IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 October 2024, accepted 12 November 2024, date of publication 20 November 2024,
date of current version 29 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3502688

RESEARCH ARTICLE

Design and Implementation of Efficient and Transparent Zero Copy Read

JIWOONG PARK[1], CHEOLGI MIN[2], HEONYOUNG YEOM[2], AND YONGSEOK SON[3]
[1]Department of Computer Engineering, Sungshin Women's University, Seoul 02844, South Korea
[2]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[3]Department of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Yongseok Son (sysganda@cau.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) funded by Korean Government (MSIT) under Grant NRF-2022R1G1A1011344 and Grant NRF-2022R1A4A5034130.

ABSTRACT In server consolidation environments, co-locating I/O-intensive and other workloads on a single physical machine often leads to significant performance degradation for memory-intensive workloads due to the memory copies required during copy-based I/O (buffered I/O) operations. Zero-copy I/O has been proposed as a solution to this problem by eliminating unnecessary memory copying between user space and kernel space, thereby improving performance. However, existing zero-copy I/O schemes fail to provide both transparent copy avoidance via standard read/write system calls and the benefits of kernel-level caching, which limits their general applicability. We introduce the z-READ framework, a solution that enables transparent zero-copy read I/O operations while leveraging kernel-level caching. It addresses two primary practical challenges: minimizing page remapping overhead by reducing the number of TLB shootdowns, and ensuring robust I/O performance under high copy-on-write fault conditions using adaptive hybrid I/O mode switching based on a performance loss model and historical I/O patterns. We implement a z-READ prototype on Linux Kernel 4.12.9 and conduct extensive evaluations. Our results demonstrate that, while traditional copy-based I/O configurations in co-located environments lead to an average performance slowdown of 84% for memory-intensive workloads, z-READ restricts this impact to only an 11% average slowdown.

INDEX TERMS Copy-on-write, page remapping, performance interference, TLB shootdown, zero copy.

I. INTRODUCTION

Performance interference is a critical issue in data centers, where server consolidation is essential for cost efficiency [2], [3], [4], [5]. In these environments, multiple applications with different resource demands (e.g., CPU, memory, storage) are co-located on a single physical server under the assumption that they will not interfere with each other [6], [7].

However, in practice, especially with memory-intensive and I/O-intensive workloads, resource contention can occur, resulting in significant performance degradation. A major cause of this degradation is the memory copy operations between user and kernel spaces during I/O. These operations, required in traditional buffered I/O, consume substantial CPU cycles and memory bandwidth, exacerbating contention for shared resources such as the last-level cache (LLC) and memory controller.

To mitigate the overhead of memory copying, zero-copy I/O schemes have been proposed [8], [9], [10], [11], [12], [13], [14]. Modern operating systems (OSs) provide mechanisms such as `direct I/O` and `mmap` to enable zero-copy access to file contents [8]. However, `direct I/O` bypasses the kernel caching layer, leading to missed opportunities for caching benefits unless applications implement their own caching mechanisms at the user level. On the other hand, `mmap` requires significant changes to application code since it uses different semantics and interfaces compared to standard POSIX read/write calls.

While various zero-copy approaches, such as direct user-level access and memory-mapped I/O, have been proposed to reduce redundant memory copying [9], [10], [11], [12], [13], [14], these approaches often require adopting

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

new APIs or programming models, imposing additional burdens on developers. Notably, previous attempts [10] to transparently achieve zero-copy using page remapping and copy-on-write (COW) techniques, face significant hurdles in modern x86 systems, particularly under storage I/O workloads.

First, page remapping can be more expensive than page copy with modern x86 multicore processors that have a few tens of cores. Page remapping involves manipulating page tables, which on modern multicore processors leads to substantial overhead due to the need for translation lookaside buffer (TLB) shootdowns across multiple cores [15].

Second, in storage I/O, COW fault overhead is unavoidable and can negate the performance gains of zero-copy. In network I/O, the COW fault overhead can be avoided by not sharing pages between user and kernel space—kernel pages with target data are directly swapped with application's user pages. It came at no cost because, in network subsystem, the same data is not read again by applications. However, applying this approach to storage system leads to abandoning kernel-level caching. To take advantages of both zero copy and caching, the COW technique is necessary to prevent modifications to shared pages. However, update on zero-copied data triggers a COW fault handler that eventually performs page copy operation. Therefore, depending on the workload, COW fault overhead can negate the benefits of zero copy or even worsen the performance compared to copy-based I/O [14].

In this paper, we present `z-READ`, an efficient, transparent and practical zero copy read file I/O scheme for storage systems, aiming to tackle the two main challenges: 1) page remapping overhead and 2) COW fault overhead. To mitigate the page remapping overhead, `z-READ` involves private page table entry (PTE) detection through page access tracking, which minimizes the number of remote TLB shootdowns and which was first introduced in Amit's work [15]. In addition, `z-READ` batches TLB flushes for a multiple page-sized read I/O request to reduce the number of TLB shootdowns per I/O.

To ensure that COW fault overhead does not surpass the performance gain from zero copy, we also propose a performance loss prediction model and adaptive hybrid I/O mode switching based on historical I/O patterns. Building a performance loss model requires a one-time offline measurement of the read I/O latencies of `z-READ` and copy-based I/O for varying sizes, which can be done with help from our simple user-level helper program. The model can be updated at `z-READ` kernel module loading time. On the other hand, `z-READ` collects the I/O behavior online on a per-page cache basis. With the prediction model and historical information, our system adaptively chooses between the traditional copy-based read I/O and `z-READ` on a per-page basis.

In our previous study [16], we focused on only one of the two challenges: page remapping overhead but not COW fault overhead. In this paper, we extend our scheme so that it provides the same performance guarantee as the copy-based I/O even under workloads that cause excessive COW faults.

We implement `z-READ` in Linux kernel 4.12.9 and conducted extensive experiments with benchmarks from FIO [17] and Filebench [18]. The results show that our scheme provides higher I/O performance than copy-based I/O in the in-memory configuration where all data are served from memory. We have found that the main reasons for the performance gains are mainly due to the reduced memory access by zero copy. We also evaluate the performance interference intensity of `z-READ`. Avoiding memory copy, `z-READ` is superior to copy-based I/O in server consolidation environments. In particular, this paper makes the following contributions:

- We design and implement `z-READ`, an efficient, transparent zero copy read I/O system for storage environments, addressing two main challenges: page remapping overhead and COW fault overhead.
- Our experiments show that `z-READ` can improve I/O performance by up to 23.3× in in-memory configuration, compared to traditional copy-based I/O.
- We show that I/O-intensive workloads using copy-based I/O can significantly interfere with co-located workloads, causing slowdowns of up to 144% (in-memory) and 82% (on-disk) due to memory copying. `z-READ` mitigates this interference, reducing the slowdowns of up to 28% and 57%, respectively.

II. BACKGROUND AND MOTIVATION

A. BACKGROUND

1) TLB SHOOTDOWN

A TLB is a per-core cache that enables fast virtual-to-physical address translation. Since modern CPUs do not automatically ensure TLB coherency, the OS is required to manually synchronize the TLB at the software level. TLB shootdown is the operation used to remove stale TLB mappings from the TLB, ensuring that it reflects current memory mappings. This operation is essential whenever page remapping occurs, following adjustments to the PTE mappings. However, TLB shootdown is costly on modern x86 multicore systems with many cores, because it necessitates sending inter-processor interrupts (IPIs) to each core, increasing in cost with the number of cores [15].

2) DIRECT TLB INSERTION

Many efforts have been devoted to mitigating TLB shootdowns, notably by Amit [15], who introduced a software-based *direct TLB insertion* technique. This involves inserting a mapping directly into a target core's TLB without setting the access bit of the PTE, which helps detect PTE sharing and avoid TLB shootdowns in common mappings: short-lived private mappings cached by one core, and long-lived idle mappings not cached by any cores. Since the CPU sets the access bit of the corresponding PTE when a page is accessed, checking the access bit after *direct TLB insertion* can indicate PTE sharing. If the access bit is set, the mapping

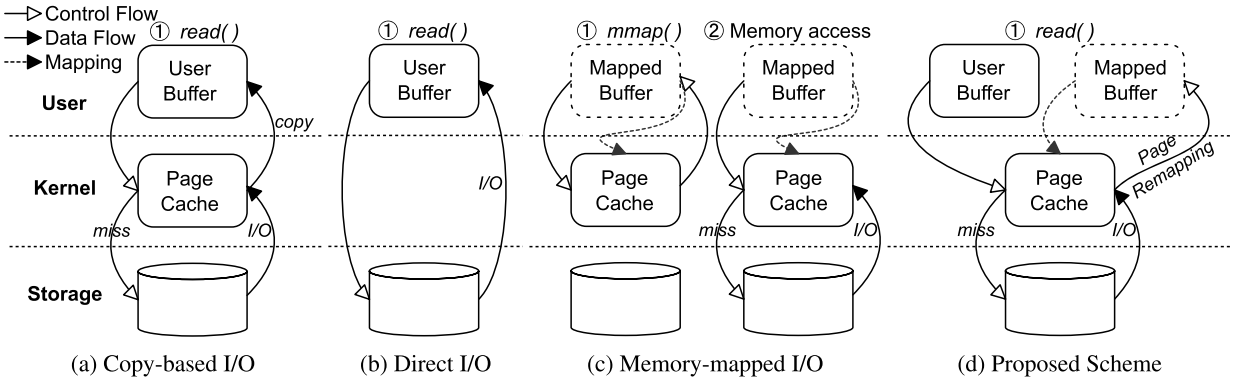


FIGURE 1. Existing file I/O interfaces and the proposed scheme.

is shared with other cores; otherwise, it remains private to the target core, which can use the locally cached mapping without setting the access bit.

One might think that direct TLB insertion could be emulated by accessing the page and then clearing the access bit. However, this approach struggles with timing issues that can lead to false indications of sharing during the interval between setting and clearing the access bit.

To address this, Amit introduced ABIS (Address-Based Isolation System), which allows *direct TLB insertion* without setting the access bit by leveraging process-context identifiers (PCIDs) and secondary page-table hierarchies. Upon a page fault, ABIS loads a secondary page-table into the CR3 register with the same PCID as the faulting thread, wiring the virtual-to-physical mapping in both the primary and secondary spaces. When the page is accessed, the hardware uses the secondary hierarchy to insert the translation into the TLB, keeping the primary's access bit clear. This entry remains valid in the primary space due to the shared PCID.

3) COPY-ON-WRITE

COW is an optimization technique initially designed to delay the costly copying of data until the first write attempt. In a transparent zero copy file I/O scheme, COW is crucial for sharing kernel page cache with applications and protecting the data from being directly modified by these applications. By setting a write protection bit in the PTE corresponding to the target page, any attempt to modify the page's contents triggers a COW fault. When a COW fault occurs, the kernel page fault handler allocates a new page, maps the faulting virtual address to the physical address of the new page, copies the contents from the old page to the new page, and performs TLB shootdowns to invalidate the stale mappings in the TLB. This allows the application to continue modifying the data at the faulting address. Consequently, the overhead from a COW fault can potentially negate the performance improvements gained from zero copy if the zero-copied data is frequently updated.

B. PROBLEMS OF COPY-BASED I/O

Traditionally, read/write system calls have been used to access files. By default, the Linux kernel employs buffered I/O (copy-based I/O) for these calls [8]. For simplicity, the term "`READ`" will be used throughout this paper to refer to a read system call with buffered I/O.

TABLE 1. Latency breakdown of a 4 KB `READ`.

Type	Page Copy	Page Search	Etc.	Total
Latency (ns)	7815 (68%)	1322 (11%)	2422 (21%)	11559

FIGURE 1 illustrates the caching behavior of `READ`. When handling read requests, the kernel first reads a section of a file in the page cache and then copies the data from the page cache to the application buffer. This caching behavior, at the cost of a single memory copy, can significantly enhance user-perceived I/O performance for subsequent accesses, helping to mask the slower speeds of the underlying storage. Thus, buffered I/O is generally the default I/O mode enabled by most OSs, and it is effective in many cases. However, there are notable drawbacks.

First, a memory copy during buffered I/O adds latency overhead. While this overhead may be negligible compared to disk I/O time for a page cache miss, it is substantial for a page cache hit. We analyze the average latency of a read I/O operation for a page cache hit by measuring the time for each function within the kernel's read path using the `do_gettimeofday()` kernel function wrapped around each target function. TABLE 1 provides a detailed breakdown of the latency for a 4 KB read in our system (see § VI-A for detailed system configuration information). The data shows that the memory copy overhead is significant, accounting for over two-thirds of the total read time.

Second, a memory copy consumes memory bandwidth. In modern systems that support high I/O bandwidth, copy-based I/O can consume significant fraction of total memory bandwidth [19]. For example, a four-lane interface speed of PCIe 4.0 NVMe SSD can reach up to 8 GB/s. By using four SSDs, the aggregate bandwidth can be up to 32 GB/s, which is comparable to the bandwidth of a two-channel DDR4-2133 memory system (17 GB/s x 2 channels = 34 GB/s). In such scenarios, co-locating I/O intensive workloads

with memory-intensive workloads can saturate the memory bandwidth.

Third, a memory copy can pollute the LLC. After a page copy, both source and destination pages are cached in the LLC. When the LLC is full, the page copy operation can cause the eviction of more useful data that will be accessed sooner. This not only decreases cache efficiency but also creates performance interference with other workloads since the LLC is shared among all cores in a CPU node.

C. LIMITATIONS OF EXISTING ZERO COPY I/O

For decades, numerous studies related to zero copy I/O have been conducted to reduce data transfer overhead in various areas in computer systems [9], [10], [11], [12], [13], [14]. Although these studies target different layers, they generally fall into categories:

1) DIRECT I/O SCHEMES

These zero copy schemes support direct user-level access to devices, removing the kernel from the critical I/O path for user applications [11], [12]. While effective in networking, where the processing time is dominated by copying packet data from the network interface card (NIC) to host memory, this approach is less beneficial for storage I/O. In storage I/O, the total read time is primarily dictated by the time required to read data from the storage device, not the memory copy overhead. Consequently, eliminating memory copy offers no real performance gain if data is served from disk and could worsen user-perceived I/O performance, as every I/O request would bypass the page cache and go directly to storage.

On the other hand, self-caching applications like database management systems (DBMS) that manage their own memory buffer pool to cache file contents do not benefit from kernel-level file caching because it leads to duplicated data in both the buffer pool and the kernel cache. This redundancy results in wasted CPU and memory resources.

In Linux, `direct I/O` is designed for self-caching applications to avoid such inefficiency by bypassing the kernel cache entirely. FIGURE 1(b) illustrates how `READ` operates when a file is opened with the `O_DIRECT` flag. With `direct I/O`, all file I/O operations involve physical disk I/O, transferring data directly between the user buffer and disk via DMA. This method uses the same interfaces as buffered I/O, allowing applications to adopt `direct I/O` with minor changes, such as aligning file offsets and memory buffers.

`Direct I/O` with a user-level buffer pool enables efficient zero-copy I/O by eliminating the CPU overhead associated with data copying between kernel and user space. It is especially effective for I/O-intensive workloads, and allows applications to implement custom replacement policies based on specific data access patterns, providing more flexibility compared to the kernel's simplified LRU replacement policy. However, `direct I/O` is constrained by the pre-configured size of the user-level buffer and does not share data across applications, which can lead to redundant data loading when multiple applications access the same file.

2) MEMORY-MAPPED I/O SCHEMES

Memory-mapped I/O schemes provide applications with access to shared buffers [9], [10], [13], [14]. Chu [10] introduced a transparent zero copy scheme based on page remapping and COW techniques, requiring minimal changes to applications and preserving existing interfaces. However, his work targeted the network subsystem, differing significantly from storage I/O. Chu's scheme eliminates the COW fault overhead by allowing applications to take ownership of kernel pages, rather than sharing it. This approach works for network data, as it is not typically read again by other applications. In storage I/O, data can be read multiple times by different applications, necessitating shared kernel pages protected by COW, making COW fault overhead unavoidable. Moreover, Chu's scheme was tested on the UltraSPARC system, where zero copy benefits outweighed page remapping costs. In modern x86 servers with many cores, page remapping costs can exceed page copying due to increased TLB shootdown costs.

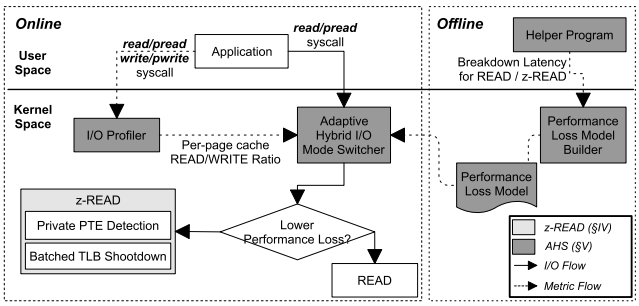


FIGURE 2. Overview of z-READ framework.

The `mmap()` system call is an alternative for zero copy storage I/O in Linux. FIGURE 1(c) shows how `mmap()` maps a file region into an application's address space. Once mapped, applications access file data directly with memory semantic interfaces without memory copy costs. This allows applications to take advantage of the kernel's page cache, which automatically handles caching and can improve overall I/O performance without requiring the developer to implement explicit data buffering.

Despite these advantages, `mmap()` presents challenges. Its interface and functionality differ significantly from traditional read/write calls, making it more complex for developers to manage [20]. Using `mmap()` requires careful handling of memory mapping, which can complicate application logic. Developers must ensure proper synchronization and account for scenarios where memory-mapped regions might be evicted from physical memory. While `mmap()` can offer performance benefits for certain workloads, it can also introduce overhead in others. For example, page faults that occur during memory access can be costly, particularly if

the application's working set exceeds the available physical memory [21].

III. DESIGN OVERVIEW

Towards a transparent zero-copy scheme in storage systems on modern x86 servers, we present the z−READ framework. Our zero-copy scheme has two key objectives: (1) providing the same interfaces as the standard read/write system calls, requiring only minor changes for adoption, similar to direct I/O, and (2) offering the benefits of kernel-level caching by dynamically mapping a file region into the calling application's address space, akin to mmap().

To achieve these goals, we focus on reducing two main overheads: page remapping and COW faults. The z−READ framework addresses page remapping overhead by implementing a *transparent zero-copy read* mechanism (z−READ, § IV) that involves efficient page remapping techniques that minimize TLB shootdown costs. Additionally, our framework avoids the COW fault overheads that could negate the benefits of zero-copy, through a per-page basis *adaptive hybrid I/O switching* mechanism (AHS, § V).

FIGURE 2 presents an overview of our z−READ framework, which consists of two core components: (1) z−READ and (2) AHS. Note that the term "*kernel page*" refers to a page in the page cache, while "*user page*" refers to an anonymous page allocated by the application.

1) z-READ

z−READ is a transparent zero-copy read mechanism that performs dynamic page remapping during I/O operations, necessitating a subsequent synchronous TLB shootdown. This shootdown overhead constitutes the majority of the page remapping time. To mitigate the TLB shootdown overheads during page remapping, z−READ employs two strategies:

- *Private PTE Detection (§ IV-B1)*: Minimizes the number of remote TLB shootdowns by identifying pages that do not require a global shootdown.
- *Batched TLB Shootdown (§ IV-B2)*: Reduces the number of TLB shootdowns per I/O operation by batching them together.

These strategies significantly reduce the page remapping overhead, enabling z−READ to outperform READ in non-intensive COW fault scenarios.

2) ADAPTIVE HYBRID I/O MODE SWITCHER

The *adaptive hybrid I/O mode switcher* is a kernel component that dynamically decides on a per-page basis whether to employ z−READ. This decision is based on data collected both online through the *I/O profiler* and offline via the *performance loss model builder*. This approach ensures that the COW fault overhead does not negate the performance benefits of zero copy, thereby maintaining the superiority of z−READ even in intensive COW fault scenarios.

Algorithm 1 z-READ Main Procedure

```
Input: file_offset, user_buf, buf_len
Output: bytes_read
1  if user_buf and file_offset are both page aligned then
2      idx = 0;
3      foreach offset in a page unit starting from file_offset do
4          page = find_page_cache(offset);
5          if page == NULL then perform disk I/O and goto 4 ;
6          else
7              if remaining data size < PAGE_SIZE then
8                  copy remaining data to user buffer;
9              else
10                 target_pages[idx++] = page;
11                 page→ref_count++;
12                 if idx == BATCH_SIZE then
13                     idx = 0;
14                     Batching Procedure();
15     if idx ≠ 0 then
           /* Handle remaining pages                    */
16         Batching Procedure();
17 else
18     fall back to READ for all pages;
```

3) PERFORMANCE LOSS MODEL BUILDER

This kernel module updates the *performance loss model table* based on experimental results from a *helper program*. The table is an array where each index i represents the average latency for an N KB read, as defined by:

$$N = 4 \times (i + 1) \tag{1}$$

For instance, index 0 indicates average latencies for 4 KB reads, while index 1 indicates average latencies for 8 KB reads. These latencies are critical for the *performance loss model*, which will be discussed in detail in § V-A.

4) HELPER PROGRAM

The *helper program* is a user-level application that measures the read latencies of z−READ and READ for varying I/O sizes. It allocates a page-aligned memory buffer, initialize it with zeros, and issues I/O requests to load the page units from a pre-generated file into the memory buffer. The program measures the average time between submitting a request and its fulfilment, as well as the average latency of read-modify pattern where the buffer content is updated after each read.

5) I/O PROFILER

The *I/O profiler* is a kernel component that monitors the number of read and write I/O operations for each *kernel page*. It provides dynamic online information to the *adaptive hybrid I/O mode switcher*, enabling it to make informed decisions about which I/O mode to use.

IV. TRANSPARENT ZERO-COPY READ

z−READ performs dynamic page remapping during I/O operations. FIGURE 1(d) shows the behavior of z−READ. z−READ's read I/O path diverges from READ after locating

the target *kernel page* in the page cache. Instead of copying data to a user buffer, `z-READ` remaps the user buffer's virtual address to the *kernel page*'s physical address, then frees the *user page*. This process includes a TLB shootdown to finalize the mapping and prevent access to the freed *user page*, which is also handling COW fault. Our focus is to minimize this TLB shootdown overhead, making transparent zero-copy reads practical and efficient in storage systems on modern x86 servers.

A. z-READ

1) PREREQUISITES

For `z-READ` to function, both the user buffer and the requested file offset must be page-aligned because page remapping is performed with page granularity. If they are not aligned, `z-READ` can simply fall back to the standard `READ` without any errors. It is important to note that page alignment is a common practice, especially in applications that support direct I/O or mmap(), such as DBMS.

The requested I/O size and file size do not need to be page-sized, provided the user buffer and file offset are page-aligned. When the requested I/O size is not a multiple of the page size, the kernel handles the last *user page* by copying the remaining data from the page cache to the user buffer (Algorithm 1, line 7). For example, if the requested I/O size is 9 KB and the file size is large enough, `z-READ` will remap the first two 4 KB *user pages* to *kernel pages*, while the remaining 1 KB of data is copied from the page cache to the last *user page* of the user buffer. This method is used to avoid accidentally overwriting data adjacent to the user buffer during page remapping.

2) READ

For reads, the kernel first checks the alignment of the user buffer and file offset and then searches the page cache for the requested blocks (Algorithm 1, lines 1–4). After finding the *kernel page* containing the requested data, the kernel stores their pointers in the page array to handle page remapping and TLB flushing in a batched manner (lines 9–10). It also increases the reference count of the target pages to prevent them from being evicted from the page cache while they are still mapped (line 11). When the number of stored page pointers reaches *BATCH_SIZE*, or the system call is about to return to the user with the unhandled pages, the kernel starts the batching operation (lines 12–16).

A single batch consists of two main jobs: page remapping and TLB flushing. For page remapping, the kernel first obtains the PTE of each *user page* by a page table walk and then checks if the *user page* is already mapped to the *kernel page* (Algorithm 2, lines 3–7). If so, the kernel just decreases the reference count of the *kernel page* pointed by the PTE to allow the page to be reclaimed in case of memory pressure (line 11). Otherwise, the *user page* that was originally pointed by the PTE is marked to be lazily freed by the kernel page reclamation mechanism later (line 9). This

Algorithm 2 Page Remapping Procedure

```
Input: start_addr, end_addr, target_pages[ ], nr_target_pages
Output: zREAD_CPU_bitmap, IPIALL_bitmap,
        OTHER_bitmap[], pte_ptrs[ ]
1  foreach addr in a page unit from start_addr to end_addr do
2      idx = calculate_from_address(addr);
3      pte = page_table_walk(addr);
4      pte_ptrs[idx] = pte;
5      user_page = pte_to_page(*pte);
6      zread_cpu = get_zread_cpu(*pte);
7      if zread_cpu == -1            /* Not mapped yet */
8       then
9          mark the user page to be lazily freed;
10     else
11         user_page→ref_count--;
12     manipulate PTE to point the target_pages[idx];
13     set write protection bit in the PTE;
14     set_zread_cpu(pte, current_cpu);
15     if access bit == 1 then
16         set the idx-th bit in IPIALL_bitmap;
17     else
18         if zread_cpu != -1 then
19             set the idx-th bit in zREAD_CPU_bitmap;
20             set the idx-th bit in OTHER_bitmap[zread_cpu];
21         else
22             set the idx-th bit in IPIALL_bitmap;
```

prevents unnecessary memory consumption and reduces the risk of memory thrashing for very large I/O operations (e.g., 1GB). The kernel then manipulates the PTE mapping to point to the *kernel page* and sets the write protection bit in the PTE to prevent unintended modification by the user (lines 12–13). The kernel also updates *zread_cpu*, which is the CPU core ID where the previous `z-READ` was performed, storing this information in six bits (52–57) of the corresponding PTE, which are unused on the x64 architecture (line 14).

After page remapping, the kernel performs TLB flushing for the manipulated PTEs. In our scheme, we minimize the number of remote TLB shootdown operations using several optimization techniques. After synchronously waiting for the remote CPUs to invalidate the stale PTEs for the requested range, the kernel returns to the user space. Note that we disable preemption during the batching operation for accurate private PTE detection (Algorithm 3, line 1). Inaccurate detection results can allow cores to access the incorrect page using stale PTEs cached in their TLBs.

3) WRITE

Since `z-READ` focuses on read file I/O, the kernel behavior on write has not been altered. Even if the application issues a write request with the user buffer already mapped to *kernel pages*, the kernel can copy the data from the buffer to the target page cache; in this case the copy is actually performed between *kernel pages*.

4) MODIFY

When an application attempts to modify the user buffer that is mapped to a *kernel page*, a COW fault occurs due to

the protection bit in the PTE set by the previous z−READ operation. The kernel COW fault handler checks if *zread_cpu* is set. If so, it allocates a new page and modifies the PTE mapping to point to the new page, followed by shooting down TLB entries. This cost can be reduced by private PTE detection optimization. After TLB flushing, the kernel decreases the reference count of the old page and then returns to the user.

5) FREE

When an application terminates or frees the user buffer, the remapped page must be properly handled. In z−READ, the kernel checks *zread_cpu* for each PTE of the remapped pages to be freed. If *zread_cpu* is set, it indicates that the page to be freed is a *kernel page*. The kernel then decreases the reference count of these *kernel pages* in a batched manner. Only pages whose reference counts reach zero are freed. Since the pages are about to be freed, the kernel does not need to roll back the pages to their original state before page remapping.

6) OPEN

For implementation and debugging convenience, we add a new file open flag, O_ZREAD. Similar to the O_DIRECT flag, z−READ is used only when a file is opened with the O_ZREAD flag.

B. TLB SHOOTDOWN OPTIMIZATION

To minimize the TLB shootdown overhead, we present techniques including *private PTE detection* and *batched TLB shootdown*. The additional instruction overhead incurred by our optimizations is substantially less consequential compared to the TLB shootdown overhead.

1) PRIVATE PTE DETECTION

a: DIRECT TLB INSERTION

The *direct TLB insertion* technique plays a significant role in detecting private mappings. Unlike in our previous study [16] where the user buffer was assumed to not be accessed by other cores during I/O, we fully implement the software-based *direct TLB insertion* scheme from Amit's work [15]. We can directly insert a PTE into the local TLB without setting the *access bit* of the PTE. This allows us to use the PTE's *access bit* to determine if the PTE has been cached by any remote TLBs.

After page remapping and TLB flushing, the kernel performs *direct TLB insertion* for private PTE detection (Algorithm 3, line 15). Local access to the virtual address is possible without setting the *access bit* due to the cached PTE in the local TLB. Access by remote cores triggers a hardware *page table walk*, setting the corresponding PTE's *access bit*.

Algorithm 3 Batching Procedure

```
   Input: start_addr, end_addr, target_pages[ ], nr_target_pages
1  preempt_disable( );
2  Page Remapping Procedure( );
3  if IPIALL_bitmap ≠ 0 then
4      bitmap = IPIALL_bitmap;
5      cpu = -1;
6      Bitmap-based Flush(bitmap, IPIALL, cpu);
7  foreach set bit i in zREAD_CPU_bitmap do
8      if OTHER_bitmap[i] ≠ 0 then
9          bitmap = OTHER_bitmap[i];
10         cpu = i;
11         if i == current_cpu then
12             Bitmap-based Flush(bitmap, LOCAL, cpu);
13         else
14             Bitmap-based Flush(bitmap, SINGLE, cpu);
15 Direct TLB Insertion( );     /* Please refer to [15] */
16 preempt_enable( );
```

b: DETECTION PROCESS

The kernel can eliminate unnecessary remote TLB shootdowns by checking the PTE's *access bit* on subsequent z−READ requests with the same user buffer (Algorithm 2, line 15). If the *access bit* is set, it indicates that some remote cores have accessed the page and may have the PTE cached in their TLBs. In this case, the kernel performs an IPI ALL FLUSH, similar to the default Linux behavior (line 16), by sending TLB flush IPIs to all remote cores sharing the same address space while also performing a local TLB flush. If the bit is clear, the kernel checks if the *zread_cpu*, the core that previously performed z−READ on the page, is the same as the current core (Algorithm 3, line 11). If so, a local TLB flush (LOCAL FLUSH) is sufficient (line 12); otherwise, a single remote TLB shootdown (SINGLE FLUSH) is performed for the *zread_cpu* (line 14). This optimization allows some remote TLB shootdown operations (IPI ALL FLUSH) to be replaced with either LOCAL FLUSH or SINGLE FLUSH.

For the first z−READ request with *user pages* that are not yet remapped, the kernel cannot exploit this optimization because the *access bits* are likely set when applications initialize the *user pages* prior to use (Algorithm 2, line 22). For single-threaded applications, we disable *direct TLB insertion* because the native Linux kernel performs only LOCAL FLUSH for such applications. In this case, our private PTE detection would only add unnecessary overhead.

2) BATCHED TLB SHOOTDOWN

For multiple page-sized z−READ requests, we reduce the number of TLB flushes by batching them rather than performing individual flushes for each page remapping. z−READ applies batched page remapping and TLB flushing with a batch size of 32.

a: BATCH SIZE

In terms of the latency, a global TLB flush (GLOBAL FLUSH) is preferable to performing numerous individual flushes (INDIV FLUSH); the GLOBAL and INDIV prefixes indicate how many TLB entries are invalidated at once in the local TLB, while the IPI ALL, LOCAL, and SINGLE

prefixes denote where TLB flushing is performed. GLOBAL FLUSH requires two instructions that read and write from/to the CR3 register, while every individual flush needs an *invlpg* instruction to invalidate a single page at a time. GLOBAL FLUSH is fast but results in more TLB misses, as non-target TLB entries are also invalidated. After GLOBAL FLUSH, the TLB must be refilled from the page table by hardware on every memory access. To balance the TLB flush latency and the TLB miss penalty, Linux uses a tunable threshold called *tlb_single_page_flush_ceiling*, with a default value is 33. When the address range to be invalidated is larger than 33 pages, the native kernel performs GLOBAL FLUSH. Similarly, we choose the default *BATCH_SIZE* to be 32 in order to limit the TLB flush latency.

In z−READ, when the requested I/O size is larger than 128 KB (32 pages), the kernel performs either GLOBAL IPI ALL FLUSH, GLOBAL LOCAL FLUSH, or GLOBAL SINGLE FLUSH. Even in this case, we still use private PTE detection to minimize the number of remote cores involved in TLB shootdown operations. After the first batch is done, no further TLB flushes are needed for cores whose TLBs were entirely flushed in the first batch. The kernel tracks these cores to prevent unnecessary TLB flushes.

b: BITMAP-BASED RANGED TLB FLUSHING

We implement a bitmap-based ranged TLB flush technique because the native Linux kernel only supports ranged TLB flushes with contiguous addresses from start to end. For batching, two types of bitmaps are used: an IPIALL bitmap for IPI ALL FLUSH, and multiple OTHER bitmaps for LOCAL FLUSH and SINGLE FLUSH. The number of OTHER bitmaps corresponds to the number of cores in the system, with each bit in a bitmap representing a page within a range starting from a base address. With the bitmaps and base address, the receiver of a TLB shootdown IPI can calculate the virtual addresses of the pages that need to be invalidated in the local TLB. This way, remote cores involved in a TLB flush receive an IPI only once per batch (once per I/O request for GLOBAL FLUSH).

During batched page remapping, the kernel checks the *access bit*, *zread_cpu*, and the current core ID for each page to determine the type of TLB flush to use. The result for each page is recorded in either the IPIALL bitmap or one of the OTHER bitmaps corresponding to the *zread_cpu*. These bitmaps are used for TLB flushing in a batch.

V. ADAPTIVE HYBRID I/O MODE SWITCHING

In intensive COW fault scenarios, where zero-copied data is frequently updated (e.g., mixed read/write workloads), the advantages of page remapping over page copying can be diminished due to the overhead associated with COW faults, as detailed in § II-A. Our previous work [16] did not ensure that z−READ would consistently outperform READ in these situations.

To address this limitation, we introduce the Adaptive Hybrid I/O Mode Switching (AHS) mechanism. AHS proactively identifies scenarios where z−READ's performance benefits are negated by COW-related overheads and dynamically switches between READ and z−READ modes as needed to optimize performance. This proactive approach ensures that z−READ can achieve the best possible performance even in demanding workload conditions.

TABLE 2. The measured average latencies (ns) for our performance loss model.

LAT	$READ$	$READ + mod$	$z\text{-}READ$	$z\text{-}READ + mod + cow$	$READ - z\text{-}READ$	cow
4 KB	1781	1926	1547	3972	234	2279
8 KB	3255	3557	2387	7054	869	4366
...	...	...	...	...	...	...
16 KB	5601	6766	3946	12367	1655	7257
...	...	...	...	...	...	...
32 KB	12910	13670	6578	28059	6331	20720

A. PERFORMANCE LOSS MODEL

Choosing the wrong I/O mode between READ and z−READ can lead to significant performance losses. For instance, if COW faults are inevitable for every read operation, z−READ loses its performance advantage over READ due to the added COW fault overhead. Conversely, if no COW faults occur, z−READ is the optimal choice because it offers lower latency than READ. Therefore, we use the latency of z−READ as a baseline to calculate the expected performance losses for both READ and z−READ in case of incorrect mode selection.

The expected performance losses for READ ($\mathbb{E}(Loss_{READ})$) and z−READ ($\mathbb{E}(Loss_{z\text{-}READ})$) are given by:

$$\mathbb{E}(Loss_{READ}) = (LAT_{READ} - LAT_{z\text{-}READ}) * (1 - P(cow)), \quad (2)$$

$$\mathbb{E}(Loss_{z\text{-}READ}) = LAT_{cow} * P(cow), \quad (3)$$

where LAT_{READ} and $LAT_{z\text{-}READ}$ are the latencies of READ and z−READ, respectively; $P(cow)$ is the probability of a COW fault occurring; and LAT_{cow} is the *COW overhead*, which is calculated as:

$$LAT_{cow} = (LAT_{z\text{-}READ+mod+cow} - LAT_{z\text{-}READ}) - (LAT_{READ+mod} - LAT_{READ}), \quad (4)$$

where $LAT_{z\text{-}READ+mod+cow}$ and $LAT_{READ+mod}$ are the latencies of z−READ and READ followed by modification of the buffer. Note that $LAT_{z\text{-}READ+mod+cow}$ includes the overhead caused by COW faults.

Our *helper program* measures LAT_{READ}, $LAT_{z\text{-}READ}$, $LAT_{z\text{-}READ+mod+cow}$, and $LAT_{READ+mod}$. TABLE 2 lists these measurements obtained from our system. Using these results, the *performance loss model builder* updates the *performance loss model table*. It is important to note that building the *performance loss model* is a one-time offline task.

B. HISTORICAL STATISTICS

To derive the values of $\mathbb{E}(Loss_{READ})$ and $\mathbb{E}(Loss_{z\text{-}READ})$, we must determine $P(cow)$, the probability of a COW fault

occurring. Before addressing this, it is essential to understand the conditions under which COW faults occur in `z-READ`. In `z-READ`, a COW fault arises when an application attempts to modify the contents of the I/O buffer mapped to the *kernel pages*. This application behavior is primarily driven by two causes.

The first cause involves the application's buffer reuse mechanism. For example, an application might initialize a user-level I/O buffer for reuse. If this buffer has already been remapped to the *kernel page*, the initialization action will trigger a COW fault. Since an application's buffer reuse mechanism does not change during runtime, such COW faults can be completely avoided by minor modifications to the buffer reuse mechanism, such as bypassing the initialization step when buffers are reused.

The second cause relates to the application's data access pattern, specifically read-modify-write operations. Consider a DBMS as an example. A typical DBMS manages a buffer pool comprised of *user pages* to cache table and index data in memory. When processing update requests, the DBMS follows a three-step process: ① *read*, ② *modify*, and ③ *write*. In step ①, it reads the target data from disk into the buffer pool (*user pages*) if the data is not already cached. In step ②, it modifies the data in the buffer pool. In step ③, it issues a write system call to commit these changes to disk (*kernel pages*). If the DBMS performs step ① using `z-READ`, then step ② will trigger a COW fault.

Due to the dynamic nature of data access patterns, $P(cow)$ cannot be statically determined; some data may initially undergo frequent updates but may transition to being read-mostly later. Therefore, predicting $P(cow)$ requires learning the time-varying data access patterns from historical I/O statistics.

The challenges lies in deciding which metrics to track and how to track them. One might consider using the ratio of the number of COW faults to the number of read requests for each *user page*, storing this information in the kernel's page structure. However, the correlation between *user page* and the data it contains tends to be transient. For instance, when the buffer pool is full, the DBMS flushes the data from the victim *user page* to disk, transferring it to a *kernel page*. The DBMS then reuses the victim *user page* for caching other data. In such scenarios, it becomes more accurate to associate the data with the *kernel page* rather than the *user page*.

However, directly correlating COW faults to specific *kernel pages* presents challenges, as the fault might not be related to the data in that *kernel page*. For instance, if buffer initialization is the fault's cause, the data within the *kernel page* is irrelevant. To address this, we adopt the read/write ratio for each *file page* as a proxy to infer the probability of a COW fault occurring. This approach aligns well with common application behaviors during persistent updates that follow a read-modify-write pattern. Therefore, we can treat a single write system call targeting a specific *kernel page* as indicative of a single COW fault.

C. AHS IMPLEMENTATION

We implement the *I/O profiler* in kernel to monitor the number of read and write I/O operations for each *kernel page*. To maintain a history of I/O events, we augment the kernel's page structure with two counters: one for reads and another for writes. The *I/O profiler* updates these counters accordingly with every I/O event, incrementing the read or write for the associated *kernel page*. Leveraging this historical data from the *I/O profiler* and the *performance loss model* from the *performance loss model builder*, the *adaptive hybrid I/O mode switcher* dynamically calculates $\mathbb{E}(Loss_{READ})$ and $\mathbb{E}(Loss_{z\text{-}READ})$. The switcher then opts for `READ` when $\mathbb{E}(Loss_{z\text{-}READ})$ exceeds $\mathbb{E}(Loss_{READ})$, and favors `z-READ` in the opposite case. For example, if a *kernel page* has historically been read 10 times and written twice, we can choose the optimal mode by comparing the performance benefits of `z-READ` for 10 reads against the COW fault overhead of `z-READ` for 2 writes, assuming that future access patterns will be similar to past ones.

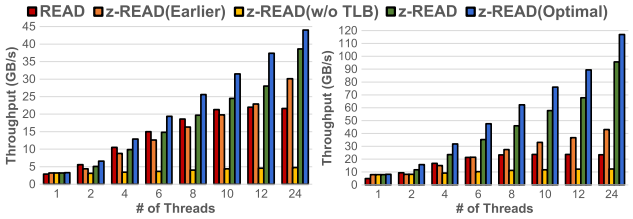
VI. EVALUATION

We implement a `z-READ` prototype in Linux kernel 4.12.9. We have modified or added less than 1000 lines of code across the read I/O path, COW fault handler, and page unmapping handler in the kernel. Since the core of these components remains unchanged in the latest kernel (e.g., 6.6.16), our scheme can be imported into newer kernels, which we plan to explore in future work. We evaluate the individual effectiveness of each of the two major optimization techniques presented in this paper and the overall effectiveness and performance interference intensity of `z-READ`. We compare several variants of our scheme to `READ`. The variants of our scheme are as follows:

- `z-READ(Earlier)`. This is the earlier version of `z-READ` from our previous work [16].
- `z-READ`. This is the enhanced version of `z-READ` that includes the full implementation of *direct TLB insertion* and the adaptive hybrid I/O mode switching (`AHS`).
- `z-READ(w/o TLB)`. This is a variant of `z-READ` for which private TLB detection is disabled.
- `z-READ(w/o AHS)`. This differs from `z-READ` in that `AHS` is disabled.
- `z-READ(Optimal)`. This represents the optimal performance of `z-READ` if we know which core caches which PTEs without the private PTE detection overhead. To emulate it, we force the kernel to perform a local TLB flush only, having each job running on a dedicated core.

A. SYSTEM CONFIGURATIONS

Experiments are performed on a non-uniform memory access (NUMA) system with two nodes, each of which is equipped with an Intel Xeon Processor E5-2650 v4 running at 2.2 GHz (12 cores/24 threads) supporting four memory channels and 128 GB (32 GB × 4) DDR4 memory running at 2133 MHz. We use two 375 GB Intel Optane DC P4800X



(a) FIO 4KB Random Read (b) FIO 16 KB Random Read

FIGURE 3. The effectiveness of the TLB shootdown optimization.

SSDs for the underlying storage devices. For the evaluations with the on-disk configuration, we configure the workload to use them simultaneously.

Since we do not have a sufficient number of storage devices to saturate the entire memory bandwidth, we choose to use only two memory channels to limit the maximum memory bandwidth, by installing memory into four DIMM slots associated with two memory channels in a node. However, the sustainable memory bandwidth of our system is not significantly different from when it uses four memory channels (25 GB/s vs. 28 GB/s).

All of the benchmarks used in the evaluations are either run until the stable experimental results are obtained or executed 10 times for the average results. We omit error bars from these results since the observed variances are negligible. In order to isolate our results from NUMA effects, we use only one node with help of the *numactl* command.

Our evaluations were conducted with swapping disabled, as our prototype currently does not support page reclamation for remapped pages. We also disable transparent huge pages (THP) due to the lack of support for our prototype. In practice, we can simply enforce the kernel to use `READ` for huge pages.

B. EFFECTIVENESS OF THE TLB SHOOTDOWN OPTIMIZATION

We first evaluate the effectiveness of our TLB shootdown optimization using FIO, a Flexible I/O tester synthetic benchmark [17] with the 4 KB/16 KB random read workload. To study the scalability of our technique, we vary the number of jobs from 1 to 24 because our testbed is equipped with 12 cores supporting 24 threads. Each job involves reading from the 128-MB-sized file to the 4 KB/16 KB user buffer allocated for the job. We always perform a dummy run with the invalidate flag clear before every experiment, fully warming up the page cache. Note that our main memory capacity is sufficiently large to hold all of the file contents in memory so that all I/Os result in a page cache hit.

We also use the pthread/thread model instead of the default fork/process model. In the fork/process model, where each job has its own address space, even the native Linux kernel can minimize the number of remote TLB shootdowns, only performing a local TLB flush on the core running the job (refer to § VIII-B for details). In other words, the results of FIO with the default fork/process model cannot show how much our TLB shootdown optimization affects the I/O performance.

a: EXPERIMENTAL RESULTS

We present the performance results in FIGURE 3. We evaluate `READ`, `z-READ(Earlier)`, `z-READ(w/o TLB)`, `z-READ`, and `z-READ(Optimal)`. The results show that our prototype is scalable and does not encounter any bottlenecks from memory bandwidth. The performance of `READ` only scales until it saturates the memory bandwidth (about 25 GB/s), while both `z-READ(Earlier)` and `z-READ` scale linearly, even surpassing the maximum theoretical memory bandwidth. This is possible because they do not consume memory bandwidth until the data is actually accessed.

Compared to `READ`, `z-READ` improves the I/O performance by up to 1.79 times for the 4 KB random read (up to 4.07 times for the 16 KB random read). The full implementation of the *direct TLB insertion* scheme in `z-READ` helps achieve a further improvement over `z-READ(Earlier)`. This is because *emulated direct TLB insertion* in our previous work [16] uses a PTE lock, which is shared by 512 pages in the native kernel, when clearing the *access bit* of each PTE. This limits the scalability when other cores access the pages among the 512 pages that share the same PTE lock. In contrast, `z-READ` can insert the PTE into the local TLB without setting the *access bit*; thus, no clearing process is needed.

A comparison of `z-READ(w/o TLB)` with `z-READ` and `z-READ(Earlier)` shows the effectiveness of private PTE detection. Without it, the performance of `z-READ` severely decreases because every page remapping requires a remote TLB shootdown to all cores sharing the same address space. The performance of `z-READ(w/o TLB)` does not scale at all. It is inferior to `READ` in every respect, except for single-threaded applications.

As shown in FIGURE 3(a), even with our TLB shootdown optimization, the I/O latency of `z-READ` can be slightly worse than that of `READ` for 4 KB read I/O when the memory bandwidth is not saturated. This means that the costs of page remapping and TLB flushing for each page can be higher than the page copy overhead for 4 KB read I/O. In FIGURE 3(b), we can observe that the performance gap between `READ` and `z-READ` becomes wider when the I/O size is increased from 4 KB to 16 KB. This result demonstrates that the TLB shootdown batching technique enhances the I/O performance for multiple-page-sized I/O requests, reducing the number of remote TLB shootdowns. We can conclude that `z-READ` becomes effective in terms of the latency when the I/O size is larger than 8 KB. Large I/O sizes are common in practice. Many high-performance computing applications, data analytics tasks, and scientific computing workloads read data in large chunks, often exceeding the size of a single page. Additionally, the default page size of the buffer pool of DBMS such as MySQL/InnoDB is 16 KB.

The result indicates that, although page remapping can increase TLB misses and shootdowns, the overhead from

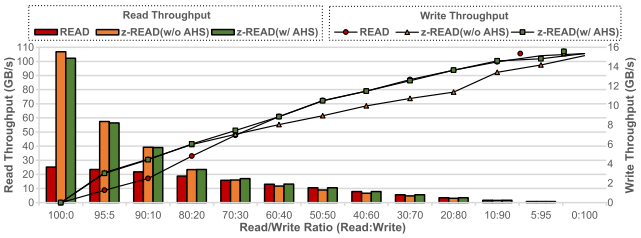


FIGURE 4. The effectiveness of AHS.

page copying still surpasses that caused by TLB misses and shootdowns when applying various TLB optimizations. There is still room for improvement in reducing the private PTE detection overhead, which can be inferred from a comparison of the increased performance of `z-READ(Optimal)` relative to that of `z-READ`. Aligned with the conclusion of Amit's work [15], the most overhead can be avoided by simple hardware enhancements.

C. EFFECTIVENESS OF AHS

In order to demonstrate the effectiveness of `AHS`, we run FIO with almost the same configuration as the previous evaluation, but a mixed random read/write workload is used instead. We vary the read/write ratio of the workload while fixing the number of jobs to 24. Please be advised that for the write workload, FIO writes random data to the user buffer before issuing a write system call; thus, every write results in a COW fault with `z-READ`. We evaluate `READ`, `z-READ(w/o AHS)`, and `z-READ(w/ AHS)`.

Results: FIGURE 4 shows the 16 KB read and write I/O throughputs for various read/write ratios. For `z-READ(w/o AHS)`, when the read/write ratio is 95:5, the read performance increases by 325% compared to that of `READ`. However, its performance becomes worse than that of `READ` as the ratio of writes increases (a slowdown of 15% for a 50:50 ratio compared to `READ`). The reason for this is due to the COW fault overhead, which is relatively large compared to the performance gain from `z-READ`.

Adaptively switching between `READ` and `z-READ` can resolve this problem. For `z-READ(w/ AHS)`, the read performance and write performance are better than or equal to those of `READ` for all read/write ratios. Its performance is preserved even under dynamic workloads because `z-READ(w/ AHS)` changes the I/O mode between `z-READ` and `READ` for each page on the basis of the read/write history. Due to the decision overhead, `z-READ(w/ AHS)` performs slightly worse than `z-READ(w/o AHS)` for read-heavy workloads (100:0, 95:5).

Since a COW fault occurs only after performing `z-READ`, there is less chance for COW faults for a write-intensive workload. Consequently, the performance gap between `READ` and `z-READ` becomes narrow after a ratio of 40:60.

D. ANALYSIS OF THE PERFORMANCE IMPROVEMENT

The key contributing factor to the improved performance of `z-READ` is the *lazy cache misses*. In `z-READ`, cache misses for the requested data can be deferred until the data are actually accessed. On the other hand, `READ` incurs cache misses during a page copy. This *eager cache miss* might be beneficial if and only if an application accesses all of the data right after a read. However, some applications may access only part of the data or access the data at a later time after it is evicted from the LLC.

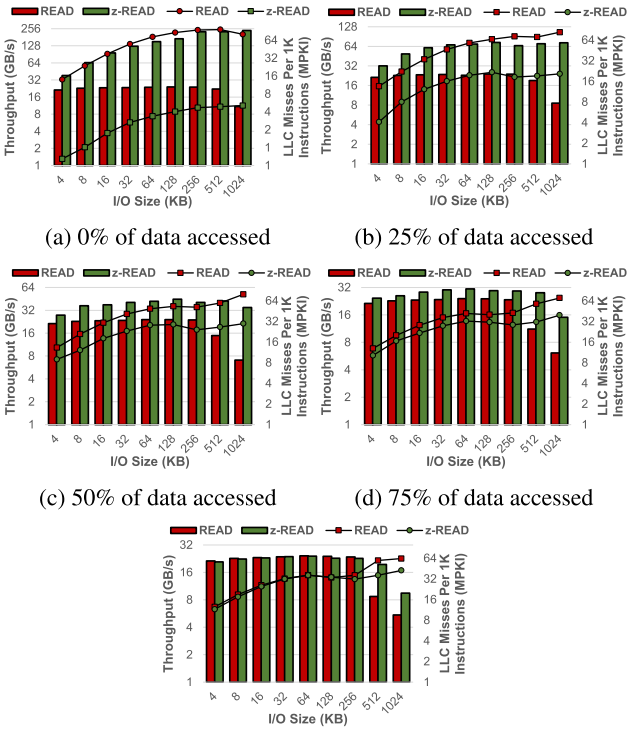


FIGURE 5. Relationship between the I/O throughput and the LLC MPKI.

To determine the relationship between the amount of data accessed after a read and the performance, we use a modified version of FIO that copies a certain amount of data from the I/O buffer to another buffer after reading data from a file. We vary the amount of copied data and the I/O size while keeping the number of jobs fixed to 24.

Results: The results are plotted on a logarithmic scale with base 2 in FIGURE 5, where the primary y axis is the I/O throughput and the secondary y axis is the number of LLC MPKI (misses per kilo instructions). As expected, the performance is inversely proportional to the number of LLC MPKI regardless of the amount of data copied. As the amount of copied data increases, the performance gap between `READ` and `z-READ` becomes narrow due to the increased number of cache misses in `z-READ`. When all of the data is copied, the performance of `z-READ` is marginally lower than that of `READ`, except for large I/O sizes.

For large I/O sizes (512 KB and 1 MB), the performance of `READ` decreases because the 30 MB of LLC in our system cannot retain the data between the time of I/O and the time of data access; cores compete for the LLC. Therefore, copying more than 50% of the data with `READ` shows a

TABLE 3. SPEC CPU 2006 benchmarks.

Memory Intensity	Benchmark
High	470.lbm, 462.libquantum
Medium	403.gcc, 401.bzip2
Low	481.wrf, 453.povray

TABLE 4. Mixed workloads.

	Workloads
mix1	lbm + lbm + gcc + gcc + wrf + wrf
mix2	libquantum + libquantum + bzip2 + bzip2 + povray + povray
mix3	lbm + libquantum + gcc + bzip2 + wrf + povray

similar tendency. However, due to the *lazy cache misses*, even with the full data copy, `z-READ` maintains better I/O performance than `READ`, avoiding some unnecessary cache misses. This characteristic is particularly favorable for server consolidation environments.

E. PERFORMANCE INTERFERENCE INTENSITY

To demonstrate that an I/O-intensive workload can cause harmful interference to co-located workloads, we run a mixture of several applications from SPEC CPU 2006 [22] in the foreground while running the 16 KB random read workload from the FIO benchmark on the other cores on the same node in the background indefinitely. We repeatedly run the foreground applications until all of them are finished at least once and then measure the execution time of each of them for the first run. To minimize the effects of other sources of interference caused by resource sharing, we allocate six physical cores to FIO while allocating the other six cores in a node to the foreground applications. For this evaluation, we set the number of jobs to 12 for FIO. Comparing the execution times of the co-located applications for each respective read I/O scheme, we indirectly evaluate the performance interference intensity of `READ` and `z-READ` under two circumstances: in-memory and on-disk.

For the on-disk configuration, we use *cgroup* to limit the memory available to FIO to 1 GB while forcing each job to read from its own 10-GB-sized file (total 120 GB for 12 threads). Note that the configured memory limit also applies to the amount of page cache used by FIO; as a result, most read requests must access the disk directly.

According to the memory intensity, we choose six benchmarks from SPEC CPU 2006 on the basis of the benchmark classification results from [23], [24]. The benchmarks used in this evaluation are listed in TABLE 3. For the foreground workloads, we run either six identical benchmarks or a mixed workload that consists of several benchmarks. Three mixed workloads are summarized in TABLE 4. We use *ref* as the input for SPEC CPU 2006.

Results: The slowdown of the foreground workloads when running with the I/O-intensive background workloads are shown in FIGURE 6. Note that we use the geometric mean to compute the average slowdown of the foreground applications. In the in-memory I/O configuration in FIGURE 6, `z-READ` scarcely affects the execution time of the co-located workloads (only a slowdown of 11%) while the workloads with `READ` suffer a slowdown of 84%. FIGURE 6 demonstrates that even in the on-disk I/O configuration, copy-based I/O (`READ`) can severely interference with the performance of co-running applications, incurring a slowdown of 31% on average. For the same case, the co-located workloads are 18% slower on average with `z-READ`. Interestingly, with `z-READ`, the on-disk I/O configuration causes more slowdown than the in-memory configuration. This originates from the default Linux kernel behavior that zeros out the new page cache before reading the data from disk, which consumes shared memory resources. By avoiding memory copy, `z-READ` can outperform `READ` by up to 2.1×.

TABLE 5. Parameters for the Filebench workloads.

Workload	Configuration
webserver	filesize=16KB, 32KB, 64KB, 128KB iosize=1MB
varmail	meanappendsize=16KB, 32KB, 64KB, 128KB filesize=16KB, 32KB, 64KB, 128KB iosize=1MB
fileserver	filesize=128KB, 256KB, 512KB, 1MB iosize=1MB meanappendsize=16KB
videoserver	readiosize=128KB, 256KB, 512KB, 1MB writeiosize=1MB filesize=500MB

F. MACROBENCHMARKS

Since our scheme is specifically optimized for file I/O, we choose the Filebench benchmark [18] as a macro benchmark for our evaluation. We use four macroworkloads from Filebench [18]: *webserver*, *varmail*, *fileserver*, and *videoserver*. We run them with the parameter settings listed in TABLE 5. For the *videoserver* workload, we eliminate the *bwlimit* workflow to measure the peak I/O performance. For each workload, we vary different parameters that affect the actual I/O size for a single read system call. For example, for *webserver* and *fileserver*, we change the file size while keeping the I/O size fixed to 1 MB because the actual I/O size is the same as the file size when the file size is shorter than the requested I/O size. On the other hand, *varmail* uses *meanappendsize* when writing data to a newly created file. Since it reads the entire file after that, *meanappendsize* is directly related to the actual I/O size and the file size.

Results: We evaluate `READ` and `z-READ` in the in-memory configuration where all data are in memory. We present the performance results from Filebench in FIGURE 7.

For the *webserver* workload in FIGURE 7(a), which emulates a web server that performs open-read-close operations, the improvement in performance obtained by `z-READ` is not noticeable (about 5%) when the file size is small (16 KB).

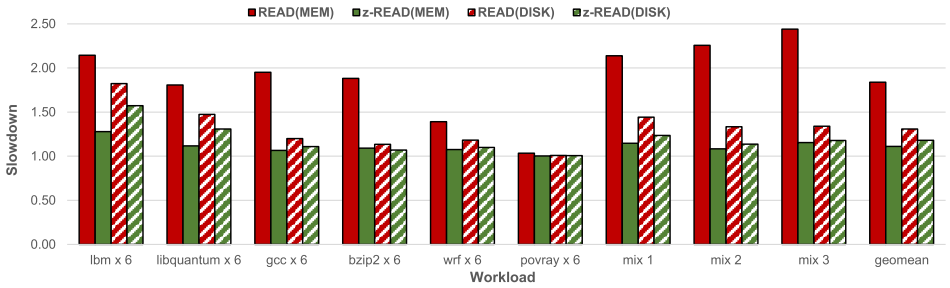


FIGURE 6. Slowdown of co-located workloads.

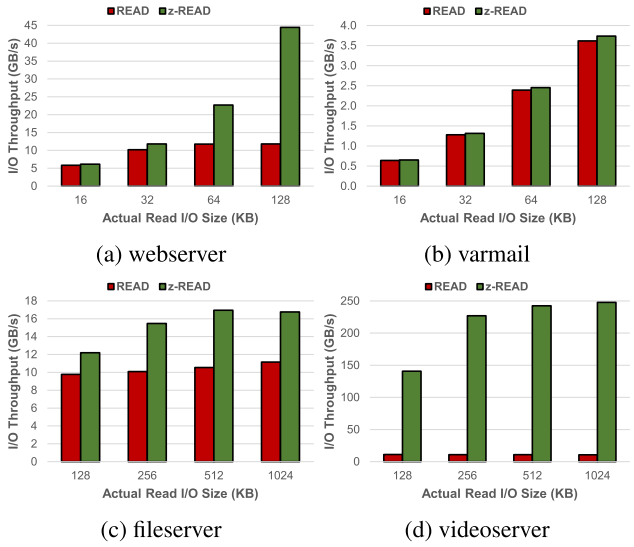


FIGURE 7. Performance on macrobenchmarks.

However, it improves the throughput by up to 3.8 times as the file size increases. Because of the in-memory configuration, the overheads for opening and closing a file are considerable, while these overheads are consistent regardless of the file size. Therefore, for a larger file size, the read performance more greatly affects the overall performance.

For the *varmail* workload in FIGURE 7(b), there is only small performance gain with `z-READ` (less than 3%). The workload emulates a mail server that performs open-create-append-sync-close, open-read-append-sync-close, open-read-close, and delete operations. As a result, the read I/O contributes only a small fraction of the application runtime; thus, the read performance has little impact on the overall performance.

In FIGURE 7(c), although the *fileserver* workload performs create-delete-append-read-write-stat operations, `z-READ` improves the throughput by up to 68% compared to `READ`. This is due to the relatively large file size. A performance gain is realized when an entire file is read.

For the *videoserver* workload in FIGURE 7(d), which performs delete-create-write-close and read operations, `z-READ` outperforms `READ` by 12.7–23.3 times. As expected, since *videoserver* is the most read-intensive workload, it can considerably benefit from the use of `z-READ`.

VII. DISCUSSION

A. LIMITATIONS

While `z-READ` generally provides better performance than `READ`, there are scenarios where `z-READ` might result in lower performance. Despite the TLB shootdown optimizations, performance can degrade if only remote TLB shootdown operations (`IPI ALL FLUSH` are occurring; refer to `z-READ(w/o TLB)` in FIGURE 3. After reading data into the user buffer with `z-READ`, access to the user buffer by a different core sets the access bit, leading to subsequent `z-READ` operations for the same buffer to trigger an `IPI ALL FLUSH`. This situation can arise in applications that separate worker threads and I/O threads. Pairing the worker thread and I/O thread per core will ensure data is always accessed by the same core that performed `z-READ`, resulting in only local flushes being required.

Additionally, if eager cache miss is more beneficial than lazy cache miss, such as in scenarios with 100% LLC cache hit rates, `READ` may outperform `z-READ`. However, this is uncommon. For most of cases, applications performing I/O access the data either partially or at a later time. For example, in DBMS, only the rows needed to process the user query are accessed from the *user page* after being read into the buffer pool.

B. VERSUS EXISTING ZERO-COPY SCHEMES

`z-READ` is not a one-size-fits-all solution. We compare `z-READ` with well-established zero-copy schemes such as `direct I/O` with user-level buffering and `mmap`.

1) vs. DIRECT I/O WITH USER-LEVEL BUFFERING

Direct I/O with user-level buffering is highly effective for zero-copy I/O, particularly in applications that manage their own caching. It eliminates CPU overhead associated with copying data between kernel and user space. When data is already present in the user-level buffer (buffer hit), not only are I/O operations avoided, but the need for system call invocation is also eliminated, further reducing latency and improving performance. Additionally, this approach allows applications to implement custom caching and replacement policies tailored to their specific workloads, offering greater control over memory management.

However, this approach bypasses the kernel page cache, which can result in redundant data loading when multiple applications access the same file. For example, in a logging system where one application writes logs and another reads them, each application must independently load the data into their own buffer.

In contrast, `z-READ`, when combined with user-level buffering, retains the benefits of `direct I/O`—such as avoiding both I/O operations and system calls on buffer hits—while also leveraging the kernel's page cache. This approach not only prevents data duplication by sharing kernel pages, similar to `mmap`, but also improves performance by keeping frequently accessed data in system memory beyond the size of the user-level buffer.

In single-application workloads or scenarios where multiple applications rarely access the same data, the advantage of `z-READ`'s shared page caching becomes less significant. In such cases, the simplicity of `direct I/O` with user-level buffering may offer better performance without the need to address the limitations associated with `z-READ`.

2) vs. mmap

`mmap` is highly efficient for zero-copy I/O, enabling applications to access file contents directly using memory-like semantics. By leveraging the kernel's page cache, `mmap` ensures that frequently accessed data is kept in memory, improving I/O performance without requiring developers to implement explicit buffering. Its straightforward approach to direct memory access makes it particularly effective for applications with smaller datasets that fit within available physical memory.

For applications already using `mmap`, switching to `z-READ` may offer limited advantages. However, for applications not currently utilizing `mmap`, `z-READ` provides a simpler integration path, minimizing the need for substantial changes to the existing code. Furthermore, `z-READ` is better suited for applications dealing with datasets larger than physical memory, as it combines the use of a small user buffer with efficient system memory management through the kernel page cache.

VIII. RELATED WORK

A. ZERO COPY I/O

1) DIRECT I/O SCHEMES

There are zero copy schemes that support direct user-level access to devices. EMP [11] is a zero copy message passing layer, completely implemented on the programmable Alteon NIC. Thanks to their message passing interface (MPI) support, a user application can directly access the NIC through the MPI. Kesavan et al. [12] explored the application of zero copy NIC DMA to in-memory databases. These works rely on the device's DMA capabilities, which allow the device to directly access the host memory. Our study is in line with these works in terms of eliminating memory copy overhead. However, in contrast to their focus on networking, we focus on storage I/O.

2) MEMORY-MAPPED I/O SCHEMES

There are other approaches that provide applications with the ability of accessing the shared buffer. Druschel and Peterson [9] presented *fbufs*, a fast buffer for a network subsystem based on immutable buffers that can be shared across a domain. Thadani and Khalidi's work [13] and IO-Lite [14] both extended *fbufs* from the network to the file system for general-purpose OSs (UNIX). While our work is also based on the memory-mapped I/O scheme, the main difference from above is the application transparency. Their non-transparent approaches may provide higher I/O performance than transparent approaches including ours, by skipping the tasks required for application transparency. However, a major disadvantage of the non-transparent approaches is that applications have to be redesigned to use new application programming interfaces (APIs) and programming models, which may significantly burden developers.

Unlike the aforementioned studies, Chu [10] presented a transparent zero copy scheme based on page remapping and COW techniques. It only required minimal changes (e.g., buffer alignment) to an application, keeping existing interfaces unchanged. However, it targeted the network subsystem, which have several different characteristics from our target, the storage I/O subsystem. Moreover, it does not consider TLB shootdown costs in multicore environments.

B. TLB SHOOTDOWN

Many efforts have been devoted to TLB shootdown mitigation [15], [25], [26], [27]. The Linux kernel tracks the address space that is active on each core and only sends TLB shootdown IPIs to the cores that currently use the address space to which the target TLB mapping belongs. By reducing the number of the cores that are involved in a TLB shootdown operation, this optimization helps to avoid some unnecessary shootdowns. For instance, for single-threaded applications, no remote TLB shootdowns are required for TLB synchronization, only local TLB flushes. However, for multi-threaded applications, the kernel still must send IPIs to all of the cores that share the same address space.

Villavieja et al. [25] presented *DiDi*, a new hardware architecture with a shared second-level TLB. This second-level TLB contains all of the information about which TLBs hold which PTEs, thus allowing lightweight TLB invalidation. Awad et al. [26] proposed the concept of self-invalidating TLB entries that allow the OS to skip TLB shootdown operations for the *expired* TLB entries. However, both studies require hardware changes.

LATR [27] is a lazy TLB shootdown approach for virtual memory operations such as *free* and *page migration*. Its key idea is to defer TLB invalidation until a context switch

and to prevent the reuse of virtual and physical memory in the meantime. However, unlike the case of *LATR*, TLB shootdown cannot be deferred in a transparent zero copy file I/O scheme. To keep the semantics of `READ`, TLB synchronization has to be handled before `READ` returns to the user mode. Otherwise, other cores might use the stale mapping to access the remapped buffer.

Amit [15] introduced a software-based *direct TLB insertion* technique to avoid the TLB shootdowns. Amit's approach, however, can impose significant overheads (up to 9%) for general applications that do not frequently modify memory mappings, because it performs direct TLB insertion for every demand paging request. Hence, only a handful of applications can benefit from these optimizations. In contrast, we introduce a new application area (storage I/O) for these optimization techniques. Software-based direct TLB insertion aligns perfectly with storage I/O because (1) storage I/O is used more repetitively across many applications compared to memory mapping operations, and (2) in `z-READ`, direct TLB insertion occurs only for read I/O requests and not for every demand paging request, avoiding performance penalties for other standard applications.

C. COPY-ON-WRITE

The COW technique has been used in many studies to improve system performance and resource utilization [28], [29], [30]. However, one can reasonably infer that the performance gains in these studies arise because COW faults are relatively rare compared to the number of avoided copy operations. When COW faults occur frequently, eager copying becomes a more effective strategy than lazy copying (COW). Our focus is to anticipate the likelihood of occurrence of COW faults and to dynamically decide whether to employ zero-copy techniques.

IX. CONCLUSION

We have presented `z-READ`, an efficient, transparent, and practical zero copy read file I/O scheme, and several optimization techniques that reduces two major sources of overhead: page remapping and COW faults. `z-READ` provides better I/O performance, even for mixed read/write workloads that incur COW faults, while significantly reducing the interference with the performance of co-located workloads in server consolidation environments.

ACKNOWLEDGMENT

This article is based on Chapter 4 of the author's dissertation [1].

REFERENCES

[1] J. Park, "Optimizing memory subsystem for efficient resource utilization of data-intensive applications," Ph.D. thesis, Seoul Nat. Univ., Seoul, South Korea, 2020.

[2] G. Somani and S. Chaudhary, "Application performance isolation in virtualization," in *Proc. IEEE Int. Conf. Cloud Comput.*, Mar. 2009, pp. 41–48.

[3] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 237–250.

[4] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI 2: CPU performance isolation for shared compute clusters," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, Apr. 2013, pp. 379–391.

[5] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Proc. 21st Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process.*, Feb. 2013, pp. 233–240.

[6] H. Liu and B. He, "F2C: Enabling fair and fine-grained resource sharing in multi-tenant IaaS clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2589–2602, Sep. 2016.

[7] L. Tomás, C. Vázquez, J. Tordsson, and G. Moreno, "Reducing noisy-neighbor impact with a fuzzy affinity-aware scheduler," in *Proc. Int. Conf. Cloud Autonomic Comput.*, Sep. 2015, pp. 33–44.

[8] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. Sebastopol, CA, USA: O'Reilly Media, 2005.

[9] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," *ACM SIGOPS Operating Syst. Rev.*, vol. 27, no. 5, pp. 189–202, Dec. 1993.

[10] H.-K. J. Chu, "Zero-copy TCP in solaris," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 1996, p. 21.

[11] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-copy OS-bypass NIC-driven gigabit Ethernet message passing," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2001, pp. 57–57.

[12] A. Kesavan, R. Ricci, and R. Stutsman, "To copy or not to copy: Making in-memory databases fast on modern NICs," in *Data Management on New Hardware*. Cham, Switzerland: Springer, 2016, pp. 79–94.

[13] M. N. Thadani and Y. A. Khalidi, "An efficient zero-copy I/O framework for UNIX," Sun Microsystems Laboratories, Santa Clara, CA, USA, Tech. Rep., 1995.

[14] V. S. Pai, P. Druschel, and W. Zwaenepoel, "IO-lite: A unified I/O buffering and caching system," *ACM Trans. Comput. Syst.*, vol. 18, no. 1, pp. 37–66, Feb. 2000.

[15] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2017, pp. 27–39.

[16] J. Park, C. Min, H. Y. Yeom, and Y. Son, "Z-READ: Towards efficient and transparent zero-copy read," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 367–371.

[17] J. Axboe. (2022). *Fio-Flexible IO Tester*. [Online]. Available: https://github.com/axboe/fio

[18] V. Tarasov, E. Zadok, and S. Shepler, "FileBench: A flexible framework for file system benchmarking," *USENIX Mag.*, vol. 41, no. 1, pp. 6–12, 2016.

[19] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled direct memory access: Isolating CPU and IO traffic by leveraging a dual-data-port DRAM," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Oct. 2015, pp. 174–187.

[20] J. Park, C. Min, and H. Yeom, "A new file system I/O mode for efficient user-level caching," in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2017, pp. 649–658.

[21] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped I/O on fast storage device," *ACM Trans. Storage*, vol. 12, no. 4, pp. 1–27, Aug. 2016.

[22] C. D. Spradling, "SPEC CPU2006 benchmark tools," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, Mar. 2007.

[23] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2013, pp. 55–64.

[24] D. H. Yoon, M. K. Jeong, and M. Erez, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2011, pp. 295–306.

[25] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, Oct. 2011, pp. 340–349.

[26] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding TLB shootdowns through self-invalidating TLB entries," in *Proc. 26th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2017, pp. 273–287.

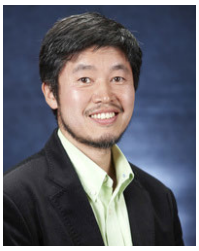
[27] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy translation coherence," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 651–664.
[28] L. Shrira and H. Xu, "Thresher: An efficient storage manager for copy-on-write snapshots," in *Proc. USENIX Annu. Tech. Conf.*, 2006, pp. 57–70.
[29] X. Wu, W. Wang, and S. Jiang, "TotalCOW: Unleash the power of copy-on-write for thin-provisioned containers," in *Proc. 6th Asia–Pacific Workshop Syst.*, 2015, p. 15.
[30] M. Chowdhury and R. Rangaswami, "Native OS support for persistent memory with regions," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2017.



JIWOONG PARK received the B.S. degree in computer science from Korea University, in 2013, and the Ph.D. degree in computer science and engineering from Seoul National University, in 2020. Before joining SSWU, in 2021, he was with Samsung Electronics as a Staff Engineer, from 2020 to 2021. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Sungshin Women's University. His research interests include operating, database, and distributed storage systems.



CHEOLGI MIN received the B.S. degree in computer science and engineering from Hanyang University, in 2016, and the M.S. degree in computer science and engineering from Seoul National University, in 2018, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include parallel programming and database systems.



HEONYOUNG YEOM was a tenured Professor with the Department of Computer Science and Engineering, Seoul National University, where he left an indelible mark over a distinguished career that spanned more than three decades. He contributed to numerous breakthroughs in the fields of OS, networks, storage systems, database, cloud computing, and distributed systems, publishing more than 300 papers in refereed journals and conferences. He passed away peacefully, in February 2024, leaving behind a profound impact on the world of computer engineering. His contributions will continue to influence the field for years to come. He will be deeply missed by his colleagues, students, and the many friends, he made throughout his career.



YONGSEOK SON received the B.S. degree in information and computer engineering from Ajou University, in 2010, and the M.S. and Ph.D. degrees from the Department of Intelligent Convergence Systems and Electronic Engineering and Computer Science, Seoul National University, in 2012 and 2018, respectively. He is currently an Associate Professor with the Department of Computer Science and Engineering, Chung-Ang University. He was a Postdoctoral Research Associate in electrical and computer engineering with the University of Illinois at Urbana–Champaign. His research interests include operating, distributed, and database systems.