ScaleCache: A Scalable Page Cache for Multiple Solid-State Drives

Kiet Tuan Pham*[1], Seokjoo Cho*[1], Sangjin Lee*, Lan Anh Nguyen*, Hyeongi Yeo*
Ipoom Jeong‡, Sungjin Lee†, Nam Sung Kim‡, and Yongseok Son*

*Chung-Ang University, †DGIST, ‡University of Illinois Urbana-Champaign

Abstract

This paper presents a scalable page cache called `ScaleCache` for improving SSD scalability. Specifically, we first propose a concurrent data structure of page cache based on XArray (`ccXArray`) to enable access and update the page cache concurrently. Second, we introduce a direct page flush (`dflush`) which directly flushes pages to storage devices in a parallel and opportunistic manner. We implement `ScaleCache` with two techniques in the Linux kernel and evaluate it on a 64-core machine with eight NVMe SSDs. Our evaluations show that `ScaleCache` improves the performance of Linux file systems by up to 6.81× and 4.50× compared with the existing scheme and scalable scheme for multiple SSDs, respectively.

Keywords: Page cache, Solid-state drives, Scalability, Concurrency, Parallelism

1 Introduction

High-performance storage devices such as NVMe-based solid-state drives (SSDs) with GB/s-level I/O bandwidth have been widely adopted in enterprise servers to satisfy ever-increasing performance requirements [22, 57, 63]. For processing massive amounts of data, it is attractive to use multiple SSDs in a RAID configuration, which increases I/O performance, reliability, and capacity [19, 20, 29, 41, 42, 57, 62]. Accordingly, multiple SSDs have been increasingly deployed for large data-intensive applications in supercomputing, big data analytics, graph analytics, enterprise storage, machine learning, and cloud services [6, 28, 29, 36, 37, 40, 54, 56].

Although multiple SSDs provide various advantages for storage systems, the current Linux storage stack exhibits poor performance scalability as it is equipped with more SSDs [20]. Figure 1 shows the performance of diverse Linux file systems and raw device on eight NVMe SSDs running write-intensive workloads. As shown in the figure, the performance of direct I/O in most file systems and that of raw device increases as the number of SSDs increases except for the cases of BTRFS and ZFS. Unlike direct I/O, even if the number of SSDs increases, the performance of buffered I/O with most file systems does not scale well or decreases. This result reveals that buffered I/O which provides many advantages to applications [27] can hinder the SSD scalability when we use multiple SSDs. According to our performance analysis, we observe two key bottlenecks: 1) coarse-grained locking to the data structure (i.e., XArray) of page cache and 2) a delayed and serialized flush operation. For example, multiple application threads concurrently try to insert and delete pages to/from the page cache for I/O operation and page reclamation under a coarse-grained lock. In addition, a single flusher thread flushes dirty pages to the storage devices.

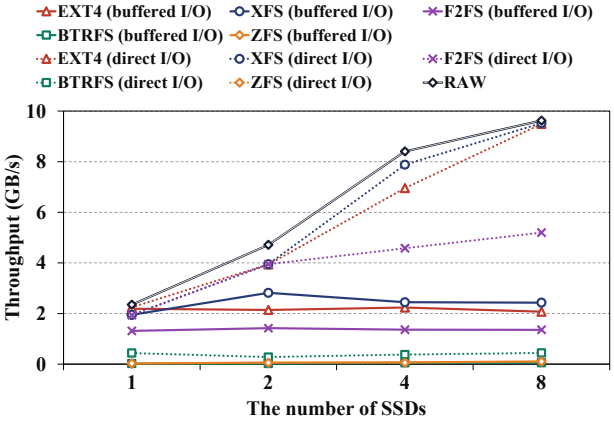


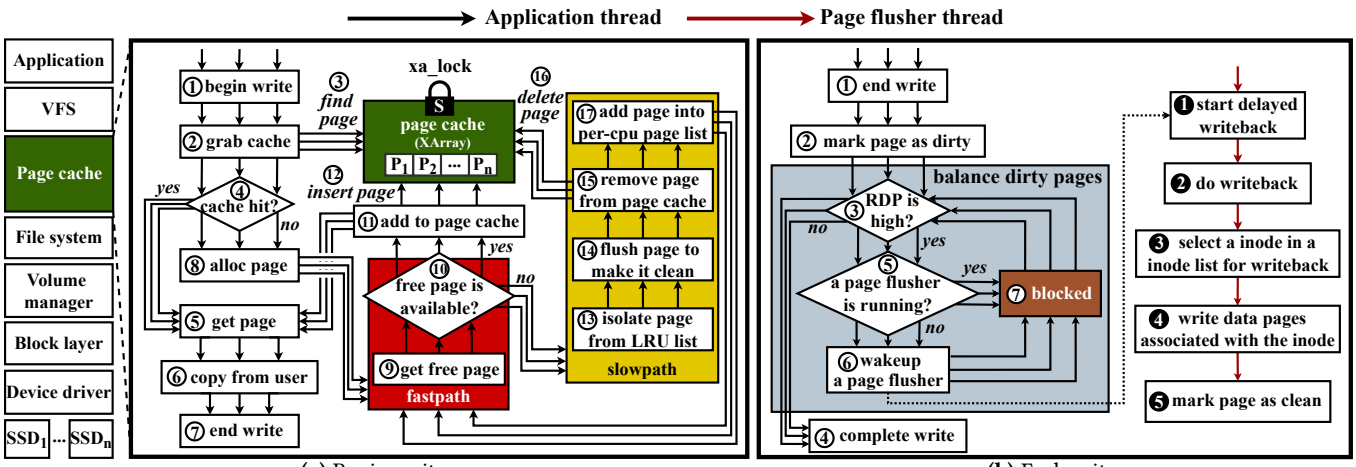
Figure 1. Throughput of buffered I/O and direct I/O with diverse file systems on eight NVMe SSDs in a RAID-0 configuration. The throughput of a single SSD is 2GB/s (i.e., ideal throughput: 16GB/s).

To improve the performance of multiple SSDs, previous studies [20, 64] have investigated the page cache and block layer. Zheng et al. [64] provide a parallel user-space page cache and dedicated I/O threads for each SSD. Falcon [20] proposes a design of per-drive I/O processing in the block layer. Our study is in line with these approaches [20, 64] in terms of investigating the performance of multiple SSDs. In contrast, we focus on the concurrency of data structure (i.e., XArray) and I/O parallelism in the Linux page cache.

[1]Equally contribution.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629588



(a) Begin write **(b)** End write

Figure 2. Procedure of existing page reclamation and flushing (S: spinlock, RDP: ratio of dirty pages).

In this paper, we present a scalable page cache called ScaleCache to improve SSD scalability. We first propose a concurrent data structure of page cache based on XArray called (ccXArray). It enables application threads to concurrently perform XArray operations such as insert, delete, and expand operations without locking. Second, we introduce a direct page flush (dflush) which enables application threads to flush pages to storage devices directly in a parallel and opportunistic manner. For example, when the ratio of dirty pages is high, the threads directly flush the pages to the storage devices without compromising consistency. We implement ScaleCache with two techniques in Linux kernel 5.4.147. We evaluate ScaleCache with EXT4 and XFS file systems on a 64-core machine with 8 Intel Optane 900P SSDs [47] using the micro/macro benchmarks [13, 45, 60] and real-world application [10]. The experimental results show that ScaleCache improves the performance by up to 6.81× and 4.50× compared with the existing scheme and a scalable scheme (i.e., Falcon [20]) for multiple SSDs, respectively. To the best of our knowledge, this is the first study that proposes concurrent XArray to access/update the page cache concurrently to achieve higher I/O performance of multiple SSDs on multi-cores. Finally, we open the source code at https://github.com/syslab-cau/ScaleCache.

2 Background and Motivation

2.1 Page Management in Page Cache

In the Linux kernel, as an important component, the page cache keeps data of files on the memory to minimize I/O operations and provide low latency (i.e., memory latency) and does not require applications to align their I/O size when performing file operations [27]. In addition, the buffered I/O can be helpful to SSD lifespan by reducing the number of I/O generated to storage as much as possible compared with direct I/O. With these advantages, buffered I/O in the Linux kernel is adopted by default. Specifically, under a non-heavy I/O workload where the total workload size does not exceed the page cache size, the performance of buffered I/O is superior to that of direct I/O. On the other hand, under a heavy I/O workload, the performance of direct I/O can be better than that of buffered I/O as shown in Figure 1 since there may not be enough free pages in the page cache to support the buffered I/O operation effectively. In realistic scenarios, there are I/O fluctuations between non-heavy and heavy I/O activities [7, 25, 32, 38, 39, 61]. Thus, under these scenarios, we aim to keep the advantages of buffered I/O while providing a comparable performance to direct I/O. The detailed procedure of buffered I/O is as follows.

Cache hit: As shown in Figure 2a, application threads look up the page cache (①,②). To perform the look-up operations, threads try to search for the pages in their target file's XArray via xas_load() using read-copy-update (RCU) read lock to enable concurrent search operations (③). In this case, XArray receives a page index as input for searching the page and returns the page corresponding to the index. If there are the target pages (i.e., cache hit (④)), the application threads get the pages (⑤) and copy their own contents to the pages without accessing the storage (⑥). Finally, they continue to finish their write operations (⑦).

Fastpath: If there is no requested page, the application threads request to allocate new free pages from a buddy allocator (⑧,⑨). If they get free pages (⑩), they will insert new pages into their associated XArray (⑪). For example, if $page_0$ belongs to $file_A$, $page_0$ is inserted into $XArray_A$ of $file_A$. To perform the insert operations, the threads try to hold an XArray spinlock (xa_lock). Thanks to the RCU-based lookup described above, multiple readers or a single writer with multiple readers can access or access/update XArray concurrently. However, we note that multiple writers require holding a lock (xa_lock) to update the pages in XArray. Subsequently, after a thread holds the lock, it inserts a new page into XArray via xas_store() (⑫).

Slowpath: If application threads cannot get free pages from the free page lists (⑩), the threads will fall into *slowpath*

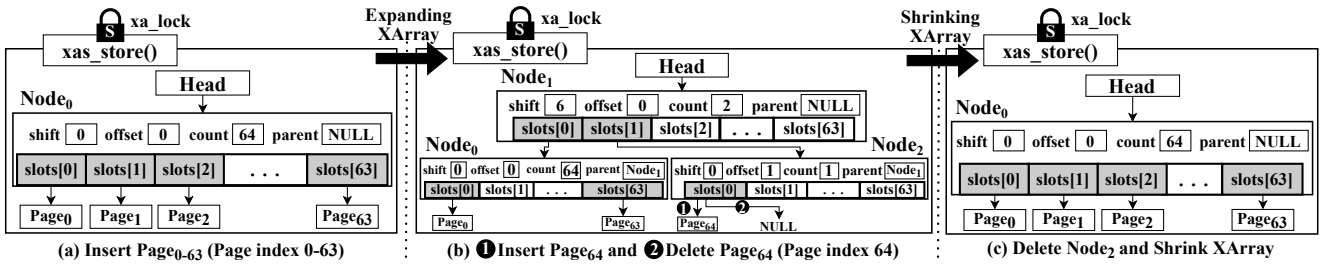


Figure 3. XArray structure and its operations. (a) and (c) denote one depth of XArray and (b) denotes two depths of XArray.

and try their best effort to reclaim new pages. In *slowpath*, the threads first scan and fetch reclaimable pages from Least Recently Used (LRU) lists [2] (⑬). Then, the threads flush the pages to make them clean pages (⑭). And then, the threads try to remove the clean pages from the page cache to use them for their buffered I/O operations (⑮). Like the insert operation, XArray is locked using the XArray spinlock (`xa_lock`) to remove the page from the page cache via `xas_store()` (⑯). Note that the threads can fetch any reclaimable page from LRU lists even if the pages are not associated with their XArray. Thus, the threads can remove the pages on any XArray unlike the insert operation. For example, $thread_0$ and $thread_1$, which try to write $file_A$ and $file_B$ associated with $XArray_A$ and $XArray_B$, respectively, can try to remove the pages from $XArray_C$ simultaneously for their buffered I/O. As a result, this induces the lock contention on XArray. After removing the free pages from XArray, each thread inserts them into its own temporary per-core page lists to use the pages freed by itself immediately (⑰). Since the threads perform the reclaim operations directly in *slowpath*, this procedure is called *direct reclaim*. Finally, each thread gets the free page for buffered I/O from its own per-core page lists (⑨,⑩), inserts the page into its own XArray with the XArray spinlock (⑪), and continues to finish its own write operation (⑤,⑥,⑦).

Balancing dirty pages: When the free pages are available, the application threads get pages and copy their contents to the pages. Then, as shown in Figure 2b, when they try to finish their write operations (①), they first change the state of the written pages to dirty (②). After then, they check whether the RDP exceeds a given threshold or not (③). If the ratio does not exceed the threshold, they finish the write operation (④). Otherwise, they check whether a page flusher thread (`kworker`) is running (⑤). If the flusher thread is not running, application threads try to wake up the flusher thread (⑥). During the flush operations, application threads are blocked for a certain amount of time (i.e., throttling mechanism) (⑦). After the time has elapsed, application threads wake up and check the ratio of dirty pages again to finish their write operations (④). The woken-up flusher thread tries to flush dirty pages in the inodes via a delayed work (❶,❷). Then, the flusher thread selects an inode in the inode list (❸) and writes the dirty pages in the inode to the storage devices (❹). The flusher thread repeats this procedure in a serialized manner until the inode list becomes empty. Finally, it changes the state of pages to clean (❺).

2.2 Data Structure of Page Cache

XArray has become a main data structure within the Linux kernel [59], replacing the radix-tree data structure [58]. XArray functions as an extensive array of pointers and provides an improved interface, iterations, and locking methodology as a part of its API. Specifically, XArray is utilized as a per-file data structure in the page cache and provides memory efficiency, cache friendliness, and RCU-based lookup for fast data retrieval. In this section, we explain how XArray manages the page cache.

XArray consists of a head and XArray nodes (`xa_node`) which are either a leaf node or an inner node. A leaf node stores pages as entries while an inner node stores nodes as entries to link to its child nodes. On a 64-bit system, an XArray node size is 576 bytes, and 7 XArray nodes are allocated in a 4KB page. As shown in Figure 3, each XArray node has its state information such as slots, parent, offset, count, and shift. `slots` is an array to store pages or nodes (i.e., inner or leaf nodes). The number of slots (`XA_CHUNK_SIZE`) in a node is 64 which is determined by a chunk shift (`XA_CHUNK_SHIFT`). For example, `XA_CHUNK_SHIFT` is 6 by default and `XA_CHUNK_SIZE` is 64 calculated by left bit shifting 1 by 6 ($1 \ll 6$). `parent` of a node points to a parent node while `offset` of the node indicates its offset in the parent node's `slots`. `count` is the number of entries in the `slots`. `shift` indicates the number of right-shift operations to be performed in each node to obtain the location in `slots` for a page index (the shift of a leaf node is 0). Whenever the depth of XArray increases, `shift` in a newly created node is increased by the chunk shift. This means that the location in `slots` in the node for a page is determined by its page index with `shift`.

Figure 3 shows an example of how XArray manages pages with a locking mechanism. XArray performs insert and delete page operations via `xas_store()` with `xa_lock`. As shown in Figure 3(a), when application threads insert their own pages ($Page_{0-63}$)[1], the threads try to hold `xa_lock`. If a thread holds the lock, it creates a new node ($Node_0$) and

[1]$Page_0$ means the page of page $index_0$

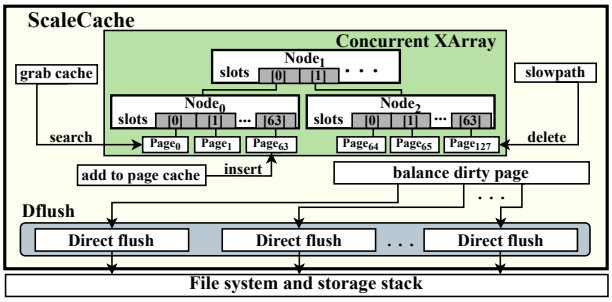


Figure 4. Overall architecture of ScaleCache.

insert its own page into the node (`xas_create()`). After then, other threads try to insert their own pages in the node under the lock. Since `XA_CHUNK_SIZE` is 64 by default, the node's slots can store the 64 pages whose indices from 0 to 63 ($Page_{0-63}$) (the count can become up to 64).

If a page whose index is larger than 63 is inserted, XArray should be expanded. As shown in Figure 3(b), when an additional page ($Page_{64}$) is inserted (❶), there is no node (i.e., $Node_2$) for $Page_{64}$. Thus, XArray is expanded by increasing its depth by allocating a new root node ($Node_1$). The previous node ($Node_0$) and the new node ($Node_2$) are allocated at `slots[0]` and `slots[1]` in the $Node_1$, respectively. Thus, the offsets of $Node_0$ and $Node_2$ are 0 and 1, respectively, based on the location in `slots` in the parent node ($Node_1$). Thus, the shift of $Node_1$ (root node) becomes 6, and the count in $Node_1$ increases to 2. Finally, the page ($Page_{64}$) is inserted into $Node_2$. Note that a page index determines its location in `slots`. `XA_CHUNK_MASK`, a bit-mask used for extracting an offset of an entry in `slots`, is 63 (`XA_CHUNK_SIZE` − 1).

When removing a page ($Page_{64}$) from XArray, the XArray slot of the page to be removed is updated with a deleted mark (e.g., a NULL address), and the count in $Node_2$ decreases to 0 (❷). If the count in the node becomes zero, the node ($Node_2$) is deleted, which decreases the count of root node ($Node_1$) to 1. Then, since the current root node ($Node_1$) is not necessary (i.e., the count of the root node is 1), XArray is shrunk while $Node_1$ is deleted. Finally, $Node_0$ becomes the root node again as shown in Figure 3(c).

2.3 Challenges in Page Cache Management

As shown in Figure 1, under large write-intensive workloads, adopting buffered I/O with the page cache hinders SSD scalability when we use multiple SSDs in a RAID configuration. There are the root causes as follows.

Limited concurrency: Application threads frequently update (e.g., insert/delete) the page cache structure (i.e., XArray) with a non-scalable spinlock (`xa_lock`) as shown in Figure 2. This spinlock prevents conflicts between multiple writers on XArray, resulting in high lock contention.

Limited I/O parallelism: Only a single flusher can perform the I/O operation at a time while blocking multiple application threads. This serialized I/O strategy limits the I/O parallelism offered by a group of SSDs.

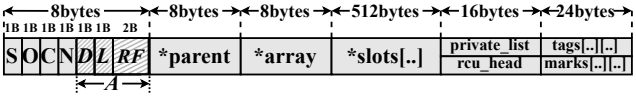


Figure 5. `ccXArray` node structure (The size is 576 bytes which is identical to the existing XArray node, S: shift, O: offset, C: count, N: nr_values, *D*: delete flag, *L*: ldflag, *RF*: reference count, *A*: alignment).

As described above, we have identified two main bottlenecks and measured their overheads which are reported in Table 2 in §4.7. As shown in the table, when considering the sequence of bottleneck points, the issue of limited I/O parallelism due to the serialized I/O strategy is first induced. When I/O parallelism increases, the bottleneck shifts to XArray, resulting in the limited concurrency by the non-scalable spinlock to protect XArray.

3 Design and Implementation

3.1 Overall Architecture

To achieve higher SSD scalability on multi-cores, we aim to reduce the lock contention and maximize I/O parallelism in the page cache. To do this, we propose a concurrent and I/O parallelized page cache to scale buffered I/O performance close to direct I/O performance while obtaining the advantages of buffered I/O. Figure 4 shows an overall architecture of `ScaleCache`. In `ScaleCache`, we first establish a concurrent data structure of page cache by devising a concurrent XArray (`ccXArray`) via a concurrent mechanism with atomic instructions. Thus, with `ccXArray`, application threads can update the page cache concurrently without the XArray spinlock (i.e., `xa_lock`). Second, we present a direct page flush (`dflush`) to enable application threads to balance dirty pages directly without blocking when the ratio of dirty pages is high. Thus, instead of a serialized flush operation, this enables parallel flush operations in an opportunistic manner.

To guarantee the correctness of our `ScaleCache` while improving concurrency and parallelism, we satisfy four properties as follows:

Property 1: All nodes created in `ccXArray` are always unique.

Property 2: A logically deleted node in `ccXArray` is invisible to the application and the node can be visible again only if the node is reused.

Property 3: An empty node in `ccXArray` is deleted physically only when there are no more insert/search operations.

Property 4: `dflush` flushes dirty pages even in parallel without sacrificing the consistency of files.

We will describe our `ScaleCache` with these properties in more detail in the following sections.

3.2 Concurrent XArray

To design a concurrent XArray, we leverage four main key strategies as follows.

Algorithm 1 Simplified procedure of entry insertion and deletion

```
1: function CC_XAS_STORE(index, new_entry, head)
2:     {leaf_node, old_entry} ← CC_CREATE_NODE(index, head) ▷ Get leaf node
3:     if new_entry ≠ NULL and old_entry ≠ NULL then
4:         return old_entry                ▷ Page already exists in the insertion case
5:     end if
6:     offset ← (index & XA_CHUNK_MASK)
7:     current ← CAS(leaf_node.slots[offset], old_entry, new_entry)
8:     if current == old_entry then
9:         if new_entry ≠ NULL then                              ▷ Insert a page
10:            atomic_add(leaf_node.count, 1)
11:        else if new_entry == NULL then                        ▷ Delete a page
12:            count ← atomic_sub_and_read(leaf_node.count, 1)
13:            if count == 0 then
14:                CC_LOGICAL_DELETE_NODE(leaf_node)
15:            end if
16:        end if
17:    end if
18:    PUT_ALL_NODES()
19:    return current
20: end function
```

Winner strategy: In the existing page cache, XArray creates/inserts node(s) or inserts/deletes page(s) with a non-scalable lock. To enable a concurrent page insertion/deletion and node creation/insertion, `ccXArray` adopts a winner strategy under competing threads [1] while keeping ***Property 1***. With this strategy, we elect a winner among threads concurrently running on `ccXArray`, allowing the winner to insert/delete a page within the node or create/insert a node in `ccXArray`.

Lazy node deletion and reuse: `ccXArray` deletes a leaf node or an inner node if there are no pages in the leaf node or no child nodes in the inner node, respectively. For a concurrent node deletion, we adopt a lazy node deletion and reuse strategy [9, 31] while keeping ***Property 2 and 3***. In this strategy, (1) lazy deletion delays deleting a node physically and (2) the node is reused if it is requested before being deleted physically. More specifically, the lazy deletion divides a delete process into two separate steps. In the first step, instead of directly freeing the node, we once atomically mark the node as logically deleted when there are no pages in the node [9]. This allows the node to be not visible to applications so that `ccXArray` can always have a consistent view during search operations. In the second step, we delete the node physically in a certain situation where any page cannot be inserted/searched in `ccXArray`. We note that this strategy can increase memory consumption in a particular situation, however, it can be effective under heavy write-intensive workloads since the pages can be frequently deleted or inserted.

Per-thread node access tracking: To avoid read/write and write/write conflicts when accessing nodes, we devise a per-thread node tracking list which does not require any lock. With this strategy, whenever the threads access or update to `ccXArray`, we can track the access of all the nodes by inserting the node into the per-thread list in an access order. It enables catching how many threads using a particular node, and we can prevent page faults or control the access on each node assisted by this strategy.

Algorithm 2 Simplified procedure of node creation

```
1: function CC_CREATE_NODE(index, head)
2:     root ← GET_NODE(cc_xas_expand(index, head))        ▷ Expand XArray
3:     shift ← root.shift + XA_CHUNK_SHIFT
4:     curr_node ← root
5:     offset ← ((index ≫ curr_node.shift) & XA_CHUNK_MASK)
6:     entry ← curr_node.slots[offset]
7:     while shift ≠ 0 do          ▷ Descend until reaching the target leaf node
8:         shift ← shift − XA_CHUNK_SHIFT
9:         if entry == NULL then                              ▷ Create child node
10:            new_node ← GET_NODE(alloc_node())
11:            new_node.shift ← shift
12:            entry ← CAS(curr_node.slots[offset], NULL, new_node)
13:            if entry ≠ NULL then
14:                free_node(new_node)
15:                curr_node ← GET_NODE(entry)
16:            else
17:                curr_node ← new_node
18:            end if
19:            atomic_add(curr_node.parent.count, 1)
20:        else                                               ▷ Follow child node
21:            curr_node ← GET_NODE(entry)
22:            while atomic_read(curr_node.ldflag) == ON do
23:                Wait for logical node deletion
24:            end while
25:            if CAS(curr_node.del, ON, OFF) == ON then      ▷ Reuse
26:                atomic_add(curr_node.parent.count, 1)
27:            end if
28:        end if
   descend:
29:        offset ← (index ≫ curr_node.shift) & XA_CHUNK_MASK
30:        entry ← curr_node.slots[offset]
31:    end while
32:    return curr_node and entry
33: end function
```

Algorithm 3 Simplified procedure of managing node references

```
1: function GET_NODE(node)
2:     Register this node into per-thread node tracking list
3:     atomic_add(node.refcnt, 1)
4:     return node
5: end function
6: function PUT_ALL_NODES()
7:     while list is not empty do
8:         Unregister node from per-thread node tracking list in a reverse order
9:         atomic_sub(node.refcnt, 1)
10:    end while
11: end function
```

`ccXArray` node structure: To realize the above strategies, we introduce three indicators in `ccXArray` node. As shown in Figure 5, to indicate a logically deleted node, we use a delete flag (i.e., `del`) by embedding it in the node structure. To notify that a node is undergoing logical deletion, we employ a logical deleting flag (i.e., `ldflag`). To track threads referencing the node, we adopt a reference count (i.e., `refcnt`). By utilizing unused space within the node data structure (i.e., `struct xa_node`), the size of the `ccXArray` node is still the same as that of the existing XArray node. This shows efficient space utilization while realizing our strategies and keeping the consistency of XArray.

3.2.1 Page Insertion and Deletion.

In this section, we describe how `ccXArray` supports concurrent page insert and delete operations. In current XArray, the pages are inserted/deleted into/from the slots of XArray according to their page indexes via `xas_store()` with `xa_lock`, as described in §2.2. Similar to the existing scheme

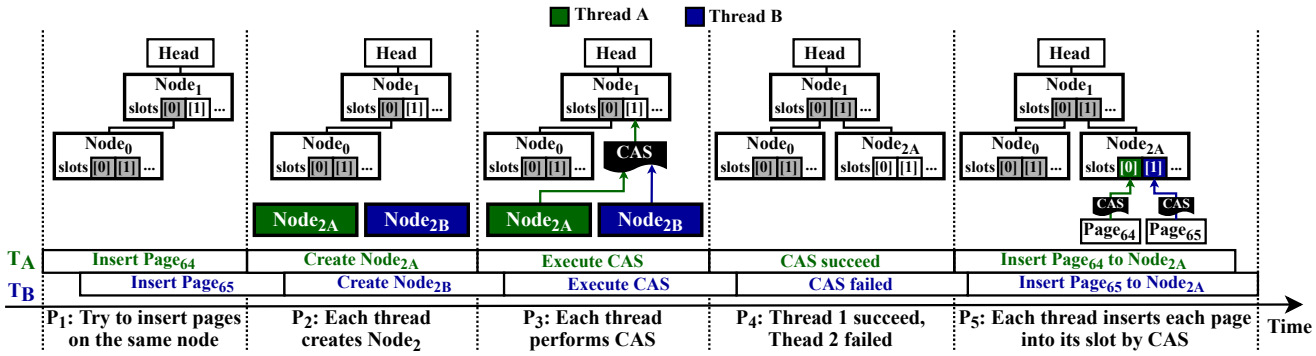


Figure 6. Concurrent page insertion and node creation in `ccXArray` (T_A: Thread A, T_B: Thread B, $Node_{2A}$ denotes $Node_2$ created by Thread A, $Node_{2B}$ denotes $Node_2$ created by Thread B, P_X: Phase X).

but without locking, `ccXArray` enables the slots to be updated concurrently with incoming pages or the deleted mark using a concurrent mechanism leveraging atomic instructions. In this manner, our mechanism resolves the conflict issue between the insert and delete operations. Algorithm 1 shows our concurrent page insertion and deletion mechanism with `ccXArray`.

Page insertion: Whenever application threads try to insert their own pages via `cc_xas_store()`, each thread has its own page index (`index`) which determines the slot location (`offset`). The new entry (`new_entry`) denotes a page to be stored in the slot (Algorithm 1, line 1). Starting from the head, threads descend toward each leaf node containing their target slots by calling `cc_create_node()` in Algorithm 2 (line 2). By doing so, at the given page index, each thread can create or find out its own leaf node (`leaf_node`) and get the entry (`old_entry`) within the leaf node. If the entry (`old_entry`) already exists in the slot (line 3), to avoid overwriting the existing page, the thread does not insert its own page (`new_entry`) and tries to search the existing page (line 4). Otherwise, each thread tries to store its own page in the slot corresponding to the page index (lines 6–7). Using an atomic Compare-And-Swap (CAS) instruction, the thread compares the current entry in the slot with the existing entry (`old_entry`). If they are the same (CAS-succeeded), the CAS-succeeded thread updates the slot by its page and increases node count (`leaf_node.count`) (lines 8–10). Otherwise, the failed threads try to search the existing page (line 19).

Page deletion: In `ScaleCache`, when application threads perform direct reclaim in *slowpath*, they delete the pages in `ccXArray` concurrently by storing deleted marks (e.g., a NULL address) at their corresponding slots. Page deletion is performed in the almost same manner as page insertion via `cc_xas_store()` as shown in Algorithm 1. Instead, in the case of page deletion, each thread tries to store the deleted mark (`new_entry`) in the slot of the page index (line 1). The following procedure is identical to that of insertion (lines 2–7). After a thread succeeds in the CAS operation (line 8), it decreases the number of entries (`leaf_node.count`) atomically (lines 11–12). Then, the thread checks whether the node is empty or not (line 13). If the node is empty, it tries to delete the node logically (line 14) which will be described in Algorithm 4.

3.2.2 Node Creation and Expanding `ccXArray`.

We use Algorithm 2 and Figure 6 to elaborate node insertion.

Node creation: As shown in Algorithm 2, when application threads try to insert their own page to `ccXArray`, they create nodes for their own page indexes (Algorithm 2, line 1). First, each thread checks whether `ccXArray` should be expanded or not via the expand operation (`cc_xas_expand()`) (line 2). Note that, to avoid conflicts, whenever each thread accesses a node, it records the node access by inserting the node (i.e., pointer) to the node tracking list (Algorithm 3, lines 1–5). After getting the node, each thread determines the required number of iterations (`shift`) to descend to the target leaf node (line 3). Then, each thread starts descending from the root node to its leaf node (lines 4–31). If a target node (e.g., inner or leaf node) does not exist (line 9), each thread creates and gets (line 10) its own node and tries to insert its own node at the slot using a CAS operation (lines 11–12). By doing so, only the CAS-succeeded thread can insert its created node to the slot. The CAS-failed threads cannot insert their node, free the created nodes, and use the node of the CAS-succeeded thread to descend (lines 13–15).

If a node already exists at the slot (line 20), they get the existing node (line 21) and check if this node is tried to be deleted logically (line 22) according to our lazy node deletion strategy (§3.2.3). If so, they wait for the logical node deletion procedure to be finished (line 23). It is necessary to avoid the conflict between threads accessing the node and deleting the node. Accessing to the node under logical deletion before the logical deletion is finished can make threads misunderstand the node as not being deleted. By removing this conflict, we ensure consistent node deletion. According to our reuse strategy (§3.2.3), if the node is already logically deleted, the node can be reused again via CAS operation (lines 25–26). This CAS operation protects the node to be reused again

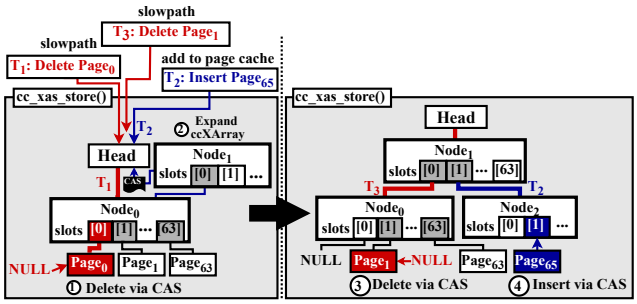


Figure 7. Concurrent expand operation (T_X: $Thread_X$).

until it is logically deleted again. After getting the node, each thread repeats this procedure until reaching the leaf node (lines 29–30 and line 7). Finally, they acquire their leaf nodes and entries (line 32).Note that after the threads finish their work (Algorithm 1, line 18), the threads put all of their referencing nodes by deleting the accessed node points from the list in the FILO manner and decreasing the node reference count via `put_all_nodes()` (Algorithm 3, (lines 6–11)).

Figure 6 shows a simplified example of concurrent page insertion and node creation in `ccXArray`. As shown in the figure, when application threads (T_A and T_B) try to insert their pages ($Page_{64}$ and $Page_{65}$) in `ccXArray`, respectively, there are two nodes (root node: $Node_1$, leaf node: $Node_0$). In $Phase_1$, T_A and T_B get the root node (i.e., $Node_1$) from `ccXArray` head. After calculating the slot offsets based on their page indexes, they have to descend through `slots[1]` in $Node_1$ because $Page_{64}$ and $Page_{65}$ should be inserted into $Node_2$ to be allocated. In $Phase_2$, since `slots[1]` does not have its node, T_A creates $Node_{2A}$ and T_B creates $Node_{2B}$ simultaneously. Note that $Node_{2A}$ and $Node_{2B}$ are the same nodes to be $Node_2$ but only one node finally will be inserted into the slot. In $Phase_3$, both threads execute the CAS operation. Then, in $Phase_4$, since T_A succeeds, $Node_{2A}$ becomes a final node to be inserted into the slot. Meanwhile, the CAS-failed thread (T_B) destroys $Node_{2B}$. This strategy demonstrates that only a winner can get an opportunity to put the node to the slot (***Property 1***). Finally, in $Phase_5$, each thread tries to insert its own page into its slot in $Node_{2A}$ via the CAS operation.

Expanding `ccXArray`: When a page index is out of range in current nodes, `ccXArray` should be expanded like XArray. Unlike XArray, we concurrently expand the depth of `ccXArray` via `cc_xas_expand()`. Figure 7 shows a concurrent expand operation while delete and insert operations are performed simultaneously. As shown in the left side of the figure, thread T_1 tries to delete $Page_0$ by updating *slots[0]* in $Node_0$ with a NULL address while T_2 tries to expand `ccXArray` to insert $Page_{65}$ (①,②). Since there is no node for $Page_{65}$ to be inserted, T_2 tries to increase depth of `ccXArray` by creating $Node_1$ concurrently via the CAS operation (②). Meanwhile, T_1 deletes $Page_0$ in the root node ($Node_0$) through *head* right before T_2 performs the CAS operation (①). As shown in the right side of the figure, after T_2 expands `ccXArray`, T_3 tries to delete $Page_1$ in $Node_0$ while T_2 tries to insert $Page_{65}$. Unlike T_1 in left side of the figure, T_3 accesses the slots of $Page_1$ through $Node_1$, since the root node has been updated to $Node_1$ instantly. Even if T_3 arrives before the root node is updated, this is safe since the delete operation by T_3 is performed in $Node_0$ through *head*. Then, T_3 deletes $Page_1$ by updating the slot with the NULL address while T_2 inserts $Page_{65}$ in the slot via the CAS operations (③,④). By doing so, in spite of during expanding nodes, `ccXArray` does not block the insert or delete operations.

Algorithm 4 Simplified procedure of logical node deletion

```
1:  function CC_LOGICAL_DELETE_NODE(node)
2:      while node is not root node do
3:          if atomic_read(node.count) ≠ 0 then
4:              return
5:          else if CAS(node.ldflag, OFF, ON) ≠ OFF then
6:              return
7:          else if atomic_read(node.parent.refcnt) > 1 then
8:              CAS(node.ldflag, ON, OFF)
9:              return
10:         else if CAS(node.del, OFF, ON) ≠ OFF then
11:             return
12:         end if
13:         atomic_sub(node.parent.count, 1)
14:         CAS(node.ldflag, ON, OFF)
15:         node ← node.parent
16:     end while
17: end function
```

3.2.3 Node Deletion and Shrinking `ccXArray`.

Logical node deletion: Algorithm 4 shows how `ccXArray` deletes nodes logically. Whenever application threads delete their pages, they check if there are no pages in the node (Algorithm 1, line 13) and if so, they try to delete their nodes logically. As shown in Algorithm 4, the thread first rechecks for any potential page insertion (line 3). Once the thread confirms that the target node is empty, it informs other upcoming threads the target node will be tried to be deleted logically by using the CAS operation (line 5). This guarantees that only a single thread can try to delete the node logically. By doing so, other threads wait for this deletion to be finished (Algorithm 2, line 22 and Algorithm 5, line 5).

Note that, even if the target node is tried to be deleted logically, if there is a potential conflict, the logical deletion will not progress further. For example, if the parent of the target node is already referenced from other threads (i.e., `node.parent.refcnt>1`) (line 7), the logical deletion stops (line 8) since a potential reference to the target node can take place. On the other hand, if no other threads are concurrently referencing the parent of the target node (i.e., `node.parent.refcnt=1`), the deleting thread can delete the target (child) node logically and safely. This is guaranteed by two rules of the accessing node in `ccXArray` as follows: 1) all threads (i.e., insert, search, delete) traverse from the root node to the leaf node (i.e., the node reference count increases in order from root to leaf node) and 2) delete threads start to delete from the leaf node to the root node.

The target node can be deleted logically only if these conditions (line 3, 5, and 7) are satisfied (line 10). Since the target

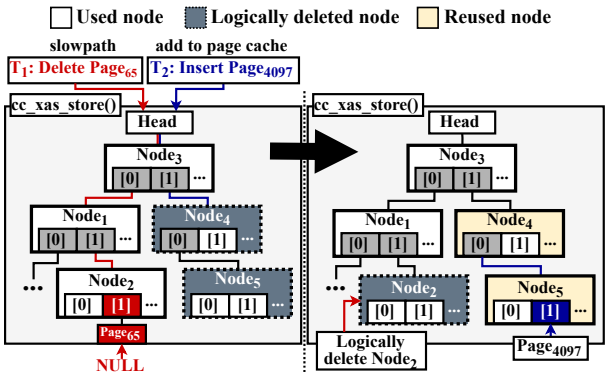


Figure 8. Logical node deletion and reuse ($Node_2$ is logically deleted and $Node_4$ and $Node_5$ are reused).

node is now considered deleted, the number of nodes of the target node's parent can be decreased (line 13). Finally, the logical deletion for the target node is finalized by using the CAS operation (line 14), and the thread moves on to the parent node (line 15) until it reaches the root node (line 2).

Figure 8 shows logically delete and reuse nodes. As shown in the left side of the figure, T_1 tries to delete a page ($Page_{65}$) and T_2 tries to insert another page ($Page_{4097}$) simultaneously. In this case, T_1 deletes the page by atomically updating its slot by a NULL address. Meanwhile, in the case of T_2, there are logically deleted nodes ($Node_4$ and $Node_5$). After then, as shown in the right side of the figure, $Node_2$ is logically deleted since there is no page in the node. Meanwhile, T_2 re-activates $Node_4$ as an inner node and $Node_5$ as a leaf node and inserts its page in the slot in $Node_5$ (***Property 2***).

Shrinking `ccXArray`: When a file is not used anymore, the lazy node deletion strategy physically removes the logically deleted nodes of the file at the safe point. For example, a file can be opened and shared by multiple threads. After file operations, the file can also be closed by multiple threads. While closing the file, when the file is not shared with the other threads, the last thread releases the file. At this point, there will be no further page insertion/search and node insertion/reuse operations in `ccXArray`. As a result, there is only page/node deletion. By using this property of file release, we can delete the nodes physically in two cases. First, the last closing thread which releases the file deletes all the logically deleted nodes in `ccXArray` physically. Second, even though the file is released, the pages of the file can be still deleted from *slowpath* as described in §2.1. Thus, in this case, the thread deleting the last page in the node deletes the node physically. In both cases, we can accomplish the safe node deletion by checking whether there are still working threads on the target node by checking the node's reference count. If there is any thread, after we wait for the thread to finish its work, we delete the logically deleted nodes physically. By doing so, we call these points *safe point* where we can safely remove the nodes physically or shrink `ccXArray`.

Figure 9 describes the example of deleting nodes physically and shrinking `ccXArray` when calling `close()`. As

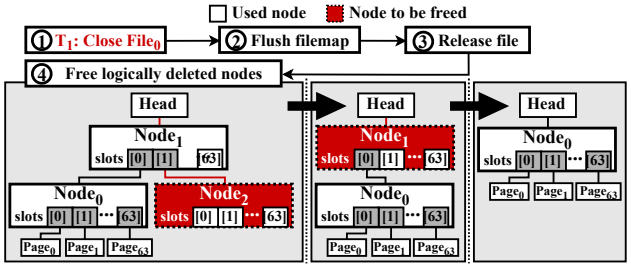


Figure 9. Physical node deletion and `ccXArray` shrinkage ($Node_2$ is physically deleted).

shown in the figure, in existing file systems, thread T_1 tries to close $File_0$ (①). During this task, T_1 flushes the filemap, where all dirty pages of $File_0$ in the page cache are being flushed (②). After then, T_1 releases $File_0$ only when there are no other threads sharing the file anymore while the write counter of the file's inode (`inode->i_writecount`) becomes 1 (③). `ccXArray` uses this file release point as a safe point to delete the nodes physically since there are no insert/search operations to the file (④). Thus, as shown in the figure, T_1 can delete $Node_2$ physically and safely during releasing the file (***Property 3***). Then, $Node_1$ becomes the root node by deleting the $Node_2$. After deleting $Node_2$, T_1 starts to shrink `ccXArray` of the file since the root node ($Node_1$) has only one child node ($Node_0$). By shrinking `ccXArray`, $Node_0$ finally becomes the root by deleting the $Node_1$. Consequently, `ccXArray` can perform the shrink operation while avoiding the lock contention via the lazy deletion and reuse strategy.

3.2.4 Node and Page Search.

Unlike existing XArray, since `ccXArray` adopts the lazy deletion and reuse strategy, a logically deleted node should be invisible when the node is searched. Algorithm 5 shows how `ccXArray` deals with deleted nodes when searching for a page. As shown in the algorithm, application threads start searching for their pages from the head (line 2). The searching threads first get the root node where they increase the reference count in the node and register it to their own node tracking list (line 4). After that, they wait for the deleting thread if one exists (lines 5–7). This ensures that the searching thread does not proceed further until the deleting thread finishes its work. Only after the node is not occupied by the deleting thread, the searching threads check if the node is logically deleted (lines 8–10). If the node has been deleted logically (i.e., `node.del` is *ON*), the threads stop descending. By doing so, the node is invisible to the application (***Property 2***). Otherwise, they descend toward child nodes to search for their pages by atomically fetching the next target node at a lower depth (lines 11–12). They repeat this iteration while they find out the node in their descending slots (line 3). Finally, if they have reached the leaf node, they get their target pages (lines 13–15). Finally, each thread untracks all accessed nodes (line 17). By doing so, the logically deleted nodes cannot be accessed unless it is reused.

Algorithm 5 Simplified search procedure

```
1: function CC_XAS_LOAD(index, head)
2:     entry ← head                          ▷ Top of the tree: root node or NULL
3:     while entry is a node do
4:         node ← GET_NODE(entry)
5:         while atomic_read(node.ldflag) == ON do
6:             Wait for logical node deletion
7:         end while
8:         if atomic_read(node.del) == ON then      ▷ Logically deleted node
9:             return NULL
10:        end if
11:        offset ← (index ≫ node.shift) & XA_CHUNK_MASK
12:        entry ← atomic_read(node.slots[offset])
13:        if node.shift == 0 then                  ▷ Leaf node
14:            goto found
15:        end if
16:    end while
   found:
17:    PUT_ALL_NODES()
18:    return entry
19: end function
```

3.3 Direct Flush

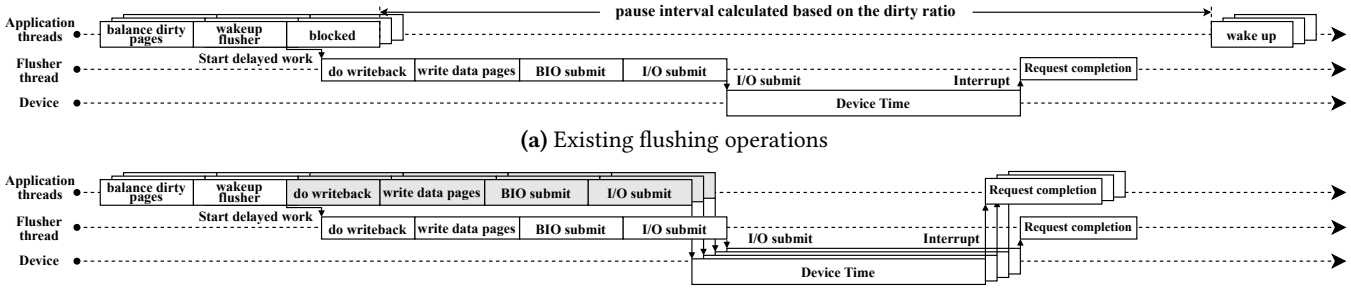
Throttling and balancing mechanism: Since pages are limited resources, the written pages should be reclaimed at some point for other purposes or uses. When the dirty page ratio is high and there are many requests for free pages, the system should quickly flush dirty pages to the backing device in order to fulfill the requests promptly. Accordingly, an efficient and fast page flushing mechanism is required. Thus, for flushing the pages to HDD, the Linux kernel adopts a throttling mechanism with a page flusher which adjusts the number of I/Os, collects the I/Os, and submits them by considering the dirty page ratio in the page cache. This leads to many benefits for single-channel HDD. Specifically, the throttling mechanism makes serialized I/O and sequential I/O patterns and reduces the amount of I/Os to HDD as much as possible. In addition, the mechanism blocks the application threads for the flushing operations. There are two reasons for the blocking operation as follows. (1) It prevents application threads from generating dirty pages anymore to get free pages. (2) Multiple flushing operations with multiple flushers can negatively affect the performance of a single-channel device.

As we discussed above, the existing throttling mechanism is useful for flushing the pages to block devices with long latency. However, the throttling mechanism can negatively throttle the parallelism of SSDs, especially for multiple SSDs in a RAID configuration which provide high parallelism with multiple channels. There are three potential negative effects of throttling mechanism on multiple SSDs as follows. (1) Even if the multiple channels are supported by multiple SSDs, since the existing throttling provides a single page flusher, the flusher only uses a channel utilization at a time. (2) The existing blocking operation which blocks the application threads hinders the opportunity to flush more dirty pages per unit time. (3) The blocking time which blocks application threads can be longer than the time required for I/O operation in the case of low-latency SSDs. Consequently, `ScaleCache` aims to enable direct and parallel flush operations without blocking by eliminating this throttling mechanism but uses the existing balancing algorithm with the existing RDP.

Direct flush (dflush): Direct flush (`dflush`) technique in `ScaleCache` allows the application threads to directly flush the dirty pages to the storage devices in parallel instead of blocking them. In the existing page cache, a single page flusher thread (`kworker`) exists per a RAID of SSDs and it is in charge of flushing dirty pages. The flusher is woken up by application threads when the current dirty page ratio in the system exceeds a threshold calculated by a specific ratio (`vm.dirty_background_ratio`). Thus, if the dirty ratio increases quickly, the flusher thread cannot catch up and exceed the threshold. In this case, the page cache blocks the application threads since the threads cannot obtain free pages anymore (throttled). In addition, the balancing dirty page algorithm determines pause intervals for the threads to be blocked and reduces the current dirty page ratio in the system. This serialized and blocking-based I/O mechanism is suitable and enough for a RAID of multiple HDDs which has low parallelism. However, for multiple SSDs, since they provide high parallelism, only a single flusher thread is insufficient.

To exploit the parallelism, we adopt multiple consumers and multiple producers strategy while keeping ***Property 4*** instead of a single consumer (`kworker`)/multiple producers. Whenever the ratio of dirty pages is high, `dflush` enables parallel and opportunistic I/O with application threads (i.e., multiple consumers) similar to the existing *direct reclaim* scheme in *slowpath*. By doing so, the threads can flush the dirty pages as a co-worker when needed. Figure 10 shows the timeline of balancing dirty pages between the existing page cache and `ScaleCache`. As shown in Figure 10a, in the existing page cache, the threads perform balancing dirty pages by waking up a page flusher thread (`kworker`). Later, the threads are blocked by calling `io_scheduler_timeout()` which induces the blocking time and context switching overhead. Finally, the flusher thread starts a delayed work to perform writeback. To do this, it fetches dirty inodes and flushes their dirty pages to storage devices (i.e., write data pages, BIO submit, and I/O submit). After the interrupt handler completes, the flusher thread finishes the post-I/O processing. Finally, the application threads wake up after the pause interval passes and try to get the free pages.

As shown in Figure 10b, in `dflush`, like the existing flushing operation, when the ratio of dirty pages is high, the application threads start balancing dirty pages by waking up the flusher thread which starts the delayed work of flushing operations. However, unlike the existing scheme, they directly perform the flushing operations instead of the delayed work while threads are not blocked without calling `io_scheduler_timeout()`. At this time, `dflush` enables to map each consumer thread for each inode by fetching the dirty inodes. Thus, each consumer completely flushes the



(a) Existing flushing operations

(b) Flushing operations of `dflush`

Figure 10. Flushing operations in existing flush and `dflush`.

dirty pages of each inode and makes them clean in parallel without stopping in the middle of the flushing work. Therefore, the flushing procedure is still consistent even if our I/O operations are parallelized (***Property 4***). This demonstrates `dflush` opportunistically allows the application threads to perform the flush operation directly and parallelize the I/O operations. As a result, `dflush` reduces the thread blocking time and context switching overhead when balancing dirty pages.

4 Evaluation

We run all experiments on a 64-core machine with four Intel Xeon Gold 6242 processors (with hyperthreading disabled) and 64GB DRAM. For storage, the machine has 8 Intel Optane 900P NVMe SSD [47] (2GB/s stable write throughput per SSD) and runs Ubuntu 20.04 LTS distribution with Linux kernel 5.4.147. To evaluate the existing page cache and `ScaleCache`, we use two common file systems: EXT4 [30] and XFS [53] since they are popular file systems used by many Linux distributions and storage backends [15, 17]. We run random and sequential writes using FIO [13] as a micro-benchmark, filebench workloads [60] and FFSB [45] as macro-benchmarks, and YCSB workloads with RocksDB as a real-world application under the different number of SSDs using Linux software RAID (MD-RAID) [24]. We run each test ten times and report the average results.

4.1 Microbenchmark

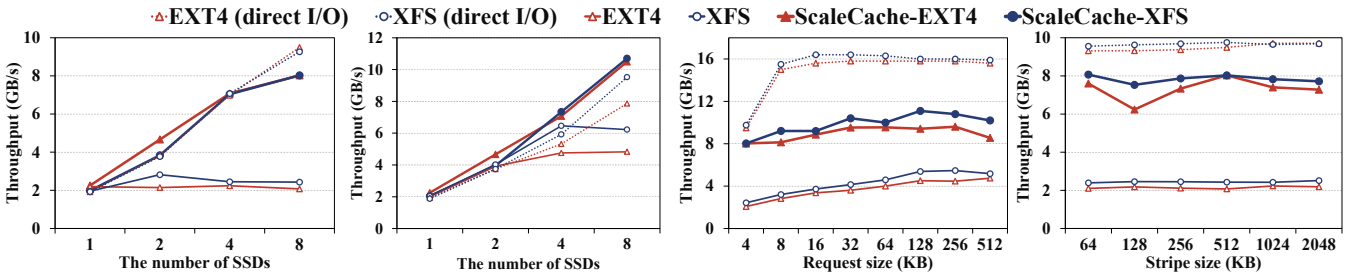
We present the performance results of the existing page cache and `ScaleCache` with two file systems in a RAID-0 configuration. We use FIO benchmark with 64 threads (the number of cores), 3GB file size per thread, 4KB request size[2]. We use the default stripe size (512KB) in all cases except for evaluating the performance on various stripe sizes. We use this FIO configuration in all the evaluations unless stated otherwise.

Random and sequential writes: As shown in Figures 11a and 11b, `ScaleCache` scales well while the existing page cache shows almost the same or even decreased performance as the number of SSDs increases. For random writes, `ScaleCache-EXT4`/`ScaleCache-XFS` improve the performance by up to 3.87×/3.30× compared with EXT4/XFS in the case of 8 SSDs, respectively. These results show that high concurrency and I/O parallelism at the page cache increase the performance almost linearly as the number of SSDs increases. We achieve higher performance up to 8221 MB/s and 10.7 GB/s in the case of random and sequential writes, respectively. `ScaleCache` demonstrates buffered I/O can reach to or even outperform direct I/O (9488 MB/s and 9753 MB/s) in the case of random and sequential writes, respectively, while we still obtain the advantages of buffered I/O.

Various request sizes: Figure 11c shows the random write performance with different request sizes on 8 SSDs. As expected, the throughput of `ScaleCache` and existing page cache increases as the request size becomes larger. In our evaluation, the throughput in both schemes is saturated at 128KB. `ScaleCache-EXT4` and `ScaleCache-XFS` improve the throughput by up to 2.08× and 2.06× compared with the existing page cache with EXT4 and XFS at 128KB, respectively. In terms of the direct I/O performance, starting from an 8KB request size, direct I/O has already reached the maximum throughput of multiple SSDs (i.e., the ideal throughput with 8 devices is 16GB/s). It is because direct I/O immediately writes the 8KB-sized I/O requests (i.e., larger-sized I/O requests) without buffering, reducing the overall number of I/O operations. Meanwhile, since buffered I/O selects dirtied 4KB pages buffered in the page cache when flushing, the selected pages cannot be combined unless they are continuous [49, 51]. Thus, buffered I/O can generate relatively smaller-sized I/O requests, increasing the overall number of I/O operations.

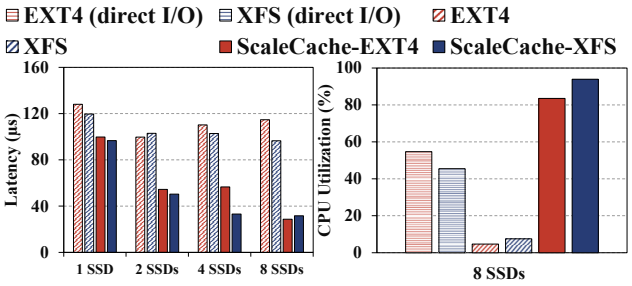
Various stripe sizes: Figure 11d shows the results of the random write workload with different stripe sizes on 8 SSDs. The performance is almost the same or decreased even if the stripe size increases. Generally, the default 512KB stripe size shows the best throughput in both the existing page cache and `ScaleCache`. `ScaleCache` achieves higher performance by up to 3.87× and 3.31× than the existing page cache with EXT4 and XFS in the case of 512KB stripe size, respectively.

[2]We mostly use 4KB as a request size since it is the default size and usually used in most OS and applications [16, 21, 44].



(a) Random write **(b)** Sequential write **(c)** Various request sizes **(d)** Various stripe sizes

Figure 11. Throughput of different I/O configurations with different number of SSDs.



(a) Average latency **(b)** Average CPU utilization

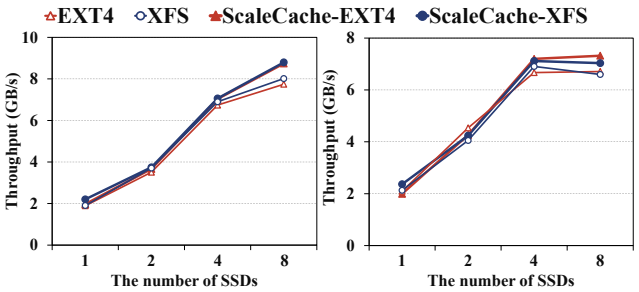
Figure 12. Average latency and CPU utilization.

Page cache \ # of SSDs	1	2	4	8
EXT4	60031 μs	58459 μs	59507 μs	55837 μs
XFS	60556 μs	60556 μs	60556 μs	60031 μs
ScaleCache-EXT4	221 μs	155 μs	204 μs	161 μs
ScaleCache-XFS	92 μs	123 μs	258 μs	449 μs

Table 1. Tail latency (99.9th) under multiple SSDs.

Average and tail latency: As shown in Figure 12a, ScaleCache reduces the average latency by up to 67.7% compared with the existing page cache under the random write workload. Especially, ScaleCache-XFS shows the shortest latency (31.63 μs) on 8 SSDs. In the case of 99.9th-percentile tail latency as shown in Table 1, the tail latency of ScaleCache-EXT4 and ScaleCache-XFS decreases by 99.7% and 99.8% of the existing page cache with EXT4 and XFS, respectively. This result shows ScaleCache outperforms the existing page cache significantly in terms of tail latency as well as throughput.

CPU Utilization: We measure the CPU utilization with 8 SSDs under random writes as shown in Figure 12b. As expected, the file systems with ScaleCache incur a higher CPU utilization compared with existing file systems with the page cache. EXT4 and XFS in the existing page cache utilize much fewer CPU resources (4.65% and 7.49%) compared with ScaleCache and direct I/O due to the long blocking time. Meanwhile, since ScaleCache eliminates the bottleneck of the existing page cache, it enables to accelerate the page reclamation and writeback with almost no blocking and spinning, resulting in high CPU utilization. As a result, our ScaleCache focuses on and achieves higher I/O scalability at the cost of high CPU utilization.



(a) Random read **(b)** Sequential read

Figure 13. Throughput of read-only workloads.

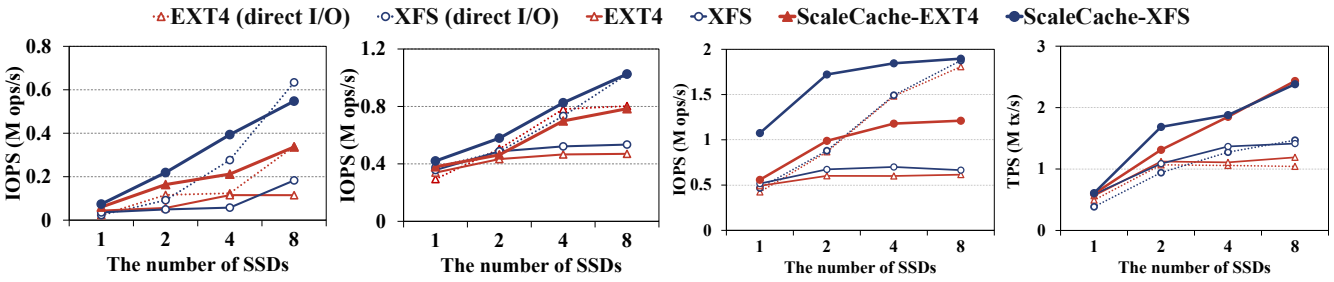
Read-only workloads: To further demonstrate the benefits of ccXArray, we measure the performance of buffered read-only workloads in Figure 13 which does not perform flushing operations since pages remain clean. In the case of 8 SSDs, ScaleCache-EXT4/ScaleCache-XFS improve the throughput by up to 1.12×/1.09× compared with existing page cache with EXT4/XFS. The performance improvement is not as pronounced as seen in the write scenarios since existing file systems have already reached the upper limit of buffered I/O performance. Even though threads also perform direct reclaim, this read operation induces less lock contention in the overall buffered I/O path. For example, in a read-only scenario, there is no need to mark pages as dirty, which requires the page cache to hold a lock.

4.2 Macrobenchmark

We use filebench (fileserver, varmail, and videoserver) and flexible filesystem benchmark (FFSB) with different number of SSDs. We use 64 threads, 64 files, 3GB file size, 4KB request size, and default stripe size (512KB).

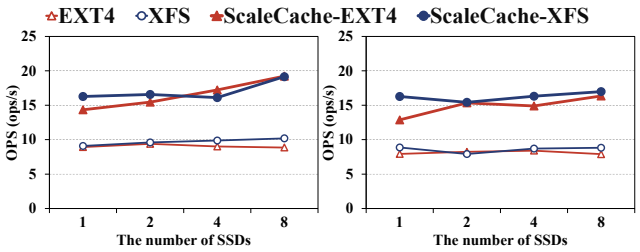
Fileserver: Figure 14a shows the result of fileserver. ScaleCache scales well and improves the performance by up to 6.81× compared with the existing page cache. Especially, ScaleCache-XFS shows more linear improvement than ScaleCache-EXT4 as the number of SSDs increases. As a result, ScaleCache yields a significant performance gain in fileserver.

Varmail: Figure 14b shows the result of varmail. The existing page cache shows almost the same performance even if the number of SSDs increases. ScaleCache exhibits less scalability in the varmail case compared with the fileserver



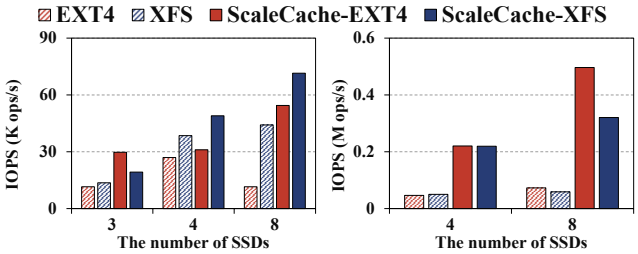
(a) Fileserver **(b)** Varmail **(c)** Videoserver **(d)** FFSB

Figure 14. Throughput of various filebench workloads and FFSB under different number of SSDs.



(a) Workload A results **(b)** Workload F results

Figure 15. Throughput of YCSB workloads with RocksDB under different number of SSDs.



(a) RAID-5 **(b)** RAID-10

Figure 16. Fileserver results under different RAID configurations.

case due to the higher read ratio in varmail. ScaleCache improves the performance by up to 1.92× in the case of 8 SSDs compared with the existing page cache. This result demonstrates ScaleCache can be effective in varmail as well as fileserver.

Videoserver: Figure 14c shows the results of videoserver. ScaleCache achieves higher performance by up to 2.85× compared with the existing page cache on 8 SSDs. Comparing cases of fileserver and varmail, the performance does not scale well in the case of two or more SSDs since videoserver also includes a heavy read workload.

FFSB: FFSB is a multi-threaded benchmark that provides a mix of read and write operations. As shown in Figure 14d, the existing page cache scales until two SSDs while ScaleCache scales well until 8 SSDs. ScaleCache-EXT4 and ScaleCache-XFS show better performance by up to 2.04× and 1.68× than the existing page cache with EXT4 and XFS on 8 SSDs, respectively.

Consequently, our ScaleCache increases the performance and scalability of file systems by reducing the lock contention and parallelizing the I/O operations in the page cache in even more realistic workloads.

4.3 Real-world Application

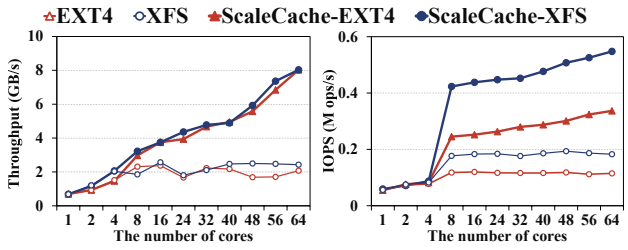
To evaluate our ScaleCache with real-world application, we use a key-value store, RocksDB [10] and YCSB [8] workloads. We use 64 threads, 1000 record counts, 1000 operation counts, an 8KB record size, and a RAID-0 configuration with its default stripe size (512KB). Figure 15 shows the results of YCSB workload A and F on RocksDB. Generally, the performance of both ScaleCache and existing page cache does not scale well as the number of SSDs increases, due to the overhead of RocksDB itself [14]. However, ScaleCache can achieve higher performance than the existing page cache. Specifically, ScaleCache improves the performance by up to 2.17× and 2.06× compared with the existing page cache in the case of workload A (update-heavy) and workload F (read/read-modify-write). This result demonstrates ScaleCache can be also effective in the real-world application such as key-value store.

4.4 Different RAID configurations

Figure 16 shows the performance of ScaleCache on multiple SSDs constructed by different RAID configurations: RAID5 and RAID10. As shown in Figure 16a, in the case of RAID-5 ScaleCache-EXT4 and ScaleCache-XFS outperform the page cache with EXT4 and XFS by up to 4.72× and 1.61× in the case of 8 SSDs. RAID-5 provides lower performance than other RAID configurations since there is the overhead of calculating the parity [57]. In the case of RAID-5, XFS shows better scalable performance than EXT4. Figure 16b shows RAID-10 performance results in the case of 4 and 8 SSDs. ScaleCache increases performance by up to 6.81× and 5.45× compared with the existing page cache in the case of EXT4 and XFS, respectively. This result demonstrates ScaleCache can be effective in various RAID configurations.

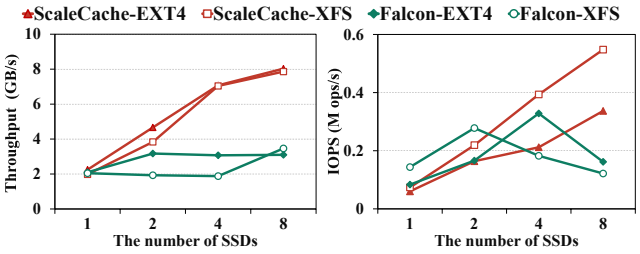
4.5 Core Scalability

To examine the core scalability as well as SSD scalability, we evaluate the performance on the different numbers of cores by fixing the number of SSDs to 8 as shown in Figure 17. As shown in Figure 17a in the case of random writes, ScaleCache-EXT4 scales the performance linearly while the



(a) FIO results **(b)** Fileserver results

Figure 17. Core scalability on 8 SSDs.



(a) FIO results **(b)** Fileserver results

Figure 18. Comparison with Falcon.

existing page cache does not scale even if the number of cores increases. As shown in Figure 17b, until 4 cores, both schemes do not scale but `ScaleCache-XFS` increases core scalability from 4 cores in the case of fileserver. These results show that `ScaleCache` also achieves high core scalability on multiple SSDs.

4.6 Comparison with a Scalable Scheme

We compare `ScaleCache` with Falcon [20], a scalable scheme for multiple SSDs. Falcon parallelizes I/O operations in the block layer for multiple SSDs using per-drive I/O processing. As shown in Figure 18, generally, `ScaleCache` shows better SSD scalability than Falcon. Meanwhile, in the case of fileserver, the performance with Falcon-EXT4 scales well until 4 SSDs, however, it sharply drops on 8 SSDs. Also, the performance with Falcon-XFS drops when the number of SSDs is more than 2. Eventually, `ScaleCache` outperforms Falcon by up to 2.59× and 4.5× on 8 SSDs in the case of FIO and fileserver, respectively. There are two reasons for the performance gap as follows. (1) Even if the I/O operation is parallelized in the block layer, there is still only one flusher thread when balancing dirty pages for a RAID of multiple SSDs. (2) Although the parallel scheme can accelerate I/O operations, the lock-based XArray limits the concurrency of the page cache level. Consequently, this result shows that our scheme can deliver better performance compared with the scheme which parallelizes I/O operations in the block layer.

4.7 Performance Breakdown

Table 2 shows the performance breakdown by measuring the time of main functions and locks using a kernel time function for higher accuracy and lower overhead. We use 8 SSDs with RAID-0 by running FIO with the same configuration. In the existing page cache, the blocking time (`io_schedule_timeout()`) takes a large portion of the total execution time. `dflush` reduces this blocking time. However, the overhead is moved to the lock contention on `xa_lock` of XArray. `ccXArray` eliminates the lock contention, and subsequently, the overhead is moved to `do_writepages()` and others. More specifically, the software MD-RAID driver (e.g., `md_handle_request()`) accounts for 38% of total execution time. This is out of scope in this paper, and we leave it as future work. Consequently, this result demonstrates `dflush` improves the throughput/latency by 2.57×/2.61×, and `ccXArray` further improves the throughput/latency by 1.50×/1.72×.

4.8 Memory Consumption

We examine the extra memory consumption that can be induced by `ScaleCache`. We measure the peak memory usage when running fileserver. As shown in Table 3, `ScaleCache` increases the memory consumption of slab by up to 4.5% and 4.9% compared with the existing page cache in the case of EXT4 and XFS, respectively. It is because the node is allocated via the slab allocator, and the lazy node deletion strategy in `ScaleCache` delays the node deletion, resulting in increasing the slab memory usage. However, this result shows that the memory overhead induced by `ScaleCache` can be minor (less than 5%) under a heavy-intensive write workload since the node can be reused instantly.

5 Discussion

In this section, we discuss the potential scalability of `ScaleCache` when using many more cores. As the number of cores increases, we expect that `ScaleCache` can meet contentions from itself or other system components, which is not exposed in our evaluation, resulting in degrading the performance and scalability. In this discussion, we focus on examining potential contention in `ScaleCache` itself as follows.

As the number of cores and threads increases, atomic instructions such as atomic-read and compare-and-swap in `ScaleCache` are more frequently executed. Thus, frequent executions of atomic instructions with many cores may incur a non-negligible performance overhead due to the cache coherency cost by the multiplying effect of cache-line ping-ponging and serialization caused by memory bus locking [11, 12, 34, 43]. Specifically, in `ScaleCache`, CAS operations may be performed excessively to determine whether a node is newly created or not in Algorithms 2 and 5. However, this CAS operation may not take much time since node creation in Algorithm 4 accompanies a finite number of atomic instructions for a single node creation without waiting for other threads. In addition, excessive atomic-read operations may be performed in waiting for whether a node is deleted logically or not in Algorithms 2 and 5. However, the cost of

Page cache schemes	Throughput	Avg latency	io_schedule_timeout()	i_rwsem	xa_lock	do_writepages()	Others
Existing page cache with EXT4	2123 MB/s	114.65 μs	90.24 %	0.05 %	0.05 %	1.67 %	7.99 %
ScaleCache(dflush)-EXT4	5463 MB/s	43.93 μs	0 %	0.11 %	9.25 %	38.43 %	52.21 %
ScaleCache(dflush+ccXArray)-EXT4	8221 MB/s	25.48 μs	0 %	0.13 %	0 %	43.88 %	55.99 %

Table 2. Performance breakdown of ScaleCache and existing page cache (i_rwsem is the inode lock).

Page cache / Benchmark	EXT4	SC-EXT4	XFS	SC-XFS
Memory Total	62.55 GB (100%)			
Anonymous page	0.15 GB (0.24%)	0.15 GB (0.24%)	0.15 GB (0.24%)	0.15 GB (0.24%)
File-backed page	60.86 GB (97.3%)	60.05 GB (96.0%)	61.38 GB (98.13%)	61.21 GB (97.86%)
Slab	2.61 GB (4.17%)	2.73 GB (4.36%)	1.27 GB (2.03%)	1.33 GB (2.13%)

Table 3. Memory consumption on 8 SSDs with fileserver (SC: ScaleCache).

atomic-read operation is lower than that of CAS operation on a contended value [33, 46]. Consequently, ScaleCache performs finite CAS operations without any waiting and low-cost atomic-read operations with waiting, minimizing the cost of cache coherency.

In future work, we have a plan to analyze each atomic instruction overhead and how it affects the ScaleCache performance on many more cores. Furthermore, we can collaborate with the previous works which deal with performance issues in executing the atomic instructions. For example, exploiting ILP (Instruction-level parallelism) [5, 26] can reorder issued atomic operations as long as non-overlapping memory regions are affected [46] to improve the atomic operation performance.

6 Related Work

Scaling file and storage systems: SpanFS [15] consists of a collection of micro file system services to scale file systems to many cores. ScaleFS [3] is a file system that decouples the in-memory file system from the on-disk file system to avoid the lock contention. Son et al. [50] present a transaction processing with a concurrent linked list and multi-threaded journal writes. Bjørling et al. [4] propose a new design for I/O management within the block layer by providing multiple I/O queues. Z-Journal [18] presents a scalable per-core journaling approach which can be progressed independently on each core. Max [23] targets improving the scalability of log-structured file system with SSD friendly design. Our study is in line with these works in terms of reducing the contention on shared resources or parallelizing the I/O operations. Unlike these studies for file systems [3, 15, 18, 23, 50] and block layer [4], we focus on scaling the page cache to improve the SSD scalability.

Improving I/O performance for multiple SSDs: Zheng et al. [64] build a user-space file abstraction on top of a local file system on each SSD for multiple SSDs. They create a dedicated I/O thread for each SSD and divide the global page cache into many small and independent sets of pages. However, this scheme can introduce complexity in the application and still has a lock for each set of pages as a fine-grained locking mechanism. Falcon [20] proposes a new block layer for per-drive I/O processing on a multi-SSD volume to parallelize I/O serving. SWAN [19] takes a spatial separation approach to alleviate the performance interference caused by GC. Shin et al. [48] eliminate bottom half contexts and background queue running contexts by a cooperative I/O processing model for flash SSDs. stRAID [57] proposes a stripe-threaded parity-RAID such as RAID-5/6 by targeting Linux MD with parity-based RAIDs. Our study is in line with these work [19, 20, 48, 57, 64] in terms of investigating the performance of multiple SSDs. In contrast, we focus on enabling concurrent access to the data structure (i.e., XArray) of Linux page cache and balancing dirty pages in parallel.

Reducing lock contention on data structures: Valois [55] provides lock-free data structures for a shared singly-linked list, which allows concurrent traversal, insertion, and deletion. Sundell et al. [52] present a lock-free algorithmic implementation of a concurrent priority queue. Natarajan et al. [35] propose a lock-free algorithm for a binary search tree based on marking edges and atomic instructions. Our study is inspired by these works [35, 52, 55] in terms of reducing the lock contention. In contrast, our study focuses on devising a scalable page cache via the concurrent data structure and parallel I/O.

7 Conclusion

As the throughput and I/O parallelism offered by modern SSDs continue to grow, leveraging a RAID of multiple SSDs has become an attractive storage solution in data centers for big data applications. However, we observe that the page cache in the Linux storage stack can limit the SSD parallelism under intensive write workloads. This paper presents ScaleCache comprising two synergistic techniques: ccXArray and dflush for multiple SSDs. ScaleCache first introduces a concurrent XArray (ccXArray) to enable concurrent access to the data structure of the page cache. Second, ScaleCache presents a direct page flush (dflush) to directly flush pages in a parallel and opportunistic manner. Through evaluation, we show that ScaleCache outperforms the Linux page cache and a scalable scheme for multiple SSDs by up to 6.81× and 4.50×, respectively.

Acknowledgments

We would like to give special thanks to our shepherd Dr. Shai Bergman and the anonymous reviewers for their valuable comments and feedback. This work was supported in part by the National Research Foundation of Korea (NRF) (No. NRF-2021R1C1C1010861, NRF-2022R1A4A5034130) and Korea Institute for Advancement of Technology (KIAT) (No. KIAT-P0012724) grant funded by the Korea Government (Corresponding Author: Yongseok Son).

References

[1] Dan Alistarh, Thomas Sauerwald, and Milan Vojnović. 2015. Lock-free algorithms under stochastic schedulers. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 251–260.

[2] Jiwoo Bang, Chungyong Kim, Sunggon Kim, Qichen Chen, Cheongjun Lee, Eun-Kyu Byun, Jaehwan Lee, and Hyeonsang Eom. 2021. Finer-lru: A scalable page management scheme for hpc manycore architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 567–576.

[3] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 69–86.

[4] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*. ACM, 22.

[5] Timothy J Callahan and John Wawrzynek. 1998. Instruction-level parallelism for reconfigurable computing. In *FPL*, Vol. 98. 248–257.

[6] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. 2015. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 233–242.

[7] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. 2017. On the performance variation in modern storage stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 329–344.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[9] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No hot spot non-blocking skip list. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 196–205.

[10] Facebook. 2023. RocksDB. http://rocksdb.org/.

[11] Eduardo José Gómez-Hernández, Juan M Cebrian, Rubén Titos-Gil, Stefanos Kaxiras, and Alberto Ros. 2021. Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 337–349.

[12] Balaji Iyengar, Edward Gehringer, Michael Wolf, and Karthikeyan Manivannan. 2012. Scalable concurrent and parallel mark. In *Proceedings of the 2012 international symposium on Memory Management*. 61–72.

[13] J.Axboe. 1998. Fiobenchmark. http://freecode.com/projects/fio.

[14] Yichen Jia and Feng Chen. 2020. From flash to 3D XPoint: Performance bottlenecks and potentials in RocksDB with storage evolution. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 192–201.

[15] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 249–261.

[16] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*.

[17] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. 2022. ScaleXFS: Getting scalability of XFS back on the ring. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 329–344.

[18] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. 2021. Z-Journal: Scalable Per-Core Journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 893–906.

[19] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 799–812. https://www.usenix.org/conference/atc19/presentation/kim-jaeho

[20] Pradeep Kumar and H Howie Huang. 2017. Falcon: Scaling io performance in multissd volumes. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association*. 41–53.

[21] Eunji Lee, Seunghoon Yoo, Jee-Eun Jang, and Hyokyung Bahn. 2012. Shortcut-JFS: A write efficient journaling file system for phase change memory. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 1–6.

[22] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. 2019. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 603–616.

[23] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 877–891.

[24] Linux software RAID. 2023. https://raid.wiki.kernel.org/index.php/Linux_Raid.

[25] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. 2014. Automatic {Identification} of Application {I/O} Signatures from Noisy {Server-Side} Traces. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 213–228.

[26] Jack L Lo, Joel S Emer, Henry M Levy, Rebecca L Stamm, Dean M Tullsen, and Susan J Eggers. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems (TOCS)* 15, 3 (1997), 322–354.

[27] Robert Love. 2010. The Page Cache and Page Writeback. In *Linux Kernel Development* (3rd ed.). Addison-Wesley, Chapter 16.

[28] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2020. A Study of SSD Reliability in Large Scale Enterprise Storage Deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 137–149.

[29] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2022. Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 165–180. https://www.usenix.org/conference/fast22/presentation/maneas

[30] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S. A. S. 2007. A AND VIVER, L. The New ext4 filesystem: current status and future plans. In *In Ottawa Linux Symposium. http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf*.

[31] Paul E McKenney. 2003. Two-phase update for scalable concurrent data structures. *OGI RPE* (2003).

[32] Ethan L Miller and Randy H Katz. 1991. Input/output behavior of supercomputing applications. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 567–576.

[33] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 103–112.

[34] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 457–468.

[35] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 317–328.

[36] NETAPP INC. AFF A-Series arrays. 2023. https://www.netapp.com/data-storage/aff-a-series/.

[37] Jinoh Oh, Wook-Shin Han, Hwanjo Yu, and Xiaoqian Jiang. 2015. Fast and robust parallel SGD matrix factorization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 865–874.

[38] Rachit Pandey. 2020. *Performance benchmarking and comparison of cloud-based databases MongoDB (NoSQL) vs MySQL (Relational) using YCSB.* Technical Report. Technical Report. https://doi. org/10.13140/RG. 2.2. 10789.32484.

[39] Tirthak Patel, Suren Byna, Glenn K Lockwood, and Devesh Tiwari. 2019. Revisiting I/O behavior in large-scale storage systems: The expected and the unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[40] Tirthak Patel, Suren Byna, Glenn K Lockwood, Nicholas J Wright, Philip Carns, Robert Ross, and Devesh Tiwari. 2020. Uncovering Access, Reuse, and Sharing Characteristics of I/O-Intensive Files on Large-Scale Production HPC Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 91–101.

[41] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz. 1989. Introduction to redundant arrays of inexpensive disks (RAID). In *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*. 112–117. https://doi.org/10.1109/CMPCON.1989.301912

[42] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '88)*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/50202.50214

[43] Yuxin Ren, Gabriel Parmer, and Dejan Milojicic. 2020. Ch'i: scaling microkernel capabilities in cache-incoherent systems. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. IEEE, 12–21.

[44] Alma Riska and Erik Riedel. 2006. Disk Drive Level Workload Characterization.. In *USENIX Annual Technical Conference, General Track*, Vol. 2006. 97–102.

[45] Jose Santos. 2013. FFSB (flexible file system benchmark). http://sourceforge.net/projects/ffsb/.

[46] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 445–456.

[47] Intel Optane SSD 900P series. 2017. https://www.intel.com/content/www/us/en/products/sku/123623/intel-optane-ssd-900p-series-280gb-2-5in-pcie-x4-20nm-3d-xpoint/specifications.html.

[48] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. 2014. OS I/O Path Optimizations for Flash Solid-state Drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 483–488. https://www.usenix.org/conference/atc14/technical-sessions/presentation/shin

[49] Yongseok Son, Hyuck Han, and Heon Young Yeom. 2015. Optimizing file systems for fast storage devices. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–6.

[50] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. 2018. High-Performance Transaction Processing in Journaling File Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 227–240. https://www.usenix.org/conference/fast18/presentation/son

[51] Yongseok Son, Heon Young Yeom, and Hyuck Han. 2016. Optimizing I/O operations in file systems for fast storage devices. *IEEE Trans. Comput.* 66, 6 (2016), 1071–1084.

[52] Håkan Sundell and Philippas Tsigas. 2005. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel and Distrib. Comput.* 65, 5 (2005), 609 – 627. https://doi.org/10.1016/j.jpdc.2004.12.005

[53] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System.. In *USENIX Annual Technical Conference*, Vol. 15.

[54] Michael Hao Tong, Robert L Grossman, and Haryadi S Gunawi. 2021. Experiences in Managing the Performance and Reliability of a Large-Scale Genomics Cloud Platform. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 973–988.

[55] John D. Valois. 1995. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. ACM, New York, NY, USA, 214–222. https://doi.org/10.1145/224964.224988

[56] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 559–571.

[57] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. 2022. StRAID: Stripe-threaded Architecture for Parity-based RAIDs with Ultra-fast SSDs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 915–932.

[58] Matthew Wilcox. 2018. [PATCH v13 00/72] Convert page cache to XArray. https://lore.kernel.org/all/20180611140639.17215-1-willy@infradead.org/.

[59] Matthew Wilcox. 2018. XArray. https://docs.kernel.org/core-api/xarray.html.

[60] Andrew Wilson. 2008. The new and improved FileBench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies*.

[61] Lianghong Xu, James Cipar, Elie Krevat, Alexey Tumanov, Nitin Gupta, Michael A Kozuch, and Gregory R Ganger. 2014. {SpringFS}: Bridging Agility and Performance in Elastic Distributed Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 243–255.

[62] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. 2018. RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 279–294. https://www.usenix.org/conference/fast18/presentation/zhang

[63] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. 2018. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 477–492.

[64] D. Zheng, R. Burns, and A. S. Szalay. 2013. Toward millions of file system IOPS on low-cost, commodity hardware. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503225