FetchBPF: Customizable Prefetching Policies in Linux with eBPF

Xuechun Cao
University of British Columbia

Shaurya Patel
University of British Columbia

Soo Yee Lim
University of British Columbia

Xueyuan Han
Wake Forest University

Thomas Pasquier
University of British Columbia

Abstract

Monolithic operating systems are infamously complex. Linux in particular has a tendency to intermingle policy and mechanisms in a manner that hinders modularity. This is especially problematic when developers aim to finely optimize performance, since it is often the case that a default policy in Linux, while performing well on average, cannot achieve the optimal performance in all circumstances. However, developing and maintaining a bespoke kernel to satisfy the need of a specific application is usually an unrealistic endeavor due to the high software engineering cost. Therefore, we need a mechanism to easily customize kernel policies and its behavior. In this paper, we design a framework called FetchBPF that addresses this problem in the context of memory prefetching. FetchBPF extends the widely used eBPF framework to allow developers to easily express, develop, and deploy prefetching policies without modifying the kernel codebase. We implement various memory prefetching policies from the literature and demonstrate that our deployment model incurs negligible overhead as compared to the equivalent native kernel implementation.

1 Introduction

With a growing demand of memory capacity in memory-intensive applications, especially machine learning programs, researchers and practitioners alike have turned their attention to improving the memory management system [24, 36]. In particular, one focus area is to optimize *memory prefetching* [6, 9, 10, 17, 26, 28, 29, 31, 32, 35, 37, 38]. Despite the progress, existing solutions face the following challenges to be widely adopted.

First, a one-size-fits-all prefetcher that provides optimal performance for all applications does not exist. A new prefetching policy might excel in one memory access pattern but not another. For example, Leap [28] outperforms the built-in Linux prefetcher – which is optimized for sequential access patterns – when an application exhibits a large strided pattern; however, it lags behind Canvas [35] for workloads that involve chasing a large number of pointers. In §2, we provide empirical results to demonstrate this challenge more concretely.

An obvious, albeit naïve, way to address the first challenge is perhaps to have multiple prefetching policies in the kernel and choose from them the most appropriate one based on specific memory access patterns [26]. However, whether this approach would scale to an arbitrary number of policies is unknown. In fact, an optimal set of policies that would accommodate all possible access patterns likely does not exist. Meanwhile, implementing a new policy in the kernel is a Herculean task. Evidence from studies of other kernel development [21] suggests that the engineering effort would typically exceed the capacity of an average developer team. Even if the implementation was successful, further complications, such as maintaining a bespoke kernel distribution or convincing the mainline kernel to adopt the policy (e.g., in the Linux case), would require a tremendous amount of labor and time [22].

To ease development and maintenance, prior work [31, 35] has considered implementing prefetching policies in user space. On the one hand, user-space applications could better inform the prefetching mechanism of finer semantic hints; on the other hand, a lack of kernel access eventually complicates development [25]. This approach also introduces significant overhead, mainly from context switches between user space and the kernel. This could negate any performance gain, as recent work [35] has shown that prefetched pages are typically used by an application within a very short time frame.

We introduce FetchBPF, an alternative solution that leverages eBPF to overcome these challenges. eBPF is a Linux framework that allows users to load customized features to the kernel without modifying the kernel's codebase [3]. Building atop the eBPF framework and expanding its functionality, FetchBPF simplifies the development of new prefetching policies in the kernel with negligible performance overhead. To design FetchBPF, we study existing prefetching policies in the literature and identify fundamental kernel mechanisms that drive their implementations. Following these observations, we modify the kernel by adding new eBPF hooks to implement custom prefetching policies and additional eBPF helper

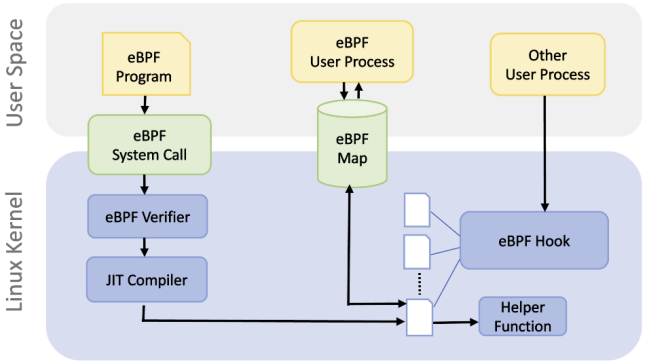


Figure 1: Workflow of the eBPF framework.

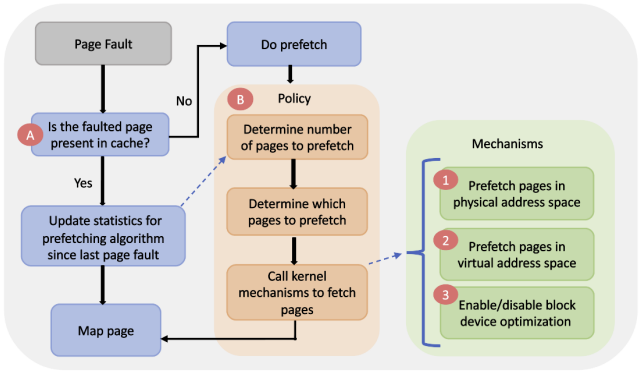


Figure 2: An overview of prefetching in Linux.

functions to evoke necessary kernel mechanisms.

Contributions

- We empirically show that applications with different memory access patterns benefit from different policies (§2).
- We propose an extension to the Linux eBPF framework to implement prefetching policies (§3).
- We implement (§4) and evaluate (§5) a number of policies from the literature to showcase FetchBPF.

2 Background & Motivation

We describe the eBPF framework in §2.1 and how prefetching works in Linux in §2.2. We then motivate FetchBPF by showing how the performances of different prefetching policies vary under different workloads in §2.3.

2.1 eBPF

The extended Berkeley Packet Filter (eBPF) is a Linux framework that allows users to extend the kernel without any modification to the kernel's source code. To ensure that the kernel extension is safe, all user-provided eBPF programs are statically checked by the eBPF verifier before they are allowed to execute. Once verified, eBPF programs are JIT-compiled into native machine code for performance. With safety and efficiency at the core of the eBPF design, eBPF is widely adopted for tracing [1] and networking [2, 5]. Furthermore, eBPF can also be used to customize and fine-tune the behavior of core kernel policies (e.g., scheduling [8] and auditing [27]).

As shown in Fig. 1, eBPF programs are event-driven, invoked when the hooks they are attached to are triggered. For example, an XDP program is a type of an eBPF program attached to the network interface card (NIC). When a packet arrives at the NIC, control is transferred to the eBPF program, where users can process the packet before it reaches the kernel. eBPF supports a variety of kernel hooks, e.g., system calls, tracepoints, and kernel probes. The type of an eBPF program dictates the set of hooks it can be attached to. It also determines the list of in-kernel helper functions it is allowed to call, as eBPF programs cannot call arbitrary kernel functions. eBPF programs use eBPF maps to persist data across programs and execution instances, and to share data with user space.

2.2 Linux Prefetching Policy

When memory is under stress, pages are swapped out. Retrieving those pages when they are needed again is expensive. To improve performance, an operating system (OS) tries to predict which swapped-out pages will be needed and prefetch them. We summarize at a high level Linux's prefetching policy and mechanisms in Fig. 2.

When a page fault is detected, the kernel first checks if the page resides in the page cache. A minor fault occurs when the page is present in the cache but not yet mapped. The kernel simply updates the page-fault statistics and maps the page. If the page is not found in the cache, then a remote request is issued to bring the page to memory. As remote accesses are expensive (e.g., when reading from the disk), in addition to loading the requested page, the kernel prefetches pages that it predicts will be used in the near future. To determine which pages to prefetch, the default Linux prefetching policy follows a sequential pattern [16]. Once the kernel has identified the pages to be prefetched, it issues remote access requests and maps the requested page.

2.3 Motivating Experiments

In Fig. 3, we show the execution time of six prefetching policies for six regimes described in §5.3. In §5, we dive into these results and discuss how a prefetching policy impacts the execution time. Each regime showcases a particular memory access pattern described in prior work [6, 11, 26, 28]. Given a regime, the best policy is the one that minimizes the execution time. We observe that no policy outperforms the others in all

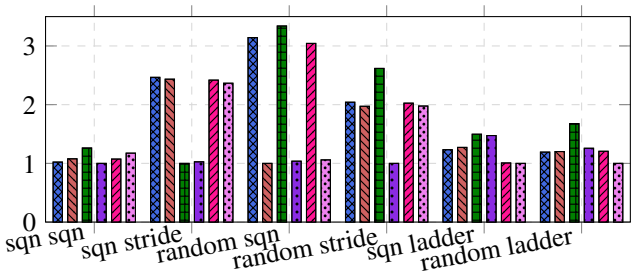


Figure 3: Execution time (lower is better) of `default`, `VMA`, `leap`, `leap VMA`, `ladder`, `ladder VMA`, under six different regimes. Values are normalized using the best performing policy on a particular regime.

regimes. For example, the `leap` policy is the best on the `sqn stride` regime but the worst on all other five regimes. On the `random sqn` regime, running the `leap` policy multiplies the execution time by more than three compared to the best performing policy. Clearly, deploying policies that best fit specific workloads can significantly improve performance.

3 Design & Implementation

FetchBPF is an extension to the existing eBPF framework; its design is guided by our analysis of existing prefetching policies (§3.1). Specifically, we modify the kernel to include two additional hook points where prefetching-related eBPF programs can be triggered (§3.2): Ⓐ when a page fault occurs and Ⓑ when the kernel decides on the specific pages to prefetch (see Fig. 2). We also include additional eBPF helper functions (§3.3); they enable developers to evoke kernel mechanisms related to page prefetching (①, ②, and ③ in Fig. 2).

3.1 Design Process

The first step in our design process is to understand existing prefetchers and identify common components and patterns in their workflows. This investigation informs us of fundamental building blocks, which guides us to build a flexible framework that supports the development and deployment of custom in-kernel prefetchers. We make the following observations of the common workflow features from existing prefetchers:

Ⓐ Prefetching policies [6, 10, 28] typically require as input a history of memory accesses, either physical or virtual. The exact type and computation of such information needed by a prefetcher is policy specific. Therefore, FetchBPF should allow developers to customize the capture and computation of the historical information through an eBPF program.

Ⓑ Prefetchers [6,10,26,28,35,37] run before a faulting page is requested; prefetching requests are then issued along with the page. FetchBPF should enable the prefetcher to call an eBPF program that implements custom prefetching logic, which typically uses the historical statistics computed from the eBPF program at Ⓐ.

①, ②, and ③ Prefetching policies issue I/O requests, but the specific I/O mechanism depends on the backend device, not the policy itself. For example, batching requests together is preferred when the backend is a disk-based device [6, 28] but not when an RDMA (remote direct memory access) backend is used [9, 28]. I/Os are functionalities provided by the kernel, which should be available to developers when they implement the policy.

3.2 eBPF Hooks

The Linux kernel executes an eBPF program when it encounters an eBPF *hook* that the program is attached to. Hooks are predefined in the kernel; they capture events such as system calls, function entries and exits, and network activities. We add two hooks in the kernel to support the expression of customized prefetching policies.

Ⓐ **`prefetch_stats`:** FetchBPF triggers an eBPF program attached to this hook every time a page fault occurs, including both major and minor faults. Policies such as Leap [28] use the history of faulting addresses to gain insight into memory access patterns. Therefore, this hook is typically used to record such data. Parameters passed to it include information about the physical and virtual address of the faulting page to support policies based on these addresses.

Ⓑ **`prefetch_policy`:** FetchBPF triggers an eBPF program attached to this hook before the execution of the default Linux prefetching policy. Like in Ⓐ, information about the physical and virtual address of the faulting page is passed as parameters. The eBPF programs attached to this hook are expected to (1) identify pages of interest and (2) request the kernel to prefetch them through helper functions (§3.3). They must return either an error code or one of the two values: `PREFETCH_RA_DEFAULT_PREFETCH` or `PREFETCH_RA_SKIP`. The former indicates that the default policy must be executed, whereas the latter skips the default policy execution. This enables developers to either completely replace the default policy or complement it (as done in Leap [28], for example).

3.3 Helper Functions

eBPF programs, unlike the built-in kernel code, drivers, or loadable kernel modules, have no access to the entire kernel address space and cannot call arbitrary kernel functions. Instead, they are limited to their program parameters and kernel-provided helper functions that perform actions on their behalf. FetchBPF implements the following helper functions:

① **`bpf_prefetch_physical_page`**: This helper function triggers the mechanism to prefetch pages based on physical addresses. It takes as input the parameters given to the eBPF program (§3.2). Specifically, the information about physical swap entry offsets is used to perform page I/O.

```c
SEC("prefetch_policy")
int policy_handler(struct swap_entry_info *ctx)
{
  unsigned int count = 0;

  // turn on block device plug
  bpf_start_block_plug(ctx->plug);

  while(count < 2){
    bpf_prefetch_physical_page(ctx);
    // increment the offset to the faulting page
    ctx->offset++;
    count++;
  }
  // turn off block device plug
  bpf_stop_block_plug(ctx->plug);
}
```

Figure 4: A simple policy fetching the faulting page and two subsequent physical pages.

② **`bpf_prefetch_virtual_page`**: This helper function triggers the mechanism to prefetch pages based on virtual addresses. It takes the same input as the previous helper function; however, the function prefetches pages based on `pte` (page table entries) and `pte` offsets.

③ **`bpf_<start/stop>_block_plug`**: These two helper functions enable a policy to control prefetching request batching. All requests made after `bpf_start_block_plug` are batched. When `bpf_stop_block_plug` is called, batched requests are processed (e.g., sorted and merged) to optimize I/O. The idea behind plugging is to build up requests to better utilize the hardware and to merge sequential requests into larger ones.

4 Policy Examples

We use a simple policy (Fig. 4) to illustrate how developers can implement a prefetching policy in FetchBPF. This policy loads the faulting page and two subsequent pages. In §5, we discuss five policies that we implemented in FetchBPF. Due to space constraints, we discuss the implementation of the `leap` and `leap VMA` policies here to illustrate the use of FetchBPF.

Leap. Leap [28] uses the Boyer-Moore algorithm [14] to find the majority stride pattern within the page fault history. We use an eBPF map (of the array type) to store the memory access history. Specifically, the `prefetch_stats` program populates the map with the address of a faulting page and its distance from the address of the previous faulting page. The `prefetch_policy` program uses this information to identify the majority stride, i.e., the most frequent distance between two faulting pages. It then calls the `bpf_prefetch_physical_page` helper function to prefetch the relevant pages.

Leap VMA (Virtual Memory Area). We modified the Leap algorithm to make prefetching decisions based on page table entries instead of physical addresses. VMA-based prefetching is shown to be more effective [26], since a program's access pattern is based on virtual rather than physical addresses. The `prefetch_stats` program now records page table entries and their relative distances. The `prefetch_policy` program is modified accordingly and calls the `bpf_prefetch_virtual_page` helper function.

5 Evaluation

We implement FetchBPF on the Linux kernel version 6.1.8. We run our expriments on an Intel i7-12700 [4] machine with 8 performance cores 2.10GHz-4.90GHz (16 threads), 4 efficient cores 1..60GHz-3.60 GHz (4 threads), 32GB DDR5 RAM, and an NVMe drive as the swap backend. We use `cgroups` to intentionally limit the amount of memory available to applications. Our evaluation focuses on answering the following two research questions:

Q1. Does implementing a prefetching policy using FetchBPF affect the policy's accuracy and coverage performance?

Q2. Is there performance degradation when using FetchBPF as compared to implementing a policy natively in the kernel?

5.1 Metrics

We use three metrics to quantitatively evaluate FetchBPF:

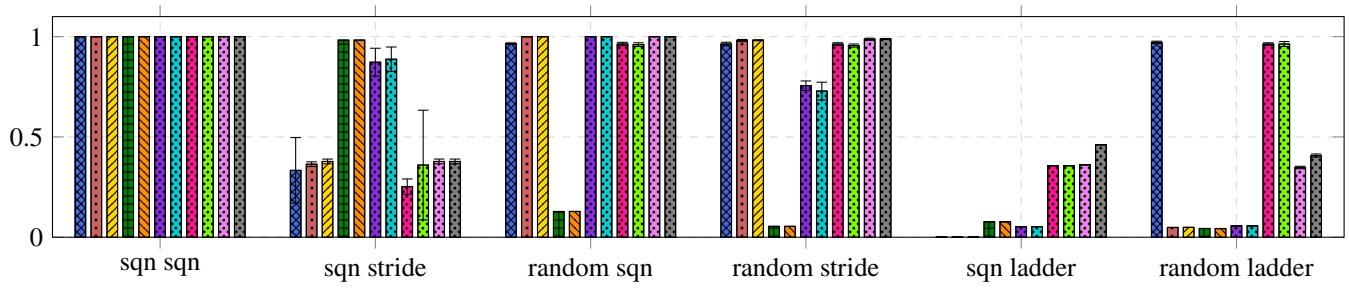
Accuracy, defined as the ratio of cache hits over the number of prefetched pages ($\frac{\text{cache hits}}{\text{\# of prefetched pages}}$), measures the number of prefetched pages that are indeed used by an application.

Coverage, defined as the ratio of cache hits over total page faults ($\frac{\text{cache hits}}{\text{cache hits+cache misses}}$), measures the extent to which the pages predicted by the prefetcher help reduce major faults.

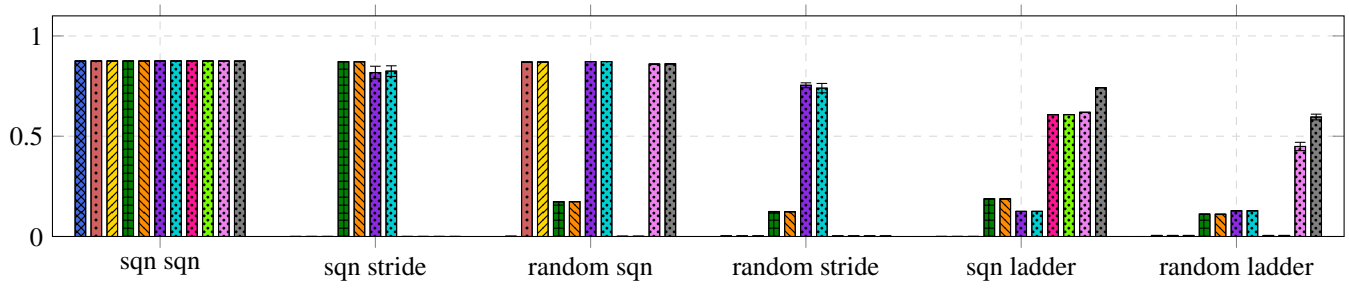
Execution time is the overall execution time of a benchmark.

5.2 Policies

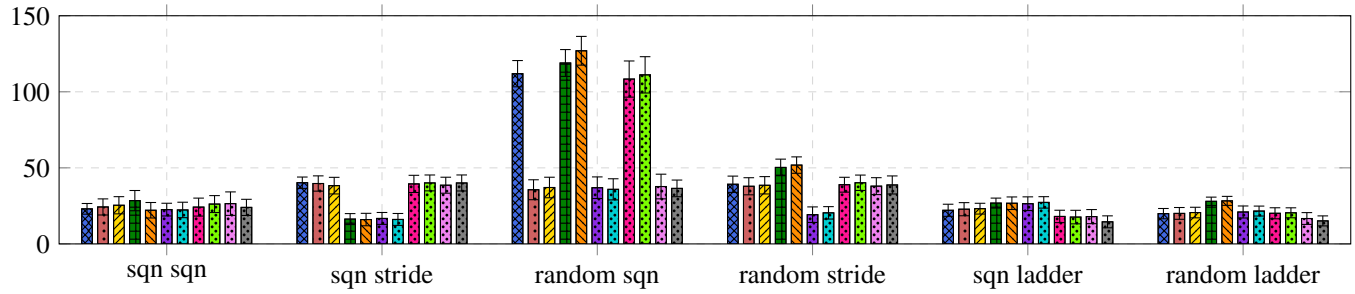
We implemented five prefetching policies in both FetchBPF and the Linux kernel. These policies are the `VMA` policy [6], the `leap` and `leap VMA` policies described in §4, and a ladder policy based on physical addresses (`ladder`) and VMA (`ladder VMA`) from prior work [26]. The `ladder` policy is tailored for applications where memory accesses initially follow a strided pattern for a short period of time but then jump to a different memory location followed by another strided pattern. Prior work [26] has shown that policies like `leap` do not perform well under this type of patterns.



(a) Accuracy (the higher the better)



(b) Coverage (the higher the better)



(c) Execution time in seconds (the lower the better)

Figure 5: Performance evaluation of `default`, `VMA`, `VMA eBPF`, `leap`, `leap eBPF`, `leap VMA`, `leap VMA eBPF`, `ladder`, `ladder eBPF`, `ladder VMA`, `ladder VMA eBPF`, under six different regimes. The first part of the regime label represents the allocation pattern and the second part the access pattern.

5.3 Experimental Results

Microbenchmarks

We design and run a suite of six microbenchmark programs to evaluate FetchBPF. Each program allocates a 5GB array but differs from each other in terms of its allocation and access pattern. We define below two allocation patterns and three access patterns; we call each unique combination of the two types of patterns a *regime*.

Allocation Patterns. (1) *Sequential allocation* (`sqn`), where we allocate array elements (pages) in order in physical memory, and (2) *random allocation* (`random`), where we allocate elements randomly; it simulates a scenario where multiple threads allocate pages simultaneously.

Access Patterns. (1) *Sequential* (`sqn`), where we access array elements in a sequential order, (2) *stride* (`stride`), where we access array elements on every third virtual page (i.e., a stride of three pages), and (3) *ladder* (`ladder`), where we access array elements with an increasing stride with every access. After accessing seven elements, we access an element that is three pages beyond the last accessed page. Then, we access another element that is 20 pages ahead of the page that we just accessed. We repeat this ladder-like access pattern.

Fig. 5 shows the accuracy (Fig. 5a), coverage (Fig. 5b), and execution time (Fig. 5c) of all six regimes. We limit DRAM allowance to 20% of working set size. As a quick sanity check, we note that a policy outperforms the other policies on the regime that it is designed for. For example, the `leap VMA` policy excels on the `random stride` regime, because (1) `VMA` is oblivious to physical memory allocation order but bases its prefetching decisions on virtual addresses, and (2) `leap` is designed to identify stride patterns of memory access. This matches our intuition and the observations in §2.3.

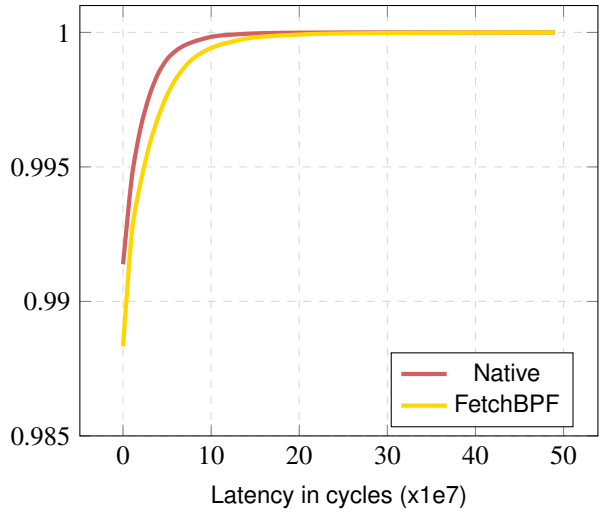


Figure 6: CDF of the latency of the `VMA` policy implemented in FetchBPF vs natively in the kernel.

More importantly, we see that the FetchBPF implementations of prefetching policies have the same accuracy and coverage as their in-kernel counterparts on all regimes, which again aligns with our expectation. For execution time, the difference between a FetchBPF and in-kernel implementation is statistically insignificant. This supports our hypothesis that eBPF is a good mechanism to implement prefetching policies.

Page-fault Latency. Fig. 6 shows the cumulative distribution function (CDF) of page-fault latencies of the `VMA` and `VMA eBPF` policies implemented in the kernel and with FetchBPF respectively. FetchBPF does not incur significant additional latency. This is not surprising, because (1) latency is dominated by page retrieval; and (2) eBPF programs are JIT compiled and executed as native machine code in the kernel [19], achieving performance similar to that of built-in kernel functions.

Macrobenchmarks

We evaluate the end-to-end performance on five benchmarks from three popular systems:

1. VoltDB running TPC-C (`voltdb`);
2. two variations of Redis' memtier benchmark, random (`redis-random`) and sequential (`redis-sequential`) key access;
3. GapBS's [13] PageRank (`gapbs-pr`) and betweeness centrality (`gapbs-bc`) algorithms on the Twitter dataset [23].

For each experiment, we limit DRAM allowance to 25% (Fig. 7a) and 50% (Fig. 7b) of the size of its working set. We compare the `VMA eBPF`, `leap eBPF`, and `leap VMA eBPF` policies implemented in FetchBPF against their native kernel implementations (`VMA`, `leap`, and `leap VMA`). The normalized performance results for each experiment are shown in Fig. 7 (and the raw data are reported in Appendix A). FetchBPF does not degrade application performance compared to the native kernel implementations. We note that the performance of different policies varies depending on the amount of memory available to them. More importantly, good prefetching decisions result in a more significant improvement on performance when the available resources become more scarce (Fig. 7a).

High-performance applications such as those in Fig. 7 are highly optimized for the Linux default prefetcher, which takes countless hours of engineering effort. FetchBPF represents a paradigm shift, where the policy instead gets optimized to improve the performance of the application. Recent advances [15, 20] indicate that this process could be automated.

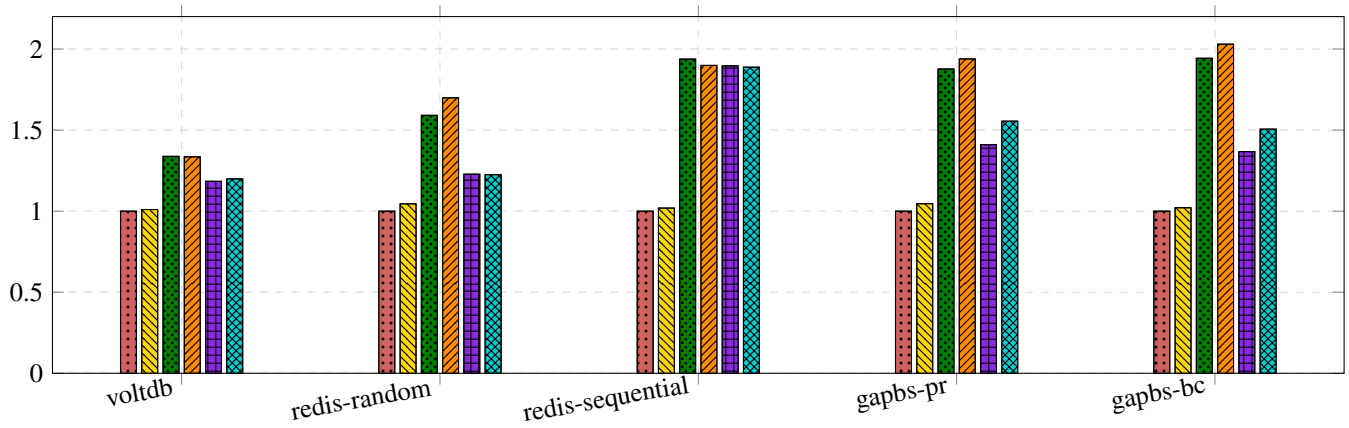
6 Related Work

eBPF and Custom Policies. eBPF is a popular framework to customize kernel behavior. For example, Lake [18] extends eBPF to enable developers to deploy custom policies to manage the use of hardware accelerators for machine learning. Google recently developed an eBPF version [7, 8] of Ghost [21] to customize the Linux scheduling policy. Prior work and ours share the same observations that the kernel's default policy does not fit all workloads, but supporting bespoke policies natively in the kernel is not sensible. eBPF is the best alternative to facilitate the expression of tailored policies.

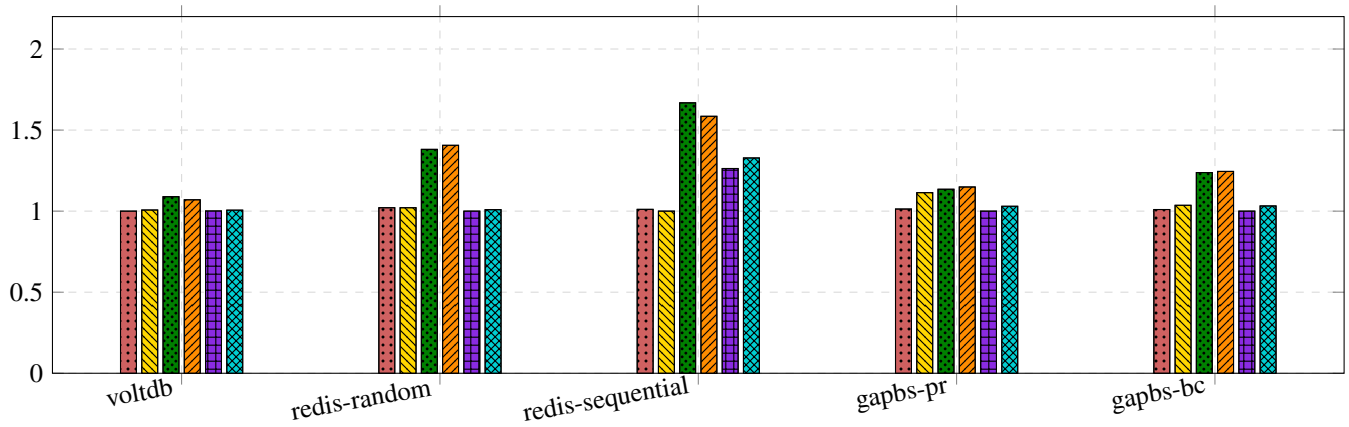
Like FetchBPF, P2Cache [25] also targets prefetching but offers far less flexibility. For example, it allows developers to customize only certain aspects of the default policy, while FetchBPF supports completely new policies. We were not able to evaluate FetchBPF against P2Cache, because, to the best of our knowledge, it is closed-source (at the time of writing) and the complexity of its API makes reimplementing P2Cache ourselves based purely on the publication difficult.

Other Types of Prefetchers. Past work [12, 26, 29] has explored hardware features to obtain more precise information about memory access patterns. FetchBPF could be extended to support such approaches, e.g., by adding dedicated helper functions to access hardware information. We could also extend FetchBPF to obtain user-space information to facilitate prefetching. Unlike prior work [35, 37] that involves solving complex engineering issues to access user-space information across kernel boundary, FetchBPF can simplify this process.

Data Path Optimization. We can optimize an application's performance by not only improving prefetching policies but also optimizing page I/O paths [9, 28, 35]. For example, Fastswap [9] optimizes page I/O by removing head-of-line blocking from critical paths. Leap [28] removes Linux's disk I/O optimization from its block I/O interface to make it suitable for faster access storage such as RDMA. Canvas [35]



(a) We limit DRAM allowance to **25%** of the size of the experiment working set.



(b) We limit DRAM allowance to **50%** of the size of the experiment working set.

Figure 7: Performance of `VMA`, `VMA eBPF`, `leap`, `leap eBPF`, `leap VMA`, `leap VMA eBPF` for five benchmarks. Values are normalized using the best performing policy under a particular workload (i.e., the lower the value, the better). Numerical results are available in Appendix A.

implements a two-tier scheduling policy for page I/O requests. While FetchBPF focuses on prefetching, a future extension could explore the possibilities of I/O data path customization.

7 Conclusion & Future Work

Customizing a prefetching policy that is tailored to an application's page access patterns can improve its performance. FetchBPF is an extension to the eBPF framework that enables users to easily deploy new prefetching policies without developing or maintaining a dedicated kernel. Our evaluation shows that FetchBPF accomplishes the same functionality but incurs no additional performance overhead when compared to a policy directly implemented in the kernel. In future work, we plan to achieve FetchBPF's full potentially by automatically generating application- and environment-specific policies through the use of application hints [15, 30, 33, 34].

Acknowledgments

We thank USENIX ATC 2024 reviewers for their feedback and their help in improving the paper. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien. This material is based upon work supported by the U.S. National Science Foundation under Grant CNS-2245442. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] bpftrace. Online (Accessed: 5th June 2024). https://github.com/iovisor/bpftrace.

[2] Cilium. Online (Accessed: 5th June 2024). https://github.com/cilium/cilium.

[3] ebpf. Online (Accessed: 5th June 2024). https://ebpf.io/.

[4] Intel Core i7-12700 Processor. Online (Accessed: 5th June 2024). https://ark.intel.com/content/www/us/en/ark/products/134591/intel-core-i7-12700-processor-25m-cache-up-to-4-90-ghz.html.

[5] Katran. Online (Accessed: 5th June 2024). https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[6] mm, swap: VMA based swap readahead, 2017. https://lwn.net/Articles/716296/.

[7] ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling, 2021. https://lwn.net/Articles/873244/.

[8] eBPF Kernel Scheduling with Ghost, 2022. https://lpc.events/event/16/contributions/1365/attachments/986/1912/lpc22-ebpf-kernel-scheduling-with-ghost.pdf.

[9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *European Conference on Computer Systems (EuroSys'16)*. ACM, 2020.

[10] Amro Awad, Sergey Blagodurov, and Yan Solihin. Write-Aware Management of NVM-based Memory Extensions. In *International Conference on Supercomputing (ICS'16)*. ACM, 2016.

[11] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying Memory Access Patterns for Prefetching. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, 2020.

[12] Reza Azimi, Livio Soares, Michael Stumm, Thomas Walsh, and Demke Angela Brown. Path: Page Access Tracking to Improve Memory Management. In *International Symposium on Memory Management (ISMM'07)*. ACM, 2007.

[13] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[14] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 1977.

[15] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 3PO: Programmed Far-Memory Prefetching for Oblivious Applications. *arXiv preprint arXiv:2207.07688*, 2022.

[16] Ali Butt R., Chris Gniady, and Y. Charlie Hu. The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. ACM, 2005.

[17] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul Gratz V, and A. L. Narasimha Reddy. Speculative Paging for Future NVM Storage. In *International Symposium on Memory Systems (MEMSYS'17)*. ACM, 2017.

[18] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher Rossbach J. Towards a Machine Learning-Assisted Kernel with LAKE. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. ACM, 2023.

[19] Mark Fleming. A thorough introduction to eBPF. Online (Accessed: 5th June 2024). https://lwn.net/Articles/740157/.

[20] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A Program-Behavior-Guided Far Memory System. In *Symposium on Operating Systems Principles (SOSP'23)*, pages 692–708. ACM, 2023.

[21] 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 *Symposium on Operating Systems Principles (SOSP'21)*. ACM, 2021.

[22] Yujuan Jiang, Bram Adams, and Daniel M German. Will my patch make it? and how fast? case study on the Linux kernel. In *Working conference on Mining Software Repositories (MSR'13)*. IEEE, 2013.

[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *International Conference on World Wide Web (WWW'10)*, pages 591–600. ACM, 2010.

[24] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, 2019.

[25] 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 *Workshop on Hot Topics in Storage and File Systems (HotStorage'23)*, pages 31–36. ACM, 2023.

[26] Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuam, Yinben Xia, Yungang Bao, Mingyu Chen, Shan Yizhou, Haifeng Li, and Ke Liu. HoPP: Hardware-Software Co-Designed Page Prefetching for Disaggregated Memory. In *International Symposium on High-Performance Computer Architecture (HPCA'23)*. IEEE, 2023.

[27] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. Secure Namespaced Kernel Audit for Containers. In *Symposium on Cloud Computing (SoCC '21)*. ACM, 2021.

[28] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *Annual Technical Conference (ATC'20)*. USENIX, 2020.

[29] Shaurya Patel, Sidharth Agrawal, Alexandra Fedorova, and Margo Seltzer. CHERI-Picking: Leveraging Capability Hardware for Prefetching. In *Workshop on Programming Languages and Operating Systems (PLOS'23)*. ACM, 2023.

[30] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Symposium on Operating Systems Principles (SOSP'95)*, pages 79–95. ACM, 1995.

[31] Zhenyuan Ruan, Malte Schwarzkopf, K. Marcos Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX, 2020.

[32] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *International Symposium on Microarchitecture (MICRO'15)*. IEEE/ACM, 2015.

[33] Andrew Tomkins, R Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):100–114, 1997.

[34] Steve VanDeBogart, Christopher Frost, and Eddie Kohler. Reducing seek overhead with application-directed prefetching. In *Annual Technical Conference (ATC'09)*. USENIX, 2009.

[35] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Harry Guoqing Xu. Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory. In *Symposium on Networked Systems Design and Implementation (NSDI'23)*. USENIX, 2023.

[36] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent Memory Offloading in Datacenters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. ACM, 2022.

[37] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. DiLOS: adding performance to paging-based memory disaggregation. In *Asia-Pacific Workshop on Systems (APSys'21)*. ACM, 2021.

[38] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *European Conference on Computer Systems (EuroSys'23)*, pages 266–282. ACM, 2023.

A End-to-end Evaluation Raw Data

Table 1 results are normalized in Fig. 7 by dividing the execution time of a policy by the minimum among all policies. We compute the values this way, so that the best performing policy normalizes to 1. This allows us to present in Fig. 7 consistent results with other benchmark experiments below. In all benchmarks, the closer a value is to 1, the better the result.

Table 2 and Table 3 results are normalized in Fig. 7 by dividing the maximum number of operations per second among all policies by the value of each policy.

Table 4 and Table 5 results are normalized in Fig. 7 by dividing the execution time of a policy by the minimum among all policies.

Policy	Execution Time for 25% DRAM Allowance (s)	Execution Time for 50% DRAM Allowance (s)
VMA	527.17	165.26
VMA eBPF	532.39	166.50
leap	705.51	180.06
leap eBPF	703.76	176.95
leap VMA	624.42	165.53
leap VMA eBPF	632.04	166.26

Table 1: Voltdb-tpcc benchmark results.

Policy	1,000 operations/sec for 25% DRAM Allowance	1,000 operations/sec for 50% DRAM Allowance
VMA	13,511	31,306
VMA eBPF	12,924	31,299
leap	8,492	23,147
leap eBPF	7,951	22,736
leap VMA	10,997	31,983
leap VMA eBPF	11,025	31,680

Table 2: Redis memtier random benchmark results.

Policy	1,000 operations/sec for 25% DRAM Allowance	1,000 operations/sec for 50% DRAM Allowance
VMA	13,089	22,399
VMA eBPF	12,838	22,648
leap	6,754	14,289
leap eBPF	6,892	13,576
leap VMA	6,898	17,049
leap VMA eBPF	6,924	17,920

Table 3: Redis memtier sequential benchmark results.

Policy	Execution Time for 25% DRAM Allowance (s)	Execution Time for 50% DRAM Allowance (s)
VMA	2,329	857
VMA eBPF	2,438	942
leap	4,373	960
leap eBPF	4,518	972
leap VMA	3,286	846
leap VMA eBPF	3,623	872

Table 4: GapBS PageRank benchmark results on the Twitter dataset.

Policy	Execution Time for 25% DRAM Allowance (s)	Execution Time for 50% DRAM Allowance (s)
VMA	1,783	323
VMA eBPF	1,821	331
leap	3,465	396
leap eBPF	3,620	398
leap VMA	2,438	320
leap VMA eBPF	2,686	330

Table 5: GapBS betweenness centrality benchmark results on the Twitter dataset.