufficient flexibility for custom policies, as we describe below.

4.2 Interface

Caching is an active area of research, with many recently-proposed eviction and admission algorithms [5–7, 64, 67, 68, 70, 72] that aim to take advantage of different features of a workload (e.g., recency, frequency, size), using various techniques (e.g., conditional probability models [5], Markov chains [7], machine learning [64]). To ensure flexibility, cachebpf should allow developers to experiment with a wide range of caching policies, including relatively sophisticated ones. We now describe cachebpf's API and demonstrate how it can be used to create a wide range of policies, addressing challenge 2 in §3.

4.2.1 Policy Functions

cachebpf allows applications to define custom eviction policies as *policy functions*, a set of eBPF programs that trace caching events and determine which folios should be evicted from the page cache. Policy functions can be triggered by five

```
// Policy function hooks
struct cachebpf_ops {
   s32 (*policy_init)(struct mem_cgroup *memcg);
    // Propose folios to evict
   void (*evict_folios)(struct eviction_ctx *ctx,
         struct mem_cgroup *memcg);
   void (*folio_added)(struct folio *folio);
   void (*folio_accessed)(struct folio *folio);
    // Folio was removed: clean up metadata
   void (*folio_removed)(struct folio *folio);
    char name[CACHEBPF_OPS_NAME_LEN];
};
struct eviction_ctx {
    u64 nr_candidates_requested; /* Input */
   u64 nr_candidates_proposed; /* Output */
    struct folio *candidates[32];
};
```

Figure 3: struct_ops for cachebpf and eviction context.

events: policy initialization, requests for eviction, folio admission, folio access, and folio removal. The policy function interface is implemented using eBPF's recent struct_ops kernel interface [46], as shown in Figure 3.

These five events are central to caching decisions in the page cache. Notably, requests for eviction and folio removal are different: the former involves the kernel asking the policy to propose folios to evict, and the latter is the kernel informing the policy that a folio was actually evicted. This distinction exists for the following two reasons. A folio can be evicted in circumvention of the "normal" eviction path if, for example, the file containing it is deleted. Conversely, in rare cases, proposing a folio for eviction does not guarantee that it will be evicted (e.g., the folio is in active use by the kernel).

We use eBPF's struct_ops feature in order to minimize the verifier changes needed to add new eBPF hooks. struct_ops was designed to allow kernel subsystems to expose modular interfaces to eBPF components, and has already been used for TCP congestion control algorithms, FUSE eBPF filesystems, handling HID driver quirks, and sched_ext [15, 18, 27, 45]. struct_ops programs are loaded into the kernel like any other eBPF program. Using struct_ops also makes it much easier to extend cachebpf and add new hooks. For example, we implemented an extension to cachebpf that added a page cache admission filter with only 15 additional lines of verifier-related code.

4.2.2 Eviction Lists

Eviction algorithms are implemented on a wide range of data structures. Nevertheless, we observe that many of these policies can be implemented either exactly or approximately

```
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 2: cachebpf eviction list API.

using linked lists, where the policy iterates over one or more lists and evicts items based on a calculated per-item score. For example, the "classic" eviction policies, (e.g., LRU, MRU) are all based on lists, with items inserted or evicted from the head or tail of a list. Similarly, families of policies like ARC [50] or segmented LRU [43] can be implemented using multiple variable-sized lists, where items are inserted into any list or moved between lists. Even recent "state-of-the-art" policies, such as LHD, S3-FIFO, or LRB either store data directly in a list [70,72], or simply select a sample of objects and evict the ones with the lowest *score* [5,64].

In order to facilitate an interface flexible enough for all these policies, cachebpf is built around an *eviction list API*, a simple interface for policies to construct and manipulate a set of variable-sized linked lists. Each node in the list corresponds to a single folio, and stores a pointer to that folio, rather than the folio itself. Importantly, the actual folios are still stored and maintained by the default kernel page cache implementation, in order to minimize changes to the kernel.

This API is implemented as a set of eBPF kfuncs (in-kernel functions that are exposed to eBPF programs) and is shown in Table 2.³ For example, init() will typically call list_create() to create a new eviction list, and folio_added() will call list_add() to add the folio to a list. Newly-created lists are added to a "registry", an internal perpolicy hash table which maps between the list IDs (exposed to eBPF) and the lists themselves. Notably, these lists are indexed – that is, given a folio pointer, the APIs can directly obtain that folio's list node. This property is necessary for operations such as deletion from the list, and is facilitated using a per-policy hash-table which maps from folios to list nodes. We discuss the use of this hash table further in §4.4.

4.2.3 Eviction Candidate Interface

To facilitate eviction, policy functions iterate over their eviction lists in order to determine which folios to evict. Note that policies do not directly evict folios – rather, they propose *eviction candidates* to the kernel, which checks if the folios

³The actual functions have a "cachebpf" prefix to prevent name collisions, but we omit it for brevity.

are indeed valid eviction targets (i.e. not pinned or in other use by the kernel) and evicts them if so.

The eBPF framework currently does not provide a clean way to iterate over the eviction lists, so cachebpf provides a new iteration kfunc which allows policy functions to specify how to iterate over a list and make decisions for each node. Specifically, list_iterate() takes a list to iterate over, an options struct, an eviction context, and a callback function. The callback function, which is also an eBPF program, is called on each node, and the policy decides whether to keep or evict that folio. Folios chosen for eviction are added to the candidates array in the eviction_ctx struct. The options struct specifies how the interface should treat evaluated folios. For example, they can be left in place, moved to the tail of the list, or moved to a different list. This enables implementing policies that make use of multiple lists and require balancing the lists, such as S3-FIFO or ARC. We also provide a "batch scoring" mode for this interface, where the callback function is used to compute *scores* for N folios, with the C folios with the lowest score chosen for eviction. This mode can be used for policies such as LFU.

In order to ensure correct verification of our callback functions, we added $\sim\!80$ lines to the eBPF verifier to "register" our iteration interfaces, on top of eBPF's existing support for callback functions. Additionally, to protect against eBPF program misbehavior, this interface performs the requisite bounds-checking and enforces loop termination.

4.2.4 eBPF limitations

We ran into a number of challenges when implementing cachebpf's eviction list API. eBPF maps, the standard way to maintain state in eBPF programs, do not provide interfaces that both store items in a specified order while also providing random access, both of which are necessary in order to implement eviction properly. Specifically, eBPF provides maps such as BPF_MAP_TYPE_QUEUE and BPF_MAP_TYPE_STACK, which provide pop() and push() operations, but do not allow deleting or accessing elements from the "middle" of the map. Conversely, BPF_MAP_TYPE_HASH provides random access, but no method to easily maintain an ordering of elements (e.g., MRU order). A notable exception is BPF_MAP_TYPE_LRU_HASH, which provides both an LRU structure and random-access, but is too deeply tied to its specific algorithm for our purposes [22]. This necessitated the development of a custom data structure for cachebpf.

While eBPF has started to introduce experimental support for custom data structures and more complex locking in eBPF, this support is not yet mature enough for our use case [2, 20, 25]. As such, we designed our list API to be managed by the kernel and exposed to eBPF via kfuncs. Additionally, in order to avoid concurrency issues and verifier limitations around locking, the provided API is concurrency-safe and makes use of locks under the hood, in the kernel implementations. As

```
u64 lfu_list;
int lfu_policy_init(struct mem_cgroup *cg) {
    lfu_list = list_create(cg);
    return 0:
void lfu_folio_added(struct folio *folio) {
    u64 freq = 1;
    list_add(lfu_list, folio, true); // Add to tail
    bpf_map_update_elem(&freq_map, &folio, &freq);
void lfu_folio_accessed(struct folio *folio) {
    u64 *freq = bpf_map_lookup_elem(&freq_map, &folio);
    __sync_fetch_and_add(freq, 1); // Increment freq
long score_lfu(int id, struct folio *folio) {
    return bpf_map_lookup_elem(&freq_map, &folio);
void lfu_evict_folios(struct eviction_ctx *ctx, struct
    mem_cgroup *cg) {
    struct iter_opts opts = { /* Set scoring mode */ };
    list_iterate(cg, lfu_list, score_lfu, &opts, ctx);
}
void lfu_folio_removed(struct folio *folio) {
    bpf_map_delete_elem(&freq_map, &folio);
```

Figure 4: Simplified LFU implementation with cachebpf.

eBPF matures, new features could further reduce overhead and provide even more flexibility for eBPF policies.

4.2.5 Example: LFU Policy

To get a better sense of how cachebpf's policy functions can be used to implement custom policies, we walk through implementing a simple eviction policy, LFU, using cachebpf. LFU evicts the least-frequently accessed item in the list, which requires storing additional metadata. Our LFU implementation uses a single list and an eBPF map to store folio access frequencies. It approximates LFU using cachebpf's batch scoring mode to select the C (e.g., 32) least-frequently accessed folios out of N (e.g., 512) folios.

A simplified version of the policy is shown in Figure 4. When the policy is loaded and $lfu_policy_init()$ is called, we create a new eviction list. When a folio is added, $lfu_folio_added()$ adds the folio to the tail of the list using $list_add()$ and initializes its frequency to 1 in the $list_add()$ and initializes its frequency to 1 in the $list_add()$ and initializes its frequency to 1 in the $list_add()$ and initializes its frequency to 1 in the $list_add()$ and $list_add()$ which a folio is accessed, we increment its frequency. When eviction is triggered, $lfu_evict_folios()$ calls $list_iterate()$, which calls the score_lfu() callback function on $lost_add()$ nodes in the list. The score function returns the frequency of each folio as its score. cachebpf then selects the $list_add()$ folios with the lowest scores as eviction candidates. When a folio is evicted by the kernel, $lfu_folio_removed()$ is called, and the folio's metadata is

removed from the map. Additionally, it is not necessary to remove the folio from the list on eviction, as this is taken care of by cachebpf. We discuss this point further in §4.4.

4.3 Isolation

We now tackle the third challenge from §3: how to allow applications to deploy their own policy functions without interfering with other applications' policies, while preserving the sharing property of the page cache, whereby applications can avoid having to load duplicate pages into memory.

We make the observation that implementing policies within a cgroup can address this challenge. This is due to the fact that within a cgroup, processes have the same custom eviction policy, and different cgroups running on the same server can each use their own eviction policy. In addition, deploying policy functions per-cgroup fits the common pattern of deploying modern applications via containers, which isolate each application in its own memory cgroup. Note that processes from cgroup A can still access page cache memory managed by cgroup B, and benefit from accessing shared data.

To support per-cgroup policies, we extend eBPF's struct_ops functionality to support cgroup-specific struct_ops (currently, it only supports system-wide policies). This involved adding a cgroup identifier (in the form of a file descriptor) to the kernel's struct_ops loading interface, along with corresponding libbpf interfaces in userspace.

4.4 Memory Safety

We must ensure that cachebpf does not allow unsafe memory accesses when interacting with folio pointers (challenge 4 from §3). Specifically, cachebpf must ensure that eBPF programs return valid pointers to the kernel (i.e. in the eviction candidate interface). Otherwise, a malicious eBPF program could return invalid values, leading to memory corruption or a kernel crash. Frameworks like sched_ext solve this in part by using PIDs as identifiers for processes in userspace dispatch. However, folios do not have analogous easily-obtainable unique identifiers, so we resort to using folio pointers.

In order to validate these pointers, we implement a registry of "valid folios" in the system. When a folio enters the page cache, cachebpf adds it to the registry. When a folio is evicted, it is removed from the registry. When a cachebpf eviction proposes a set of folio eviction candidates, the kernel uses the registry to verify that each candidate is indeed a valid folio before proceeding with eviction. This registry is implemented as a hash table with a per-bucket lock, which also stores a folio's list node (as described in §4.2.2), which maps from folio pointer to list node. We find that this design incurs minimal overhead, which we evaluate in §5.3. Future developments in eBPF may make it easier to keep track of "trusted" pointers, potentially allowing us to remove this check and further reduce overhead.

We also protect against adversarial behavior by providing a fallback for eviction. For example, if the kernel asks a faulty policy to evict 10 folios, but it only proposes 5 candidates, the kernel will fall back to its default policy and evict additional folios. Similarly, when a folio is evicted, the kernel ensures that it is removed from any eviction list it is present in, in order to release memory resources and minimize stale references lying around. Similar fallbacks are present in other frameworks, such as sched_ext, which implements a watchdog that forcibly removes misbehaving policies.

4.5 Kernel Implementation Complexity

Implementing cachebpf required adding ~2000 lines to the kernel. Only a fraction of these lines touched core kernel code: 210 lines in the page cache (most of which are the new eBPF hooks), 80 lines in the verifier (supporting our callback functions), and 80 lines in cgroup code. Additionally, implementing per-cgroup struct_ops required 220 lines in the kernel and 75 lines in libbpf. The remaining lines implemented pure cachebpf functionality: 750 lines for cachebpf's eviction list kfuncs, and 580 lines to implement registry operations and register cachebpf with the verifier.

5 Evaluation

We aim to answer the following questions:

- Q1: Is cachebpf flexible enough to implement a variety of eviction policies? Can cachebpf policies improve application performance with low developer effort? (§5.1)
- **Q2:** Can different applications use different policies without interfering with each other? (§5.2)
- **O3:** What is the overhead of cachebpf? (§5.3)

System configuration. We conduct our experiments on Cloudlab [24] c6525-25g machines, with a 16-core AMD Rome CPU, 128GB of memory and a 480GB SSD drive. We use CPU-pinning and disable SMT, swap, and address space randomization to make our results more reproducible. We also drop the page cache before each test. We run Ubuntu 22.04 with Linux v6.6.8 as the kernel.

5.1 Case Studies: Custom Policies (Q1)

In this section, we describe several case studies on how applications can utilize cachebpf to achieve better performance. First, we show how cachebpf can be used to implement a wide range of eviction policies, tailored to different applications: from simple "classic" policies (MRU and LFU) to state-of-the-art policies such as LHD [5], which uses conditional probabilities to model different page features (e.g., age, frequency), and S3-FIFO [70]. We then explore how an

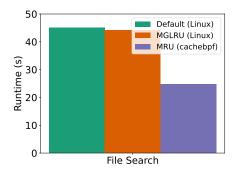


Figure 5: File search workload results (MRU policy).

application can make its eviction policy *aware* of application-specific information, such as assigning different priorities to specific types of requests.

5.1.1 Most-Recently Used (MRU)

Scan-like workloads do not perform well under LRU-like policies when the scan length is larger than the size of the LRU list. For example, one such scan-heavy workload is searching through files in a codebase. Consider a developer working on a large codebase, such as the Linux kernel, and continuously searching for certain terms. In such a scenario, an LRU-like policy would evict the files that were least-recently searched, but those are precisely the files that are required at the start of the next search. While readahead can help mitigate this issue for single-file scans by prefetching sequential blocks, it cannot predict which blocks to fetch across files.

We use cachebpf to develop an MRU eviction policy for this use case. In contrast to LRU, MRU will evict the folios that were most recently searched, and will be used again furthest in the future. In order to facilitate this, our policy adds folios to the head of the list on insertion, and moves them back to the head on access. No metadata is required, as nodes are stored in the correct order in the eviction list. In a simplified version of the policy, when eviction is triggered, the first 32 nodes in the list are selected as eviction candidates using cachebpf's iterate interface. However, if the policy decides to evict folios right after they were added to the page cache, they may still be in use by the kernel to service the I/O request. This would lead to the kernel refusing to evict the folios and resorting to the fallback path to evict folios. Therefore, we skip a small fixed number of folios when iterating the eviction list before proposing eviction candidates.

Evaluation. To evaluate the policy, we construct a file search workload that searches the Linux kernel codebase (v6.6), using the multi-threaded *ripgrep* CLI tool [32]. More specifically, we perform 10 searches within a 1GiB cgroup, which is roughly 70% the size of the codebase (excluding Git history). We compare the cachebpf MRU policy with the default Linux kernel policy as well as the kernel's experimental

MGLRU policy. The results in Figure 5 show that cachebpf is almost $2 \times$ faster than both baseline and MGLRU, since both policies suffer from the scan "pathology" of LRU.

5.1.2 Least-Frequently Used (LFU)

An additional disadvantage of LRU-like algorithms is that they only take into account a single feature (recency) when making eviction decisions. However, other features, such as access frequency, may be better suited for certain workloads, especially skewed workloads where the access distribution is static or slow-changing. One such workload is the popular YCSB benchmark, made to model cloud OLTP applications, in which the probability of each key being requested is drawn independently from a static distribution (by default, Zipfian).

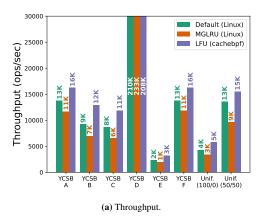
We use cachebpf to implement an LFU policy, which takes access frequency into account when evicting folios by evicting those with the lowest access frequency among the eviction candidates. Our LFU policy is an *approximate* LFU policy, as it does not evict the global least-frequently used folio, but only the least-frequently ones among the current batch of folios considered for eviction. An exact policy would either yield higher overhead or require more complex data structures, which eBPF does not yet support. We described the implementation of our LFU policy in §4.2.5.

Evaluation. We evaluate our LFU policy by running LevelDB [33], a popular key-value store, on the YCSB (Zipfian) workloads, as well as against uniform and uniform-read-write workloads. We compare this custom policy against both the default and MGLRU Linux policies, using a 100GiB database with a 10GiB cgroup. Our results in Figure 6 show that cachebpf's LFU policy outperforms both the default and MGLRU, for all the YCSB Zipfian workloads and the uniform workloads, except for YCSB D, which only uses the latest key-value pairs and as such is cached entirely in-memory. cachebpf achieves up to 37% better throughput than the default Linux policy, and interestingly, it outperforms MGLRU by an even greater margin. We also measure the P99 read latency, for which cachebpf beats the default policy by up to 55%. Note that YCSB D's tail latency barely registers in the figure due to its lack of disk accesses. We also evaluated the YCSB workload with our other policies, but LFU outperformed those policies as well, so we omit those results.

Takeaway 1: cachebpf can significantly improve application performance even with simple policies (e.g., MRU, LFU) that match the application's access patterns.

5.1.3 S3-FIFO

S3-FIFO [70] is a recent caching policy designed for keyvalue caches, which uses three FIFO queues to quickly remove "one-hit wonders" (keys that are accessed only once). It has been shown to yield significant throughput gains on a number



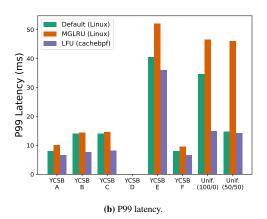


Figure 6: YCSB workload results (LFU policy).

of workloads. We implement S3-FIFO using cachebpf and evaluate it on Twitter production cache traces below.

S3-FIFO uses a main FIFO and a small FIFO to hold $\sim 90\%$ and 10% of the objects, respectively. Upon insertion, objects are typically added to the small FIFO. It uses a ghost FIFO to track recently-evicted objects, in order to promote them quickly to the main FIFO on readmission. The small FIFO is used to filter out short-lived objects, while objects that are accessed more often are promoted from the small FIFO to the main FIFO. The access frequency of the objects is tracked, but is capped at a maximum of 3, in order to ensure that a burst of accesses does not prevent objects from being evicted.

We implement the main and small FIFOs as two eviction lists, and the ghost FIFO as a BPF_MAP_TYPE_LRU_HASH map. The map will then automatically remove entries from the ghost FIFO in LRU order when it hits capacity. When a folio is evicted, we create a ghost entry using a pointer to its struct address_space (which represents a file's contents), along with the folio's offset in the file, as the key. Note that we cannot use folio pointers as the key, as they are not persistent across evictions. While we cannot implement the ghost FIFO as an eviction list (as they operate on *valid* folios), it is more performant and simpler to use an existing eBPF map. We find that the combination of existing eBPF features and cachebpf is sufficiently flexible to implement complex eviction policies.

On folio addition, we set its access frequency in an eBPF map, and update it on access. We use eviction candidate requests to evict folios, but also to maintain the 90%-10% ratio between the main and small lists. If the small list is overrepresented, we evict from it. We use cachebpf's eviction iteration interface: if a folio's access frequency is greater than 1, we move it to the tail of the main list, balancing the lists. Otherwise, we propose the folio for eviction, and move it to the tail of the small list so that it isn't considered again before it is evicted. When evicting from the main list, we use the iteration interface to find folios with access frequency of 0. If we cannot find enough of those, we search for folios with access frequency of 1, and then 2, and so on. All folios

that are considered for eviction have their access frequency decremented and are moved to the tail of the main list.

5.1.4 Least Hit Density (LHD)

LHD is a relatively sophisticated eviction policy that uses conditional probabilities to predict which objects are most likely to be accessed in the future [5]. LHD uses a metric called *hit density* in order to determine which objects should be evicted, along with a *dynamic ranking* approach which allows it to automatically tune its eviction policy over time. We implement LHD using cachebpf, based on the implementation in libcachesim [69, 70, 72].

Our implementation only uses one eviction list. However, folios are divided into *classes* partially based on when they were last accessed and their age at that time. Each class stores its own statistics (e.g., hits, evictions, hit densities) for different folio ages. Folios are not explicitly "owned" by classes – instead, they use metadata from classes based on which class they most closely correspond to at a given time. When a folio is added, we store its metadata, such as its last access time and age at that time, in an eBPF map. That metadata is updated on folio access, and removed when the folio is evicted.

Folios are selected as eviction candidates based on their hit density (or more accurately, the hit density of the class and age they most closely correspond to). LHD iterates over the list and selects the folios with the lowest hit density as eviction candidates. While this process is fairly straightforward, it is enabled by accurate computation of hit densities over time. LHD requires periodically "reconfiguring" its hit densities and other statistics in order to ensure that its probability distributions are accurate and aged appropriately over time using an exponentially weighted moving average (EWMA).

This reconfiguration process needs to run every N folio admissions or insertions (where N is a relatively large number – e.g., 2^{20}). However, reconfiguration is a relatively expensive process, requiring iterating over all of the policy's metadata and adjusting it. In order to avoid performing this operations

in the page cache's insertion or access hot paths, we use an eBPF ring buffer to notify userspace that reconfiguration needs to take place. Userspace then calls an eBPF program of type BPF_PR0G_TYPE_SYSCALL, which allows running an eBPF program without attaching it to a specific hook. This program then performs the required reconfiguration, including computing updated hit densities, and scaling or compressing distributions as necessary. We use atomic operations to ensure that the page cache can continue using these values, albeit with some potential inaccuracy, which we permit for the sake of performance. While we could implement this reconfiguration step in userspace, doing so would have required numerous syscalls to interact with eBPF maps, and atomic updates would not have been possible. Additionally, we note that in a standard LHD policy, hit densities and other parameters are stored as floating-point values. However, eBPF does not support floating-point operations, so we resort to scaling values by a large constant in order to approximate such calculations.

Evaluation. For many real-world workloads, it may not be obvious in advance which policy works best for a given workload. cachebpf makes experimentation easy, allowing developers to implement a set of policies and *empirically* choose the best one for each workload.

We evaluate our LHD, LFU, and S3-FIFO policies on production traces taken from the Twitter cache workloads [69]. The workloads divide the traces by cluster ID. We compare these policies to Linux's default and MGLRU policies. Each cluster was evaluated with a cgroup size set to 10% of the cluster's data size using LevelDB. As shown in Figure 7, we find that, in general, there is no single policy that is best for all workloads. While LHD beats the default and MGLRU policies by 13% and 30%, respectively, on cluster 34, and LFU beats them by 13% and 34% on cluster 52, MGLRU dominates on clusters 17 and 18. In cluster 24, the default policy is best, while MGLRU consistently resulted in out-of-memory errors, hence its 0 throughput result. Meanwhile, S3-FIFO beats or matches the baseline on clusters 34 and 52, but does not outperform the other cachebpf policies.

Takeaway 2: There is no one-size-fits-all policy that performs best for all workloads. Customization and experimentation are necessary to maximize performance.

5.1.5 Application-Informed Policy (GET-SCAN)

In addition to enabling the implementation of a variety of general eviction algorithms, cachebpf enables applications to use eviction algorithm tailored to their design. In other words, the eviction algorithm can be made *aware of application-level abstractions* and uses this information to make better decisions. To illustrate the value of application-informed policies, consider the case of heterogeneous queries in databases. For example, a database serving financial transactions could see many small queries for individual payments, while also per-

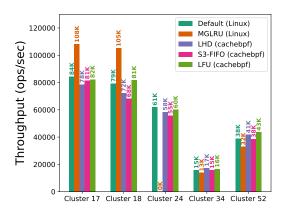


Figure 7: Twitter workload results (LHD, S3-FIFO, and LFU policies) using LevelDB. No one policy performs best across the different clusters.

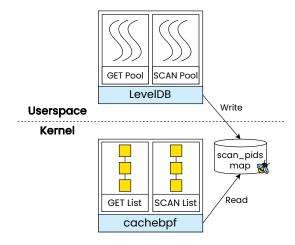


Figure 8: Overview of GET-SCAN policy implementation.

forming slower scan-like queries in the background to conduct fraud detection, reconciliation, and other business processes. While these scan-like queries are important, they typically have more relaxed service-level objectives. However, these large scan-like requests can "pollute" the page cache and degrade the performance of the smaller requests, as generic eviction algorithms struggle to isolate the folios used by these requests. Ideally, the page cache should prioritize the small requests over the large ones in the presence of memory pressure. Using cachebpf, we can build an application-informed policy that fulfills this requirement.

To simulate the application we describe above, we run LevelDB with a mixed SCAN/GET workload. This workload is highly skewed and is composed of 99.95% GET requests with a small amount of SCANs (0.05%). We use a separate thread-pool for SCAN requests to avoid head-of-line blocking at the scheduling level, as per prior work [42], with a disjoint set of threads handling GET requests. While the workload exhibits good cache locality for GETs, it has poor locality for SCANs, which span a large number of folios and exhibit high reuse distance. The existing kernel eviction policy cannot

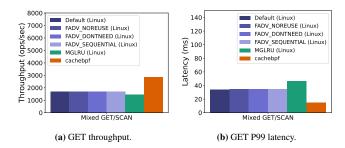


Figure 9: Mixed GET-SCAN workload results.

handle this scenario well, leading to cache pollution due to a large number of SCAN folios.

Using cachebpf, we design a policy that is aware of the different request types. We observe that a folio accessed by a SCAN should not be worth the same as a folio accessed by a GET. To implement prioritization, the policy uses two eviction lists: one for folios inserted by GET requests, and the other for those inserted by SCAN requests. When loading the policy, the application initializes an eBPF map with the PIDs of the SCAN threads. When a folio is inserted, the policy checks whether the PID of the current task is present in the map to determine which eviction list to add the folio to. Each eviction list independently maintains an approximate LFU policy, as described in §5.1.2. When the kernel requests eviction candidates, the policy prioritizes evicting folios from the SCAN list. Figure 8 illustrates this policy.

Evaluation. To evaluate the policy, we compare against Linux's default and MGLRU policies, and various fadvise() options: FADV_DONTNEED, FADV_NOREUSE and FADV_SEQUENTIAL (on top of the default policy). We apply these fadvise() options to files used by SCAN requests, in order to inform the kernel in advance that we plan to read the files sequentially or only once (SEQUENTIAL and NOREUSE) or that we no longer need the folios after their use (DONTNEED). As shown in Figure 9, cachebpf's application-informed policy achieves $1.70\times$ the throughput and 57% lower P99 latency for GET requests, while SCAN requests experience an 18% throughput decrease. In addition, the fadvise() options do not help much, demonstrating the inadequacy of existing kernel page cache interfaces compared to cachebpf. MGLRU performs even worse than the default LRU.

Takeaway 3: Even very simple application-aware eviction policies can significantly improve performance.

Takeaway 4: Existing Linux page cache customization interfaces are ineffective.

5.1.6 Implementation Complexity

Table 3 shows the lines of eBPF and userspace loader code necessary to implement each of the aforementioned policies, along with a simple FIFO policy. The policies are all implemented in at most a few hundred lines of code, a much smaller

| Policy | eBPF LoC | Userspace LoC |
|----------|----------|---------------|
| FIFO | 56 | 118 |
| MRU | 101 | 87 |
| LFU | 221 | 107 |
| S3-FIFO | 287 | 139 |
| GET-SCAN | 324 | 107 |
| LHD | 366 | 152 |

Table 3: Lines of eBPF and userspace loader code in cachebpf policies.

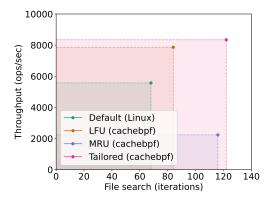


Figure 10: Using cachebpf, multiple applications can run different eviction policies, yielding better performance for all.

amount than would be necessary to implement them within the kernel (or even in userspace). We find that cachebpf reduces the complexity of developing new policies, by using the pre-defined list and policy function abstractions, as well as by relying on the kernel's existing page cache to actually store the folios. In addition, developer experience and velocity is greatly improved, since eBPF prevents kernel crashes and many types of bugs, enabling developers to focus on the policy logic. Thus, cachebpf allows developers to accelerate their applications with a relatively modest amount of effort.

Additionally, we plan to open source all of our policies, allowing developers to easily try them with their applications, lowering the barrier to entry for using cachebpf.

5.2 Isolation (Q2)

The Linux page cache already provides a measure of isolation by giving each cgroup its own set of LRU lists. cachebpf takes advantage of this design by enabling each cgroup to have its own custom policy. We demonstrate that this is a useful capability by simulating and comparing against "global" policies, as opposed to cachebpf's per-cgroup policies. We create two cgroups, one running a YCSB C workload with LevelDB, and the other running a file search workload with ripgrep. The YCSB cgroup is allocated 10GiB and the file search cgroup is allocated 1GiB. We run these workloads under four configurations: both cgroups using the default policy, both using LFU, both using MRU, and a "tailored" setup: YCSB with LFU and file search with MRU.

Evaluation. Figure 10 shows that the tailored setup beats the other three configurations, yielding 49.8% and 79.4% improvements for YCSB and file search, respectively, over the baseline configuration. While the other two cachebpt configurations provide performance improvements for the workloads corresponding to their policy, they can significantly degrade the performance of the other workload, demonstrating that global policies are indeed not a viable solution. Note that YCSB improves further in the tailored setup compared to the LFU configuration (and vice versa for file search compared to the MRU configuration). This is due to improved caching of the workloads yielding reduced disk contention. Additionally, the file search workload improves in the LFU configuration for the same reason.

Takeaway 5: Using cachebpf with per-cgroup policies allows for fine-grained control and improved performance.

5.3 Memory and CPU Overhead (Q3)

The advent of faster and larger storage devices means that the page cache (and cachebpf) must be able to handle millions of events per second. We run a number of micro-benchmarks to investigate cachebpf's memory and CPU overhead.

5.3.1 Memory Overhead

cachebpf's primary memory usage is the valid folios registry hash table (§4.4). In the worst case, we set up the hash table with as many buckets as there are 4KiB pages in the cgroup (based on its configured size). Each bucket requires 16 bytes to store the hash table's internal list pointers. Thus, the memory overhead for an empty registry is:

$$\frac{\left(\frac{\text{cgroup_size}}{\text{page_size}}\right) \times 16}{\text{cgroup size}} = 0.4\%$$

Each filled entry in the hash table uses 32 bytes for the cachebpf list node. The full registry memory overhead is:

$$\frac{\left(\frac{\text{cgroup_size}}{\text{page_size}}\right) \times (16+32)}{\text{cgroup_size}} = 1.2\%$$

Therefore, the memory overhead for cachebpf's registry is between 0.4%-1.2% of a policy's cgroup's memory. We believe that this overhead could be significantly reduced with recent improvements to eBPF's handling of kernel objects, allowing eBPF to directly ensure that some pointers are trusted.

5.3.2 CPU Overhead

To measure the CPU overhead, we run the fio microbenchmark [3] with 8 threads on a *randread* workload and record the IOPS and CPU usage. We do this for both the default Linux policy and a cachebpf no-op policy, meaning that it uses the default eviction policy while still maintaining

| Cgroup Size | Baseline | cachebpf | Overhead (%) |
|-------------|----------|----------|--------------|
| 5 GiB | 234.80 | 236.51 | 0.72% |
| 10 GiB | 217.48 | 221.14 | 1.66% |
| 30 GiB | 197.67 | 198.01 | 0.17% |

Table 4: cachebpf μ CPU usage per I/O operation using fio.

cachebpf data structures. Then, we calculate and compare the CPU usage per I/O operation (measured in μ CPUs, i.e. one-millionth of a CPU). Table 4 shows that the CPU overhead of cachebpf is at most 1.7%.

Takeaway 6: cachebpf incurs relatively low CPU and memory overhead, while improving performance.

6 Related Work

There have been three predominant approaches to allow applications to customize the page cache. The first approach, which was explored in the 80's and 90's, was to design clean-slate extensible kernels [1, 8, 28, 60], which allow applications to customize kernel interfaces and policies. For example, VINO [60, 62] and SPIN [8] allow applications to customize buffer cache eviction, admission, and prefetching policies. These OS designs never achieved widespread use, even though some of their underlying ideas have become relevant again with the adoption of eBPF, which enables extensibility within monolithic kernels like Linux or Windows.

The second approach, introduced in the 90's, is to design customizable file systems, which allow applications to customize the page cache. ACFS [10, 11] is an application-controlled file system which enables customizing caching and prefetching. The XN [41] libOS file system enables running a userspace-level file system within the exokernel OS, which can be fully customized. More recent work in this vein is Bento [52], which allows custom file systems written in Rust to be installed in the kernel, without disrupting applications. None of these approaches would work with existing Linux or legacy file systems.

The third approach is for applications to simply implement their own userspace cache, with the option of bypassing the OS page cache with direct I/O. There are many examples of data systems that implement a userspace cache [14, 48, 65]. TriCache [30] is a recent framework that helps applications customize their own userspace caches. Nonetheless, many popular data systems still rely on the page cache, sometimes in conjunction with userspace caches [21, 34, 54, 58, 65].

There has been more recent work on customizing memory management policies using eBPF, such as huge page placement, page fault handling, and page table designs [55,61,77]. Most relevant to cachebpf, FetchBPF allows customizing Linux's memory prefetching policy, and could easily be integrated into cachebpf as an additional hook [12]. P2Cache

is conceptually similar to cachebpf, but only allows LRU or MRU ordering and changes the global page cache policy [47]. Additionally, the P2Cache paper is a work-in-progress workshop paper, is closed-source, and does not contain many details about its design, implementation, or evaluation.

7 Conclusion

This work explores the design of a new eBPF framework to implement custom eviction policies in the kernel, enabling applications to choose a policy according to their needs and making the latest caching research accessible to the kernel. We believe there are significant future research challenges in this area, such as exploring ML-based eviction algorithms and integrating more parts of the page cache into the cachebpf framework (e.g., writeback and prefetching). Furthermore, we are aware of efforts in the eBPF verifier to support more complex data structures and believe that cachebpf could benefit from these efforts.

8 Acknowledgments

We would like to thank Kostis Kaffes, Tanvir Ahmed Khan, and Yuhong Zhong for their feedback. We also thank the CloudLab team for their help in supporting our experiments. This work was supported by IBM, and NSF awards CNS-2143868 and CNS-2106530. Tal Zussman was supported by NSF award DGE-2036197. Ioannis Zarkadas is an Onassis Foundation scholar.

References

- [1] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference, Altanta, GA, USA, June 1986*, pages 93–113. USENIX Association, 1986.
- [2] Daroc Alden. A proposal for shared memory in BPF programs, 2024. https://lwn.net/Articles/961941/.
- [3] Jens Axboe. fio: Flexible I/O tester. https://github.com/axboe/fio.
- [4] Nidhi Bansal. An Overview of Caching for PostgreSQL, 2020. https://severalnines.com/blog/overview-caching-postgresql/.
- [5] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 389–403, Renton, WA, April 2018. USENIX Association.

- [6] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Sid-dhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching dynamic reallocation from Cache-Rich to Cache-Poor. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 195–212, Carlsbad, CA, October 2018. USENIX Association.
- [7] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 483–498, Boston, MA, March 2017. USENIX Association.
- [8] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In Proceedings of the fifteenth ACM symposium on Operating systems principles, pages 267–283, 1995.
- [9] Bryan Cantrill. A crime against common sense (MADV_DONTNEED), 2015. https://www.youtube.com/ watch?v=bg6-LVCHmGM&t=3518s, OmniTi Surge 2015.
- [10] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. ACM Trans. Comput. Syst., 14(4):311–343, nov 1996.
- [11] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In First Symposium on Operating Systems Design and Implementation (OSDI 94), Monterey, CA, November 1994. USENIX Association.
- [12] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. FetchBPF: Customizable prefetching policies in linux with eBPF. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 369–378, Santa Clara, CA, July 2024. USENIX Association.
- [13] PyTorch contributors. PyTorch torch.load. https://pytorch.org/docs/stable/generated/ torch.load.html.
- [14] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 49–63. USENIX Association, July 2020.
- [15] Jonathan Corbet. The extensible scheduler class. https://lwn.net/Articles/922405/.

- [16] Jonathan Corbet. Clarifying memory management with page folios, 2021. https://lwn.net/Articles/849538/.
- [17] Jonathan Corbet. The multi-generational LRU, 2021. https://lwn.net/Articles/851184/.
- [18] Jonathan Corbet. BPF for HID drivers, 2022. https://lwn.net/Articles/909109/.
- [19] Jonathan Corbet. Merging the multi-generational LRU, 2022. https://lwn.net/Articles/894859/.
- [20] Jonathan Corbet. Red-black trees for BPF programs, 2023. https://lwn.net/Articles/924128/.
- [21] Yifan Dai, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Symbiosis: The art of application and kernel cache cooperation. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 51–69, Santa Clara, CA, February 2024. USENIX Association.
- [22] Kernel development community. BPF_MAP_TYPE_HASH, with PERCPU and LRU variants. https://docs.kernel.org/bpf/map_hash.html.
- [23] Western Digital. Western digital PC SN8000S NVMe SSD, 2024. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/pc-sn8000s-nvme-ssd/data-sheet-pc-sn8000s-nvme-ssd.pdf.
- [24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 1–14, Renton, WA, July 2019. USENIX Association.
- [25] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24, page 249–264, New York, NY, USA, 2024. Association for Computing Machinery.
- [26] eBPF.io authors. eBPF. https://ebpf.io/.
- [27] Jake Edge. The FUSE BPF filesystem, 2023. https://lwn.net/Articles/937433/.
- [28] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM*

- SIGOPS Operating Systems Review, 29(5):251–266, 1995.
- [29] Facebook. Memory usage in RocksDB, 2024. https://github.com/facebook/rocksdb/wiki/ memory-usage-in-rocksdb.
- [30] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent block cache enabling High-Performance Out-of-Core processing with In-Memory programs. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 395–411, Carlsbad, CA, July 2022. USENIX Association.
- [31] Wi Fengguang, Xi Hongsheng, and Xu Chenfeng. On the design of a new Linux readahead framework. *SIGOPS Oper. Syst. Rev.*, 42(5):75–84, jul 2008.
- [32] Andrew Gallant. ripgrep is faster than {grep, ag, git grep, ucg, pt, sift}, 2016. https://blog.burntsushi.net/ripgrep/.
- [33] Google. LevelDB. https://github.com/google/ leveldb/.
- [34] The PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source relational database. https://www.postgresql.org/.
- [35] Tejun Heo. cgroup-v2. https://docs.kernel.org/admin-guide/cgroup-v2.html.
- [36] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.
- [37] Claire Huang, Stephen Blackburn, and Zixian Cai. Improving garbage collection observability with performance tracing. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, page 85–99, New York, NY, USA, 2023. Association for Computing Machinery.
- [38] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & flexible user-space delegation of Linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

- [39] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with eBPF, 2023.
- [40] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 31–42, New York, NY, USA, 2002. Association for Computing Machinery.
- [41] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery.
- [42] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.
- [43] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.
- [44] The kernel development community. bpf-ringbuf. https://www.kernel.org/doc/html/next/bpf/ringbuf.html.
- [45] Martin Lau. BPF extensible network, 2020. https://lpc.events/event/7/contributions/687/attachments/537/1262/BPF_network_tcp-cc-hdr-sk-stg_LPC_2020.pdf.
- [46] Martin KaFai Lau. struct-ops, 2020. https://lwn.net/ Articles/809092/.
- [47] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. P2Cache: An application-directed page cache for improving performance of data-intensive applications. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 31–36, New York, NY, USA, 2023. Association for Computing Machinery.
- [48] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys

- from values in SSD-conscious storage. *ACM Trans. Storage*, 13(1), mar 2017.
- [49] Linux man pages project. madvise linux manual page, 2024. https://man7.org/linux/man-pages/man2/madvise.2.html.
- [50] N. Megiddo and D.S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.
- [51] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. SIG-COMM Comput. Commun. Rev., 49(3):2–17, November 2019.
- [52] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High velocity kernel file systems with Bento. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 65–79. USENIX Association, February 2021.
- [53] Milvus. Milvus chunk cache. https://milvus.io/ docs/chunk_cache.md.
- [54] MongoDB. WiredTiger storage engine. https:// www.mongodb.com/docs/manual/core/wiredtiger/.
- [55] Konstantinos Mores, Stratos Psomadakis, and Georgios Goumas. ebpf-mm: Userspace-guided memory management in linux with ebpf, 2024.
- [56] MySQL. InnoDB Buffer Pool Optimization, 2024. https://dev.mysql.com/doc/refman/8.4/en/innodb-buffer-pool-optimization.html.
- [57] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993* ACM SIGMOD International Conference on Management of Data, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.
- [58] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and Andre Brinkmann. Combining buffered I/O and direct I/O in distributed file systems. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 17–33, Santa Clara, CA, February 2024. USENIX Association.
- [59] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. *SIGMETRICS Perform. Eval. Rev.*, 18(1):134–142, apr 1990.

- [60] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.
- [61] Dimitrios Skarlatos and Kaiyang Zhao. Towards programmable memory management with eBPF, 2024. https://lpc.events/event/18/contributions/1932/attachments/1646/3414/Towards%20Programmable%20Memory%20Management%20with%20eBPF%20(LPC%202024).pdf.
- [62] Christopher A Small and Margo I Seltzer. Vino: An integrated platform for operating system and database research. Harvard Computer Science Group Technical Report, 1994.
- [63] Solidigm. Solidigm d7-ps1030. https: //www.solidigm.com/products/data-center/d7/ ps1030.html#configurator.
- [64] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed Belady for content distribution network caching. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 529–544, Santa Clara, CA, February 2020. USENIX Association.
- [65] Meta Open Source. RocksDB. https://rocksdb.org/.
- [66] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, jul 1981.
- [67] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 373–386, Santa Clara, CA, February 2015. USENIX Association.
- [68] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Baleen: ML admission & prefetching for flash caches. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 347–371, Santa Clara, CA, February 2024. USENIX Association.
- [69] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 191–208. USENIX Association, November 2020.

- [70] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.
- [71] Ioannis Zarkadas, Tal Zussman, Jeremy Carin, Sheng Jiang, Yuhong Zhong, Jonas Pfefferle, Hubertus Franke, Junfeng Yang, Kostis Kaffes, Ryan Stutsman, and Asaf Cidon. BPF-oF: Storage function pushdown over the network, 2023.
- [72] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 1229–1246, Santa Clara, CA, April 2024. USENIX Association.
- [73] Da Zheng, Randal Burns, and Alexander S. Szalay. A parallel page cache: IOPS and caching for multicore systems. In 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 12), Boston, MA, June 2012. USENIX Association.
- [74] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 375–393, Carlsbad, CA, July 2022. USENIX Association.
- [75] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1391–1407, Boston, MA, April 2023. USENIX Association.
- [76] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel distributed transactions with eBPF. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 401–417, Santa Clara, CA, April 2024. USENIX Association.
- [77] Tal Zussman, Teng Jiang, and Asaf Cidon. Custom page fault handling with eBPF. In *Proceedings of the* ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions, eBPF '24, page 71–73, New York, NY, USA, 2024. Association for Computing Machinery.