StreamCache: Revisiting Page Cache for File Scanning on Fast Storage Devices

Zhiyue Li and Guangyan Zhang, *Tsinghua University*

https://www.usenix.org/conference/atc24/presentation/li-zhiyue

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

StreamCache: Revisiting Page Cache for File Scanning on Fast Storage Devices

Zhiyue Li, Guangyan Zhang*
Department of Computer Science and Technology, BNRist, Tsinghua University

Abstract

Buffered I/O via page cache is used for file scanning in many cases as page cache can provide buffering, data aggregation, I/O alignment and prefetching transparently. However, our study indicates that employing page cache for file scanning on fast storage devices presents two performance issues: it offers limited I/O bandwidth that does not align with the performance of fast storage devices, and the intensive background writeback onto fast storage devices can significantly interfere with foreground I/O requests.

In this paper, we propose StreamCache, a new page cache management system for file scanning on fast storage devices. StreamCache exploits three techniques to achieve high I/O performance. First, it uses a lightweight stream tracking method to record the states of cached pages at the granularity of sequential streams. Second, it uses a stream-based page reclaiming method to lower the interference to foreground I/O requests. Third, it uses a two-layer memory management method to accelerate page allocation by leveraging CPU cache locality.

We implement StreamCache in XFS. Experimental results show that compared with existing methods, StreamCache can increase the I/O bandwidth of scientific applications by 44%, and reduce the checkpoint/restart time of large language models by 15.7% on average.

1 Introduction

In high performance computing, many applications exhibit file scanning I/O patterns as shown in previous works [7,8,49,56]. Figure 1 illustrates these I/O patterns. Each file may be independently accessed by one process (N-N mode) or shared by multiple processes (N-1 mode) during each I/O stage. For example, scientific applications perform file scanning when loading initial data, doing checkpoint/restart [43,50] or visualizing results [15,31]. AI training jobs also scan files when loading data files or performing checkpoint/restart [30,51].

*Corresponding author: gyzh@tsinghua.edu.cn

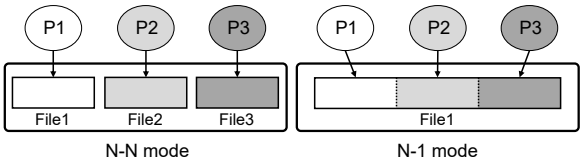


Figure 1: Typical I/O patterns of file scanning.

File scanning has a low data reuse rate because different processes will mostly access different parts of each file, even in the N-1 mode.

Buffered I/O via page cache is used for file scanning in many cases as page cache can provide buffering, data aggregation, I/O alignment and prefetching transparently. For example, modern HPC clusters build SSD-based burst buffer systems [16,17,32,48,49] to quickly absorb bursty I/O requests from the applications [25]. Representative burst buffer systems like HadaFS [16], Cray Datawarp [17] and GekkoFS [48] use kernel file systems and page cache to manage SSDs within one burst buffer node. Another example is Safetensors [4], a widely used machine learning checkpoint/restart library that loads model parameters and writes checkpoint files using the buffered I/O mode.

Although the buffered I/O mode can provide transparent caching, our study indicates that using the page cache for file scanning on fast storage devices like NVMe [53] SSDs has two performance issues. To demonstrate these issues, we conduct sequential I/O tests using the **fio** [6] benchmark, imitating the most straightforward file scanning workload. The kernel page size in this experiment is 4KB.

First, we find that buffered I/O scales poorly as the device bandwidth increases, limiting its performance on fast storage devices. We evaluate the bandwidth of accessing a 10GB file with a 4MB I/O size in the buffered I/O mode, and alter the device bandwidth limit by aggregating different numbers of Intel Optane 905p NVMe SSDs [20] into RAID-0 arrays. By comparison, we repeat the experiment with the direct I/O mode that bypasses the kernel page cache. As shown in Fig-

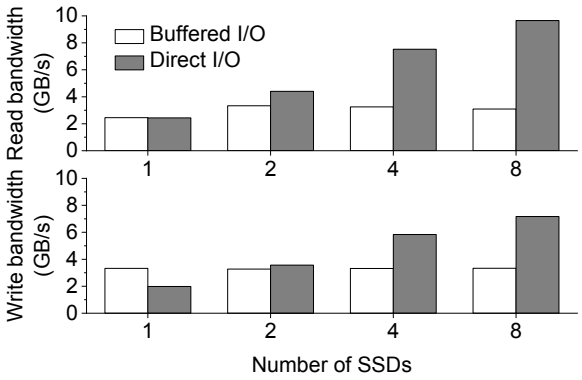


Figure 2: Sequential I/O bandwidth upon RAID-0 arrays with different number of SSDs. The experiment was performed on XFS, but we found similar trends in other file systems, such as Ext4 and F2FS.

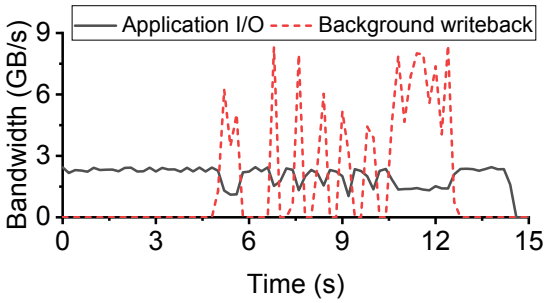


Figure 3: Interference between foreground I/O requests and background writeback.

ure 2, buffered I/O performs better than direct I/O on one SSD, thanks to the readahead and write buffering mechanisms in the page cache. However, buffered I/O performance has little improvement as the storage device becomes faster. When the number of SSDs is larger than one, buffered read bandwidth increases no more than 35%, and buffered write bandwidth does not have obvious improvement. Instead, direct I/O can achieve better scalability with such a large I/O size. This experiment demonstrates that current page cache management limits the scalability of buffered I/O in file scanning.

Although direct I/O manifests better scalability than buffered I/O for file scanning in this experiment, we argue that file scanning can benefit from the page cache for three reasons. First, file scanning of small I/Os performs better with the buffered I/O mode than the direct I/O mode for both read and write workloads [42]. This improvement is attributed to prefetching for read workloads and I/O aggregation for both read and write workloads. Second, the page cache can buffer data for write workloads and perform asynchronous writeback, effectively reducing I/O latency in the critical path. Third, when serving both read and write requests, using direct I/O requires additional effort for I/O alignment due to the strict restrictions on I/O addresses and sizes [18].

Second, we find that the performance of write-intensive file scanning can be significantly affected by background writeback. We demonstrate this by sequentially issuing buffered writes on a 30GB file, during which the kernel triggers background writeback to restrict the memory dirty ratio. As depicted in Figure 3, buffered writes can achieve relatively consistent performance at the beginning of the test, as the kernel writeback threads have not yet been woken up. However, foreground I/O bandwidth can be significantly impacted when background writeback is active, resulting in a 31.7% performance degradation during the background writeback phase. Background writeback is more intensive on fast storage devices due to their abundant device bandwidth, resulting in significant degradation of foreground I/O performance. Throttling the background writeback rate is not a permanent solution because it leads to the accumulation of dirty pages in memory, which subsequently blocks foreground I/O requests.

The above issues arise due to the combined impact of workload characteristics and the in-kernel page cache management. File scanning has a low data reuse ratio, which results in *high page cache miss rates* and *rapid growth of memory dirty ratio*. On the one hand, page cache allocates free pages in the critical path when a page cache miss happens, and the substantial overhead associated with page allocation in the kernel hinders foreground I/O performance. On the other hand, the page cache maintains dirty states, such as page dirty and writeback, at the page granularity. This design causes intensive lock contention between foreground write and background writeback as they both manipulate the shared index XArray [54].

This paper proposes StreamCache, an efficient page cache management system for file scanning workloads on fast storage devices. StreamCache includes three techniques to solve the above problems. First, a lightweight stream tracking method is designed to maintain the metadata of cached pages at the granularity of sequential streams. Second, a stream-based page reclaiming method is designed to perform page writeback and eviction based on the tracked streams, keeping interference to foreground I/O requests low. Finally, a two-layer memory management method is designed to accelerate page allocation by leveraging CPU cache locality.

We compare StreamCache with the original kernel page cache and FastMap-cache. FastMap-cache integrates the key techniques from FastMap [36], a new memory-mapped I/O (MMIO) design for fast devices, into the buffered I/O stack. FastMap-cache manages its own memory pool with per-core free-page lists to enhance the multi-core scalability of page allocation. Besides, it separates clean and dirty trees by designing dedicated per-core dirty trees to index the dirty pages. However, we find that allocating each page directly from global free-page lists does not provide good CPU cache locality in buffered I/O processing, and contention still exists in the dedicated dirty trees as both foreground write and background writeback need to get a spinlock to modify the dirty trees.

We implement StreamCache and FastMap-cache in XFS, and perform extensive experiments under both real-world and synthetic workloads. Experimental results show that, compared with the existing methods, StreamCache can increase the I/O bandwidth of scientific computing applications by 44% and reduce the checkpoint/restart time of large language models by 15.7% on average.

2 Background and Motivation

In this section, we first present the necessary background for buffered I/O. Then, we analyze the problems of poor scalability and high writeback interference when using page cache for file scanning on fast storage devices. Finally, we introduce the most relevant work FastMap [36] and analyze why it cannot fully solve the above problems.

2.1 Buffered I/O and Page Cache

Buffered I/O is a file I/O mode that can leverage the page cache to provide buffering, data aggregation, I/O alignment and prefetching transparently. It is used in many scenarios for file scanning, like local storage management in HPC burst buffer [16, 17, 48] and checkpoint/restart library for machine learning [4].

Although buffered I/O implementations vary in different kernel file systems, the ways they interact with the page cache are roughly the same. By referring to the source codes of representative kernel file systems (XFS [11], Ext4 [27], F2FS [22] and BtrFS [44]), we summarize the procedure of buffered I/O as following five parts:

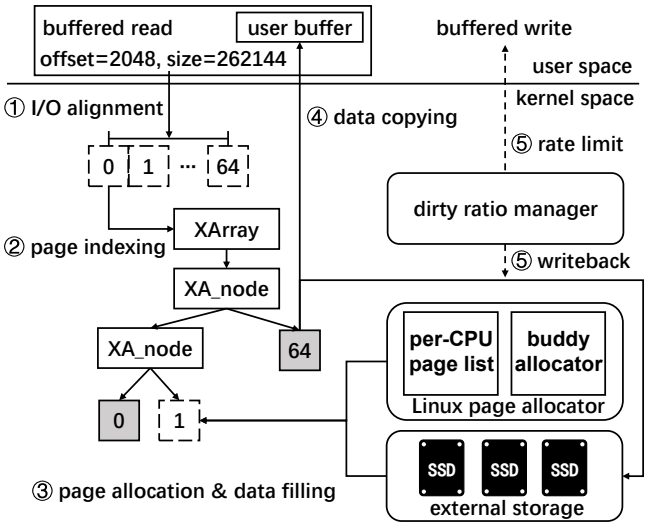


Figure 4: Buffered I/O procedure via page cache. Assuming that page 0 and page 64 are in the XArray initially, and page 64 is dirty. The page size is 4KB in this example.

I/O aligning and page-level processing. One advantage of buffered I/O is that applications do not need to do I/O alignment explicitly, even though block devices require I/O requests aligned to a certain size (e.g. 4KB). This is realized by I/O alignment in the kernel. For a buffered I/O request accessing the logical range $[offset, offset + length)$ within one file, the kernel aligns it with the page size (typically 4KB) as shown in Figure 4. After the alignment, the kernel processes each page within the aligned range sequentially. Each page has a *page index* to describe its position within the file.

Page indexing. File data can only be accessed by buffered I/O requests after it is cached in the page cache. Linux uses a per-file data structure called XArray [54] to index the cached data at the page granularity. As shown in Figure 4, XArray is a tree-index that consists of many *xa_nodes*. Each *xa_node* has three major fields: (1) pointer fields like the *slots* and *parent* that store the pointers to target pages, next-layer *xa_nodes* or the father *xa_node*, (2) indexing fields like the $shift$ and $offset$, which are used to decide the target pointer in *slots* when searching for a certain page, (3) tag fields like the *tags* that mark whether the sub-tree of this *xa_node* indexes any page that is dirty or under writeback.

For read-write contention, XArray uses RCU [28] lock to guarantee consistency. Major pointer fields in each *xa_node* are marked as read-side critical sections. It provides good lookup performance as the index modifications (e.g. inserting a new page or modifying the tag fields) will never block the index readings (e.g. looking up the page pointers). However, XArray still relies on the spinlock to guarantee consistency for write-write contention, like concurrent index modifications.

Page allocating and data populating. If an accessed page is not found in the page cache, the kernel will allocate a free page and insert it into the XArray and the kernel LRU list. Kernel page allocation is first served by the per-cpu page list [3], which is designed to retain small amounts of free pages for frequent page allocation and deallocation. If no page is left in the per-cpu page list, the kernel will supply free pages from the system-wide buddy allocator [52] in a batch manner. The buddy allocator maintains free pages in different orders, each of which is a double-link list protected by a spinlock [55]. After allocating the page, the kernel may need to populate it with relevant data from the device.

Data copying. After getting the desired page, the kernel copies data between the user and kernel space. Buffered reads copy data from the kernel pages to the user memory buffer and buffered writes manipulate data in the opposite direction.

Dirty ratio limiting. The kernel monitors the memory dirty ratio and limits its value according to the user configuration. Two configurable thresholds related to the memory dirty ratio

play an important role in this process. When the memory dirty ratio exceeds *dirty_background_ratio*, the kernel wakes up background writeback threads to write the dirty pages back. When the memory dirty ratio exceeds *dirty_ratio*, the kernel blocks buffered writes to limit the growth rate of the memory dirty ratio.

2.2 Performance Issues of Buffered I/O

Although it's convenient to use the buffered I/O mode for file access, we find that running file scanning workloads with the buffered I/O mode has performance issues, especially on high-performance NVMe SSDs. This section analyzes the issues of poor scalability and high background interference of buffered I/O under file scanning workloads.

We use **fio** [6] benchmark to simulate the simplest file scanning workload that sequentially reads or writes one file in the buffered I/O mode. The tested workloads include a sequential read workload, a sequential write workload and a sequential write workload with active background writeback. Each workload manipulates a 10GB file in XFS. The block device for XFS is a RAID-0 array of 8 NVMe SSDs, which simulates a next-generation high-performance NVMe SSD. In the third workload, we let the kernel flush the dirty pages actively by setting the *dirty_background_ratio* to 0, which can demonstrate the interference from background writeback more clearly.

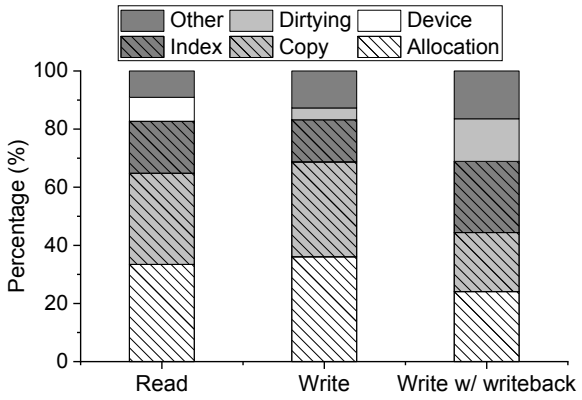


Figure 5: CPU occupation breakdown of buffered I/O.

For each workload, we use the performance analysis tool **perf** [5] to capture CPU occupation breakdown of buffered I/O. Figure 5 gives the CPU cycle breakdown of each test. We can get three conclusions from the experiments.

Firstly, page allocation (*Allocation*) takes major time among the three workloads, with the proportions of 33.45%, 36.06%, and 24.09%, respectively. This result reveals that high page allocation overhead significantly contributes to buffered I/O's poor scalability, especially for buffered reads and buffered writes without writeback interference. The ratio of reused data in a file scanning workload is low, resulting in a high page cache miss ratio. It is worth mentioning that the kernel needs to allocate a free page for each page cache miss.

Allocating from Linux page allocator may introduce three expenses. First, free pages are maintained in different orders. A page of order k contains 2^k physically continuous basic pages. When the lower-order pages run out, page-splitting overhead is introduced to supply the lower-order pages with the higher-order pages [55]. Second, pages of the same order are maintained in the same double-linked list that is protected by a spinlock, which can cause concurrency bottleneck when different CPU cores allocate free pages in parallel [55]. Third, the kernel may be configured to clean the page on allocation by writing all bytes to zero, introducing extra CPU overheads.

Secondly, background writeback increases the page indexing and dirty state maintenance overhead in foreground I/O requests. Buffered write does not interact with the device directly (i.e. no *Device* part in write workloads) because the page cache buffers the newly written data. However, the comparison between two write workloads reveals that background writeback increases the proportions of *Index* and *Dirtying*. The *Dirtying* refers to the operations to change the dirty states, like the XArray *tags* field in the kernel page cache. The dirty states are important to efficiently locate the dirty pages for writeback and make sure that these pages are synchronized to the external storage under an *fsync* command. For write workloads without active writeback, the proportions of index and dirtying are 14.53% and 4.03% respectively (with absolute times of 0.42s and 0.12s). However, under active background writeback, their proportions increase to 24.51% and 14.67% respectively, with absolute times increasing to 1.16s and 0.7s.

Further investigation reveals that this interference stems from the spinlock contention in XArray. Ideally, writing back a dirty page only needs an XArray search to get a pointer to the page, and this XArray search does not need to get a spinlock under the RCU mechanism. However, XArray couples page indexing and dirty state maintenance, so writeback operations must get the XArray spinlock to change the tag fields of *xa_nodes*. During writeback, the state of a page transitions from "dirty" to "writeback", and finally to "clean". In this process, XArray spinlock will be repeatedly acquired and released to change the dirty and writeback tags in the *xa_nodes*. The above problem is more severe in a file scanning workload as its ratio of reused data is low, causing a rapid increase in the memory dirty ratio and triggering the background writeback.

Lastly, copying data between the user space buffer and the kernel space pages is costly. However, in this work we do not optimize this part as data copying between the user space and the kernel space is necessary for the buffered I/O mode.

2.3 Page Cache Management in FastMap

The most relevant work to this paper is FastMap [36], which optimizes memory-mapped I/O on fast storage devices. With

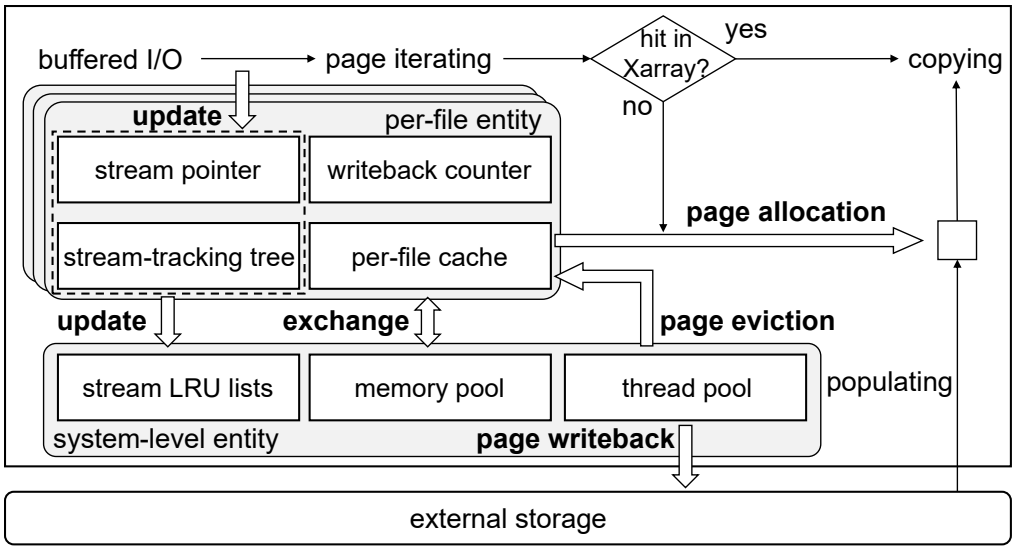


Figure 6: The architecture and workflow of StreamCache.

respect to the two problems in Section 2.2, two techniques in FastMap may help. However, our analysis reveals that these techniques are inadequate to exploit the performance of next-generation storage devices under file scanning workloads.

First, FastMap allocates a dedicated memory pool to accelerate page allocation. Zero-order free pages are allocated in advance and maintained in per-core double-linked lists. This can avoid the page-splitting overhead and lower lock contention on accessing the free-page lists when multiple CPU cores allocate free pages in parallel. However, as each page allocation is directly served from the system-wide free-page lists, we find that it is not cache-efficient for bandwidth-demanding applications.

Second, for the writeback interference problem, FastMap separates the dirty page index from the clean page index by maintaining dedicated dirty trees. In this way, FastMap does not need to maintain dirty tags to locate the dirty pages. Furthermore, FastMap scatters the pages into per-core page indexes to lower conflicts under concurrent accesses. However, separating the dirty pages from the clean pages only eliminates the lock contention between inserting clean pages and dirtying existing clean pages. There is still a conflict when a foreground I/O operation dirties pages while a background thread simultaneously performs writeback as they both manipulate the dirty trees.

3 StreamCache Overview

This paper proposes StreamCache, a high-performance page cache management system for file scanning workloads to better utilize the bandwidth of next-generation NVMe SSDs. Figure 6 presents StreamCache's architecture and workflow.

Design rationale. StreamCache consists of three modules to tackle the issues analyzed in Section 2.2.

First, a lightweight stream tracking module is proposed to maintain the dirty states of cached pages at the sequential stream granularity. For each opened file, StreamCache maintains a stream tracking tree and updates the tree when a buffered I/O request arrives. Furthermore, StreamCache keeps a stream pointer to lower the stream tracking overhead when the I/O pattern is highly sequential. All streams are maintained in two system-level stream LRU lists for page writeback and eviction.

Second, a stream-based page reclaiming module is proposed to perform page writeback and eviction. StreamCache has a pool of kernel threads that leverage the stream tracking module to quickly locate the dirty pages for writeback. Then, the pointers of these dirty pages can be extracted from the XArray without acquiring a spinlock, lowering the interference to foreground I/O requests. Besides, StreamCache uses per-file writeback counters for file synchronization. When StreamCache detects an insufficient quantity of spare pages, it exploits a stream-based eviction to reclaim the clean pages.

Third, a two-layer memory management module is proposed to enable fast page allocation. StreamCache has a system-level memory pool that maintains zero-order free pages in per-core double-linked lists. On top of the memory pool, StreamCache further designs the per-file cache to batch page allocation from the memory pool for bandwidth-demanding applications.

The components of each module can be categorized into system-level entities and per-file entities. The system-level entities are shared by all opened files to provide a global view and resource pooling. Each opened file also has its per-file entities to maintain local data structures.

Buffered I/O stack in StreamCache. In Figure 6, the solid arrows denote the reused page cache workflow in StreamCache. The bold arrows are the new workflow introduced by StreamCache. Before performing the page iterating, each request gets a *range lock* of its accessed logical range and updates the stream tracking module. Then, the request iterates through the accessed pages and processes them sequentially as the page cache does. If a target page is not found in the XArray, StreamCache allocates a free page from the two-layer memory management module. The request releases the range lock when all the pages have been processed.

4 Key Techniques

This section presents the detailed designs of StreamCache's key modules and how they can optimize the background write-back interference and page allocation overhead.

4.1 Lightweight Stream Tracking

The goal of the lightweight stream tracking module is to maintain the metadata for cached pages, both clean and dirty, at the stream granularity. It is based on an observation that the cached pages of typical file scanning workloads can be aggregated into continuous logical ranges that are much larger than a single page. Maintaining the dirty states of cached pages at the stream granularity enables kernel background threads to locate the pages for writeback quickly, lowering the interference to foreground I/O requests compared to existing methods that maintain page-level dirty states.

The stream tracking is done in a per-file manner, as mixing I/O requests at whole system may blur the simple I/O patterns within each file. Each file records the page indexes of its cached data by merging the continuous page indexes into streams. The streams are stored as key-value pairs in a per-file B-tree-based data structure called *stream tracking tree* (STT). A stream's key is its start page index, considering that extending a stream sequentially only increases its end page index. This design can avoid frequent STT modifications, especially under file scanning workloads. The stream's value is a pointer to its stream descriptor. All streams in a valid STT state are guaranteed to have no intersection. Figure 7 gives an example of an STT.

A stream descriptor contains several fields to describe a stream. The address range [*start*, *end*) records the start and end page indexes of the stream. The dirty range [$dirty_{start}$, $dirty_{end}$) records the dirty page indexes within the address range. A stream is *dirty* if it has a valid dirty range. These range fields are protected by a per-stream spinlock for concurrent modifications from foreground and background operations. Besides, each stream has an *upper_limit* field that denotes the start page index of its following stream. If a stream has no following stream, the *upper_limit* is set to *infinite*.

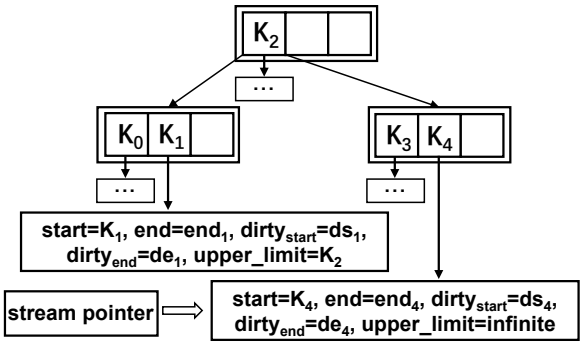


Figure 7: An example of stream tracking tree and stream metadata.

For each buffered I/O request, STT merges its page index range with existing streams to keep them non-intersected. We design an algorithm to search each intersected stream in $O(\log n)$ complexity, in which n is the number of streams in the STT. Considering that the number of streams is limited in file scanning workloads, this search only introduces minor overhead. When merging an I/O with an existing stream, the derived address and dirty ranges are the unions of previous ranges. When merging streams with intersected address ranges, their dirty ranges may not intersect. In this case, STT splits the merged address range into two streams, each maintaining one dirty range. The above process is performed recursively as the merged stream may still intersect with other streams in the STT.

StreamCache further introduces a *stream_pointer* to lower the STT searching overhead for sequential I/O. As shown in Figure 7, *stream_pointer* is a per-file pointer to the stream to which the last I/O request on this file belongs. Considering that I/O patterns of file scanning workloads are highly sequential, the *stream_pointer* provides a "cache" to accelerate I/O tracking. For each I/O request, the *stream_pointer* is first checked to see if the start page index of the request falls within the range [start, end] of the cached stream (which is referred to as a stream hit). A stream hit can eliminate the need for the STT search. Otherwise, this I/O request incurs a stream miss, in which StreamCache will perform normal stream tracking and update the *stream_pointer* to the newly accessed stream.

In an STT, some streams may have start page indexes larger than the end page index of the cached stream. Among these streams, the one with the smallest start page index is called the stream following the cached stream. Extending the cached stream without referring to the STT may cause the cached stream to intersect with its following stream. StreamCache introduces a per-stream *upper_limit* field to solve this problem. When a stream becomes the cached stream, the start page index of its following stream is recorded in this field. Each stream hit only requires referring the *upper_limit* to decide if the extended cached stream will intersect with its following

stream without searching the STT.

Extending the cached stream within the *upper_limit* will not violate the non-intersected property as modifications on STT are serialized by a spinlock. However, there is a false positive case in which the *upper_limit* field of the cached stream is smaller than the actual start page index of the following stream due to the background page eviction (see Section 4.2). This case will not violate the non-intersected property, only introducing a probably unnecessary STT search. It has little performance impact considering its rarity in file scanning workloads.

Finally, we talk about STT consistency. As stated above, stream tracking of I/O requests on the same file are serialized by a spinlock. This won't cause high contention as: (1) The I/O sizes of many file scanning workloads are much larger than one page (e.g. hundreds of KB in parallel applications [9]), and tracking each I/O only need to get the spinlock once. (2) Many kernel file systems serialize the buffered writes with the inode lock, which decreases the application-level concurrency [21,29]. However, batching updates to page dirty states at the I/O granularity does not come with no cost. It makes the dirty states between STT and XArray be eventually consistent because the STT is updated before the pages are dirtied. In view of this, StreamCache implements a per-file range lock manager to synchronize foreground I/O requests and background reclaiming on the same page range. The I/O tracking and page accesses in the buffered I/O mode should perform within the relative range lock.

4.2 Stream-based Page Reclaiming

StreamCache employs a stream-based page reclaiming method. It maintains the order of page writeback and eviction at the stream granularity and keeps multiple reclaiming threads running in the background.

Figure 8 presents the system-level LRU lists in StreamCache. The dirty streams in all STTs are linked in a double-linked list called *dirty stream list*. Likewise, all streams, whether dirty or not, are linked in another double-linked list called *stream list*. StreamCache aggregates the updates to LRU lists to lower maintaining overhead. A stream updates its position in LRU lists immediately if another stream becomes the cached stream due to a stream miss.

Based on the system-level LRU lists, StreamCache can efficiently perform writeback and eviction by finding a bundle of target pages with one search. This can lower the interference to the foreground I/O requests.

Page writeback. Like the Linux kernel, StreamCache writes dirty pages back when a certain condition is met (e.g., the memory dirty ratio reaches the *dirty_background_ratio* in StreamCache). The main difference from the Linux kernel is that StreamCache does not rely on the *tag* field in the XArray to locate the dirty pages. Instead, the reclaiming threads

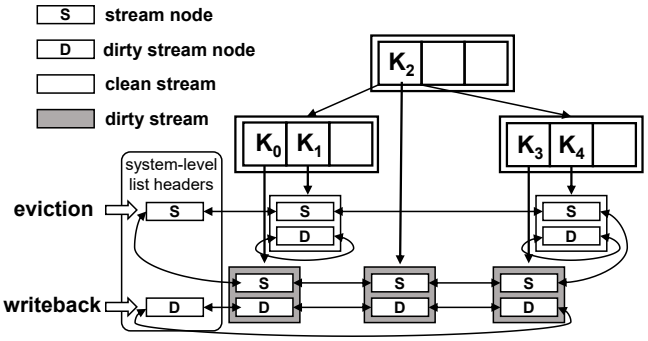


Figure 8: System-level LRU lists and stream-based page reclaiming.

traverse the system-level dirty list to find a dirty stream for writeback. For each traversed dirty stream, the reclaiming thread picks a range of dirty pages and tries to acquire the range lock on this range. If the range lock is acquired, the reclaiming thread will submit a writeback of relevant pages, shrink the dirty ranges (perhaps the stream is removed from the dirty stream list if all dirty pages are written back), and release the range lock. Otherwise, it repeats the above process until it succeeds. StreamCache issues asynchronous I/O for page writeback. After submitting write requests to the block device, the reclaim thread finishes the writeback and releases the range lock.

When a file synchronization command like *fsync* is called, StreamCache writes back all relative dirty pages in the STT of the file. A per-file writeback counter is used to indicate the writeback completion without the need for manipulating the writeback tags in the XArray. When a reclaiming thread submits an I/O to write *nr* dirty pages back, it increases the writeback counter by *nr* atomically. In the I/O completion callback function, the relevant writeback counter is deceased. File data synchronization operations like *fsync* and *fsync_range* can return if all relevant dirty pages are submitted for writeback and the writeback counter becomes 0.

During the writeback, StreamCache still needs to extract the pointers of target pages from the XArray with their page indexes. However, as StreamCache decouples the dirty state maintenance from the XArray, the background writeback only needs to get an RCU read lock for page searching instead of an RCU write lock (i.e. XArray spinlock). When a foreground I/O request requires a write lock to insert new pages, this design brings great benefits by converting the write-write contention to the read-write contention, and the RCU mechanism can handle the read-write contention efficiently [28]. FastMap does a similar optimization by recording dirty pages in separated data structures, but it cannot solve the background interference problem as both foreground writes and background writeback need a write lock to modify the same dirty page index.

Page eviction. When StreamCache detects an insufficient quantity of spare pages, it makes the reclaiming threads traverse the system-level stream list for page eviction. Like the page cache and FastMap, StreamCache only evicts the clean pages to avoid writeback blocking. Dirty streams are also traversed because their dirty ranges may not equal to their address ranges, indicating that the streams contain clean pages eligible for eviction. Unlike the page writeback, page eviction will change the STT when the first page of a stream is evicted as STT indexes each stream with its start page index.

4.3 Two-layer Memory Management

StreamCache manages the free pages with the two-layer memory management module to mitigate the high page allocation overhead. Figure 9 gives the main components of this module. It contains a system-level memory pool to manage the majority of free pages. Per-file caches are designed to accelerate page allocation for bandwidth-demanding applications.

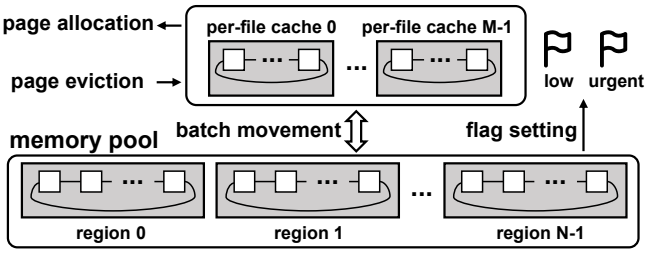
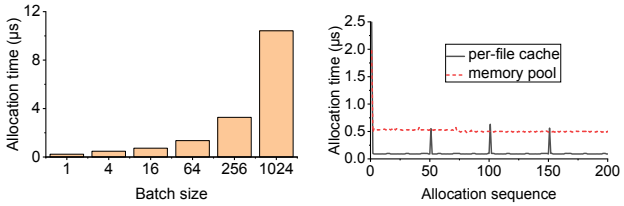


Figure 9: Two-layer memory management in StreamCache.

The memory pool consists of multiple memory regions, each of which allocates a preset number of zero-order pages when StreamCache kernel module is loaded. The number of memory regions equals the number of physical cores in the machine. The administrator can adjust the total pool size at runtime. Memory regions maintain the free pages with double-linked lists and per-region spinlocks. The free-page lists do not introduce additional memory overhead as they reuse the LRU fields of Linux page descriptors.

On top of the memory regions, StreamCache further designs the per-file caches to store a small number of free pages in file-local list headers. When a file is opened for the first time, StreamCache will allocate a per-file cache for it. All page cache misses on this file will allocate free pages from the relative per-file cache. If a per-file cache is empty on page allocation, StreamCache tries to allocate *low_mark* (50 by default) free pages from one memory region in a batch manner. The target memory region is decided by the application's currently running CPU core ID. If the preferred memory region does not have enough free pages, StreamCache will traverse other memory regions to steal free pages until *low_mark* pages are allocated or wait for page eviction if the allocation still fails after StreamCache traversing all memory regions once.



(a) Memory pool allocation time with different batch sizes.

(b) Allocation time comparison of each page allocation.

Figure 10: Allocation performance of memory pool and per-file cache.

We find that the per-file cache plays a key role in optimizing the bandwidth-demanding workloads compared to the memory-pool-only solution (like the memory management in FastMap). We perform two experiments on these two memory allocation methods to demonstrate this point. First, we test the average time of allocating free pages from the memory pool for the per-file cache with different batch sizes. For each batch size, we repeat the allocation 1024 times and calculate the average time. As shown in Figure 10(a), the allocation time does not grow so fast as the batch size increases within 64. This is because the allocation overhead is dominated by referring to the system-level free-page list for page allocation, even though no contention exists in this experiment. Second, we perform another test by repeating the page allocation 200 times with the per-file cache and the memory pool, respectively. Figure 10(b) demonstrates the time of each page allocation. It is shown that although the per-file cache sees latency spikes every 50 allocations due to its batch allocation from the memory pool, it can achieve relatively low allocation latency other times. This is because the local free-page list of a per-file cache can be embedded into the frequently used per-file entity, providing better CPU cache hit rates than the memory pool does.

The two-layer memory management module uses a global memory flag to control the page eviction. One background reclaiming thread is chosen to traverse all memory regions periodically, count the remaining free pages and set the global memory flag according to the free page ratio. StreamCache uses *threshold_low* and *threshold_urgent* for memory management. If the free-page ratio is lower than the *threshold_low* but higher than the *threshold_urgent*, the global memory flag is set to "low" so that all reclaiming threads start to evict pages from the system-level stream LRU list. If the free-page ratio is further lower than the *threshold_urgent*, the flag is set to "urgent" so that all reclaiming threads will additionally shrink all per-file caches by half and reclaim these pages to the memory pool. This can rebalance the free pages among different files quickly.

Evicted pages are put into the relevant per-file caches for potential reuse. Each per-file cache has a *high_mark* to restrict the number of free pages it can keep. StreamCache sets the

high_mark to be twice as the *low_mark*. StreamCache makes a further optimization by moving the page cleaning operation from the allocation to the eviction. Linux kernel can be configured to clean the page on allocation, like setting all bytes in the page to zero. Thanks to memory pool management in StreamCache, all pages can be cleaned when evicted, saving the cleaning cost from the allocation critical path.

5 Implementation

We replace the default page cache in XFS [11] with StreamCache and compile it as a new kernel module. To use StreamCache, the user needs to format a block device with the XFS formatting tool [24] and mount the device to a certain directory. Buffered I/O requests to files in this directory will be handled by StreamCache.

The design of StreamCache is not coupled with XFS and can be integrated into other file systems. To ease these new implementations, StreamCache functions are categorized into three types with respect to the positions they are inserted:

Module-level functions. Module-level functions only run once during the kernel module lifetime. They are inserted into the module initialization and exit functions and take effects when *insmod* and *rmmod* commands are called. During the module initialization, StreamCache starts its core modules and allocates global data structures. The memory pool is initialized at this stage to allocate free pages from the kernel. StreamCache also starts a number of background reclaiming threads at this stage for page writeback and eviction. The exit functions do the opposite things.

File-level functions. File-level functions are called when a file is opened or closed. These functions are mainly used to allocate and reclaim the per-file entities. During the buffered I/O requests handling, StreamCache may frequently refer to the per-file entity, like accessing the STT during I/O tracking or getting free pages from the per-file cache when a page cache miss happens. StreamCache maintains a concurrent-friendly hash table to map the file inode ID to its per-file entity. Multiple applications may share a per-file entity when a file is opened multiple times, and a new entity is allocated when no relative per-file entity exists. StreamCache maintains the same semantic with the kernel page cache that the cached pages are not written back or evicted immediately when all applications close the file. Background reclaiming threads will periodically do page writeback and eviction on cached data of zero-referred files.

I/O-level functions. I/O-level functions are called during each I/O request. These functions include I/O tracking, acquiring/releasing the range lock and the page allocation for page cache misses. Some background operations that manipulate the page cache also need to call the I/O-level functions, like the asynchronous readahead when a sequential access pattern is detected by the kernel [2]. These functions should also be tracked by STT to provide a holistic view of the cached pages in StreamCache.

6 Performance Evaluation

Our evaluation tries to answer the following questions:

- How does StreamCache perform on real-world file scanning workloads? (§6.1)
- How does StreamCache perform under synthetic workloads, with different parameter settings? (§6.2)
- What are the effects of individual techniques used by StreamCache? (§6.3)
- How much overhead will be introduced by the stream tracking in StreamCache when the I/O workload is not file scanning? (§6.4)

Platform. We conduct the experiment on a server that consists of a 32-core AMD Rome EPYC 7542 CPU and 128GB DDR4 memory, running Ubuntu 18.04 with Linux kernel version 5.4. The server can support ten PCIe 4.0 devices in parallel.

For evaluation, we use eight Intel Optane SSDs [20], the same as our motivation test in Section 2. In our platform, we measure that each SSD can provide 2581MB/s sequential read bandwidth and 2179MB/s sequential write bandwidth, very close to those in the specification [20]. To simulate a future high-performance NVMe SSD, we use Linux **md** (Multiple Devices) to aggregate these SSDs into a RAID-0 array with 64KB chunk size. Raw device test reveals that our platform can achieve 19558.4MB/s aggregated read bandwidth and 14745.6MB/s aggregated write bandwidth, approaching the aggregated device limits. Unless otherwise stated, the following tests are performed on this eight-SSD array. We evaluate the influence of different device bandwidth in Section 6.2.

Page cache management methods. We compare StreamCache with the original kernel page cache and FastMap-cache. As applications cannot directly interact with the page cache, we perform all our tests on XFS with different page cache management methods, abbreviated to as *Page Cache*, *FastMap-cache* and *StreamCache* hereafter. As FastMap [36] is designed for memory-mapped I/O, we implement its separated clean and dirty trees as well as the dedicated DRAM cache in XFS according to the paper and its source code [10].

I/O workloads. The evaluation is performed on real-world workloads and synthetic workloads. For real-world workloads, we test the I/O performance of scientific computing and the checkpoint/restart in the large language model (LLM), which are all typical file scanning workloads. Additionally, we use **fio** [6] as a micro-benchmark to evaluate how StreamCache performs under various parameters.

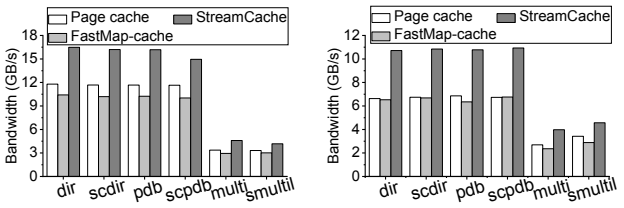
The memory pool sizes of both FastMap-cache and StreamCache are 64GB. When evaluating the Page Cache, we limit memory usage to 64GB with **cgroup** [1]. One exception is that for LLM I/O tests, we don't limit the memory usage as the application itself uses a non-negligible amount of memory and **cgroup** counts memory usage of both user space memory and kernel page cache. In all tests, background writeback is triggered when 10% of the maximum available pages become dirty, the same as the default value in Linux. Before each test, we drop all cached pages to exclude their impacts.

6.1 Performance on Real-world Applications

Scientific computing. We use the PF3DIO [26] I/O kernel to test how StreamCache behaves under parallel applications, arguing that it is a solution for local storage management in burst buffer [16, 17, 32]. I/O kernels extract the I/O patterns of scientific applications and are widely adopted in testing HPC file system performance [16, 32, 49]. PF3DIO is derived from a laser-plasma simulation application developed by Lawrence Livermore National Lab (LLNL). It contains six workloads that write checkpoint files in different formats. We test these workloads in both small problem size and large problem size. The total checkpoint file size of each workload is shown in Table 1.

Table 1: Total checkpoint file sizes of PF3DIO workloads.

problem size	dir	scdir	pdb	scpdb	multi	smulti
small	1.5GB	1.5GB	1.5GB	1.5GB	1.5GB	156MB
large	30.4GB	30.4GB	30.4GB	30.4GB	30.4GB	3.0GB



(a) Small problem size. (b) Large problem size.

Figure 11: I/O performance of PF3DIO scientific computing.

As is shown in Figure 11, StreamCache consistently outperforms the other two systems under different workloads. The first four workloads primarily write large arrays of floating-point numbers and two of them (*scdir* and *scpdb*) additionally write some scalar floating-point numbers after each array. These workloads mainly consist of large sequential I/O requests, and StreamCache provides 28%-62% improvements compared to the best-behaving one of the existing methods. The *multi* and *smulti* workloads have many small unaligned I/O requests, making their bandwidth lower than the other workloads. However, StreamCache can still provide 26%-48% improvements. That is because these workloads have high page cache miss ratios which need to allocate free pages frequently, and the two-layer memory management in StreamCache can provide faster page allocation than the other two methods.

Larger problem sizes make the checkpoint files larger, triggering the background writeback and degrading the foreground I/O performance. However, thanks to stream-based page reclaiming in StreamCache, the background interference to foreground I/O performance is smaller than that of the other two methods. Comparing Figure 11(a) and Figure 11(b), it can be seen that StreamCache can provide larger improvements on a large problem size (53% on average) than that on a small problem size (35% on average).

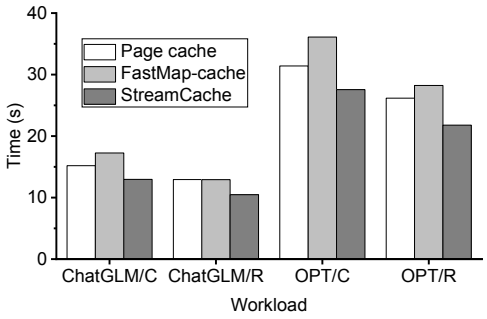
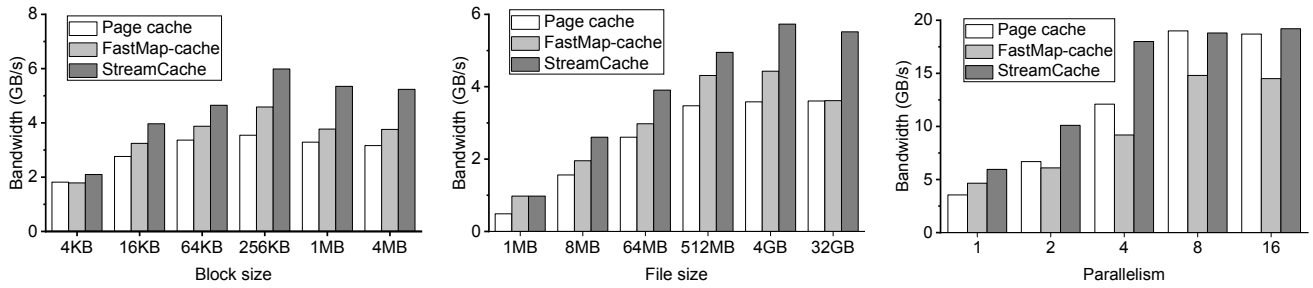


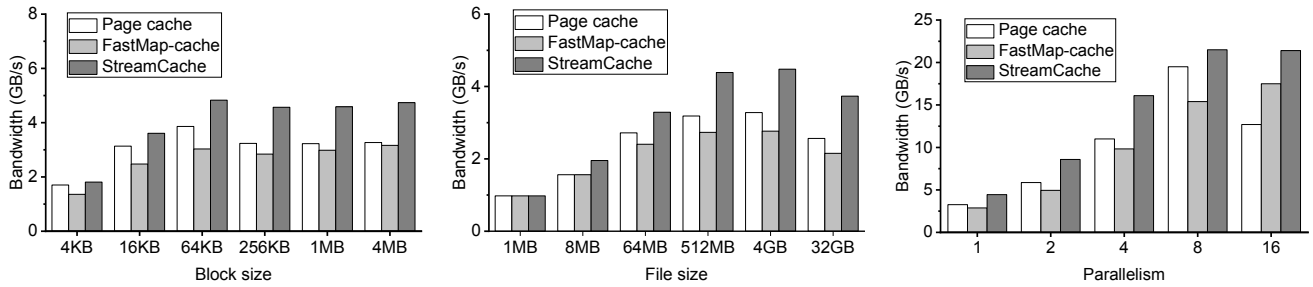
Figure 12: LLM checkpoint and restart performance.

Large language models. Previous works [30, 51] indicate that checkpoint/restart (C/R) introduces high overhead in machine learning, especially in large models. We demonstrate how StreamCache can boost the LLM training by testing the C/R performance of two large language models, ChatGLM-6B [46] and OPT-13B [13].

Figure 12 presents the checkpoint/restart times with different page cache management methods. The total checkpoint sizes of ChatGLM-6B and OPT-13B are 12GB and 24GB, respectively, which will trigger background writeback during the checkpoint process. Consequently, we observed longer checkpoint times than restart times in both models. Despite this background interference, StreamCache can cut the checkpoint times by 15% and 12% for the two models, thanks to its faster page allocation and stream-based page reclaiming. Additionally, StreamCache can reduce the restart times by 19% and 17% for the two models, mainly due to its faster page allocation.



(a) Read bandwidth with different block sizes. (b) Read bandwidth with different file sizes. (c) Read bandwidth with different parallelisms.



(d) Write bandwidth with different block sizes. (e) Write bandwidth with different file sizes. (f) Write bandwidth with different parallelisms.

Figure 13: I/O performance under synthetic workloads.

6.2 Performance under Synthetic Workloads

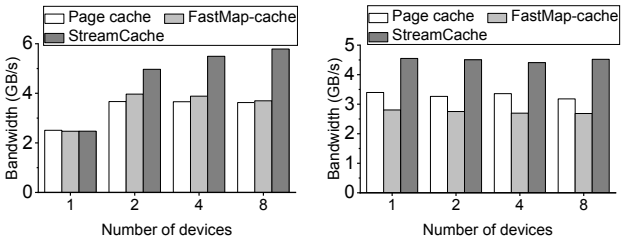
We use **fio** benchmark to produce synthetic workloads and evaluate how StreamCache performs under different I/O parameters, including varying I/O sizes, total file sizes, parallelisms, and device bandwidth.

Different block sizes. Figure 13(a) and Figure 13(d) present the bandwidth of accessing a 1GB file with different block sizes. Both FastMap-cache and StreamCache outperform the page cache under read workloads due to their optimized memory management. Additionally, StreamCache achieves higher bandwidth than FastMap-cache because its per-file cache can batch page allocation. As for the write workloads, FastMap-cache does not perform better than the page cache. This is because FastMap-cache needs to access multiple dirty trees to record the dirty pages, resulting in worse cache locality than just manipulating one tree (as the page cache does) under sequential I/Os.

Different file sizes. Figure 13(b) and Figure 13(e) present the bandwidth of accessing files of different sizes with a 256KB block size. For read workloads, although FastMap-cache achieves better performance on small files, its bandwidth drops with larger file sizes. This is because it divides the system memory into 32 (the number of physical cores) free-page lists, incurring memory-stealing overhead when the preferred free-page list runs out of pages. This problem also influences the page allocation in StreamCache, but the per-file cache batches page allocation and amortizes this overhead. The write bandwidth of all methods drops at a 32GB file size because the high dirty page ratio triggers the background writeback. However, StreamCache achieves lower foreground performance degradation than the page cache and FastMap-cache thanks to its stream-based page reclaiming.

Different parallelisms. Figure 13(c) and Figure 13(f) present the aggregated I/O bandwidth of different number of processes, each of which accesses a 1GB file with a 256KB block size. Although FastMap-cache achieves better read performance than the page cache with one process, its performance grows slower than the page cache when the number of processes increases. This is because the page cache can batch the page allocation from the buddy system with per-CPU page lists. StreamCache has the best performance as it not only eliminates page-splitting overhead and lowers multi-process contention with the memory pool but also batches the page allocation with the per-file cache. The write performance of the page cache drops under 16 processes because the high parallelism accelerates the rise of memory dirty ratio and triggers background writeback. Both FastMap-cache and StreamCache do not see degraded performance under 16 processes due to their optimized index designs.

Different device bandwidth. The above experiments demonstrate that StreamCache can boost the performance of file scanning workloads on a high-bandwidth storage device. In this part, we evaluate how StreamCache performs



(a) Read workload. (b) Write workload.

Figure 14: Impact of device bandwidth.

with varying device bandwidth.

We simulate a storage device of different bandwidth by constructing RAID-0 arrays with different numbers of NVMe SSDs. Figure 14 presents the results of running the **fio** benchmark on a large file (10GB). For the read workload, all three methods could approach the device bandwidth limit (2.5GB/s) upon one SSD, thanks to the readahead mechanism. However, as the device becomes faster, StreamCache can better utilize fast devices due to its fast page allocation. The bandwidth for the write workload does not change with the increased device bandwidth because data is buffered in memory. StreamCache achieves better performance due to its fast page allocation and lower interference from background writeback. We observe similar trends when the file size is small.

6.3 Effects of Individual Techniques

To isolate the improvement brought by StreamCache's key techniques, we use PF3DIO with a large problem size as an example and progressively add three techniques to test the performance.

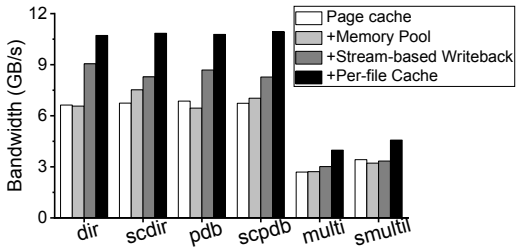


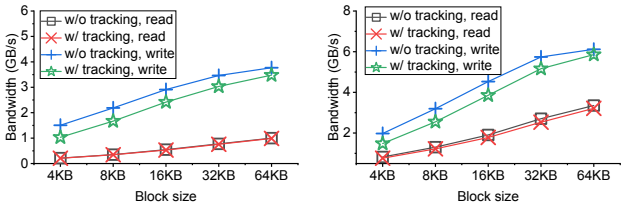
Figure 15: Effects of individual techniques.

Figure 15 shows that designing a dedicated memory pool boosts the performance of some workloads (like *scdir* and *scpdb*), as the memory pool can eliminate page-splitting overhead and reduce multi-process lock contention on free-page lists. However, in some bandwidth-demanding workloads that mostly consist of large I/O requests (like *dir* and *pdb*), allocating from the memory pool at the page granularity can impair the performance compared to the page cache. Using the memory pool only brings a 1.3% improvement on average.

Adding the stream tracking module and stream-based page reclaiming can additionally bring a 21.3% improvement on average. This is because the large problem size triggers background writeback during the checkpoint, and the stream-based page reclaiming decouples the dirty state maintenance from page indexing in the XArray. This lowers the background interference to the foreground I/O requests and enhances overall performance.

Finally, adding the per-file cache can additionally bring a 27.5% improvement because it batches page allocation from the memory pool, providing good cache locality and lowering contention in the memory pool under parallel file I/O requests.

6.4 Stream Tracking Overhead



(a) Uniform distribution. (b) Zipfian distribution.

Figure 16: Stream tracking overhead under random I/O requests.

Finally, there is a concern about whether the stream tracking in StreamCache will introduce significant overhead when the workload is not sequential. We use **fio** to compare the random I/O performance between XFS without stream tracking and with stream tracking. As shown in Figure 16, the stream tracking only introduces minor overhead under random I/O requests. This is because stream tracking is done at the I/O granularity, and this overhead is at least comparable to that of the XArray indexing and page copying, considering that the page cache processes buffered I/O requests at the page granularity.

7 Related Work

Page cache management. Previous works that optimize page cache management mainly fall into two categories. The first category aims to make the page cache index friendly to concurrent accesses by redesigning the page cache index [36, 40, 41] or recording dirty pages in dedicated indexes [34–36]. However, these works do not address the contentions on dirty state maintenance between foreground I/O requests and background writeback. The second category focuses on promoting performance isolation between multiple applications, proposing solutions like per-VM eviction list [45] or weight-aware request scheduling for buffered

I/O [37]. These works are orthogonal to StreamCache, and their techniques can be integrated into our system.

Kernel page allocator. The page allocator is the basic module in the Linux kernel to support memory allocation, and the allocation overhead is vital to application performance. The Linux kernel uses the buddy system to manage free pages and proposes the per-CPU page list to batch page allocation and deallocation [3]. Some works design dedicated memory management modules for the page cache by storing the free pages in multiple double-linked lists [23, 36, 47]. LLFree [55] discards the list-based free page management in the Linux kernel and allocates the free pages with bit-vector searching. StreamCache differs from these works by devising a two-layer memory management method for cache-friendly page allocation.

Designing user space cache. One work [12] points out that the transparent kernel page cache may not be the best choice for database management systems (DBMS), and some works design their own user space cache [38, 39, 57] for DBMS. While the user space cache allows convenient customization and can exploit the user space I/O stacks like the SPDK [19], this solution is not transparent for applications [14], and the cache hit performance can be worse than the kernel cache [33]. Tricache [14] provides a transparent user space cache, but it needs compile-time instrumentation on application source codes and may perform worse than the kernel page cache under cache hits. StreamCache differs from these works in that it does not need any extra work on application programs.

8 Conclusion

In this work, we propose a new page cache management system called StreamCache and implement it in XFS to exploit the performance of fast storage devices. StreamCache can boost the performance of buffered I/O requests under various file scanning workloads with three key techniques: lightweight stream tracking, stream-based page reclaiming, and two-layer memory management. Extensive experiments demonstrate that StreamCache can outperform existing page cache management systems in both real-world applications and various synthetic workloads.

Acknowledgment

We thank all reviewers for their insightful comments and helpful suggestions. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed on June 5, 2024.

[2] Page cache readahead. https://www.halolinux.us/kernel-architecture/page-cache-readahead.html. Accessed on June 5, 2024.

[3] Physical page allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html. Accessed on June 5, 2024.

[4] Safetensors. https://huggingface.co/docs/safetensors/index. Accessed on June 5, 2024.

[5] Linux perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2023. Accessed on June 5, 2024.

[6] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio. Accessed on June 5, 2024.

[7] John Bent, Sorin Faibish, Jim Ahrens, Gary Grider, John Patchett, Percy Tzelnic, and Jon Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*, pages 1–5, 2012.

[8] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pages 1–12, 2009.

[9] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[10] CARV-ICS-FORTH. Fastmap. https://github.com/CARV-ICS-FORTH/FastMap. Accessed on June 5, 2024.

[11] Jonathan Corbet. XFS: The filesystem of the future? https://lwn.net/Articles/476263/, 2012.

[12] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system? In *CIDR*, 2022.

[13] Facebook. Opt-13B. https://huggingface.co/facebook/opt-13b. Accessed on June 5, 2024.

[14] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. Tricache: A user-transparent block cache

enabling high-performance out-of-core processing with in-memory programs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*, pages 395–411, 2022.

[15] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. I/O acceleration with pattern detection. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HDPC '13)*, pages 25–36, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Xiaobin He, Bin Yang, Jie Gao, Wei Xiao, Qi Chen, Shupeng Shi, Dexun Chen, Weiguo Liu, Wei Xue, and Zuo-ning Chen. HadaFS: A file system bridging the local and shared burst buffer for exascale supercomputers. In *21st USENIX Conference on File and Storage Technologies (FAST '23)*, pages 215–230, 2023.

[17] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. Architecture and design of cray datawarp. *Cray User Group (CUG)*, 2016.

[18] IBM. Considerations for the use of direct I/O (O_DIRECT). https://www.ibm.com/docs/en/spectrum-scale/5.0.5?topic=applications-considerations-use-direct-io-o-direct, 2022.

[19] Intel. SPDK: Storage Performance Devlopment Kit. https://spdk.io/. Accessed on June 5, 2024.

[20] Intel. Intel Optane SSD 905P series specification. https://www.intel.com/content/www/us/en/products/sku/147529/intel-optane-ssd-905p-series-960gb-2-5in-pcie-x4-3d-xpoint/specifications.html, 2018.

[21] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 21–32, 2019.

[22] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[23] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proceedings of the ACM on Management of Data*, 1(1):1–25, 2023.

[24] Arch Linux. xfsprogs. https://archlinux.org/packages/core/x86_64/xfsprogs/. Accessed on June 5, 2024.

[25] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*, pages 1–11. IEEE, 2012.

[26] LLNL. PF3DIO benchmark. https://github.com/LLNL/PF3DIO. Accessed on June 5, 2024.

[27] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[28] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, pages 509–518, 1998.

[29] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC '16)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[30] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, fine-grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST '21)*, pages 203–216, 2021.

[31] National Energy Research Scientific Computing (NERSC) Center. Trinity / NERSC-8 Use Case Scenarios. https://www.nersc.gov/assets/Trinity--NERSC-8-RFP/Documents/trinity-NERSC8-use-case-v1.2a.pdf, 2013. Accessed on June 5, 2024.

[32] Yoshihiro Oyama Osamu Tatebe, Shukuko Moriwake. Gfarm/BB—Gfarm file system for node-local burst buffer. *Journal of Computer Science and Technology*, 35(1):61, 2020.

[33] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped I/O on steroids. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*, pages 277–293, 2021.

[34] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*, pages 490–502, 2018.

[35] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. Kreon: An efficient memory-mapped key-value store for flash storage. *ACM Transactions on Storage*, 17(1):1–32, 2021.

[36] 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, 2020.

[37] Jonggyu Park and Young Ik Eom. Weight-aware cache for application-level proportional I/O sharing. *IEEE Transactions on Computers*, 71(10):2395–2407, 2021.

[38] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing*, pages 71–78. IEEE, 2019.

[39] Ivy B. Peng, Maya B. Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. Enabling scalable and extensible memory-mapped datastores in userspace. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):866–877, 2022.

[40] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. Scalecache: A scalable page cache for multiple solid-state drives. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys '24)*, pages 641–656, 2024.

[41] Nick Piggin. A lockless page cache in linux. In *Proceedings of the Linux Symposium*, volume 2. Citeseer, 2006.

[42] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. Combining buffered I/O and direct I/O in distributed file systems. In *22nd USENIX Conference on File and Storage Technologies (FAST '24)*, pages 17–33, 2024.

[43] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K Panda. A 1 PB/s file system to checkpoint three million MPI tasks. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (HPDC '13)*, pages 143–154, 2013.

[44] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3), aug 2013.

[45] Prateek Sharma, Purushottam Kulkarni, and Prashant Shenoy. Per-vm page cache partitioning for cloud computing platforms. In *2016 8th International Conference on Communication Systems and Networks*, pages 1–8, 2016.

[46] THUDM. ChatGLM-6B. https://huggingface.co/THUDM/chatglm2-6b. Accessed on June 5, 2024.

[47] Brian Van Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. DI-MMAP—a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18:15–28, 2015.

[48] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. GekkoFS - A temporary distributed file system for HPC applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER '18)*, pages 319–324, 2018.

[49] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, pages 807–818, 2016.

[50] Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. TRIO: Burst buffer based I/O orchestration. In *2015 IEEE International Conference on Cluster Computing (CLUSTER '15)*, pages 194–203, 2015.

[51] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pages 364–381, 2023.

[52] Wikipeida. Buddy memory allocation. https://en.wikipedia.org/wiki/Buddy_memory_allocation. Accessed on June 5, 2024.

[53] Wikipeida. NVM Express. https://en.wikipedia.org/wiki/NVM_Express. Accessed on June 5, 2024.

[54] Matthew Wilcox. XArray. https://docs.kernel.org/core-api/xarray.html. Accessed on June 5, 2024.

[55] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and optionally-persistent page-frame allocation. In *2023 USENIX Annual Technical Conference (USENIX ATC '23)*, pages 897–914, 2023.

[56] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, Weiguo Liu, and Wei Xue. End-to-end I/O monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, pages 379–394, 2019.

[57] Karim Youssef, Niteya Shah, Maya Gokhale, Roger Pearce, and Wu-chun Feng. Autopager: Auto-tuning memory-mapped I/O parameters in userspace. In *2022 IEEE High Performance Extreme Computing Conference*, pages 1–7. IEEE, 2022.