CrossPrefetch: Accelerating I/O Prefetching for Modern Storage

Shaleen Garg[*]
Rutgers University
USA

Jian Zhang[*]
Rutgers University
USA

Rekha Pitchumani
Samsung
USA

Manish Parashar
University of Utah
USA

Bing Xie
Microsoft
USA

Sudarsun Kannan
Rutgers University
USA

Abstract

We introduce **CrossPrefetch,** a novel *cross-layered* I/O prefetching mechanism that operates across the OS and a user-level runtime to achieve optimal performance. Existing OS prefetching mechanisms suffer from rigid interfaces that do not provide information to applications on the prefetch effectiveness, suffer from high concurrency bottlenecks, and are inefficient in utilizing available system memory. CrossPrefetch addresses these limitations by dividing responsibilities between the OS and runtime, minimizing overhead, and achieving low cache misses, lock contentions, and higher I/O performance.

CrossPrefetch tackles the limitations of rigid OS prefetching interfaces by maintaining and exporting cache state and prefetch effectiveness to user-level runtimes. It also addresses scalability and concurrency bottlenecks by distinguishing between regular I/O and prefetch operations paths and introduces fine-grained prefetch indexing for shared files. Finally, CrossPrefetch designs low-interference access pattern prediction combined with support for adaptive and aggressive techniques to exploit memory capacity and storage bandwidth. Our evaluation of CrossPrefetch, encompassing microbenchmarks, macrobenchmarks, and real-world workloads, illustrates performance gains of up to 1.22x-3.7x in I/O throughput. We also evaluate CrossPrefetch across different file systems and local and remote storage configurations.

[*]The authors contributed equally to this paper.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/3617232.3624872

ACM Reference Format:
Shaleen Garg, Jian Zhang, Rekha Pitchumani, Manish Parashar, Bing Xie, and Sudarsun Kannan. 2024. CrossPrefetch: Accelerating I/O Prefetching for Modern Storage. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617232.3624872

1 Introduction

Despite the introduction of ultra-fast nonvolatile block storage technologies (e.g., NVMe), the performance gap between compute and storage devices remains considerable. Consequently, main memory caching and buffering are widely employed in OS, user-level file systems [32], and I/O runtimes, and continue to play a critical role to hide the performance gap and reduce I/O bottlenecks [23, 26, 30, 37]. Although varying in techniques and strategies, the existing OS caching and prefetching designs are all application-transparent [21, 24, 31, 33]. A well-designed memory caching design seeks to optimize the overlap between computation and I/O. The effectiveness of caching is heavily dependent on the effectiveness of the underlying I/O prefetching mechanisms, which predict upcoming I/O accesses and load the corresponding I/O operations in a timely manner. In OSes such as Linux, prefetching is accomplished through a generic **readahead** component [19] implemented by the virtual file system (VFS) layer. As prior studies [6, 12, 14, 20, 27] and our analysis demonstrate that, a good prefetching mechanism can reduce I/O overhead by 2×-3.7× even for well-tuned production-class applications (e.g., key-value stores).

Unfortunately, state-of-the-art prefetching designs often fall short of delivering higher performance gains from I/O prefetching and fail to fully exploit the available bandwidth in modern fast storage devices like SSDs. These designs include application-customized prefetching approaches [1], OS-level I/O prediction schemes [20, 27], machine learning (ML)-based strategies [8, 14], and compiler-directed file layout optimizers [11], which primarily focus on advancing prediction accuracy but neglect cross-layered coordination and prefetching effectiveness (detailed in §2). The major shortcomings of current designs can be summarized into the following facets.

First, the effectiveness of current OS prefetchers is limited due to their rigid interfaces and conservative policies, which makes it difficult to efficiently leverage cache state (i.e., pages in cache). System calls such as `readahead`, `fadvise`, and `madvise` do not provide visibility to the information such as total bytes prefetched, leading to the possibility of applications either under-prefetching or over-prefetching. Consequently, due to the lack of visibility, several applications (e.g., RocksDB) implement custom prefetching logic using prefetching system calls. However, as a response, the OSes may under- or over-prefetch, with no guarantee of the bytes actually prefetched.

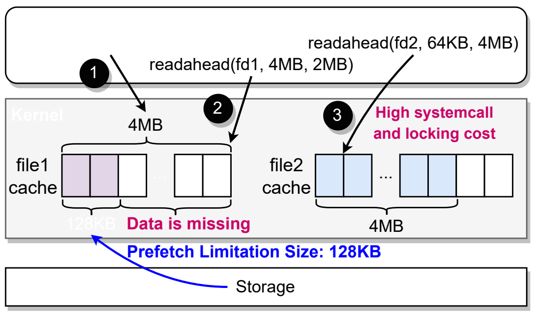
Figure 1 presents a simple example of the OS under-prefetching for readahead, where *request 1* results in an incorrect offset for *request 2*, leading to cache misses. Similarly, over-prefetching results in higher prefetching system calls and interference with blocking I/O (e.g., read) operations, negatively impacting application performance. Although OS support is available to query the cache state of a file through system calls such as **fincore** [18], as we discuss in §3 and §2.1, these system calls incur significant performance overhead as well as applications changes.

Secondly, the current prefetching operations face a significant concurrency bottleneck. This is mainly because prefetching and regular I/O operations (e.g., `read`) use the same data structures (like Xarray [17]) for caching and prefetching, causing contention for locks. Additionally, when multiple threads share a file, they contend for file-level locks such as inode `rw-lock` and redundantly issue prefetch operations due to the lack of cache awareness.

Finally, existing OSes, as well as state-of-the-art OS techniques [20, 27, 35], and application-specific approaches [8, 14], do not effectively coordinate the cache state between the OS and applications. This leads to missed opportunities to fully utilize available memory capacity and I/O bandwidth resources, ultimately resulting in poor application performance.

To overcome the aforementioned challenges, we propose **CrossPrefetch**, I/O prefetching system that accelerates I/O without amplifying memory requirements. CrossPrefetch is designed to be cross-layered, application-transparent, and scalable, with three major innovations.

Firstly, CrossPrefetch **disaggregates and distributes tasks** between user space and the OS to improve prefetching precision, minimize unnecessary I/O operations, and address cache state visibility challenges. In this cross-layered approach, the OS component (Cross-OS) maintains a per-inode bitmap alongside the OS's per-file cache tree to assess prefetching effectiveness. It conveys this information to the user-level runtime (Cross-Lib) through a new multi-purpose `readahead_info` system call, used for performing readahead operations, exporting OS-level cache bitmap state of a file,



readahead(fd2, 64KB, 4MB)
1
readahead(fd1, 4MB, 2MB)
Kernel
4MB
2
3
High systemcall and locking cost
file1 cache
file2 cache
128KB
Data is missing
4MB
Prefetch Limitation Size: 128KB
Storage

Figure 1. I/O Prefetching Pathology: The figure illustrates scenarios demonstrating the impact of a lack of cache awareness. ❶ The application issues a *readahead* request from offset 0MB to 4MB. However, due to static limits, the OS only prefetches 128KB. ❷ The application, assuming full prefetching, issues subsequent requests starting at 4MB, but accesses between 128KB and 4MB incur cache misses. ❸ The application requests prefetching between 64KB and 4MB, already cached, resulting in unnecessary system calls and locking costs due to cache unawareness.

and exporting OS-level telemetry. By leveraging this information, Cross-Lib optimizes and reduces prefetch system calls, improves application thread-level prefetching for private and shared files, and enables prefetch customizations bypassing complex OS cache layer modifications.

Secondly, to enhance scalability and mitigate concurrency bottlenecks, CrossPrefetch segregates the I/O path for regular I/O and prefetch operations. This facilitates fast cache state lookup through `readahead_info` system call. To accommodate thread sharing and non-conflicting access to file regions, CrossPrefetch employs fine-grained indexing via range trees. This empowers threads to query their cache status and initiate informed prefetching requests simultaneously. Augmented with lightweight access pattern prediction, Cross-Lib adjusts prefetch requests according to per-inode cache states, thereby reducing the need for excessive prefetch system calls.

Thirdly, to maximize the use of available memory resources, CrossPrefetch implements an aggressive prefetching/eviction policy based on the available free memory (budget) approach. With per-file cache awareness, CrossPrefetch switches between aggressive prefetching and eviction, depending on the memory budget available. This adaptive approach improves overall performance. *Finally,* Cross-OS optimizes I/O paths by allowing for larger prefetch requests and dynamic adjustment of prefetching limits based on available memory budget, resulting in better system performance.

Evaluation: We implement CrossPrefetch in Linux OS, and a user-level runtime. CrossPrefetch shows significant performance gains over application-controlled and OS-delegated prefetching. On NVMe SSDs with ext4, we observe up to 1.97x gains in microbenchmarks and 2.1x gains in RocksDB (a key-value store [1]), while reducing cache misses and lock contentions. Under low memory, for the Snappy application, CrossPrefetch shows up to 1.22x gains. Finally,

CrossPrefetch on RDMA-based remote storage and SSD-optimized F2FS file system shows high-performance benefits.
Contributions: To summarize,

1. I/O prefetching awareness: We propose a novel cross-layered design to accelerate I/O prefetching for modern storage without requiring application modifications.

2. Improved concurrency: We introduce techniques to reduce concurrency bottlenecks and provide fine-grained prefetching support for threads sharing files.

3. Low-interference prefetching and fine-grained prediction: Utilizing the cross-layered design, we create a low-interference and fine-grained prefetching approach that combines access pattern prediction with cache-state awareness.

4. Memory-efficient aggressive prefetching and eviction: We develop memory budget-efficient aggressive prefetching and eviction methods to enhance I/O acceleration.

2 Background and Related Work

We present a background on OS I/O prefetching followed by state-of-the-art I/O prefetching techniques, their capabilities, and limitations.

2.1 OS I/O Prefetching

OS-level I/O prefetching, also known as pre-paging or readahead, is a common technique used to accelerate I/O for both slow and fast storage devices. In Linux, incremental prefetching is used, where the prefetcher (the `readahead` component in the VFS) reads a portion of the file into memory in anticipation of a process/thread reading or writing to that portion, which reduces I/O wait time. To predict when to prefetch, Linux maintains a per-file `PG_readahead` marker on pages at the edge of the populated cache and incrementally prefetches (up to 128KB) when the marked page is accessed or shrinks for random accesses based on a file's hit rate.
I/O Prefetching System Calls: OSes like Linux provide system calls and data access pattern hints to override the OS's access pattern identification to enable applications and runtimes to control prefetching. For example, applications like RocksDB use system calls such as `readahead()` with offset and bytes to prefetch from a file or access pattern hints like `fadvice()`. The `POSIX_FADV_NORMAL` hint lets the OS determine I/O patterns, while `POSIX_FADV_SEQUENTIAL` hints that a file will be accessed sequentially, and the OS can increase prefetch window size. The `POSIX_FADV_RANDOM` hint informs the OS of a random access file and turns off prefetching, and `POSIX_FADV_WILLNEED` is analogous to the `readahead` syscall to populate the cache quickly. Finally, `POSIX_FADV_DONTNEED` hints to the OS to remove cache pages of a file not needed in the future.

However, despite these different interfaces and hints, applications are often unaware of the effectiveness of explicit `readahead` or advice calls, resulting in high cache misses, software overheads, and performance impact.

Understanding the Cache State using Fincore: Linux system calls, `fincore` [18] and `mincore`, enable applications to determine which pages of a file or an address range that are currently resident in the memory. These calls work by locking the entire memory address space and walking the cache tree and a process address space to build this information. Therefore, because they are expensive, frequent use of these calls can result in high overhead and performance degradation, as we will show shortly.

2.2 Related Work

We next provide an overview of the state-of-the-art OS, application, ML, and compiler prefetching techniques for improving I/O performance.
OS-level Prefetching: Several techniques have been proposed to improve I/O prefetching beyond the prefetching support in the Linux OS. Lynx [27] proposes a learning-based SSD prefetching mechanism that captures random access patterns using Markov chains. However, Lynx only works for memory-mapped files and turns off prediction when page cache misses increase. ATS [20] uses partitioned context modeling (PCM) in the OS cache layer to exploit disk data layout for prefetching. In contrast, FastMap [35] and MMap on Steroids [34] improve `mmap` performance, but at the cost of disabling caching. Leap [6] detects remote file-access patterns based on page faults on the client and prefetches them using current Linux ABIs. However, all of these techniques fail to address the mismatch between application requests and OS prefetching because of a lack of awareness across these layers.
Application-specific Prefetching: Several application-level prefetching techniques have been proposed. For example, VSS, a storage system for video analytics [15], reduces cache usage by down-sampling or deleting low-frequency and high-frequency prefetching videos. Similarly, HTTP-based streaming video servers aggressively prefetch sequential video files [7]. Further, Dong et al. [12] propose a correlation-based prefetching for Hadoop files drawing correlations from I/O patterns to replication/prefetch files from data nodes. Frog [38] is a context-based file system that adjusts I/O settings for different application contexts. Finally, I/O prefetching has been proposed for reducing the load time of mobile applications [22].
ML Techniques: In recent years, machine learning (ML) techniques have emerged as a promising approach to improve prefetching prediction. Akgun et al.[14] propose a kernel-level ML framework that enhances storage system performance by identifying and increasing or decreasing prefetching through user-defined training functions. Clairvoyant, an OS ML design [14] for distributed ML, observes that ML applications are trained in batches, and the batch numbers generated using a pseudo-random number generator can be used to predict and prefetch data used for the next timestamp. Similarly, ML techniques have been proposed at

the SSD device layer, with Chakrabortti et al.[8] proposing to learn access patterns and improve prefetching. Additionally, Stacker [36] designs an autonomic data movement engine for high-performance computing (HPC) in-situ workflows, which uses the N-gram model to learn strided access patterns for prefetching.

Compiler Techniques: Prior research has investigated static compiler-directed file layout optimization and prefetching for hierarchical storage in HPC systems. Specifically, prior work using the IBM Blue Gene compiler minimizes disk block reads by using a polyhedral model for n-dimensional data objects [11]. While compiler-optimized prefetching is beneficial, a lack of coordination with OS prefetching prevents exploiting maximum benefits from prefetching.

To the best of our knowledge, no prior work has focused on the lack of cross-layered prefetching awareness and their resulting implications on I/O performance.

3 Motivation and Analysis

Regardless of whether prefetching is application-centric or entirely delegated to the OS, current techniques face three key problems. Firstly, they lack awareness of the prefetching state. Secondly, they encounter scalability and concurrency bottlenecks. Lastly, they fail to utilize memory efficiently.

3.1 Lack of Prefetching Progress Awareness

A significant challenge in existing application-level and OS-level prefetching designs is the lack of synergy and awareness between application I/O needs and OS prefetching effectiveness. Applications often issue prefetch requests without certainty about page prefetching status, leading to imprecise results. For instance, during sequential access to large files, applications may issue large prefetch requests (e.g., using `fadvise(SEQUENTIAL)`) to overlap computation and I/O. However, OSes like Linux limit initial prefetch size to 128KB, even without memory pressure. Consequently, applications assume successful prefetching of the entire request and experience diskI/O for accesses beyond the limit. Naively increasing prefetch thresholds could hurt performance under memory pressure or high I/O loads.

Additionally, some applications disable prefetching (using `fadvise(...FADV_RANDOM..)`) for perceived random access patterns, which can degrade performance. For instance, RocksDB [1] proactively deactivates prefetching for workloads involving random access, mistrusting the OS's capacity to identify such patterns. This introduces two significant concerns: Firstly, a random read or an update often generates additional I/O requests for activities like search, update, or compaction, which exhibit non-random behavior and could benefit from effective caching. Secondly, when multiple threads share database files, they share the cache, and cache misses could be reduced even for random access. On the other hand, disabling caching prevents threads from capitalizing on the shared cache. Lastly, the Linux OS prefetcher operates in batches of 32 blocks, deeming subsequent accesses as sequential if it falls within this range, consequently triggering the prefetch of another batch. Disabling OS prefetching can increase disk I/O when access strides are shorter than 32 blocks.

3.2 Concurrency Bottlenecks and Software Overheads

Current prefetching mechanisms can incur high software bottlenecks, such as system calls, concurrency, and data movement. These challenges become more pronounced in scenarios where multiple threads concurrently access a file, as exemplified by RocksDB, where database and log files are shared among multiple readers and writers.

The bottlenecks associated with locking and concurrency originate from several key factors. Firstly, both prefetching and regular I/O operations like `read` share the same data structures, such as the cache tree used for lookup and updates. This cache tree is implemented in Linux using Xarray, which employs a global reader-writer lock. Consequently, prefetch calls can obstruct regular I/O operations, even those related to lookup tasks [17]. Conversely, prefetch threads reading data from the disk and updating the Xarray can lead to blockages in regular I/O operations. Secondly, the situation is further exacerbated when employing system calls like `fincore` [18] for cache state queries. These calls construct cache awareness dynamically and lock the entire virtual memory within the process, resulting in an overall application slowdown.

The bottlenecks stemming from system calls are due to the frequent issuance of prefetch calls. For example, in RocksDB, multiple threads share access to log and database files. These threads could initiate prefetch operations without knowing the cached pages' existence, thereby increasing system calls.

3.3 Failure to Exploit Memory Budgets

The I/O prefetchers in OSes do not exploit available system memory. For instance, incremental prefetching, which is commonly used in Linux and FreeBSD [2, 16], is not sufficiently aggressive or adaptive to the available memory space. The conservative incremental approach limits prefetching to 128KB, irrespective of the available system memory. Unfortunately, this results in high cache misses for several I/O-heavy applications that perform I/O during startup or switch between low and high I/O phases. Only implementing aggressive prefetching without adaptive eviction to reduce memory usage could impact prefetch effectiveness.

3.4 Analysis

In Figure 2, we analyze the aforementioned issues of OS-delegated and application-centric techniques for RocksDB

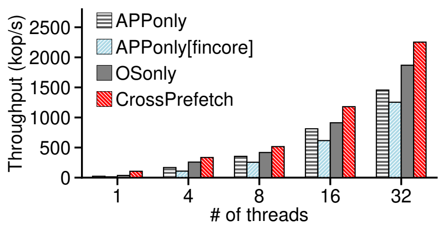


Figure 2. RocksDB Analysis: The graph shows throughput for a multi-threaded multi-read random workload where 32 threads collectively read 120GB database. We compare APPonly, APPonly[fincore], and OSonly, representing application-only, application-only using fincore [18], and OS-only prefetching methods, respectively, alongside our proposed CrossPrefetch.

	APPonly	APPonly[fincore]	OSonly	CrossPrefetch
Locking (%)	16	34	27	19
Cache Misses (%)	98.2	91.5	84.3	63.7

Table 1. Lock Overhead and Avg. Cache Misses (in %)

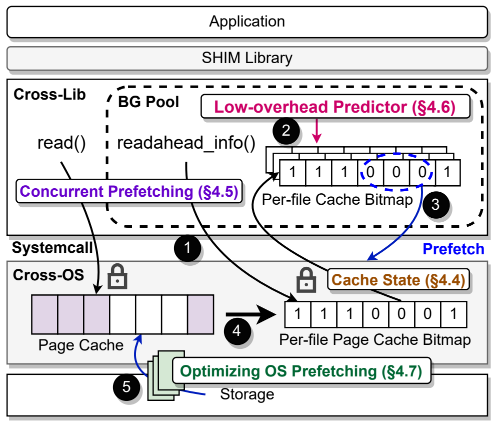


Figure 3. CrossPrefetch Design Overview. The system is divided into two parts Cross-OS and Cross-Lib. Cross-Lib intercepts I/O system calls, predicts access patterns, and decides on prefetching size. Cross-OS exports cache state, per-file memory use, and optimized I/O path. Operations 1 to 5 show request execution cycles when prefetch is triggered.

with 32 application threads performing batched-but-random reads using *db_bench*. The y-axis shows the throughput, and we use a total data size of 100GB (without warm-up) that fits within the system memory (128GB). Table 1 shows the average cache miss (%) and the time spent on the lock (%). We compare four approaches: (1) *OSonly*, where the OS handles prefetching; (2) *APPonly*, which disables OS and application prefetching for random access; (3) *APPonly[fincore]*, which uses a background prefetching thread to use `fincore` [18] to query cache state and issue prefetch operations; and finally, (4) the proposed *CrossPrefetch* approach.

First, we observe that the *OSonly* approach outperforms the unmodified RocksDB application (*APPonly*) that turns off I/O prefetching for random accesses. The *APPonly* approach suffers from higher cache misses, highlighting the need for more informed prefetching in complex applications. Next, the *OSonly* approach performs better than *APPonly*, but it employs an incremental prefetching technique that restricts memory use despite the availability of free memory. Due to random access, the prefetching window reduces initially and only improves towards the end. In contrast, the *APPonly[fincore]* approach provides cache visibility but suffers from high concurrency bottlenecks. In fact, these overheads prevent effective prefetching, resulting in a high cache miss.

Unlike the above approaches, in our proposed cross-layered CrossPrefetch, the OS exports cache visibility to the user-level runtime to understand prefetching effectiveness. Using the exported information from the OS, the runtime reduces system calls, provides concurrency, and implements memory budget-centric aggressive prefetching and eviction policies. Further, CrossPrefetch also adds other OS optimizations. These combined design capabilities contribute to the performance gains, which we will detail shortly.

4 Design and Implementation

4.1 Design Goals

At its core, CrossPrefetch is designed to enhance the current OS prefetching by adhering to the following principles.

Goal 1: Disaggregate I/O prefetching responsibilities between the OS and a user-level runtime. CrossPrefetch achieves this disaggregation by utilizing cross-layer awareness of the OS cache state. This awareness empowers higher-level layers (e.g., runtime) to perform precise and effective prefetching. This approach prevents under or over-prefetching, redundant prefetch system calls, and contention for page cache locks between regular and prefetch operations by clearly defining their access paths.

Goal 2: Support concurrent prefetching and lightweight prediction. For files shared among threads, CrossPrefetch facilitates concurrent prefetching for non-conflicting blocks and offers low-overhead access pattern detection that enhances prediction accuracy.

Goal 3: Enable aggressive prefetching and eviction without impacting memory budget. Using per-application access patterns, per-file cache states, and available free memory, Cross-Lib dynamically adjusts prefetching aggressiveness to reduce cache misses and mitigate I/O bottlenecks.

4.2 Our Approach: Cross-layered Prefetching

In CrossPrefetch, we seek a balance in dividing the prefetching responsibilities across the user-level library (Cross-Lib) and the OS (Cross-OS). First, CrossPrefetch eliminates unnecessary application-level prefetch calls. The user-level library (Cross-Lib) employs cache state bitmaps to discard application prefetch system calls (`fadvice(), readahead()`)

when the requested data blocks are already cached. This reduction in system calls minimizes overhead and lowers locking costs (as illustrated in Table 1). In the absence of Cross-Lib, the OS must handle application system calls.

Secondly, Cross-Lib captures and optimizes the application's thread-level prefetching by capturing their access pattern, which is challenging to accomplish in an OS without visibility of application-level threads. Specifically, we implement a concurrent range tree that captures a thread's access patterns for private and shared files and optimizes per-thread prefetching (see §4.5).

Third, CrossPrefetch facilitates simpler customizations without further complicating the existing complex OS caching layers. This involves effortless integration of policies like aggressive prefetching (given available memory) and eviction within Cross-Lib. While transferring these policies to the Linux OS is feasible, it introduces complexities throughout the OS virtual memory stack (including the page allocator, VFS page cache, and LRU-based eviction) and limits the adaptability for application-specific configurations.

4.3 CrossPrefetch Layers Overview

This section demonstrates the prefetching steps of CrossPrefetch. We present the overview in Figure 3 and iterate the steps in detail below.

User-level: Cross-Lib optimizes I/O prefetching of individual files in an application. It employs a shim layer to transparently intercept POSIX I/O, detect access patterns, and initiate prefetch calls. Upon opening a file, Cross-Lib creates a user-level file-descriptor structure to maintain file-level access pattern and prefetch information (❶). When an I/O request (e.g., read, write) is issued, Cross-Lib predictor identifies a file's access pattern across application threads and decides on the bytes to prefetch (❷). Subsequently, Cross-Lib uses our newly introduced `readahead_info` system call for three purposes: to prefetch blocks, to export the per-file cache state from the OS, and to export OS-level telemetry like cache memory usage of a file (❸). Furthermore, we use dedicated background threads to issue prefetch calls to prevent impacting application thread performance. Finally, Cross-Lib adapts to the available memory budget and carries out aggressive prefetching and cache evictions.

OS Component: Cross-OS maintains the cache state and the exported prefetch information for use by Cross-Lib. Within the OS, to reduce contention between threads issuing regular I/O and those initiating prefetch operations, Cross-OS segregates the regular I/O and prefetching paths. Contentions mainly arise from acquiring distinct locks, including inode (file), page cache, journal, and memory manager locks. During a regular I/O operation (read or write), Cross-OS updates the bitmap when pages are fetched into or evicted from the cache (❹). Handling the `readahead_info` call involves initial checks for the presence or absence of requested blocks, followed by the adjustment of prefetch requests and the issuance of the request. Upon return, the OS exports the file's cache bitmap to Cross-Lib. Ultimately, Cross-OS introduces optimizations to the I/O prefetching path and prefetch parameters, such as removing predefined limitations (❺).

Listing 1. `readahead_info` simple code use example

```
void prefetcher(uinode* inode, int fd, off_t offset, size_t
        prefech_size)) {
    struct cache_info info;
    size_t prefetch_limit = 0;
    prefetch_limit = offset + prefech_size;
    info->cache_state_bitmap = inode->cache_state_bitmap;
    while (offset < prefetch_limit) {
        readahead_info(fd, offset, size, &info);
        offset = predict(&info);
    }
}
```

4.4 Provide Visibility on I/O Prefetching State

Applications and runtimes benefit from the visibility into the cache state and the progress/status of prefetching requests, as it allows them to evaluate the effectiveness of prefetching and adjust future requests accordingly. However, providing visibility should not significantly impact application performance or resource usage. Unfortunately, current mechanisms like `fincore()`, discussed in §3, do not meet these requirements.

In contrast, CrossPrefetch achieves cache visibility and prefetching awareness with minimal performance overhead and no application changes. CrossPrefetch introduces a new `readahead_info` call with an `info` parameter, which is a structure that stores per-file bitmap and other information about the file. The `readahead_info` system call extends the existing prefetch call, `readahead()`, as shown in Code 1. Cross-Lib uses `readahead_info` call and `info` structure to read the per-file bitmap for prediction and future prefetch operations.

Besides storing bitmap information, each `info` structure maintains other fields used for communication between the OS and Cross-Lib. This includes information for control-plane operations or for telemetry. Control-plane operations include files for which prefetching can be disabled, the offset and bitmap range to copy to userspace, and optimizations (§4.7) to increase the prefetch window flexibly. The telemetry fields include per-application and per-file cache usage, which are used for aggressive memory prefetching and eviction (§4.6). Additionally, telemetry includes counters for the number of per-file page cache hits and misses, providing insights into the effectiveness of a prefetching policy.

Challenges: Implementing cache visibility poses the following challenges: first, the cost of scanning the OS cache data structure to check whether a block has been prefetched for a file's per-inode Xarray [17] is expensive for applications with many large files. This is because it requires acquiring spin locks for each inode and its Xarray, which can compete with regular I/O operations. Second, copying the page cache state frequently between the protected OS and the unprotected runtime can incur high data copy costs. Lastly, fast updates to the per-inode cache bitmap are critical to deciding which set of blocks to prefetch. This is particularly important since Linux and other OSes incrementally shrink or grow pages using background threads that use LRU-based cache management.

Delineated Prefetching Path: We propose delineating the I/O prefetching and regular I/O paths when feasible to address the above challenges. CrossPrefetch utilizes a stateful per-file bitmap at the OS level to track the cache state and improve prefetching efficiency. This complements the per-file cache trees already present. Each bit in the bitmap represents a block in the file (by default), and the bitmap is an array of unsigned longs that dynamically grows and shrinks with the file size. This bitmap is also imported to Cross-Lib.

Updating the Bitmaps in Cross-OS: The per-inode bitmap is continually updated during read, write, and prefetch operations. We introduce a **slow and a fast path** to reduce the contention of a single big per-file cache-tree lock between regular I/O and prefetch operations. During regular I/O (i.e., `read()` and `write()`), the OS uses a slow path that involves checking for the requested block's presence in the cache by walking the per-file Xarray. This walk is done using a page vector called `pvec`, which records the availability of multiple blocks in the cache. If a block is missing from the cache, a read request is issued for it, and the per-inode cache bitmap is updated. During this process, the cache tree's lock is held, which can contend with concurrent prefetch operations and impact regular I/O and prefetching effectiveness.

Fast Prefetch Lookup: First, Cross-Lib always uses the user-level copy of the bitmap to check for cached pages. When a `readahead_info` call is issued, the OS employs the fast path for the file's bitmap lookup to reduce lock contention with non-prefetching I/O. This is achieved by acquiring the bitmap's `rw-lock`. If additional pages are requested and inserted into the cache, the `readahead_info` call employs a slow path, requiring the acquisition of a write lock. However, this process remains fast due to its reliance on simple bitmap operations. To further mitigate contention, Cross-OS updates the per-inode bitmap only once after completing the entire walk rather than updating it for each page.

Importing Cache Bitmaps to Cross-Lib: Figure 3 illustrates a sample use case of how Cross-Lib imports the per-inode bitmap from Cross-OS into a user-level buffer. When Cross-Lib calls `readahead_info`, Cross-OS first checks the per-inode cache bitmap before initiating a prefetch operation to determine if the requested blocks are already present in the cache; this step avoids the need for cache tree traversal, leading to three possible scenarios: (1) all blocks are already present in the cache, and the userspace `info` structure is updated; (2) all blocks need to be prefetched, prompting a read request followed by a bitmap update; (3) some block pages are already in the cache, resulting in the modification and issuance of the prefetch request. The `readahead_info` system call serves a dual purpose: it copies the per-inode cache state from the OS to Cross-Lib while also triggering prefetch operations for blocks required but not yet present in the OS cache. This consolidation eliminates the need for separate calls, even though prefetch operations occur less frequently than read or write operations. To minimize the overhead of copying the per-file bitmap from the OS to Cross-Lib buffers, Cross-Lib can specify offset and range values in the `info` structure for selective copying, as demonstrated in Listing 1.

System Call and Bitmap Memory Overhead: CrossPrefetch reduces unnecessary prefetch system calls by first checking for pages already in the cache and issuing prefetch requests only when necessary. Similarly, the per-file bitmaps in CrossPrefetch consume only a small fraction of memory. Consider a 1TB file requiring just 32MB (> 0.005% memory cost). To further reduce memory consumption, the OS component copies a select bitmap range from the OS to the Cross-Lib, such as 64 bytes for a 2MB prefetch. Additionally, as discussed in §4.6, another memory-centric optimization could be to use a bitmap bit to represent a range of multiple blocks.

4.5 Scalable and Concurrent Prefetching

We next discuss the mechanics of Cross-Lib with a focus on enabling concurrent prefetching for private and shared files with a focus on reducing system call overheads.

Challenges: Supporting concurrent prefetching across tens and hundreds of application threads is of utmost importance. For threads accessing private files, Cross-Lib maintains per-file (i.e., per-inode) cache bitmaps in userspace. This approach enables threads operating on entirely different files to work concurrently by utilizing the current I/O prefetch request.

However, certain applications, including RocksDB, HPC-based molecular simulations, and databases, employ per-thread file descriptors to enable concurrent access to shared files across multiple threads or processes. These threads use their file descriptors to read from or write to specific regions of the files. For instance, RocksDB employs per-thread file descriptors for concurrent I/O to share log and database files between client and background threads.

Using per-file prefetching with a shared per-file cache bitmap and access pattern prediction can introduce scalability challenges. To be more precise, application threads inspect the per-inode cache bitmap while a dedicated group of helper threads sends actual prefetch requests to the OS. Consequently, concurrent updates and access to per-inode bitmaps must be serialized using read-write locks (`rw-locks`). The fundamental design of CrossPrefetch employs per-inode bitmap locks to enable concurrent access across threads to the shared file. However, scalability can be impacted as the file size increases.

Range Bitmaps with Scalable Range Tree: One approach to reducing synchronization bottlenecks across application threads is to maintain a separate bitmap for each thread or a file descriptor and import a file's cache bitmap from the OS. Besides the complexity of keeping the bitmaps consistent, this approach can also increase per-file memory.

Therefore, we introduce a concurrent per-file range tree. This tree tracks a range of blocks accessed by each thread using their private or shared file descriptor. Each range tree node represents a contiguous range of blocks and includes an embedded bitmap, with each bit representing a single block within the range. Each node's range can dynamically grow or shrink along with the bitmap. For concurrency, each node has its own lock, which is acquired solely when accessing a block within that range.

By maintaining a range tree with per-node ranges and per-node locks, multiple application threads utilizing the same or different file descriptors can concurrently access non-conflicting ranges of a file and their corresponding bitmaps. This approach not only mitigates scalability bottlenecks but also eliminates the need to replicate the bitmaps. Importantly, threads accessing overlapping blocks share the bitmap and benefit from the awareness of pages already in the cache, reducing redundant prefetch requests and associated overheads. In §4.6, we discuss maintaining separate access pattern prediction for each thread using their respective file descriptors.

4.6 Low-overhead Prediction and Prefetching

For effective prefetching that adapts to different access patterns, Cross-Lib first detects the access pattern of a file by intercepting POSIX I/O operations and deciding on the number of blocks to prefetch. The pattern detector identifies different access patterns, including sequential, random, forward/backward strides, and changes in access patterns.

Internally, Cross-Lib uses a simple n-bit counter for detecting a file's access pattern. The counter indicates the level of sequentiality and can represent a file in seven different states: highly random (000, access distance beyond the maximum prefetch distance of 128KB), random (001, random but within 128KB distance), partially random (010, a mix of sequential and random access), likely-sequential (011, frequent sequential interspersed with random access), sequential (100, sequential but with strides), and definitely sequential (110). During a read or write, Cross-Lib increments/decrements the counter's value based on the sequentiality of the access, and the counter's value determines the blocks to prefetch.

When a file is opened, we begin in a "definitely random" state, signifying that no blocks are prefetched. However, as

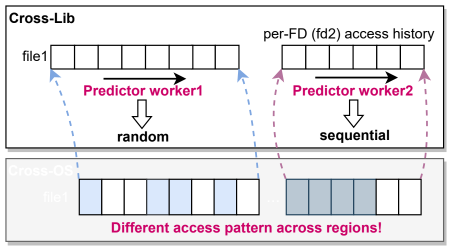


Figure 4. Fine-grained Predictor for Shared Files.

sequential accesses accumulate, the number of prefetched blocks grows exponentially by 2^n, where n represents the value of the access pattern counter. To prevent issuing prefetch calls for blocks already in the page cache, Cross-Lib examines the cache bitmap and modifies the prefetch request only for blocks that are not yet in the cache.

Cross-Lib can identify a variety of I/O access patterns, such as sequential, random, forward/backward strides, etc. This predictor intercepts each I/O and quickly assesses a file's transition or oscillation between access patterns. Moreover, the number of bits used for the per-file counter can be configured to improve prefetch accuracy. However, for the workload we analyze, a 3-bit counter provides the best performance without over-prefetching for different workloads with varying access patterns. To optimize the predictions and reduce the overheads of pattern detection, once a steady state is reached (i.e., definitely sequential or random), Cross-Lib delays predictions for the next n accesses.

Overall, Cross-Lib utilizes the cross-layered capabilities of OS and runtime prefetching, reducing frequent system calls and related overheads. Our future work will focus on enhancing Cross-Lib with sophisticated domain-specific predictors.

Support for File-descriptor Prefetching: To prefetch files accessed by multiple threads or processes, Cross-Lib utilizes file-descriptor prefetching and range-tree mechanisms (§4.5). It maintains an access pattern detector for each descriptor and a userspace file descriptor structure containing block range information and access pattern counters. Figure 4 illustrates how Cross-Lib executes prefetching. For instance, if `Thread 1` accesses `fd1` sequentially while `Thread 2` accesses `fd2` randomly, prefetching takes place exclusively for non-overlapping regions of the file accessed by `Thread 1`. In scenarios involving overlapping accesses across file descriptors, Cross-Lib leverages cache awareness to avoid redundant prefetching while ensuring cache hits. When configured, Cross-Lib could use a single bitmap bit to represent the entire range in the tree to reduce memory use.

Memory-aware Aggressive Prefetching and Eviction: CrossPrefetch improves I/O performance for varying memory availability and changing access patterns, including high and low-intensity I/O phases. It delegates prefetching control to Cross-Lib, which uses pages cache awareness to adjust prefetching and eviction based on memory budgets set by applications, containers, VMs, or system administrators. The key insight is to utilize the available memory to aggressively prefetch from the start of an application and *reduce high compulsory cache misses (i.e., blocks loaded to the cache for the first time).* This contrasts with OSes that employ incremental prefetching and suffer from high initial cache misses.

Aggressive I/O Prefetching: To maximize the effective use of available memory, Cross-Lib continually monitors memory usage and adapts prefetch aggressiveness accordingly. Cross-Lib assesses system memory availability and defines higher and lower threshold values to signify when to cease aggressive prefetching and when to halt all prefetching, respectively. System administrators can customize these values via a configuration file.

When using aggressive prefetching, when a file is opened, Cross-Lib optimistically assumes that the access pattern is sequential and prefetches a number of blocks (defaulting to 2MB) before sufficient I/O has been executed to ascertain the actual access pattern. If this optimistic prediction proves accurate and the file is marked as "definitely" sequential, CrossPrefetch issues larger prefetch requests (memory budget permitting), thereby accelerating access to the file and reducing cache misses. If the prediction turns out to be incorrect, Cross-Lib reverts to regular prefetching behavior and stops prefetching when a file is identified as having a random access pattern. Moreover, developers can easily extend Cross-Lib to accommodate custom prefetching policies and window sizes based on a file's priority.

Aggressive Reclamation: To aggressively reclaim cache pages, CrossPrefetch adopts a two-pronged strategy. Firstly, Cross-Lib maintains per-process memory budgets and monitors active and inactive files using the LRU mechanism. When memory budgets become constrained, CrossPrefetch evicts inactive file cache blocks, circumventing the need to traverse each file's range tree. Secondly, for large files, apart from the block-level evictions managed by the OS LRU, CrossPrefetch expels infrequently accessed blocks by leveraging per-file cache states through the `fadvice()` function. Much like Linux, we adopt a 30-second duration to designate a file as inactive, relocating it to the forefront of the inactive LRU files list. Under persistent memory pressure, Cross-Lib traverses the subsequent set of LRU files, evicting LRU ranges as deemed necessary. While we assess the benefits of this approach in §5, our future work will explore fine-grained (per-inode) LRUs within the OS to expedite memory reclamation.

Support for Memory-Mapped I/O: `mmap` is used by applications for read-intensive workloads to reduce system call and data copy overheads between the OS and user space. However, predicting and prefetching for *mmap* I/O without explicit I/O calls is challenging because the application performs byte-level load and store operations on mmap'ed files, making it harder to capture the application's access pattern. Note that the underlying block device still fetches blocks and adds them to the cache pages at the OS layer.

To avoid the prohibitive cost of intercepting each load and store, Cross-Lib's background thread uses the cached bitmap exported from the OS to detect access patterns (i.e., how pages were accessed) and performs prefetching, also varying the prefetch window size. While our basic approach provides benefits, it resembles Linux OS prefetching and is somewhat inaccurate. To the best of our knowledge, accurate prefetching techniques for memory-mapped files are lacking, and our future work will exclusively focus on this by exploring ideas like userspace fault handling to detect memory access.

4.7 Optimizing OS Prefetching Path

We observe that modern OSes (e.g., Linux) enforce strict I/O limits even when using fast storage (e.g., NVMe) and reasonably large memory. It's not surprising that relaxing these limits is crucial for efficient I/O prefetching. For instance, Linux sets the incremental prefetch limit to 32 pages (128KB), regardless of the available disk bandwidth or memory capacity. In contrast, we extend the OS to grant higher-level layers (e.g., Cross-Lib) the capability to (dynamically) increase the prefetch limit using the `info` structure in the `readahead_info` call. For instance, while Cross-Lib could issue a prefetch request of up to 1.2 GB to match our NVMe bandwidth, the prefetch requests do not exceed 64MB. Larger requests do not significantly impact blocking I/O (e.g., read-/write) since the VFS layer limits an I/O request to a maximum of 2MB, and implements congestion control to postpone prefetch requests that would delay blocking I/O.

In summary, CrossPrefetch reduces the overhead of prefetch operations to iterate the Xarray and reduces cache hit costs, uses delineated path for prefetch and blocking I/O operations, and reduces contention of cache tree locks, all of which improve prefetching effectiveness.

4.8 System Implementation

CrossPrefetch is implemented in approximately 7.6K lines of code distributed across Cross-Lib (5.6K LOC) and Cross-OS (2K LOC), integrated into Linux kernel 5.14. Cross-OS is implemented within the VFS and the memory management layer. As evaluated in §5, CrossPrefetch remains agnostic to the underlying OS file systems. Moreover, CrossPrefetch does not necessitate application changes. To utilize CrossPrefetch, applications link with Cross-Lib, which intercepts POSIX I/O operations to predict access patterns and prefetch accordingly. Cross-Lib also implements a shim for employing the `readahead_info` system call to prefetch data, access cache state using a bitmap, retrieve per-process memory usage from the OS, and enable communication between Cross-Lib and the OS to relax stringent prefetch limits.

Mechanism	Description
APPonly	Application tailored prefetching using `readahead` calls
OSonly	Prefetching delegated to OS and application prefetching is disabled
CrossP[+predict]	Fine-grained and low-interference prediction avoiding prefetching entire file
CrossP[+predict+opt]	*CrossP[+predict]* without OS limits but also provide memory-centric aggressive prefetching and eviction
CrossP[+fetchall+opt] (memory insensitive)	Proposed CrossPrefetch that uses cache state awareness to prefetch missing blocks of a file using `readahead_info()` assumes all data fits in memory

Table 2. Comparison Approaches

	APPonly	*OSonly*	*CrossP[+predict]*	*CrossP[+predict+opt]*	*CrossP[+fetchall+opt]*
shared-rand	93	89	69	75	91
shared-seq	19	18	17	14	6

Table 3. Microbench: Avg. Cache Misses (in %)

5 Evaluation

The primary goal is to investigate the following design aspects.

1. To what extent does CrossPrefetch's cross-layered prefetching improve I/O throughput?
2. Can CrossPrefetch scale across threads within an application and across processes in multiple applications?
3. To what extent is CrossPrefetch effective across different file systems and storage devices?
4. How effective are CrossPrefetch's aggressive memory prefetching and eviction strategies?
5. How does CrossPrefetch perform on real applications?

5.1 Experimental Setup and Methodology

CrossPrefetch is implemented in the Linux 5.14 kernel and evaluated on various systems with different configurations, as detailed in Table 2. Due to its slower performance compared to other methods, Fincore [18] is omitted from our comparison for brevity, as indicated by our motivation analysis (Figure 2). Irrespective of whether prefetching operations are initiated by applications or CrossPrefetch, we do not modify applications. We assess CrossPrefetch's performance on a system featuring two memory sockets totaling 80GB, a 64-core, 2.8 GHz AMD 7543 processor, and a 1.6 TB NVMe SSD. The NVMe SSD provides maximum read and write bandwidths of 1.4GB/s and 0.9GB/s, respectively, and is partitioned using the ext4 file system as the default.

To understand CrossPrefetch's benefits across file systems other than ext4, we replicate some experiments using the same hardware configuration, but this time utilizing the state-of-the-art F2FS file system [29], which is optimized for flash storage. Lastly, we analyze the benefits and implications of CrossPrefetch on a distinct storage medium: a remote NVMe storage connected to the host system through RDMA-based NVMe-oF (NVMe over Fabric) technology. Before each experiment, we clear the page cache.

5.2 Impact of Cross-layered Prefetching

We first evaluate microbenchmarks to understand CrossPrefetch's prefetching capabilities, concurrent access scalability, and the effectiveness of its mmap implementation.

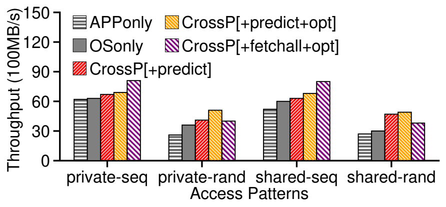


Figure 5. Microbench: Figure shows sequential & random access w/o file sharing; *private* denotes private per-thread file access, *shared* denotes file sharing, *seq* denotes sequential access, and *rand* denotes random access.

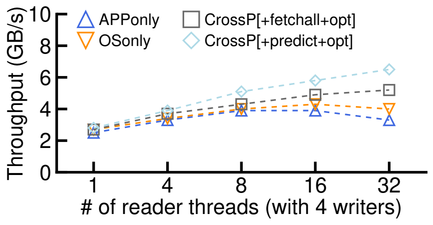


Figure 6. Microbench with File Sharing: Figure shows the aggregated write throughput when concurrent readers and 4 writers share a 128 GB file randomly accessing a non-overlapping range.

5.2.1 Microbenchmark Study We conduct experiments on a custom multi-threaded microbenchmark to analyze the impact of CrossPrefetch on private and shared file accesses and for sequential and random workloads. Threads issue 16KB I/O reads on private files, either sequentially (*private-seq*) or randomly (*private-rand*). The benchmark threads also share a large file with sequential (*shared-seq*) or random (*shared-rand*) reads to simulate HPC applications that share files and update non-overlapping regions in the file [4]. For all access patterns, the file size is limited to 200GB (2.15× larger than the available memory), and the memory cache is cleared before execution. We use *APPonly* and *OSonly* as baselines, as shown in Table 2. We also study shared file performance in the presence of multiple writers and readers Figure 6. To decipher the reasons for CrossPrefetch gains, we show cache misses (%) for two workloads in Table 3 for brevity.

Observations: For random reads on private and shared files, the *APPonly* approach turns off OS prefetching, resulting in high cache misses. The *OSonly* approach performs incremental prefetching limited to 128KB despite high memory availability at the start. Unfortunately, with incremental prefetching, random accesses reduce prefetch window (initially to 0), which increases cache misses and leads to lower throughput.

In contrast, the *[+fetchall+opt]* approach monitors missing blocks using the bitmaps exported from the OS and prefetches entire file(s). While idealistic, this approach is

Workloads	*APPonly*	*OSonly*	*CrossP[+predict+opt]*
readseq	578.55 MB/s	829.52 MB/s	1270.32 MB/s
readrandom	84.35 MB/s	484.34 MB/s	751.57 MB/s

Table 4. mmap: Sequential and Random workloads

impractical, specifically when the available memory is constrained. Despite using a workload that is 2.5x the size of available memory, *[+fetchall+opt]* still provides 1.54x benefits. However, it triggers cache pollution, increasing misses for shared and private files (see Table 3).

Furthermore, *[+predict]* uses precise prediction and achieves higher performance gains through higher cache hits. *[+predict+opt]* removes OS prefetch restrictions and performs aggressive-but-adaptive prefetching beyond 128KB limits based on available memory and I/O bandwidth. This results in increased cache hits, reduced prefetch system calls (not shown for brevity), and provides 1.81x and 1.97x gains over the *APPonly* for shared and private files, respectively. For sequential access, most approaches perform reasonably well due to low cache miss rates (Table 1). CrossPrefetch further reduces system calls and improves cache hits for sequential access to both private and shared files.

Concurrent Access Analysis: In Figure 6, we vary the number of concurrent readers on the x-axis while maintaining concurrent writers to four threads. First, *APPonly* and *OSonly* approaches suffer from the global reader-writer lock of the OS cache-tree. Additionally, *[+fetchall+opt]* struggles to scale as threads increase due to the per-file bitmap lock and insufficient memory. Conversely, in the case of *[+predict+opt]* with the scalable range-tree, each node (range) has its own read-/write lock, substantially enhancing the concurrent access performance for non-overlapping access.

Effectiveness of Memory-Mapped Access: To understand the effectiveness of prefetching support when workloads use `mmap`, we evaluate *CrossP[+predict+opt]* by varying the thread count and compare the throughput against *APPonly* and *OSonly* approaches in Table 4. As shown, *APPonly* turns off prefetching using `madvice`, resulting in a substantial slowdown. In contrast, *[+predict+opt]* improves performance by monitoring the cache state and using the Cross-Lib predictor to estimate and prefetch more blocks. However, as discussed in §4.6, because *[+predict+opt]* periodically fetches the cache state from the OS to identify access pattern as opposed to intercepting all load/store operations, our current implementation leaves performance on the table.

5.3 CrossPrefetch Impact on RocksDB

We next evaluate the cross-layered prefetching impact using the widely used RocksDB's *dbbench* benchmark. RocksDB is a production-scale LSM-based NoSQL database [5] designed to exploit SSD's parallel bandwidth and multi-core parallelism. We analyze RocksDB's workloads with multiple approaches, as detailed below. We use 40 million keys (̃120GB) as the database for our analysis.

Methodology: Table 2 summarizes the five approaches compared in this analysis. *APPonly* relies solely on application-controlled prefetching based on its perception of the access pattern, while the *OSonly* approach delegates prefetching to the OS. *[+predict]* uses cross-layered prediction without prefetching the entire file, while *[+predict+opt]* dynamically sets prefetch parameters based on available memory and I/O bandwidth, thereby removing static limits in the OS. Finally, *[+fetchall+opt]*, an idealistic but impractical approach, attempts to prefetch the entire file when opened using cross-layered prefetch information.

Sensitivity to Thread Count: Figure 7a illustrates the effect of varying thread count on the throughput for multi-readrandom workload. As thread count increases, the miss-ratio reduces, implying the advantage of threads benefiting from a shared cache state. The *APPonly* approach incurs performance degradation due to the lack of prefetching support for most files, whereas the *OSonly* approach performs incremental prefetching for all files with prefetching limits (128 KB), which reduces the benefits. In contrast, *[+predict]* and *[+predict+opt]* achieve 1.39x gains over the *APPonly* and 1.22x gains over the *OSonly* approaches without the need to load all database files to memory. Finally, *[+fetchall+opt]* provides maximum gains by prefetching all database files but significantly amplifies memory usage.

Performance Breakdown: Table 5 shows the breakdown of CrossPrefetch's incremental performance gains on RocksDB for 32-threads multi-read random workload. CrossPrefetch performance improvements are primarily through the following incremental factors: first, the cache visibility (*+cache visibility*) increases cache hit rate and reduces system calls by exporting cache states to the user space. Second, the support for concurrent prefetching using the scalable per-file range tree (*+range tree*) adds to the performance gains. Third, the support for understanding the memory budget and aggressively prefetching the data (*+aggr. prefetch*) further reduces cache misses, all leading to higher performance gains.

Sensitivity to Prefetch Limit Size: In Figure 10, we illustrate the impact of varying the prefetch limit size in the kernel along the x-axis from 32KB to 8MB. We observe that merely increasing the prefetch limit size for both *APPonly* and *OSonly* does not enhance performance due to their lack of cache state awareness and limited prefetching concurrency. In contrast, CrossPrefetch does not adhere to such prefetch limits. Increasing the prefetch size for each request (without memory pressure) provides performance gains, but the gains are limited, highlighting that only increasing the prefetch limit does not provide all the benefits of CrossPrefetch.

Performance Across Access Patterns: For different *db_bench* access patterns shown in Figure 7b, CrossPrefetch shows similar trends. For sequential reads, *OSonly* performs better than *APPonly* through efficient prefetching.

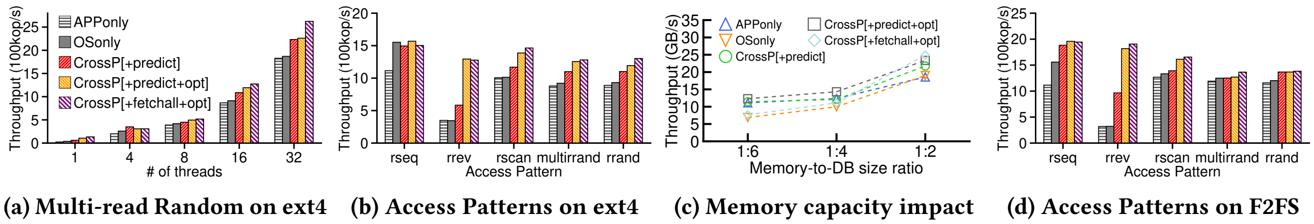


Figure 7. RocksDB DBbench on Local NVMe: (a) throughput for multi-read random workload varying thread counts; (b) throughput for access patterns using 32 threads; (c) RocksDB performance impact when varying the memory capacity to DB size ratio; (d) access patterns on F2FS file system (32 threads).

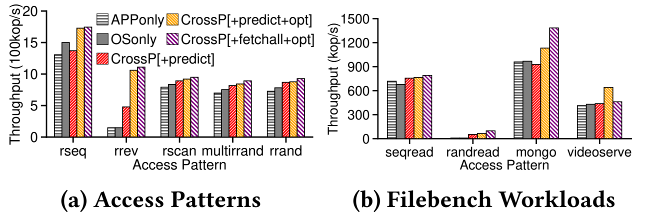


Figure 8. RocksDB on Remote NVMe (8a) and Filebench (8b).

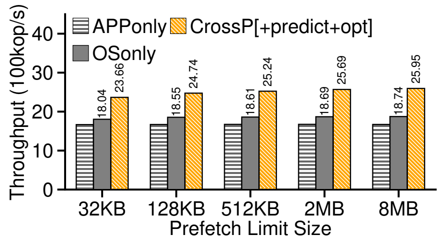


Figure 10. Prefetch Limit Impact: Figure shows the multi-read random workload for RocksDB when varying prefetch limit size (32 threads).

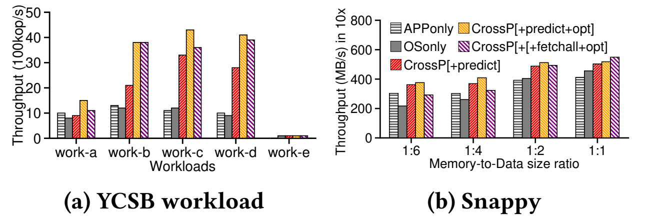


Figure 9. Real-world Workloads: (a) YCSB workloads with varying access characteristics (16 threads); (b) Snappy varying memory to on-disk size using 16 threads for a 120GB database; 1 : 6 in the x-axis denotes memory capacity set to (20GB).

APPonly	*OSonly*	CrossPrefetch [+cache visibility]	CrossPrefetch [+cache visibility +range tree]	CrossPrefetch [+cache visibility +range tree +aggr. prefetch]
1688 kop/s	1834 kop/s	2143 kops/s	2379 kop/s	2642 kops/s

Table 5. Breakdown of CrossPrefetch incremental gains.

[+fetchall+opt] and *[+predict]* provide higher gains for access patterns like read-while-scanning (readscan), while *[+predict+opt]* provides considerable gains over *OSonly* or *APPonly* for reverse read access, resulting in 3.7x gains.

Sensitivity to Memory Capacity: In Figure 7c, we evaluate RocksDB with ˜120GB database and vary the memory-to-disk size ratios along the x-axis from 1:6 to 1:1, where 1:6 denotes the memory capacity set to 20GB. We observe that *OSonly* without the understanding of memory budget, under-performs, specifically when the memory capacity is constrained. In contrast, *APPonly*'s lack of awareness of cache state results in more system calls but shows better performance compared to *OSonly* when the memory is constrained as it generally turns off prefetching for the batched multi-read random workload. In contrast, *[+fetchall+opt]* without aggressive eviction performs on par with *APPonly* and *OSonly* under low-memory scenarios. However, *[+predict+opt]* delivers superior performance with its aggressive prefetching and eviction.

Sensitivity to Filesystems and Remote Storage: We evaluate five RocksDB access patterns using F2FS and ext4-based remote storage (NVMe-oF). Apart from sequential reads, CrossPrefetch generally outperforms other approaches, particularly for reverse read, where it achieves up to 5.68x gains. This can be attributed to cache awareness, access pattern detection, and reduced lock contention. This demonstrates the adaptability of CrossPrefetch, as it consistently delivers performance gains across file systems and local/remote storage.

5.4 Multi-instance and Multi-process Workloads

To evaluate CrossPrefetch's I/O benefits on data-intensive multi-instance workloads, we use the Filebench macrobenchmark with sequential (*seqread*) and random read (*randread*), metadata-intensive MongoDB (*mongodb*) that creates thousands of files, and streaming video server (*videoserve*) workloads. We run 16 instances for each workload with an overall workload of 160GB. We compare CrossPrefetch's *[+predict+opt]*, *[+predict]*, and *[+fetchall+opt]* approaches against *APPonly* and *OSonly* baselines in Figure 8b. As observed, *APPonly*'s prefetch operations increase system call overheads and interference between prefetch and read-/write operations. *OSonly* avoids overheads but experiences interference and contention with the 128KB prefetching limit. *[+predict+opt]* reduces overheads, and limits interference with a dedicated bitmap structure, thereby improving performance. For example, *[+predict+opt]* compared to *[+fetchall+opt]* for *videoserve*, reduces cache pollution, and provides 55% performance improvement.

5.5 CrossPrefetch Impact on Real-world Workloads

Finally, we study two real-world workloads: RocksDB [13] with widely-used YCSB [9] and Snappy compression [10].

RocksDB + YCSB: We evaluate CrossPrefetch using the YCSB [9] cloud benchmark with workloads A-F, encompassing varying read/write ratios and utilizing the Zipfian distribution [25]. YCSB includes both a warm-up phase (write-only) and a run phase, and we conduct our experiments during the run phase employing 16 client threads. Figure 9a illustrates the throughput using a 4KB value size and 40 million keys. Workload A is write-intensive (50%), and its performance is dominated by new block writes without benefiting from the cache. In contrast, for the random read-intensive workloads B and C, CrossPrefetch's ability to prefetch blocks alongside read operations concurrently results in performance gains. *[+predict+opt]*, with its fine-grained prefetching, avoids prefetching the entire file and, therefore, outperforms *[+fetchall+opt]*. We observe similar gains for workload D, where most accesses involve recently inserted values. For scan-intensive workload E, the overall throughput is low due to application-level software overheads, but both *[+predict+opt]* and *[+fetchall+opt]* provide a twofold increase in throughput. Lastly, for workload F with 50% updates, CrossPrefetch accelerates the overwrite operations.

Snappy Compression: To evaluate CrossPrefetch's sensitivity to memory capacity, we use Snappy [10], a file compression engine widely used as a backend compression engine in various applications [3, 13, 28]. We modify Snappy to compress files in parallel using 16 threads. Snappy reads the entire file into memory before compressing and demands a significant amount of memory size, which can impact performance. We compress a 120GB dataset consisting of multiple 100MB files using Snappy and vary the memory-to-disk size ratios from 1:6 to 1:1 (i.e., increase memory) on the x-axis. Each thread opens a file, issues one or two read operations (mostly sequential), and moves on to the next file, creating a streaming-like access pattern. To exploit sequential access, we modify the application to explicitly issue a `fdavise` after opening a file in the *APPonly* approach. We observe that *APPonly* is limited by excessive system calls, while *OSonly* is ineffective with incremental prefetching. *[+fetchall+opt]* lacks aggressive eviction under low-memory conditions, resulting in a similar performance to *APPonly* and *OSonly*. However, *[+predict+opt]* enables aggressive prefetching and eviction, providing up to 31% performance gains with a 1:2 memory-to-disk ratio.

6 Conclusion

This paper focuses on accelerating I/O prefetching for modern storage and introduces the design of **CrossPrefetch**. CrossPrefetch optimizes prefetching by designing a cross-layered stack between the OS and user runtimes, where the OS provides cache visibility to the higher-level runtime. The runtime uses cache visibility to provide concurrent and fine-grained prediction prefetching to exploit storage bandwidth and memory availability. Evaluation of CrossPrefetch with various benchmarks and applications shows significant performance improvements ranging from 1.22x to 3.7x on modern fast storage devices.

Acknowledgements

We thank Jiri Schindler (our shepherd) for insightful comments to improve the quality of this paper. We also thank anonymous reviewers and the members of RSRL for their valuable feedback. This research was supported by funding from Samsung (Memory Solutions Lab), and NSF CNS grants 1910593, 2231724. This work was partially carried out on the experimental platform funded by NSF II-EN grant 1730043.

References

[1] Facebook RocksDB. http://rocksdb.org/.
[2] FreeBSD. https://www.freebsd.org/ports/references/.
[3] Google LevelDB . http://tinyurl.com/osqd7c8.
[4] A machine learning framework to improve storage system performance.
[5] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.
[6] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.
[7] Benjamin Cassell, Tyler Szepesi, Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. Disk prefetching mechanisms for increasing http streaming video server throughput. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018.
[8] Chandranil Chakraborttii and Heiner Litz. Learning i/o access patterns to improve prefetching in ssds. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 427–443. Springer, 2020.
[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
[10] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. Snappy Compession. https://github.com/google/snappy.
[11] Wei Ding, Yuanrui Zhang, Mahmut Kandemir, and Seung Woo Son. Compiler-directed file layout optimization for hierarchical storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.
[12] Bo Dong, Xiao Zhong, Qinghua Zheng, Lirong Jian, Jian Liu, Jie Qiu, and Ying Li. Correlation based file prefetching approach for hadoop.

In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 41–48, 2010.

[13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[14] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[15] Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. Vss: A storage system for video analytics. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 685–696, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, 2017. ACM.

[17] https://docs.kernel.org/core api/xarray.html. Xarray documentation.

[18] https://man7.org/linux/man pages/man1/fincore.1.html. man page fincore.

[19] https://man7.org/linux/man pages/man2/readahead.2.html. man page readahead.

[20] Song Jiang, Xiaoning Ding, Yuehai Xu, and Kei Davis. A prefetching scheme exploiting both data layout and access history on disk. *ACM Trans. Storage*, 9(3), aug 2013.

[21] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[22] Yongsoo Joo, Dongjoo Seo, Dongyun Shin, and Sung-Soo Lim. Enlarging i/o size for faster loading of mobile applications. *IEEE Embedded Systems Letters*, 12(2):50–53, 2020.

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. SOSP '19: Symposium on Operating Systems Principles, New York, NY, USA, 2019. ACM.

[24] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 241–255, USA, 2018. USENIX Association.

[25] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[26] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[27] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. Lynx: a learning linux prefetching mechanism for ssd performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2016.

[28] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[29] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015.

[30] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Jing Liu, Andrea C. Arpaci-Dusseau, Remz H. Arpaci-Dusseau, and Sudarsun Kannan. File systems as processes. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 14, USA, 2019. USENIX Association.

[32] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 277–293, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827. USENIX Association, July 2020.

[36] Pradeep Subedi, Philip Davis, Shaohua Duan, Scott Klasky, Hemanth Kolla, and Manish Parashar. Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 920–930, 2018.

[37] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, 2016.

[38] Ji Zhang, Xunfei Jiang, Xiao Qin, Wei-Shinn Ku, and Mohammed I. Alghamdi. Frog: A framework for context-based file systems. *ACM Trans. Storage*, 11(3), jul 2015.

A Artifact Appendix

A.1 Abstract

CrossPrefetch artifact comprises OS-level components user-level library and application, which are required for the execution. The artifact comprises real-world applications (RocksDB, Snappy, YCSB workload) besides microbenchmarks, as shown in the paper. The artifact includes steps to compile the OS, user-level library, and application benchmarks, and steps to run these workloads. More details are presented below.

A.2 Artifact Check-List (Meta-Information)

- **Program: user-level library and Linux kernel**
- **Compilation: The artifact includes a series of scripts to compile the user-level library, modified Linux kernel, and all other real-world application workloads and benchmarks.**
- **Run-time environment: Experiments are run on a baremetal Linux machine.**

- **Hardware: We conduct experiments on local and remote storage. For all experiments on local storage, we use a Linux system with at least 32 AMD/Intel CPUs, 128GB of free RAM, a 512GB Samsung PM173X NVMe SSD, and superuser access permission to install packages. For the experiments to be run on remote storage, two machines (Client and Storage server) connected via InfiniBand with RDMA interface are required.**
- **Output: The output generates regular execution logs and a text table to summarize the performance. The artifact describes the output files and information.**
- **How much time is needed to prepare workflow (approximately)?: 30 minutes**
- **How much time is needed to complete experiments (approximately)?: The artifacts have two types of workloads: (1) workloads that run for a shorter duration (< 60 minutes with relatively smaller input; and (2) longer workloads with large storage and memory requirements that can take about 4-6 hours (depending on the machine configuration) to complete for all data points.**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: GPLv2**
- **Archived (provide DOI)?: Yes https://doi.org/10.1145/3617232.3624872**

A.3 Description

A.3.1 How to Access The artifacts are available on GitHub: https://github.com/RutgersCSSystems/crossprefetch-asplos24-artifacts

A.3.2 Hardware Dependencies For the local experiments, a system with an x86-64 CPU is required. The minimum system needs 32 cores of CPU, 64GB of RAM, and 250 to 500 GB of available solid-state drive. The instructions for the experiments are tailored to and tested on an x86 host system but may work on other ISAs.

A.3.3 Software Dependencies We run workloads in a Ubuntu 18.04/20.04 system with a modified Linux 5.14.0. The README in the repository provides the following script to install all packages.

```
$ scripts/install_packages.sh
```

Note that all our instructions are for Debian-based packages. Similar packages exist for other OS flavors, which could be easily installed.

A.4 Installation

The artifact includes scripts to compile the CrossPrefetch libraries and benchmarks and automated steps to install packages, compile, and install the modified Linux kernel.

Manual Installation: First, the repository provides a step-by-step guide to compile and install each workload and the scripts to run the experiments and generate results. We recommend using these instructions first to understand the overall workflow, steps, and any installation errors that may occur. The repository provides steps for the Linux installation script in `compile_modified_deb.sh`. There are other helper scripts, as mentioned in the artifact README file.

Automatic Installation: Beyond this manual approach, the repository README provides information to use a single script to compile and install all applications and user-level libraries at once (`compile_all.sh`). When using this automatic installation script, the progress logs for the compilation can be found in the following path:

```
$ scripts/compile/
```

A.5 Evaluation and Expected Results

We provide a comprehensive step-by-step README on GitHub to reproduce the experiment in the paper. As a brief overview of the evaluation, we illustrate how to execute the real-world YCSB workload (Figure 9a) with CrossPrefetch. More evaluations can be found on our GitHub page.

Before running, we assume the modified kernel (Cross-OS) is installed on the machine (please see the README file) and the current work directory is in the project's root directory.

1. First, compile and install the user-level library (Cross-Lib):

```
$ source ./scripts/setvars.sh;
$ cd $BASE/shared_libs/simple_prefetcher;
$ ./compile.sh
```

2. Next, to run YCSB, please use the following commands:

```
$ cd $BASE/appbench/apps/RocksDB-YCSB;
$ ./compile.sh;
$ ./release-run-med.sh
```

3. We also provide a script to extract the results and present a text table.

```
$ python3 release-extract-med.py;
$ cat RESULT.csv
```

Expected Results for YCSB: According to our observation on a CloudLab Machine (r6525). CrossPrefetch significantly outperforms both *APPonly* and *OSonly* for workloads A to F, offering gains between 1.17x to 4x improvements across write and read-intensive workloads. The results could vary based on the storage throughput, CPU speed, and available memory, but we expect the overall performance to be high.

A.6 Experiment Customization

Our scripts offer easy-to-customize options for different workloads, access patterns, prefetching parameters, including bitmap size `CROSS_BITMAP_SHIFT`, prefetching size (`PREFETCH_SIZE_VAR`), prefetcher threads (*NR_WORKERS_VAR*) and more. These parameters can be configured within the library compile file (`compiler.sh`). In addition, each benchmark script can be edited to adjust application parameters. For brevity, we provide some examples in our github repository.