# P2Cache: An Application-Directed Page Cache for Improving Performance of Data-Intensive Applications

Dusol Lee
Seoul National University

Inhyuk Choi
Seoul National University

Chanyoung Lee
Seoul National University

Sungjin Lee
DGIST

Jihong Kim
Seoul National University

## ABSTRACT

We propose P2Cache, an application-directed kernel-level page cache that allows an application developer to build a custom kernel-level page cache that matches the I/O characteristics of a target application. P2Cache extends a Linux kernel page cache by adding new probe points that are used to support application-programmable kernel page caches by eBPF programs. Our experimental results show that custom page caches implemented with our P2Cache achieve up to 32% performance improvement in data-intensive graph applications with little effort.

## CCS CONCEPTS

• **Software and its engineering** → *Software creation and management*.

## 1 INTRODUCTION

Modern data-intensive applications [1–6] require a large amount of data movements between an SSD and the DRAM memory of a host system. In order to efficiently manage such slow data transfers, most operating systems employ page caches in the host DRAM memory that exploits the reference locality of I/O accesses. Although OS-level page caches were widely used for improving the I/O performance of various applications, their performance improvements are often disappointing because the cache management policies of a page cache may not match well with the specific I/O characteristics of an application. For example, when an application repeatedly accesses a large range of I/O addresses in a looping pattern, the MRU (Most Recently Used) replacement policy may outperform the commonly used LRU (Least Recently Used) replacement policy. If an application accesses the I/O address space randomly, the standard read-ahead policy used by the page cache may prove ineffective.

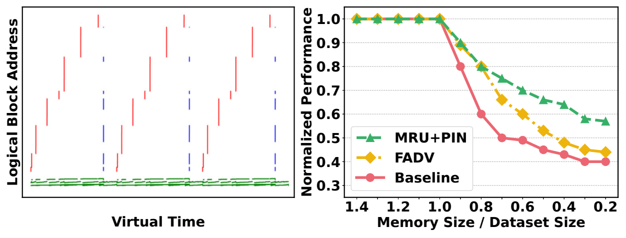
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*HotStorage '23, July 9, 2023, Boston, MA, USA*
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0224-2/23/07…$15.00
https://doi.org/10.1145/3599691.3603408

In order to mitigate the mismatch problem between the I/O characteristics (of an application) and the policies (of a kernel-level page cache), data-intensive applications often implement their own page caches at the application level [3–7]. However, supporting a user-level page cache at the application level presents several practical challenges.

If a user-level page cache is implemented directly without utilizing a kernel-level page cache (e.g., as in Jaydio [8] or RocksDB's Direct-IO [9]), it may not be able to take advantage of certain kernel-supported functions, such as those for ensuring data protection and data consistency. Moreover, if there are significant changes in either the SSD or the host memory system, a user-level cache would need to be re-implemented.

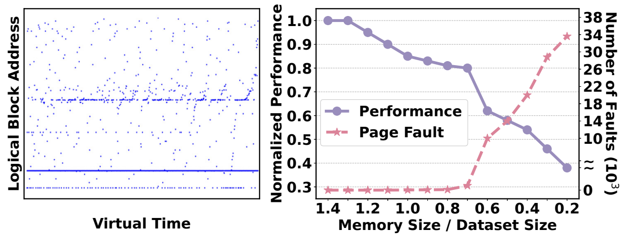
In this paper, we propose an *application-directed* kernel-level page cache, P2Cache (Programmable Page Cache), that allows an application developer to build a *custom kernel-level* page cache that matches the I/O characteristics of a target application. P2Cache aims to leverage the advantages of a kernel-level cache by enhancing the existing Linux page cache with four additional probe points. These probe points enable the integration of user-level customization through eBPF programs [10] with the Linux kernel-level page cache. Using a P2Cache-specific API, P2C API, application developers can create customized kernel-level page caches. At run time, when application-specific eBPF programs are loaded into a kernel-level page cache via the proposed probe points, the default Linux kernel-level page cache is re-configured to a custom page cache.

To assess the effectiveness of P2Cache, we implemented it on a Linux server with a high-performance 2-TB NVMe SSD running kernel version 5.8. We used two open-source data-intensive graph processing frameworks (Lumos [2] and GraphWalker [4]) as benchmarks. For each application, we implemented a custom page cache using the P2C API proposed in this work. Our experimental results show that



(a) I/O patterns (b) Performance variations

**Figure 1: I/O patterns and performance trends of Lumos with varying memory sizes.**



(a) I/O patterns (b) Performance variations

**Figure 2: I/O patterns and performance trends of GraphWalker with varying memory sizes.**

P2Cache can reduce the average overall execution time of the benchmark programs by 15% with just a few lines of C code (using the P2C API).

## 2 MOTIVATIONS

In this section, we present the limitations of existing caching strategies. We evaluate three common techniques, OS-level caching, hint-based OS-level caching, and user-level caching, under data-intensive applications with different I/O patterns.

### 2.1 Limitations of OS-level Page Cache

Modern operating systems commonly employ an OS-level page cache to reduce I/Os between disks and host memory. It employs an LRU replacement policy combined with read-ahead algorithms and generally achieves a high hit ratio if applications have moderate locality [11]. Since the caching service is performed in the OS kernel, it not only ensures robust data protection but also provides efficient data sharing across applications [12–14].

The OS-level page cache, however, often shows disappointing performance particularly under data-intensive applications. Existing data-intensive applications deal with huge amounts of data with highly customized algorithms, which results in complicated I/O patterns. Unfortunately, owing to its general-purpose design, the OS-level page cache often fails to capture unique behaviors of individual applications, thereby providing sub-optimal performance even if a higher hit ratio can be achieved. Figure 1(a) plots I/O reference pattern of graph application, Lumos [2], which executes graph processing algorithm - Pagerank [15]. Since it optimizes the graph processing engine using specialized data structures and optimization techniques, generated I/O access patterns are quite complicated. Lumos maintains multiple files and scans them simultaneously, sending mixed I/O patterns to the disk. Lumos also uses several metadata files and repeatedly reads them, resulting in highly localized I/O patterns.

We evaluate the performance of the application while decreasing its memory sizes (see Figure 1(b)). The performance is normalized to an optimal case where the memory size is large enough to fully fit the dataset size. After the dataset size exceeds the system memory, the applications start suffering from cache thrashing, providing serious performance drops (see Baseline in Figure 1(b)). This is a common phenomenon under memory-hungry situations, but the performance penalty is more severe than our expectation.

Our analysis found that the low performance is due to ineffective memory management of the OS-level page cache, which uses LRU and read-ahead policies without considering input workload patterns. The OS-level page cache prioritizes evicting the least-recently referenced pages, but these pages are actually referenced again soon, particularly under looping I/O patterns.

We have observed that using the MRU policy with small data-pinning, instead of LRU, results in higher performance, showing a 25% improvement as shown in MRU+PIN of Figure 1. However, it is difficult to change an in-kernel cache replacement policy adapting to input workloads.

As an alternative, some applications (e.g., GridGraph [1] or SQLite [7]) attempt to better manage cached data by providing application-level hints to the kernel via `fadvise` [16] and `madvise` [17]. While retaining the same advantages of the kernel-level cache management – robust data protection and efficient data sharing, it is able to achieve higher cache hit ratios by embedding important cache management decisions (e.g., WILLNEED, SEQUENTIAL, DONTNEED) in application codes. However, it also has drawbacks. First, it requires non-trivial effort in modifying existing application codes. Second, it is hard to fine control the kernel-level page cache just by injecting hints. We carefully added hint information in the code to manage the kernel cache in the MRU manner. The modified version exhibits higher hit ratios but still shows much slower performance than using the MRU (see FADV in Figure 1). This is because, as soon as the application's memory usage exceeds the system memory, cache thrashing begins to occur, leading to performance degradation.

## 2.2 Limitations of User-level Page Cache

Instead of relying on the kernel-level page cache and user-level hints, some data-intensive applications (e.g., GraphWalker [4], MyRocks [18]) employ their own caching algorithms, keeping and managing data in the user-level memory space. Based on detailed knowledge on application's behaviors, this approach is able to maximize cache hit ratios, thereby achieving excellent performance. However, it requires developers to create the caching algorithm from scratch, and it cannot utilize the kernel's protection and sharing features, making it more vulnerable to data protection issues and requiring extra effort to enable data sharing.

Another drawback of the user-level caching is that it suffers from interference by the kernel's page cache policy. Once the dataset size exceeds the system memory, the kernel tries to evict pages assigned to application's cache memory. It evicts pages based on LRU without respecting application's caching policy. As a result, it often demotes useful ones to the disk, interfering with application's caching mechanism.

To evaluate the impact of the kernel on application performance, we conducted experiments using Simrank [19], a graph application with its own user-level cache. As shown in Figure 2(a), I/O reference patterns of Simrank are mostly random because it caches popular data in the user-level cache. Thanks to such application-specific management, GraphWalker exhibits higher performance than Lumos for the same dataset (see Figure 2(b)). However, GraphWalker experiences a higher performance drop than Lumos when it runs out of memory. As mentioned earlier, this is owing to the kernel's intervention that evicts useful pages, violating the application's intention.

In summary, the kernel-level page cache fails to deliver high performance due to its inability to consider application-specific I/O patterns. Application-level hints mitigate the problem, but have limited effectiveness compared to the optimal. While the user-level custom cache works efficiently, it cannot leverage the kernel's infrastructure and suffers from performance drop by kernel intervention. To tackle the issues mentioned above, we propose P2Cache that enables developers to create a custom kernel-level page cache. P2Cache leverages eBPF to enable developers to control the detailed behaviors of the kernel's page cache with the application's high-level knowledge. Since the cache management is entirely performed inside the kernel, it allows us to benefit from the advantages of kernel-level caching and is not impacted by kernel's internal policy.

## 3 DESIGN OF P2CACHE

### 3.1 Overview

To support a custom kernel-level page cache with a user-defined page-cache management policy, it is necessary to safely execute the user policy code within the kernel. For this purpose, we use the eBPF framework [10] of Linux which guarantees the safe execution of eBPF programs inside the Linux kernel. To leverage the benefits of eBPF, the current page cache code must be expanded to accommodate new probe points that can be linked to user-level eBPF programs. To ease the overhead of kernel-level programming at the user level, a proper API support for customizing kernel-level page caches is important as well. In order to meet the above requirements, P2Cache 1) defines new probe points for page cache customizations inside the Linux kernel and 2) provides P2C API that simplifies the implementation of different cache management policies.

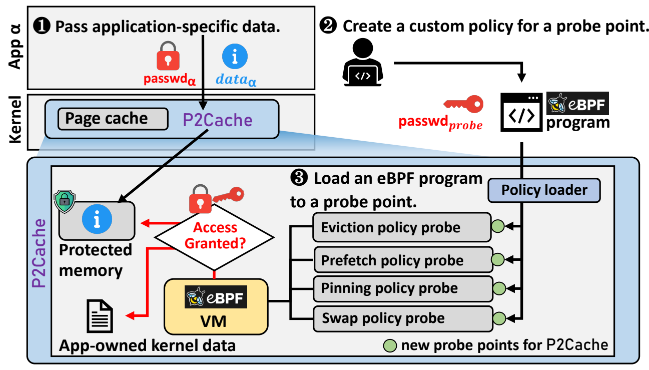


**Figure 3: An operational overview of P2Cache.**

Figure 3 shows an operational overview of P2Cache. To create a custom page cache for an application $\alpha$, application-specific data data$_\alpha$ of the app $\alpha$ may be required for developing a new cache policy. If they are needed, data$_\alpha$ is moved to the kernel's protected memory (❶). To avoid unauthorized accesses to data$_\alpha$ from other applications, a secret password (passwd$_\alpha$) is assigned for each custom page cache. Using P2C API functions along with the kernel data for the app $\alpha$, an eBPF program is implemented for each probe point of P2Cache (❷). After loading eBPF programs into their respective probe points (❸), a custom page cache for the app $\alpha$ becomes effective by executing the eBPF programs whenever the kernel's execution flow reaches those points. Before an eBPF program is executed, an extended eBPF VM verifies the program's authorization to access kernel data owned by the application $\alpha$ as well as application-specific data data$_\alpha$. This is achieved by comparing the passwd$_{probe}$ of the eBPF program with the passwd$_\alpha$ passed from the corresponding application.

### 3.2 Implementation Details

**Probe Points for Page Cache.** We integrated four new probe points into P2Cache, as depicted in Figure 3. These probe points were selected based on experimental validation results, which showed that their configurations significantly

**Table 1: A summary of functions in P2C API.**

| Group | Function name | Description |
|---|---|---|
| Configuration Management (CM) | move_data_to_kernel (`app_data, passwd_app`) | Transfer the `app_data` from user memory to kernel memory protected by the `passwd_app`. |
| | bpf_prog_load (`program, target_probe, passwd_probe`) | Load the eBPF `program` into the `target_probe` with the `passwd_probe` (used to access kernel data.) |
| | bpf_prog_unload (`program, target_probe, passwd_probe_ld`) | Unload the eBPF `program` attached to the `target_probe` if the `passwd_probe_ld` matches the current passwd of `passwd_probe`. |
| Policy Implementation (PI) | struct list_head `page_list` = get_page_list (`application_name`) | Return a `page_list` owned by the `application_name`. |
| | struct page/`file` = get_page_data (page) / get_file_data (page) | Return the page/file metadata of the page. |
| | char *`ptr` = get_app_data (`application_name`) | Get the pointer to the data transferred by the `application_name`. |
| | void iterate_page_list_lru/mru (`page_list, page`) | Iterate over the `page_list` of type page in the LRU/MRU order. |
| | void set_eviction_list (page) | Add the page to the eviction list with candidate eviction pages. |
| | void pin_page (page) | Lock the page into a page cache. |
| | void create_map (`table_name, key, value`) | Create a hash table named `table_name` that maps key to `value`. |
| | char *`ptr` = get_map_value (`table_name, key`) | Get the pointer to the value corresponding to the key in the `table_name`. |

impact the I/O performance of data-intensive applications. The eviction policy probe determines a page-cache eviction heuristic, while the prefetch policy probe defines a prefetch technique. Additionally, the pinning policy probe decides which pages to be pinned in a kernel page cache, and the swap policy probe is used to identify the pages that will be selected for swapping.

**P2C API.** The P2C API consists of two groups of functions, ones in the configuration management (CM) group and the others in the policy implementation (PI) group, as summarized in Table 1. The CM functions are used to move application data to the kernel memory. Furthermore, the loading/unloading of user-defined eBPF programs to/from the probe points are supported by the functions in the CM group.

```
  /* Application code */
1.  void main() {
     ...
2.   move_data_to_kernel (file_list_in_even_steps, passwd_app);
3.   move_data_to_kernel (file_list_metadata, passwd_app);
4.  }
```

```
  /* An eBPF Program for Eviction Policy Probe */
5.  void MRU_prioritize_even_steps(*ctx) {
6.    pg_list= get_page_list (“Lumos”);
7.    iterate_page_list_mru (pg_list, page) {
8.      if (page does not belongs to even iteration steps)
9.        set_eviction_list(page);
10.   }
11.  }
```

```
  /* An eBPF Program for Pinning Policy Probe */
12. void pinning_meta_files_only(*ctx) {
13.   pg = get_page_data(page);
14.   f = get_file_data(page);
15.   if (f is a meta file)
16.      pin_page(pg);
17. }
```

**Figure 4: An example implementation of a custom page cache for Lumos.**

The PI functions are used to access application data transferred by the CM functions or app-owned kernel data (such as a page list) for a custom page cache policy. Furthermore, the PI group functions support the use of a hash table for efficiently storing and retrieving information needed to implement a tailored page cache policy.

Figure 4 illustrates an example of a custom page cache implementation using Lumos as a target application. As depicted in Figure 1(b), Lumos frequently accesses small metadata files and conducts sequential scans on large data files in a repetitive manner. Notably, certain data files are exclusively accessed during even iteration steps. Hence, an efficient page cache for Lumos can be implemented by retaining the small metadata files in memory and adopting an MRU (Most Recently Used) policy that emphasizes the prioritization of data files accessed during even iteration steps. Initially, the list of data files accessed during even iteration steps and metadata files are transferred to kernel-protected memory (lines 2-3). At the pinning policy probe, pages associated with metadata files can be pinned in memory (lines 13-16). At the eviction policy probe, a customized MRU policy can be employed, which preferentially evicts pages belonging to data files that are not accessed during even iteration steps (lines 6-9).

**Per-App Kernel Access Isolation.** To prevent unauthorized eBPF programs from accessing other applications' data stored in the kernel, a simple authentication mechanism is employed in P2Cache. As shown in Figure 3, two passwords (such as passwd$_{\alpha}$ and passwd$_{probe}$) are used to ensure per-application kernel data isolation. When transferring application data to the kernel, a corresponding password (e.g., passwd$_{\alpha}$) is passed to P2Cache. Similarly, when an eBPF program is loaded onto its designated probe point, a matching password (e.g., passwd$_{probe}$) is passed as well. When the eBPF

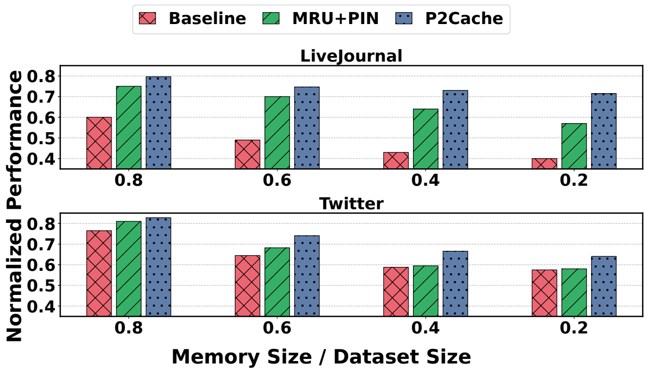


**Figure 5: Lumos performance comparisons among the default kernel page cache, simple optimization, and custom page cache.**

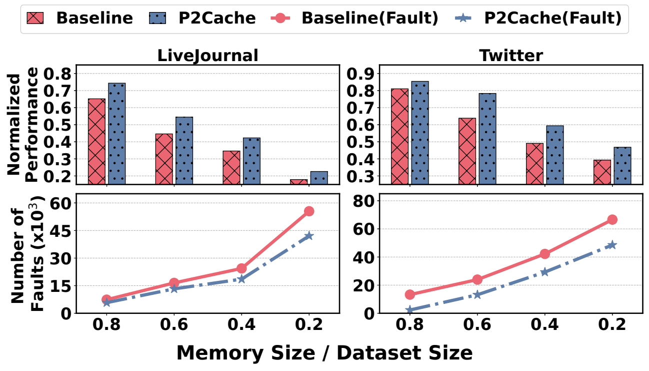


**Figure 6: GraphWalker performance comparisons between the default kernel page cache and custom page cache.**

program is executed within the eBPF VM, the password authentication mechanism, which is embodied in the PI functions of the P2C API, checks if two passwords match. Only when they match, the PI functions are executed.

## 4 EXPERIMENTAL RESULTS

### 4.1 Experimental Setup

In order to evaluate the effectiveness of P2Cache, we implemented P2Cache on a Linux server (kernel version 5.8) with a high-performance 2-TB NVMe SSD. Two open-source out-of-core graph processing frameworks, Lumos [2] and GraphWalker [4] were used as benchmark data-intensive applications. Lumos was evaluated using Pagerank [15] algorithm, and GraphWalker was evaluated using Simrank [19] algorithm. We employed two real-world graph datasets (LiveJournal [20] and Twitter [21]) to assess the effectiveness of custom page cache policies. By utilizing Linux's cgroups [22], we carried out evaluations by varying the ratios between available system memory size and dataset size.

**Table 2: A summary of custom page cache implementations for Lumos and GraphWalker.**

| Probe | Lumos | GraphWalker |
|---|---|---|
| Eviction Policy | Evict pages by MRU order, prioritizing those from data files not accessed in even iteration steps. | No customization. |
| Prefetch Policy | Adjust the read-ahead sizes, considering memory size to avoid evicting soon-to-be-used pages. | No customization. |
| Pinning Policy | Pin small and frequently used metadata files. | No customization. |
| Swap Policy | No customization. | Swap user-level pages belonging to subgraph with small walk counts. |

### 4.2 Evaluations

We implemented custom page caches for two applications using the proposed P2C API. Table 2 provides a summary of the custom page cache implementations for two applications.

**Lumos Optimizations.** Figure 5 compares the performance efficiency of a custom page cache over the default kernel page cache (Baseline) and its optimized version with user-level hints (MRU+PIN) while varying available memory sizes. All values were normalized to when sufficient memory was allocated for each dataset. The performance enhancement of Lumos, utilizing a custom page cache implemented with P2Cache, results in up to a 32% improvement compared to Baseline. Additionally, when compared to MRU+PIN, P2Cache consistently exhibits performance gains of up to 15%. Among three policy optimizations applied for Lumos, the customized MRU eviction policy, which differentiates the importance of accessed files for each iteration, has a distinction from the basic MRU policy.

**GraphWalker Optimizations.** Figure 6 compares the performance efficiency of a custom page cache over the default kernel page cache (Baseline) while varying available memory sizes. The performance in Figure 6 were normalized to when sufficient memory was allocated for each dataset. The performance enhancement of GraphWalker, utilizing a custom page cache implemented with P2Cache, results in up to a 14% improvement compared to Baseline even with a swap policy customization only. Under the default Baseline page cache, GraphWalker experiences a significant performance degradation from a large number of page faults as memory becomes scarce, as useful pages are frequently evicted by the kernel's existing swap mechanism. The bottom graphs in Figure 6 show that the number of page faults increases by up to 3.2 times when the available memory size is reduced by 1/3.

To enhance GraphWalker's performance using P2Cache, we adopted an optimization strategy from a user-level page caching technique [4] employed in GraphWalker. In the original GraphWalker's user-level page cache implementation, cache blocks with the lowest walk counts are evicted first, while walk counts are adjusted by the application's own optimization algorithm. To replicate this approach, we transferred the walk-num values for each block to the kernel after every iteration step and implemented a custom eBPF program for the swap policy probe, which adheres to the walk-conscious caching scheme [4] of GraphWalker.

## 5 CONCLUSIONS

We have presented P2Cache, an application-directed kernel-level page cache for Linux-based systems. P2Cache enables application developers to create custom kernel page caches that can better support applications' I/O workloads. By extending the existing kernel page cache with the introduction of four new probe points, P2Cache leverages all the benefits of the existing kernel page cache. To facilitate easy programmability of P2Cache, we developed the P2C API, which simplifies the process of developing a safe custom kernel-level page cache. Our evaluation results demonstrate that P2Cache can be an effective solution for improving the I/O performance of data-intensive applications such as graph processing applications.

## ACKNOWLEDGMENTS

This work was supported by SNU-SK Hynix Solution Research Center (S3RC) and by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2022R1A2C201052912 and NRF-2018R1A5A1060031). The ICT at Seoul National University provided research facilities for this study. (Corresponding author: Jihong Kim.)

## REFERENCES

[1] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.
[2] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *Proceeindgs of the USENIX Annual Technical Conference (ATC)*, 2019.
[3] Zhiyuan Shao, Jian He, Huiming Lv, and Hai Jin. FOG: A Fast Out-of-Core Graph Processing Framework. *International Journal of Parallel Programming*, 45(6):1259–1272, 2017.
[4] Rui Wang, Yongkun Li, Yinlong Xu Hong Xie and, and John C. S. Lui. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.
[5] MyRocks. MyRocks | A RocksDB Storage Engine with MySQL. http://myrocks.io/, 2020.
[6] PostgreSQL. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/, 2023.
[7] SQLite. About SQLite. https://sqlite.org/index.html, 2023.
[8] Stephen Macke - GitHub. smacke/jaydio: A Java Library to Perform Direct I/O in Linux, Bypassing File Page Cache. https://github.com/smacke/jaydio, 2021.
[9] EighteenZi - GitHub. Direct-IO.md. https://github.com/EighteenZi/rocksdb_wiki/blob/master/Direct-IO.md, 2017.
[10] eBPF. eBPF Documentation. https://ebpf.io/what-is-ebpf/, 2022.
[11] Viacheslav Fedorov, Jinchun Kim, Mian Qin, Paul V Gratz, and AL Narasimha Reddy. Speculative Paging for Future NVM Storage. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2017.
[12] The kernel development community. Memory Management. https://www.kernel.org/doc/html/latest/admin-guide/mm/index.html, 2023.
[13] Jonathan Corbet. The future of the page cache. https://lwn.net/Articles/712467/, 2017.
[14] Hao Luo, Pengcheng Li, and Chen Ding. Thread Data Sharing in Cache: Theory and Measurement. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.
[15] Wikipedia. PageRank. https://en.wikipedia.org/wiki/PageRank, 2023.
[16] Linux. fadvise(1) — Linux manual page. https://man7.org/linux/man-pages/man1/fadvise.1.html, 2022.
[17] Linux. madvise(2) — Linux manual page. https://man7.org/linux/man-pages/man2/madvise.2.html, 2021.
[18] RocksDB. RocksDB | A persistent key-value store. https://rocksdb.org/, 2022.
[19] Glen Jeh and Jennifer Widom. SimRank: A Measure of Structural-Context Similarity. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2002.
[20] Stanford University. LiveJournal social network. https://snap.stanford.edu/data/soc-LiveJournal1.html, 2006.
[21] Stanford University. Twitter follower network. https://snap.stanford.edu/data/twitter-2010.html, 2010.
[22] Linux. cgroups - Linux control groups. https://man7.org/linux/man-pages/man7/cgroups.7.html, 2021.