

PipeBWrite: Shortening the Buffered Write Critical Path with Deferred Non-Critical Steps

Luofan Chen[†] Jiyang Wang[†] Guoyu Wang[†] Chongzhuo Yang[†] Sam H. Noh[§]
Kang Chen[‡] Cheng Li[†¶]

[†]*University of Science and Technology of China* [‡]*Tsinghua University* [§]*Virginia Tech*
[¶]*Institute of Artificial Intelligence, Hefei Comprehensive National Science Center*

Abstract

Buffered I/O is the dominant storage interface, but its synchronous and monolithic write path limits performance for write-intensive applications. Prior work either applies narrow optimizations to individual steps of the write path or adopts asynchronous designs that require application changes.

In this paper, we present PipeBWrite, a transparent buffered write scheme that retains the synchronous interface while extracting the performance benefits of asynchronous execution. Based on data dependency analysis, PipeBWrite decomposes buffered writes into a two-stage pipeline: the foreground thread performs only the critical data copy needed for immediate user-buffer reuse, while non-critical kernel bookkeeping is offloaded to background threads. To ensure correctness, PipeBWrite introduces a lightweight inter-request visibility tracker that preserves POSIX semantics despite decoupled execution. It also employs holistic optimizations, including adaptive allocation, inline zeroing, and request batching, to minimize coordination overhead. PipeBWrite is implemented in the Linux Virtual File System layer and requires no application changes. Across four applications with the ext4 and XFS filesystems, PipeBWrite improves throughput by up to 25% for RocksDB and 21% for Kafka over vanilla Linux and StreamCache. For FIO synthetic workloads, it achieves up to 2.34× higher throughput and reduces write latency by up to 58%.

1 Introduction

Buffered I/O is a transparent and easy-to-use page caching mechanism that leverages DRAM for fast data access while maintaining compatibility across diverse filesystems. It is widely used by applications in domains such as databases, analytics, and AI/HPC. Table 1 shows that all 15 representative applications we surveyed primarily use buffered I/O.

Despite its critical role, the buffered write path imposes performance bottlenecks on applications. When an application issues a write, it is synchronously blocked on the critical path until the kernel completes a complex sequence of operations. This blocking nature significantly hinders application responsiveness and scalability. For example, we observe that buffered writes account for 34% of the total execution time in RocksDB (YCSB-A workload), 62% in the `cp` command, and 28% in Kafka's producer benchmark.

A deeper analysis of this critical path yields two key observations. First, the buffered write path involves far more than simply copying data from user space to kernel memory; it has additional steps including page cache XArray lookup and insertion, filesystem metadata mutations, and dirty tracking. Second, the latency breakdown reveals a counterintuitive fact. While "data copying" is the single largest contributor to latency, the remaining non-copy operations collectively consume over half of the total execution time.

Prior work has optimized individual steps along the buffered write path, such as page allocation [24, 40, 55] and metadata operations [46, 57]. However, since the latency is distributed across all six steps, these approaches focusing on point optimizations independently yield limited overall benefits. Another line of work proposes asynchronous write interfaces with approaches like xsyncfs [44] and io_uring [4]. However, these benefits require altering OS interface semantics, mandating substantial application refactoring. Our survey of 15 applications finds that only three support such interfaces, and even then only as an optional feature.

To address these issues, we propose PipeBWrite, a new buffered write mechanism that combines the transparency of synchronous interfaces with the performance benefits of asynchronous execution. PipeBWrite decomposes the monolithic write path into a two-stage pipeline. The foreground thread performs only the minimal steps required to preserve POSIX synchronous semantics, offloading the remaining time-consuming operations to background kernel threads. This enables earlier return to user space, reducing application blocking on the write critical path, while background threads utilize spare CPU cycles to improve throughput. Implemented at the Linux VFS layer and adapted to ext4 and XFS, PipeBWrite is fully transparent to applications.

To realize PipeBWrite, we conduct a rigorous data dependency analysis that identifies the minimal set of operations

Table 1: Survey of I/O interfaces in representative software systems. "()" refers to optional usage.

Domain	Application	Buffered I/O	Direct I/O	Async Interface	Data to be written
Database	PostgreSQL [29]	✓	(✓)	(`io_uring`)	Use buffered writes for WAL and data pages.
	MongoDB [8]	✓	(✓)	×	Use buffered writes for journaling and data files.
	RocksDB [15]	✓	(✓)	(`io_uring`)(read)	Use buffered writes to do compaction.
	InfluxDB [33]	✓	×	×	Use buffered writes for WAL and data files.
Data Analytics	Apache Kafka [1]	✓	×	×	Use buffered writes to write to log files.
	GraphChi [38]	✓	×	×	Use buffered writes for shard updates.
	Python (NumPy) [10]	✓	×	×	Use buffered writes to do standard file operations.
System Infrastructure	QEMU/KVM [14]	✓	(✓)	(`io_uring`)	Use buffered writes by default to cache guest I/O.
	Git [5]	✓	×	×	Use buffered writes to write object and packfile.
	Nginx [9]	✓	(✓)	(`libaio`)	Use buffered writes to write logs.
	CLI Utils [3]	✓	(✓)	×	Most utilities (`cp`, `tar`) use buffered I/O.
	Log4j [2]	✓	×	×	Use buffered writes to write logs.
AI & HPC	PF3DIO [19]	✓	×	×	Use buffered writes to checkpoint data.
	Nek5000 [20]	✓	(✓)	×	Use buffered writes to checkpoint data.
	LLM Frameworks [13]	✓	(✓)	×	Use buffered writes to save model checkpoints.

that must complete synchronously before the application can safely reuse its buffer. It reveals that only data copying itself and the two prerequisite steps it depends on, namely page cache lookup and page allocation on a cache miss, must be performed immediately. The remaining kernel bookkeeping operations, such as XArray updates, filesystem metadata mutations, and dirty page marking, can be safely deferred. Following this, we introduce a *Copy-Centric Stage Decoupling* method, which splits a single write request into a foreground stage and a background stage.

However, respecting the intra-request dependencies can still leave an inter-request visibility issue. This is because deferring XArray updates to the background creates a "window of vulnerability": a write request may return to the application, but before its background stage completes, subsequent reads might fetch stale data, and subsequent writes might trigger redundant page allocations. To overcome this challenge, PipeBWrite introduces a lightweight, inode-level tracker that explicitly records in-flight writes (foreground completed but background pending). This ensures perfect inter-request visibility and strict POSIX semantics with negligible overhead.

Furthermore, introducing background threads inevitably incurs extra costs from cross-thread communication, lock contention, context switching, and CPU cache coherence. If left uncontrolled, this overhead could easily offset the latency benefits of the pipeline. To address this, PipeBWrite applies three optimizations. (1) *Joint Adaptive Page Allocation* uses background threads to proactively replenish a page pool, relieving the foreground from allocation burdens while adapting to dynamic workloads. (2) *Inline Page Zeroing* defers page zeroing to the foreground copy stage, eliminating costly cross-core cache transfers and enabling partial or skipped zeroing. (3) *Request Batching and Merging* utilizes per-CPU local batches to minimize shared workqueue contention and enables early merging of small writes.

Finally, to guarantee system resilience under extreme concurrency where background CPU resources are exhausted, PipeBWrite features a dynamic fallback mechanism. It automatically redirects a subset of traffic back to the conventional write path, ensuring that throughput does not collapse even in the worst-case scenarios.

In summary, we make the following contributions:

- **In-depth analysis of buffered writes:** We reveal that non-copy kernel bookkeeping operations severely bottleneck the critical path. We also demonstrate that decoupling and offloading these non-urgent steps can still maintain correctness, as long as it respects the data dependency among the steps.
- **Novel architecture design:** PipeBWrite is a transparent, two-stage pipelined buffered write scheme at the VFS layer. By combining *Copy-Centric Stage Decoupling* with visiblity tracking and holistic optimizations, PipeBWrite can shorten user-perceived latency and improve throughput while preserving POSIX synchronous interfaces. It has been integrated into ext4 and XFS and open-sourced at [12].
- **Significant performance gains:** PipeBWrite's evaluation, across four real-world applications and FIO synthetic workloads, shows that it significantly improves overall application performance. Compared to vanilla Linux buffered writes, PipeBWrite improves throughput by up to 25% in RocksDB, 21% in Kafka, and 17% in Log4j, while reducing CLI utility execution time by up to 47%. On FIO workloads, it achieves up to 2.34× and 1.94× write speedups on ext4 and XFS, cuts average latency by up to 58% and 49%, and reduces P99 tail latency by up to 60% and 55%, respectively. Against StreamCache [40], it delivers up to 134% higher throughput and 56% lower latency.

2 Background and Motivation

2.1 Linux Buffered I/O

Linux buffered I/O is a mechanism in which the operating system transparently interposes a page cache between user space applications and block storage, caching file data in main memory to accelerate access [11]. All file operations performed through the standard `read()` and `write()` system calls interact with the page cache. On a `read()`, the kernel searches the page cache for the requested blocks and either copies the data directly from the cache to the application buffer on a hit or blocks for disk I/O to populate the cache on a miss. On a `write()`, the kernel copies data from the application buffer into the page cache, marks those pages dirty, and returns immediately while asynchronous threads subsequently flush the data to disk. This design amortizes expensive disk I/O by exploiting temporal and spatial locality, coalescing small writes into larger transfers, and enabling concurrency between computation and I/O, without requiring any special management from applications using the standard POSIX I/O interfaces.

2.2 Write Performance Impacts

Buffered I/O is the dominant write mechanism in modern software systems. As shown in Table 1, all 15 applications in our survey, spanning databases, analytics engines, system infrastructure, high performance computing, and AI/ML frameworks, use buffered I/O in their default configurations. In contrast, only about half of them provide direct I/O as an optional mode. This contrast suggests that buffered I/O is the common case in practice, and its performance therefore has broad impact across diverse real world workloads.

Many applications are highly performance sensitive, either requiring low latency or demanding high scalability under heavy concurrency [25, 30, 43, 52]. However, achieving these goals is often challenging in practice due to the cost of buffered writes. During normal execution, these applications commonly generate substantial write workloads, including write ahead logging, journaling, checkpointing, metadata updates, and other persistence related operations [26, 49]. Because such writes often lie on the critical path, buffered write performance directly impacts application responsiveness and can hinder scalability.

To quantify this impact, we measure three representative applications from our survey: RocksDB with the YCSB-A workload, `cp` when copying large files, and Kafka's producer test. Buffered writes account for 34% of the total execution time in RocksDB, 62% in `cp`, and 28% in Kafka. These results show that optimizing buffered writes can directly reduce user-perceived latency and improve application scalability under write-intensive workloads.

2.3 Long Buffered Write Execution Path

The substantial overhead of buffered writes indicates significant potential benefits from optimizing this path. We begin our optimization design by examining the subsystems involved for completing buffered writes and the end-to-end write workflow. Figure 1 illustrates the key system components and their interactions in the Linux ext4 filesystem [42] for performing buffered writes. While the subsequent discussion centers on Linux and ext4, our findings generally apply to other operating systems and filesystems.

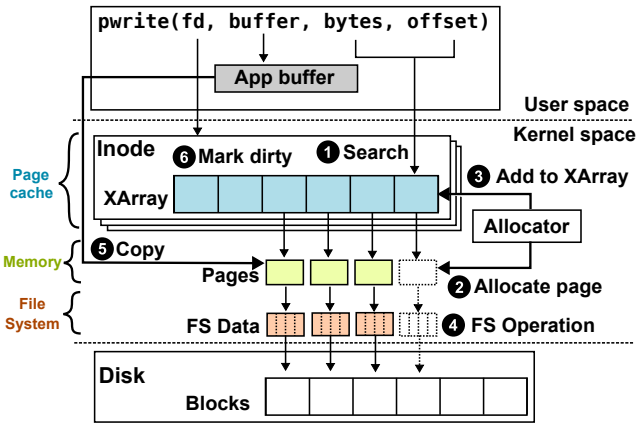


Figure 1: The Linux buffered write workflow and involved subsystems.

Involved subsystems: The buffered write path orchestrates three tightly coupled kernel subsystems. First, the memory subsystem manages DRAM memory resources in fixed-size page (e.g., 4 KB) granularity, allowing dynamic memory allocation and reclamation. Second, the page cache subsystem acts as a fast in-memory index, using an XArray data structure [18] to map a file's logical offsets to physical memory pages. Third, the filesystem (or in short, FS) subsystem is responsible for translating the in-memory page layout to the on-disk block layout, which involves operations on multiple data structures like inodes, superblocks, buffer heads [17] and iomap [7].

Workflow: *The buffered write path extends well beyond a simple data copy.* In practice, it consists of the following six steps in sequence. When a user space application invokes a `write()` or `pwrite()` system call to initiate a buffered write, the kernel first identifies the correct file inode based on the provided file descriptor. It then ❶ searches the file's page cache (XArray) using the supplied page index to determine if the target location already exists in memory . If the page is found in the cache, the kernel retrieves it and acquires the necessary lock. If no page exists, ❷ the kernel allocates a new page, initializes its contents to zeros, and ❸ inserts it into the XArray. Meanwhile, the filesystem needs to ❹ deal with buffer heads and FS-specific structures like the inode and superblock. After the preparation of the page and relevant data structures, ❺ the kernel copies data from the application's buffer to the appropriate location within the page. Following a successful copy, ❻ the kernel marks both the page structure

Table 2: Percentage of time spent in each step of a buffered write. "w/ Allocation" shows the time breakdown when page allocation is required, while "w/o Allocation" shows the breakdown when it is not.

Step (%)	❶	❷	❸	❹	❺	❻
w/ Allocation	2.3	28.4	8.4	10.4	40.1	10.4
w/o Allocation	11.0	0	0	10.7	48.8	29.5

and the related buffer heads as dirty. The steps described above constitute the entire execution path of a buffered write before the operation returns. Subsequent activities, such as dirty page writeback, are performed asynchronously by kernel threads in the background.

Timecost breakdown: In Table 2, we break down the buffered write latency using the FIO benchmark with 1 thread, 64 KB block size, and 32 GB file size. The results reveal the following key insights. First, data copying (Step ❺) is indeed a major contributor to buffered write latency, accounting for 40.1% and 48.8% of the total time for writes with and without page allocation, respectively. However, it still contributes less than half of the overall latency. Second, the majority of the time is spent in the steps beyond copying, which together exceed the cost of the copy itself. These additional steps are primarily associated with metadata updates and kernel bookkeeping that enable applications to safely continue using the copied data and allow the kernel to subsequently manage writeback and flush dirty data to storage.

2.4 Curtailing the Critical Path

Prior work has explored two main directions for improving buffered write performance. The first preserves the synchronous `write()`/`pwrite()` interface and accelerates specific steps in the write path. For example, ScaleCache [46] reduces contention in page cache indexing, while StreamCache [40] lowers page allocation overhead. However, these optimizations are inherently limited, e.g., they often apply only to particular cases, and each optimized step accounts for only part of the end-to-end latency.

The second direction improves performance through asynchronous execution. Interfaces such as `io_uring` [4] and recent systems such as Copier [31] let applications submit requests and continue execution without waiting for completion on the calling thread. This can substantially shorten the application visible critical path, but requires applications to be restructured around asynchronous control flow and completion handling. Such changes are especially difficult for legacy software. As a result, despite their benefits, asynchronous interfaces remain optional in practice. For instance, only 4 of the 15 applications in Table 1 support them, and even then only as optional modes.

We aim to combine the advantages of both directions. Like asynchronous designs, we remove selected buffered write steps from the application critical path and offload them to background kernel execution. Unlike prior asynchronous interfaces, however, we preserve the conventional synchronous `write()`/`pwrite()` interface by keeping only the minimal necessary operations in the foreground. This design benefits universally all applications making use of conventional buffered write interfaces, including existing legacy applications. However, our two-stage buffered write design introduces challenges in both correctness and performance that need to be carefully addressed.

Correctness challenge: Minimizing the foreground stage helps reduce latency, but moving critical steps such as data copying into the background may violate correctness guarantees. Therefore, the partitioning of the foreground and background stages must be carefully decided. In addition, certain steps related to visibility could be deferred to the background stage. As a result, a write may temporarily remain invisible to subsequent operations after the foreground stage returns, potentially leading to incorrect behavior. Our design must therefore incorporate additional mechanisms to ensure proper inter-request visibility.

Performance challenge: While a two-stage pipeline intuitively reduces request latency, the communication and coordination between the foreground and background stages may introduce additional overhead, potentially affecting overall throughput. Our design must minimize these newly introduced costs and fully leverage the optimization opportunities enabled by asynchrony, so as to improve latency without significantly compromising resource utilization or system throughput.

3 Overview of PipeBWrite

We propose PipeBWrite, a new buffered write mechanism that reorders the conventional buffered write workflow and splits its execution into a foreground stage and a background stage. The foreground stage performs only the minimal steps needed to determine the write outcome before returning to the application, while kernel threads complete the remaining steps in the background under dependency preserving execution order, without requiring any completion side interaction with the application thread.

3.1 Copy-Centric Stage Decoupling

The core idea of PipeBWrite is to minimize the buffered write work performed on the foreground application thread, thereby reducing application visible I/O overhead. To achieve this goal, we first identify which steps along the buffered write path must be executed synchronously before the return point (hereafter referred to as "urgent" steps), while the others can be deferred to the background.

Our design adopts copy-centric stage decoupling, where the foreground and background stages are divided around the copy step and its dependencies. The copy step must remain in the foreground because, once a buffered write returns, the application may immediately reuse or free the user buffer.

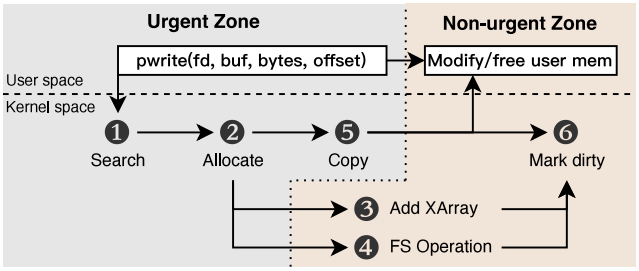


Figure 2: Data dependency graph of buffered write internal steps defined in Figure 1. For correctness, operations in the urgent zone (gray) must finish before the write call returns to user space, while those in the non-urgent zone (light beige) can safely overlap with user-space memory operations.

Deferring the copy would therefore risk reading modified data or dereferencing invalid memory. To preserve the semantics of synchronous write interfaces, data copying must complete before returning to the application.

We then examine the dependencies between the copy step and the other steps in the six-step buffered write workflow. As shown in Figure 2, Step ❺ requires a prepared destination page, which in turn depends on locating or allocating that page. These prerequisite steps must therefore also remain in the foreground. The remaining steps are not required for the safe completion of copying and can be deferred to the background without violating correctness.

Based on this analysis, PipeBWrite executes Steps ❶, ❷, and ❺ in the foreground, while deferring Steps ❸, ❹, and ❻ to the background. This partition minimizes application-thread work while preserving the correctness of the original buffered write workflow.

3.2 Inter-Request Visibility Coordination

The above stage decoupling described in Section 3.1 preserves the semantics of PipeBWrite for each individual write request, as execution across the foreground and background stages still respects the dependencies of the original buffered write workflow. However, inter-request problems can arise because all XArray-related work is deferred to the background stage. Since a buffered write becomes searchable, and thus visible to subsequent operations, only after Add XArray (Step ❸) completes, there exists a window in which a write has returned from its foreground stage but is not yet discoverable through the page cache.

This delayed visibility can break correctness in two cases. A read issued during this window may miss the latest data because it cannot find the new page in the XArray. Similarly, a later write to the same offset may fail to observe that the page has already been allocated, potentially leading to redundant allocation. Thus, while stage decoupling is correct for a single request, preserving inter-request semantics requires additional coordination.

PipeBWrite addresses this problem with a lightweight per-

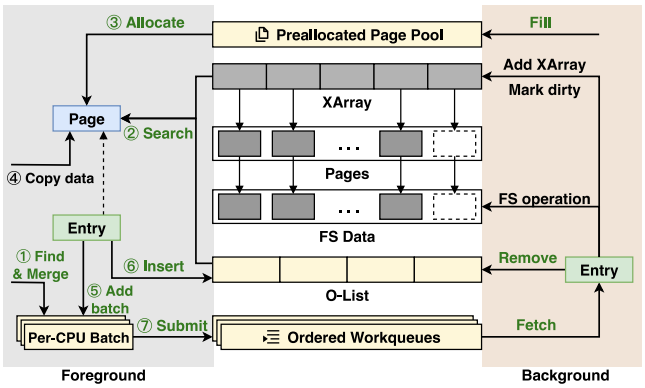


Figure 3: Architecture of PipeBWrite. Original buffered-write data structures are shown in gray, and newly introduced ones in yellow. Of the six buffered-write steps, only Search and Allocate are modified (green); newly added operations are also shown in green. To avoid confusion with Figure 1, this figure uses hollow circled numbers for step labels.

inode *Ongoing List* (O-List) that tracks writes whose foreground stages have completed but background stages are still running. Subsequent operations consult the O-List before the XArray, ensuring immediate visibility of ongoing writes without moving Add XArray to the foreground.

Concretely, the O-List is a per-inode hash table keyed by page index, where each entry records the page pointer along with the write offset and length. Each bucket is synchronized independently with RCU, so lookups proceed lock-free while foreground insertions and background removals are serialized only at the granularity of a single bucket. As a result, both lookups and updates incur minimal overhead.

3.3 PipeBWrite Architecture

With the aforementioned partition, we present the overall architecture of PipeBWrite in Figure 3. PipeBWrite resides beneath the `write()` and `pwrite()` interfaces and reuses the existing buffered write mechanism as its foundation.

PipeBWrite preserves the core kernel components of buffered writes (shown in gray in the figure), including the XArray, page cache pages, and filesystem (FS) data structures. In addition, it introduces several new components (shown in yellow): Ongoing List (O-List), Ordered Workqueues, Preallocated Page Pool and Per-CPU Batches. The O-List tracks all in-flight write operations within an inode, ensuring that subsequent read and write requests observe correct semantics (Section 3.2). The Ordered Workqueues operate as FIFO (First-In, First-Out) communication channels that dispatch deferred foreground work to background threads for asynchronous execution. The Preallocated Page Pool optimizes page allocation to further reduce foreground-stage latency (Section 4.1). The Per-CPU Batch accumulates write requests on the current CPU before they enter the background stage, thereby reducing frequent cross-CPU contention on the workqueue and enabling intra-page request merging (Section 4.3).

Table 3: Percentage of foreground and background time in the two-stage pipeline. "w/ Allocation" are writes requiring allocation while "w/o Allocation" are writes that do not.

Stage (%)	Foreground	Background
w/ Allocation	69.9	30.1
w/o Allocation	46.9	53.1

Execution logic. PipeBWrite introduces new operations on the Batch (① Find&Merge, ⑤ Add batch), O-List (⑥ Insert, Remove), Ordered Workqueues (⑦ Submit, Fetch), and Preallocated Page Pool (Fill, ③ Allocate). These and the modified original steps are highlighted in green in Figure 3.

Foreground. When a write request arrives, the issuing thread first tries to merge it into an existing entry for the same page in its Per-CPU Batch (①). If no such entry exists, the request enters the foreground workflow: it searches the O-List and the XArray for the target page (②); if absent, it allocates a new page from the Preallocated Page Pool or, on pool miss, from the system allocator (③); and it copies the user data into the page (④). After the copy, the page and its metadata are packaged into a new Batch entry (⑤) and registered in the O-List (⑥). When the Batch fills, all of its entries are submitted in bulk to the corresponding workqueue (⑦) and the foreground returns to the application.

Background. A background thread fetches tasks from its workqueue and executes the remaining conventional steps (Steps ❸, ❹, and ❻). On completion, the corresponding O-List entry is removed.

Compatibility with fsync: Applications commonly invoke `fsync()` after buffered writes because `write()` and `pwrite()` do not guarantee persistence. In the conventional buffered write path, preceding writes have already copied data into the page cache and marked the affected pages dirty; `fsync()` therefore flushes those dirty pages to storage before returning, thereby providing persistence.

PipeBWrite preserves the same semantics, but doing so requires additional coordination. Because PipeBWrite defers dirty-page marking to the background stage, some writes issued before an `fsync()` call may still be in flight and not yet visible to the conventional writeback path. Allowing `fsync()` to proceed immediately would risk missing such writes and violating persistence guarantees.

To avoid this problem, PipeBWrite treats `fsync()` as a synchronization barrier. Upon an `fsync()` call, PipeBWrite first drains all pending buffered write work from the workqueues and the O-List, ensuring that all writes issued before the call complete their deferred processing and become visible to dirty-page tracking. It then invokes the conventional writeback path to flush the resulting dirty pages to storage. This preserves semantic equivalence with conventional buffered writes while remaining compatible with PipeBWrite's split execution model.

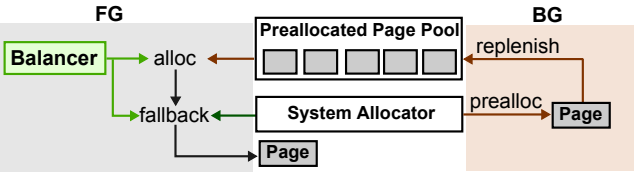


Figure 4: Joint Adaptive Page Allocation in PipeBWrite.

4 Performance Optimizations

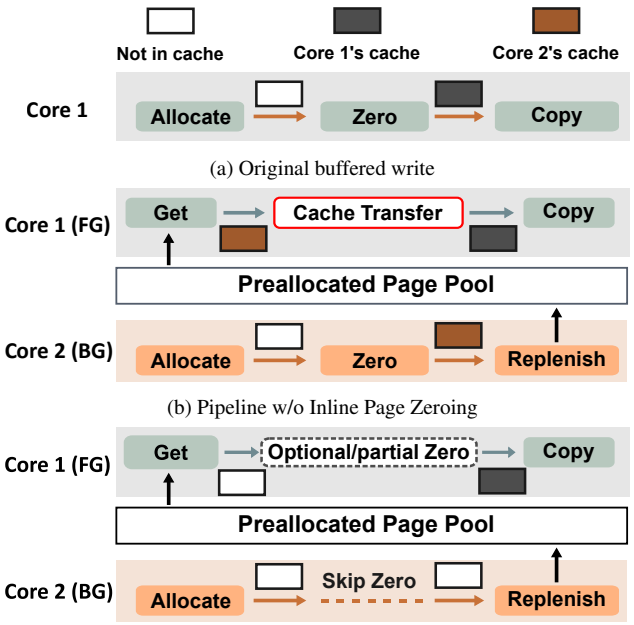
Introducing background threads inevitably incurs extra costs from cross-thread communication, lock contention, context switching, and CPU cache coherence. Therefore, here, we aim to fully exploit the performance opportunities enabled by the two-stage pipeline design while minimizing the communication and coordination overhead it introduces.

4.1 Joint Adaptive Page Allocation

I/O patterns that require allocating new pages, such as file append writes, are widely used in practice [21,40,54]. However, as shown in Table 2, page allocation constitutes the largest single source of overhead in buffered writes aside from data copying. Based on the stage partitioning results in Section 3.1, we further evaluate the impact of page allocation overhead on our new two-stage execution using a single-threaded FIO workload with a 64 KB block size. Table 3 shows that allocating new pages causes the foreground stage to account for nearly 70% of the total write pipeline, significantly increasing the latency on the critical path compared to cases where no new page allocation is required. This additional overhead substantially weakens the benefits of PipeBWrite's two-stage pipeline design.

Prior work uses large preallocated page pools to let the foreground obtain pages cheaply [40,47]. However, this approach incurs significant memory overhead from over-provisioning, and replenishment of the pool must still occur on the critical path. PipeBWrite also maintains a page pool to reduce per-request allocation overhead. Unlike prior designs, however, PipeBWrite periodically replenishes the pool using background threads, thereby offloading allocation work from the foreground stage, meanwhile achieving finer-grained balancing between foreground and background execution.

Specifically, PipeBWrite adopts a joint adaptive page allocation mechanism, coordinated by a *Balancer*. Background threads replenish the pool opportunistically when resources are available, reducing foreground allocation overhead while preserving a balanced pipeline, as shown in Figure 4. We initialize the pool with a fixed number of pages (a configurable value that is set to 64 in our experiments). When the foreground stage requires a page, the Balancer checks the pool. If pages are available, the foreground stage immediately obtains one. This action asynchronously signals the corresponding background stage to replenish the pool by fetching a new page from the system allocator. This design is inherently adaptive. Pool exhaustion indicates the background stage is overloaded.



(a) Original buffered write

(b) Pipeline w/o Inline Page Zeroing

(c) Pipeline with Inline Page Zeroing

Figure 5: Illustration of pipelined buffered write with and without Inline Page Zeroing.

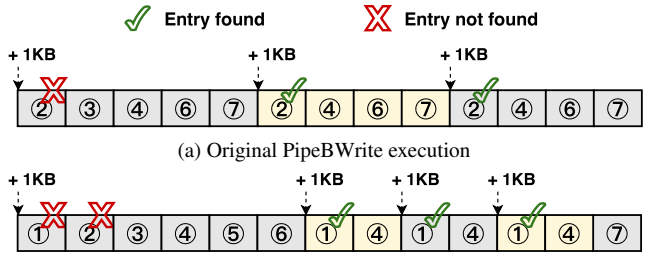
In response, the foreground stage temporarily reverts to the original system allocator. This fallback mechanism naturally throttles the foreground, granting the background stage time to recover and refill the pool, resulting in a self-stabilizing pipeline.

4.2 Inline Page Zeroing

To prevent potential data leakage across processes, the kernel traditionally zeroes a page at allocation time. Under PipeBWrite, however, this policy introduces extra CPU cache transfer overhead because allocation is usually performed by a background thread, whereas the page is later consumed by a foreground thread.

In the conventional buffered write path (Figure 5a), the same thread allocates, zeroes, and then copies into the page. Zeroing brings the page into that core's cache, so the later copy does not incur an additional cache transfer. Unlike this, in PipeBWrite (Figure 5b), a background thread allocates and zeroes the page before placing it into the preallocated page pool. When a foreground thread later consumes that page, the cache lines must be transferred from the background core to the foreground core, adding coherence traffic and overhead.

PipeBWrite eliminates this extra cache movement by deferring zeroing until the page is actually used by the foreground copy path (Figure 5c). As a result, allocation in the background does not load the page into the background core's cache, and the page is brought only once into the foreground core's cache when needed for copying. PipeBWrite further prefetches the page to reduce cache miss latency. Deferring zeroing also enables partial zeroing, where PipeBWrite zeroes



(a) Original PipeBWrite execution

(b) Execution with batching and merging

Figure 6: Foreground execution logic of 1 KB append write, with and without batching and merging. Circled numbers are the steps from Figure 3. Green checkmark means that the entry is successfully found (e.g., page exists in XArray/O-List) in that step. Red cross means the expected entry is not found.

only the bytes not overwritten by the write, and skips zeroing entirely when the write fully covers the page.

4.3 Request Batching and Merging

PipeBWrite's two-stage design introduces communication overhead between the foreground and background stages. Foreground threads must submit deferred work to the workqueue, and background threads must fetch it for execution. While necessary, this coordination adds overhead to asynchronous processing.

We reduce this cost with request batching. Rather than submitting each write to the workqueue immediately, the foreground thread first accumulates requests in a per-CPU local batch and then submits them in bulk. Batching further enables early merging of writes to the same page. As illustrated in Figure 6, later appends can often be merged locally into an earlier batched request before reaching shared cross-CPU data structures, avoiding repeated O-List updates and redundant background task creation. This combined batching and merging design is especially effective for latency-sensitive small append writes.

4.4 Write Path Fallback

Although PipeBWrite effectively reduces write latency via asynchronous offloading, it relies on the availability of spare background CPU cycles to process requests. Under highly concurrent workloads where application threads saturate the CPU, background threads can become under-provisioned, leading to pipeline stalls and throughput degradation. Since naively spawning additional background threads to match foreground concurrency only exacerbates context-switching overhead, we introduce a dynamic Write Path Fallback mechanism. When PipeBWrite detects that background threads are overloaded, it progressively redirects a subset of foreground threads to bypass the pipeline and fall back to the conventional buffered write path. This adaptive routing restores pipeline balance: the remaining pipelined threads continue to enjoy latency reductions, while the fallback threads prevent

system-wide throughput regressions by averting excessive thread contention.

5 Implementation Details

We implement the PipeBWrite prototype within the Linux kernel 6.9 I/O stack. Consisting of approximately 13K lines of code, it is open-sourced at [12].

5.1 Transparency & FS Integration

PipeBWrite is transparent to applications and filesystem-agnostic. Its core logic is implemented in the Virtual File System (VFS) layer, mainly modifying generic functions such as `generic_perform_write()` and `filemap_get_folio()`. To support PipeBWrite, a filesystem needs only two adaptations: it must invoke `pipebwrite_foreground_write()` to perform foreground operations, and it must provide a write function with page allocation and data copy removed for background invocation. We have adapted PipeBWrite to ext4 and XFS, allowing existing applications using these filesystems to benefit without modification.

5.2 Background Thread Management

By default, PipeBWrite places one dormant background thread on each CPU core, and to mitigate scheduler-induced jitter, PipeBWrite pins background threads to each CPU. These threads are awakened on demand, starting from the least loaded cores. This strategy maximizes throughput when CPU resources are plentiful, while relying on the Write Path Fallback mechanism to avoid contention with application threads when CPU resources become scarce. Users can also configure background thread counts via a `sysfs` tunable and enforce strict core isolation for latency-sensitive deployments.

5.3 Correctness

Our kernel implementation carefully handles correctness issues introduced by deferred write processing. PipeBWrite modifies the generic filemap API layer to consult the O-List in all page-cache lookup paths, including regular file and mmap accesses, preserving visibility semantics across file system interfaces. It also coordinates page/inode lifecycle operations with pending writes by delaying LRU insertion until background processing completes, draining the O-List before truncate, and adding inode references for non-empty O-List. Background-stage errors are reported through subsequent `write()`, `fsync()`, or `close()` calls, following the delayed-error model of ext4 and XFS. We further validated our prototype with real-world workloads and CrashMonkey [41]; PipeBWrite passed all 328 test cases.

5.4 Limitations

Heavy `fsync` workloads: `fsync` calls force PipeBWrite to drain internal workqueues and the O-List before initiating disk writeback, temporarily serializing the pipeline, so very frequent fsync calls diminish its benefits. PipeBWrite mitigates this by transparently falling back to the conventional write path when it detects such a pattern. Specifically, it estimates synchronization pressure from the ratio of writes drained at each `fsync` to those completed since the previous one, and falls back to the original path if the ratio exceeds a certain threshold. In practice, this fallback rarely triggers since most applications balance persistence against performance rather than syncing on every write [53]. For example, RocksDB under YCSB-A syncs only about once per 1,000 writes, making PipeBWrite broadly applicable.

High CPU utilization: CPU-saturating workloads trigger the fallback of Section 4.4, neutralizing PipeBWrite's advantages, although performance does not degrade below the baseline. In practice, average CPU utilization usually remains low across cloud [28,36], desktop [22,37], and mobile environments [27], so PipeBWrite delivers tangible benefits in predominant real-world scenarios.

6 Evaluation

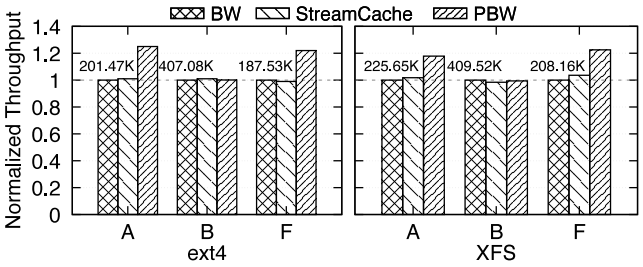
6.1 Experimental Setup

Platform: We conduct all experiments on a server equipped with four 20-core Intel Xeon Gold 6248 CPUs (80 cores total), 384 GB of memory, and a 1.6 TB Intel DC P4610 NVMe SSD (3,200/2,100 MB/s peak read/write throughput) [16]. The system runs Ubuntu 24.04 with Linux kernel 6.9.0.

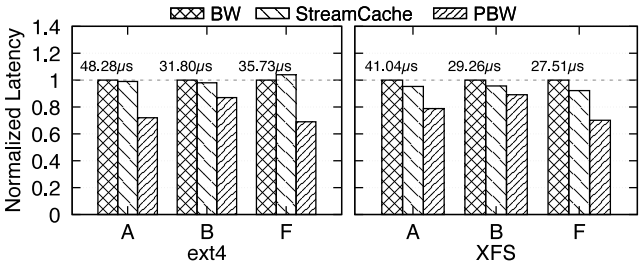
Applications & workloads: We evaluate four representative software systems from Table 1 running unmodified on ext4 and XFS: RocksDB [15] (with read/write-heavy YCSB workloads), Apache Kafka [1] (with built-in producer benchmark), standard command-line tools [3] (file copy and archiving), and Log4j [2] (with representative logging benchmarks). For each workload, we run the evaluation three times and report the average results.

Baselines: We compare PipeBWrite (PBW) against the conventional buffered write path (BW) on ext4 and XFS. We exclude non-transparent asynchronous interfaces (e.g., `io_uring`) as they require substantial application refactoring. Among transparent optimizations, we exclude ScaleCache [46] (no measurable gains in our setup) and WSBuffer [56] (closed-source with substantial page-cache modifications that are hard to reproduce). We evaluate against StreamCache [40] via a faithful reimplementation, which reproduces their reported trends.

Configurations: To ensure fairness, application thread placement is identical across all systems. For PipeBWrite, we provision one on-demand standby background thread per CPU core, the default configuration according to Section 5.2; we report overall CPU utilization in selected experiments to explicitly quantify this background overhead. Unless specified, CPUs operate in performance mode with SMT disabled to avoid variance.



(a) RocksDB throughput.



(b) RocksDB update latency.

Figure 7: RocksDB-YCSB performance. The absolute throughput (ops/s) and latency (μs) numbers of BW are annotated.

6.2 Real-World Applications

6.2.1 RocksDB

We run the full YCSB suite and report results for the representative write-intensive workloads A and F, and the read-intensive workload B. The remaining workloads C, D, and E involve very few writes and show behavior similar to Workload B, so we omit them for brevity. All experiments use 1 KB records and 10 million records. We set the number of YCSB threads to 8, since increasing the thread count beyond this point does not improve throughput even for the BW baseline; all other parameters remain at their default settings.

Figure 7a and Figure 7b show overall throughput and update latency, respectively, both normalized to BW. For the write-intensive workloads, PipeBWrite improves throughput by 18–25% and reduces update latency by 21–31% across ext4 and XFS. For the read-intensive Workload B, PipeBWrite still reduces update latency by 11–13%, but because updates account for only 5% of operations, the overall throughput improvement is modest.

By comparison, StreamCache provides less than 5% throughput improvement across all workloads. This is expected because YCSB's default 1 KB record size rarely triggers page allocation during writes. Since StreamCache mainly optimizes page allocation on the write critical path, its benefit in this setting is limited.

We also measure the additional CPU cost of PipeBWrite in the RocksDB experiments (not shown in the figure). PipeBWrite incurs negligible extra CPU usage for the read-dominant Workload B, only 1.5% on ext4 and 1.9% on XFS. In contrast, for workloads with substantial writes, PipeBWrite uses more spare CPU cycles to improve performance. For instance, for Workload A, CPU utilization increases by 22.7–23.5%, and for Workload F, by 21.6–22.0%. Although PipeBWrite consumes more CPU cycles, RocksDB is already saturated with 8 application threads on the baseline. Thus, PipeBWrite improves throughput without increasing the number of application threads, leading to better application scalability.

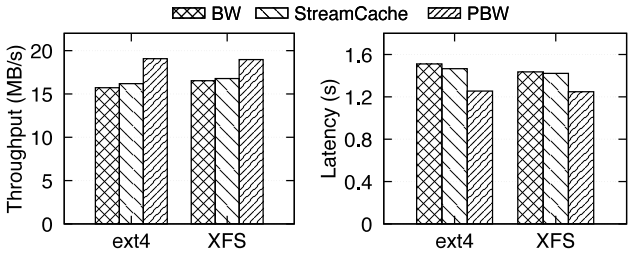


Figure 8: Kafka producer throughput and latency.

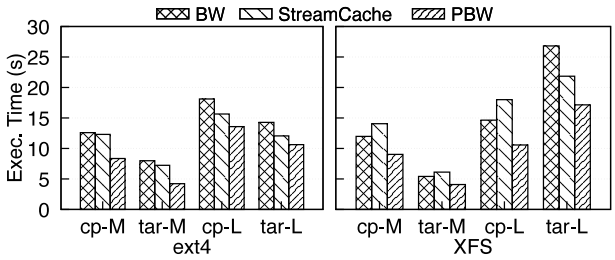


Figure 9: The execution time of command line tools. The "-M" suffix denotes a mixed dataset (Linux source code); "-L" denotes a workload on larger datasets.

6.2.2 Kafka

We use Kafka's built-in producer performance benchmark to evaluate PipeBWrite against the baseline, with 60 producer threads, 20 server threads, and 4 KB record size. The results are shown in Figure 8. Kafka's write operations lie on its critical path, hence the write optimizations of PipeBWrite translate directly into Kafka's performance gains. On ext4 and XFS, PipeBWrite improves Kafka producer throughput by 21% and 15%, respectively, while reducing message latency by 17% and 13%. In contrast, StreamCache does not systematically shorten the critical path as PipeBWrite does, and therefore delivers under 5% improvements in both throughput and message latency.

6.2.3 Command Line Applications

To evaluate PipeBWrite on common file operations, we benchmark the standard `cp` (copy) and `tar` (archive) utilities.

We first test a mixed I/O size workload using the Linux v6.9

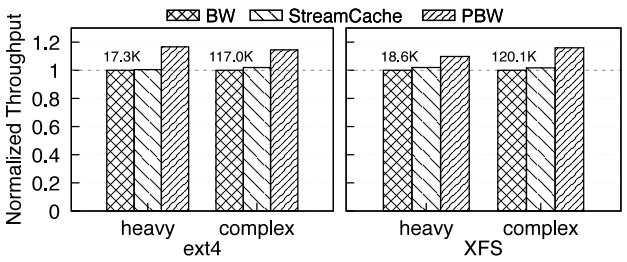


Figure 10: Throughput of Log4j under the heavy and complex logging workloads. The absolute throughput (ops/s) numbers of BW are annotated.

source code, under two page cache conditions. As shown in Figure 9 (with "-M" suffix), with a warm page cache (dataset already in memory), PipeBWrite reduces the execution time of `cp` and `tar` by up to 33.7% and 47.1% respectively. StreamCache also improves performance on ext4, but its gains are smaller than those of PipeBWrite. On XFS, StreamCache sometimes even increases the latency due to its incompatibility with large folio [24], a page structure that can be much larger than the conventional 4 KB page. With a cold cache, disk read latency dominates execution time (not shown). As a result, even though PipeBWrite halves write time, the overall completion time improves only marginally.

Next, we evaluate the utilities using a larger 3 GB dataset (10 files of 300 MB each) with a cold cache (Figure 9 with "-L" suffix). PipeBWrite reduces overall execution time for `cp` by 33% on ext4 and 38% on XFS, and for `tar` by 34% and 56%, respectively. Under this cold-cache configuration, the kernel's *readahead* mechanism prefetches data from the disk into the page cache, reducing read latency and allowing the benefits of PipeBWrite's write-path optimization to become more visible than in the mixed workload.

Due to the pipelined design, most background work overlaps with foreground execution in PipeBWrite. Consequently, in Figure 9, all the remaining background work finishes within no more than 0.09 ms.

6.2.4 Log4j

Logging frameworks rely on sequential writes to persist application logs, making their performance sensitive to write latency. We evaluate PipeBWrite using Log4j, a widely used Java logging framework, configured in its default synchronous mode. As shown in Figure 10, PipeBWrite improves throughput by 17% on ext4 and 10% on XFS for intensive logging workloads (denoted 'heavy'), and by 15% and 16% for complex workloads involving placeholders, markers, and mapped diagnostic contexts (denoted 'complex'). In contrast, StreamCache shows only marginal gains on these workloads.

6.3 Synthetic Workloads

We additionally use FIO [6] to conduct controlled microbenchmarks that isolate PipeBWrite's behavior under different I/O patterns and stress conditions. We first measure average latency, P99 latency, and throughput for pure sequential writes with block sizes of 64 B, 4 KB, and 64 KB under 1, 40, and 80 threads, representing light, medium, and heavy CPU pressure. We then examine CPU usage, break down the performance, and evaluate PipeBWrite under mixed read/write workloads, different page-pool sizes, and varying `fsync()` frequencies. Unless otherwise specified, writeback to disk is disabled in these experiments to isolate the in-memory buffered write path and avoid storage-device bottlenecks.

6.3.1 Latency

We first report the average and 99th-percentile (P99) latency in ext4 in Figure 11. XFS results show similar trends.

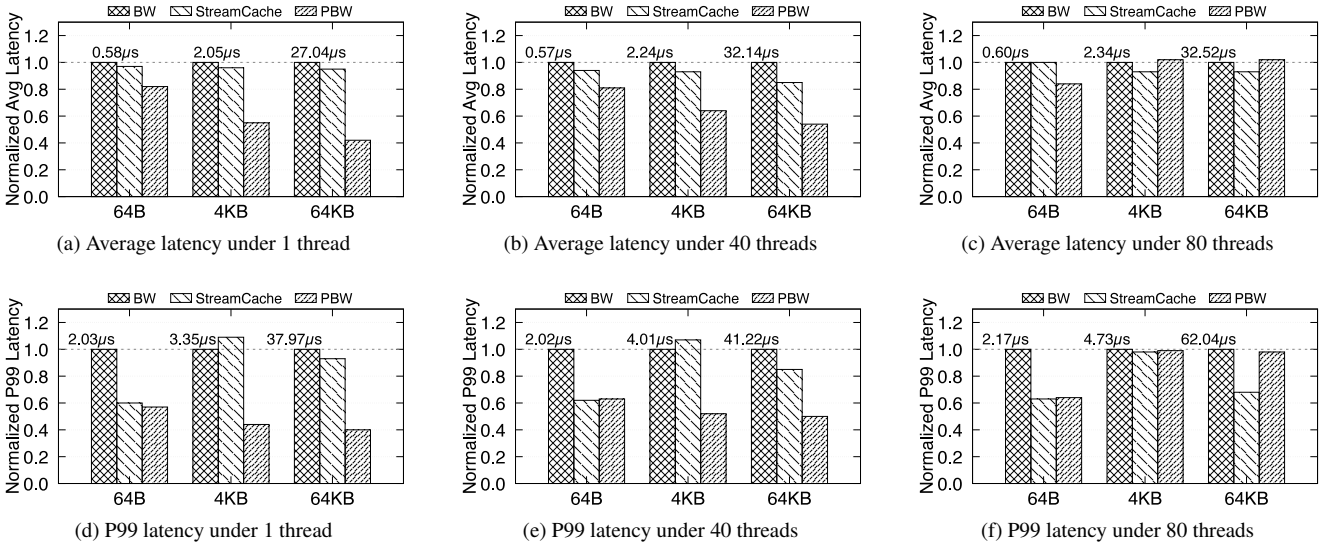
Average latency: Under low and medium concurrency, PipeBWrite reduces average latency by up to 58% on ext4 and 49% on XFS. The gains are smaller for the 64 B I/O size, since page allocation and XArray insertion occur less frequently under small I/O sizes, but PipeBWrite nonetheless achieves reductions of no less than 19% on ext4 and 33% on XFS.

Under high concurrency, CPU resources are heavily occupied by foreground threads. As a result, for 4 KB and 64 KB I/O sizes, PipeBWrite triggers fallback due to insufficient background-thread resources and reverts to the conventional buffered write path, showing identical performance compared to BW. For 64 B I/O, however, request batching and merging greatly reduce background work, allowing the background threads to remain effective with only minimal CPU resources. PipeBWrite therefore continues to provide 16% and 31% latency reduction on ext4 and XFS, respectively.

Unlike PipeBWrite, which systematically restructures the buffered write path, StreamCache optimizes only page allocation, yielding a modest latency reduction (only up to 15%). This narrower scope also comes at a memory cost: on our 80-core system, its per-CPU pools consume 160 GB of preallocated memory regardless of workload demand, whereas PipeBWrite's Joint Adaptive Page Allocation requires only a small number of preallocated pages per CPU (64 in our system), amounting to roughly 20 MB system-wide. Furthermore, StreamCache's mechanism does not support large folios, providing no benefit for 64 KB XFS writes. Compared to StreamCache, PipeBWrite improves the latency by up to 55% on ext4 and 56% on XFS.

Tail latency: PipeBWrite also substantially reduces P99 latency by up to 60%/55% on ext4/XFS under low concurrency and 50%/48% under medium concurrency. Even for small 64 B writes, where page allocation is infrequent, PipeBWrite reduces P99 latency by no less than 37% on ext4 and 35% on XFS under low and medium concurrency. Under high concurrency (80 threads), its preallocated page pool continues to absorb tail latency spikes for small I/O sizes, still achieving a 36% P99 reduction for 64 B writes on ext4 and 33% on XFS.

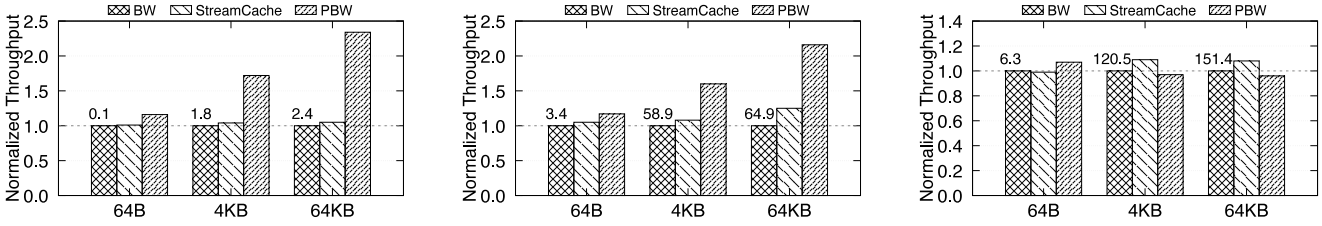
StreamCache, in contrast, exhibits highly variable tail latency. At 4 KB write size, StreamCache even incurs 9% higher tail latency compared to BW. This is because its page pool



(a) Average latency under 1 thread (b) Average latency under 40 threads (c) Average latency under 80 threads

(d) P99 latency under 1 thread (e) P99 latency under 40 threads (f) P99 latency under 80 threads

Figure 11: The average and P99 latency of FIO sequential write. The absolute latency (μs) numbers of BW are annotated.



(a) Throughput under 1 thread (b) Throughput under 40 threads (c) Throughput under 80 threads

Figure 12: The throughput of FIO sequential write. The absolute throughput (GB/s) numbers of BW are annotated.

must be replenished upon exhaustion, and the resulting refill stalls cause occasional latency spikes that elevate tail latency.

6.3.2 Throughput

Figure 12 shows the sequential write throughput in ext4. Results on XFS show similar trends and are omitted. Under low concurrency, PipeBWrite improves throughput by up to 2.34× on ext4 and 1.94× on XFS, with the largest gains at 64 KB. Under medium concurrency, PipeBWrite achieves throughput improvements of up to 2.16× on ext4 and 1.57× on XFS. For small I/O sizes (64 B), the throughput gains are modest: 16% on ext4 and 36% on XFS under low concurrency.

Under high concurrency, for page-sized or larger I/O requests, PipeBWrite activates its fallback mechanism, resulting in no throughput improvement. But for 64 B writes, request merging and batching allow PipeBWrite to still achieve a 7% throughput improvement on ext4 and 12% on XFS.

On the other hand, StreamCache provides only marginal throughput improvements in most configurations, typically 1–8% on ext4. By comparison, PipeBWrite improves throughput by 11% to 134% over StreamCache across all low- and medium-concurrency configurations.

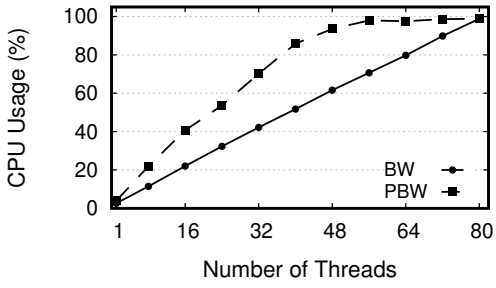


Figure 13: CPU usage under different thread counts.

6.3.3 CPU Usage

Figure 13 reports the CPU usage of 64 KB FIO writes under different thread counts. When spare CPU resources are available, PipeBWrite can leverage additional CPU cycles to improve write performance. For example, at 40 threads, PipeBWrite increases CPU usage by 66.0%, while improving throughput by 2.16× and reducing average latency by 46%. As the thread count further increases, foreground application threads gradually saturate the CPU. In this case, PipeBWrite detects CPU pressure and progressively routes only a subset of writes through the pipelined path, while falling back the

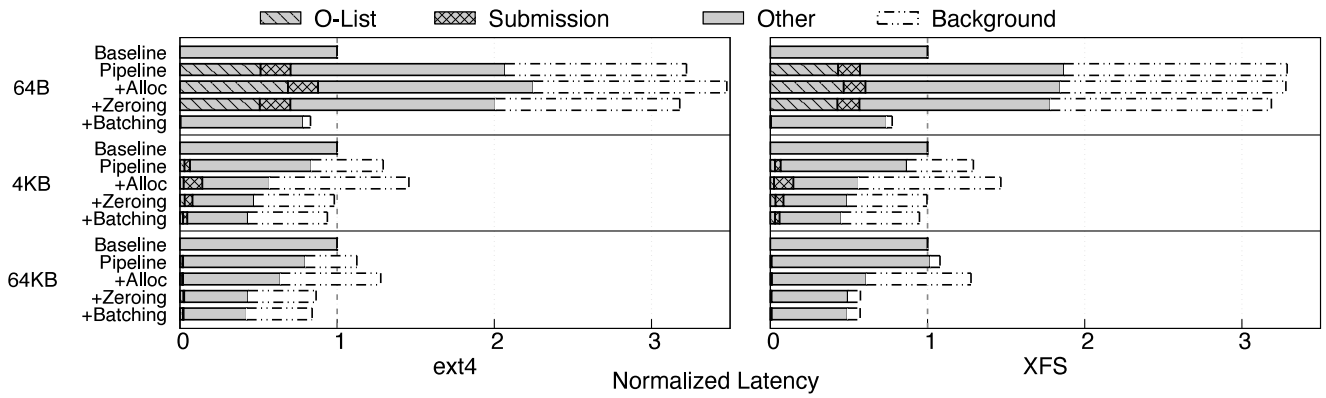


Figure 14: Latency breakdown of PipeBWrite atop ext4 and XFS across three I/O sizes, separated into foreground and background stages, as optimizations are incrementally enabled. All values are normalized to BW.

remaining writes to the conventional buffered write path.

6.3.4 Performance Breakdown

To understand the overhead introduced by PipeBWrite's new operations and the contribution of each optimization, we perform an incremental performance breakdown using single-thread FIO on ext4 and XFS, as shown in Figure 14.

4 KB and 64 KB writes: At these I/O sizes, the naïve pipeline reduces foreground-stage latency by up to 16% (4 KB) and 20% (64 KB) relative to the baseline. However, most of the total normalized latency still resides in the foreground stage. Enabling Joint Adaptive Page Allocation (+Alloc) moves page allocation and zeroing to the background stage, balancing the two stages: the foreground now accounts for only 39–50% of total latency, a further ∼33% reduction relative to the naïve pipeline. However, the background threads must zero pages and transfer cache lines to the foreground CPU, increasing total (foreground + background) overhead by 13–16% compared to the naïve pipeline.

Inline Page Zeroing (+Zeroing) addresses this by deferring zeroing to the copy stage and prefetching the page into the local cache before copy. As a result, foreground latency drops by a further 15–26% relative to previous optimizations, and the total pipeline overhead falls by 32–44%.

Finally, Request Batching and Merging (+Batching) amortizes per-request metadata costs such as O-List maintenance, workqueue submission and cuts remaining overhead by roughly 35% at 4 KB. With all four optimizations active, PipeBWrite achieves its lowest foreground latency and overall overhead across both filesystems.

64 B writes: At 64 B, the naïve pipeline introduces extra overhead. Because the data copy itself is negligible for such small requests, the O-List lookup, workqueue submission, and the page contention between foreground and background account for a disproportionate share of foreground latency.

Moreover, page allocation is less frequently triggered under 64 B writes. Joint Adaptive Page Allocation and Zeroing Optimization therefore contribute minimally to the latency improvement at this I/O size. The gains instead come from Request Batching and Merging, which eliminates repeated O-List lookups and workqueue submissions in the foreground and substantially reduces the background processing burden.

ext4 vs. XFS background overhead: An additional observation from the breakdown is that, for 64 KB writes, PipeBWrite's background-stage overhead is notably lower on XFS than on ext4. The reason is XFS's support for large folios. A coarser page granularity means fewer pages per I/O request, which in turn requires fewer per-page bookkeeping operations (XArray insertion, dirty marking, filesystem metadata updates) in the background stage.

6.3.5 Mixed Read/Write

We further quantify how PipeBWrite performs under mixed read/write workloads and the O-List lookup overhead on the read path. We run 4 KB random R/W FIO on a warm cache and sweep the write ratio from 0% to 100%. The results are shown in Figure 15. PipeBWrite's throughput matches BW for pure reads, and its advantage grows steadily with the write fraction, reaching 1.22× for random writes only. The O-List introduces only a lightweight read-path lookup, which costs a few tens of CPU cycles per access; consequently, PipeBWrite's average and P99 read latencies remain within 2% of BW across all read/write mixes.

6.3.6 Page-Pool Sensitivity

To evaluate the impact of pool sizing, we sweep the per-CPU page pool size from 4 to 256 pages under 4 KB FIO, and the results are shown in Figure 16. As the pool grows, the miss rate, i.e., the fraction of foreground allocations that fall back to the system allocator, drops from 94% with a 4-page pool to 11% with 64 pages and to near zero beyond 128. Throughput rises correspondingly from about 2.6 to 3.0 GB/s, as more allocations are served directly from the pool. However, the throughput gain becomes marginal beyond 64 pages, while each additional page must remain pinned in memory on every core. We therefore adopt 64 pages as the default, which already reaches 97% of the peak throughput with 90% hit rate.

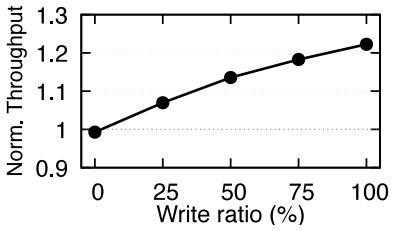


Figure 15: FIO 4 KB random R/W throughput under varying write ratios.

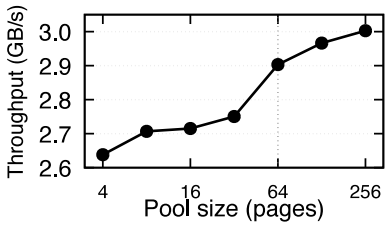


Figure 16: FIO 4 KB write throughput under varying page-pool sizes.

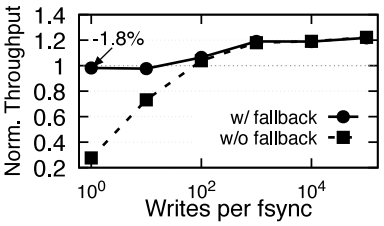


Figure 17: FIO 4 KB write throughput under varying `fsync()` frequencies.

Larger pool eliminates misses and is an equally reasonable choice when memory is plentiful.

6.3.7 Fsync Fallback

We evaluate the effectiveness of the fsync fallback mechanism of Section 5.4 by sweeping the `fsync()` frequency of FIO 4 KB writes from one call per 100,000 writes to one per write, and the throughput, normalized to BW, is shown in Figure 17. Without the fallback, frequent syncs erode PipeBWrite's benefit. Throughput drops by 72% at one `fsync()` per write and by 27% at one per 10 writes. With the fallback enabled, PipeBWrite detects the sync-heavy pattern and keeps throughput within 1.8% of BW even at one `fsync()` per write. At this frequency, 98% of writes are routed through the conventional path; the fallback ratio decreases to 88% at one `fsync()` per 10 writes and 44% at one per 100 writes. At one `fsync()` per 1,000 writes or lower, the fallback no longer engages and PipeBWrite retains its full speedup.

7 Related Work

Improving buffered I/O performance: Some prior studies observe that existing page cache implementations cannot fully exploit the performance of high-throughput storage devices. They therefore propose either selectively bypassing the page cache for I/O [32, 34, 48, 56] or optimizing specific data structures and workflows involved in buffered I/O to reduce their overhead [23, 24, 35, 40, 45, 46, 50, 51]. For cache-bypassing approaches, PipeBWrite remains essential, as it provides effective performance benefits for small writes and other scenarios in which buffering is indispensable. For work that optimizes buffered I/O itself (e.g., reducing page allocation overhead or alleviating lock contention), these approaches are primarily point optimizations targeting individual components. In contrast, PipeBWrite proposes a systematic architectural redesign of the entire buffered write critical path while retaining most of the original structure. Consequently, these single-component optimizations are complementary and can be integrated into PipeBWrite's foreground or background stages for further performance gains.

Asynchrony: Another line of work improves system performance by offloading operations from the calling thread through asynchrony. Some of these approaches require changes to operating system interface semantics [4, 31, 44], which necessitates substantial application refactoring. In contrast, PipeBWrite remains completely transparent and preserves existing synchronous POSIX interface semantics.

Other studies preserve the original interface semantics while deferring selected execution steps within the interface implementation [23, 39]. While sharing a similar spirit of transparent asynchrony, PipeBWrite fundamentally differs from them in its design scope and methodology. For instance, Non-blocking Writes [23] focuses specifically on hiding synchronous page fetch latency during cache misses. PipeBWrite, however, systematically accelerates the entire in-memory buffered write process (handling both cache hits and misses) through stage decoupling. Similarly, the Asynchronous I/O Stack (AIOS) [39] replaces synchronous operations with asynchronous ones at the block layer to overlap CPU work with device I/O for ultra-low latency SSDs. Instead of focusing on the block layer, PipeBWrite targets the VFS and page cache layers.

8 Conclusion

PipeBWrite rethinks buffered writes as a transparent two-stage pipeline. This design yields two key insights. First, decoupling urgent data transfers from background bookkeeping overcomes the synchronous I/O interface bottleneck, while our stage-partitioning method and lightweight inter-request visibility tracker strictly preserve POSIX semantics. Second, the inherent coordination overheads of asynchrony can be fully neutralized via adaptive allocation, deferred zeroing, and request batching and merging. Ultimately, PipeBWrite delivers significant reductions in both average and tail latencies, along with substantial throughput gains, with zero application source code modifications.

9 Acknowledgment

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd, Sanidhya Kashyap for his guidance in our camera-ready preparation. This work is supported in part by the National Natural Science Foundation of China under Grant No. U25B2020, NSF grant (2312785), and the IITP(Institute of Information & Communications Technology Planning & Evaluation)-ITRC(Information Technology Research Center) grant funded by the Korean government (Ministry of Science and ICT) (IITP-2026-RS-2021-II211817). Guoyu Wang and Cheng Li are the corresponding authors.

References

[1] Apache Kafka. https://kafka.apache.org/.

[2] Apache Log4j. https://logging.apache.org/log4j/2.x/index.html. (accessed 2026-03-07).

[3] Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/coreutils.html.

[4] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. (accessed 2026-03-07).

[5] Git. https://git-scm.com/.

[6] GitHub - axboe/fio: Flexible I/O Tester. https://github.com/axboe/fio.

[7] KernelProjects/iomap - Linux Kernel Newbies. https://kernelnewbies.org/KernelProjects/iomap.

[8] MongoDB: The World's Leading Modern Database | MongoDB. https://www.mongodb.com/.

[9] Nginx. https://nginx.org/en/.

[10] NumPy. https://numpy.org/.

[11] Page cache — the linux kernel documentation. https://docs.kernel.org/mm/page_cache.html.

[12] PipeBWrite. https://github.com/PipeBWrite/PipeBWrite. (accessed 2026-03-18).

[13] PyTorch. https://pytorch.org/.

[14] QEMU. https://www.qemu.org/.

[15] RocksDB | A persistent key-value store. http://rocksdb.org/.

[16] Select Intel(R) SSD DC P4610 Series, PCN 116700-00, Product Design, Firmware Update. https://www.intel.com/content/www/us/en/content-details/810965/select-intel-r-ssd-dc-p4610-series-pcn-116700-00-product-design-firmware-update.html.

[17] VFS iomap Documentation — The Linux Kernel documentation. https://docs.kernel.org/filesystems/iomap/index.html.

[18] XArray — The Linux Kernel documentation. https://docs.kernel.org/core-api/xarray.html.

[19] LLNL/PF3DIO. https://github.com/LLNL/PF3DIO, October 2022.

[20] Nek5000/Nek5000. https://github.com/Nek5000/Nek5000, September 2025.

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

[22] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. *ACM SIGARCH Computer Architecture News*, 38(3):302–313, June 2010.

[23] Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. Non-blocking Writes to Files. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 151–165, 2015.

[24] Jonathan Corbet. Clarifying memory management with page folios. *LWN.net*, March 2021.

[25] Jace Courville and Feng Chen. Understanding storage I/O behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, May 2016.

[26] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021.

[27] Cao Gao, Anthony Gutierrez, Madhav Rajan, Ronald G. Dreslinski, Trevor Mudge, and Carole-Jean Wu. A study of mobile device utilization. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234, March 2015.

[28] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 171–180, September 2007.

[29] PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, September 2025.

[30] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 168–183, Shanghai China, October 2017. ACM.

[31] Jingkai He, Yunpeng Dong, Dong Du, Mo Zou, Zhitai Yu, Yuxin Ren, Ning Jia, Yubin Xia, and Haibo Chen. How to copy memory? coordinated asynchronous copy

as a first-class os service. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 1062–1081, New York, NY, USA, 2025. Association for Computing Machinery.

[32] Frank Herold and Sven Breuner. Introduction to BeeGFS. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf.

[33] InfluxData, Inc. Influxdb platform overview. https://www.influxdata.com/products/influxdb-overview/, 2026. (accessed 2026-07-22).

[34] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[35] Jongseok Kim, Chanu Yu, and Euiseong Seo. Revitalizing buffered i/o: Optimizing page reclaim and i/o throttling. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pages 475–482, 2023.

[36] A. Kochut and K. Beaty. On Strategies for Dynamic Resource Management in Virtualized Server Environments. In *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 193–200, October 2007.

[37] Andrzej Kochut, Kirk Beaty, Hidayatullah Shaikh, and Dennis G. Shea. Desktop workload study with implications for desktop cloud resource optimization. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[38] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[39] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 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)*, pages 603–616, 2019.

[40] Zhiyue Li and Guangyan Zhang. StreamCache: Revisiting Page Cache for File Scanning on Fast Storage Devices. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1119–1134, 2024.

[41] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Automatically Test File-System Crash Consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

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

[43] David T. Nguyen. Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, UbiComp '14 Adjunct, pages 337–342, New York, NY, USA, September 2014. Association for Computing Machinery.

[44] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, 2006.

[45] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827, 2020.

[46] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. ScaleCache: A Scalable Page Cache for Multiple Solid-State Drives. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, pages 641–656, New York, NY, USA, April 2024. Association for Computing Machinery.

[47] Aidi Pi, Junxian Zhao, Shaoqi Wang, and Xiaobo Zhou. Memory at your service: fast memory allocation for latency-critical services. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 185–197, New York, NY, USA, 2021. Association for Computing Machinery.

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

[49] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista-Gomez, and Rosa M. Badia. A study of checkpointing in large scale training of deep neural networks. *CoRR*, abs/2012.00825, 2020.

[50] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. FastCommit: resource-efficient, performant and cost-effective file system journaling. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 157–171, Santa Clara, CA, July 2024. USENIX Association.

[51] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 363–378, 2023.

[52] Yi Su, Dan Feng, Yu Hua, and Zhan Shi. Understanding the latency distribution of cloud object storage systems. *Journal of Parallel and Distributed Computing*, 128:71–83, June 2019.

[53] Guoyu Wang, Xilong Che, Haoyang Wei, Shuo Chen, Puyi He, and Juncheng Hu. Boosting File Systems Elegantly: A Transparent NVM Write-ahead Log for Disk File Systems. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 19–34, 2025.

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

[55] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. LLFree: Scalable and Optionally-Persistent Page-Frame allocation. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 897–914, Boston, MA, July 2023. USENIX Association.

[56] Yekang Zhan, Tianze Wang, Zheng Peng, Haichuan Hu, Jiahao Wu, Xiangrui Yang, Qiang Cao, Hong Jiang, and Jie Yao. Rearchitecting Buffered I/O in the Era of High-Bandwidth SSDs. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, pages 579–595, 2026.

[57] Da Zheng, Randal Burns, and Alexander S. Szalay. A parallel page cache: IOPS and caching for multicore systems. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, page 5, USA, 2012. USENIX Association.

A Artifact Appendix

Abstract

The artifact contains PipeBWrite's kernel implementation together with a self-contained benchmark harness. The kernel side is our modified Linux 6.9 tree with the PipeBWrite write path and its ext4 and XFS adaptations. The harness side pins all workload dependencies and automates environment setup, kernel verification, benchmark execution, and figure generation for the experiments reported in this paper.

Scope

The artifact reproduces the paper's main performance claims, namely the real-world application results on RocksDB YCSB, Kafka, command-line utilities, and Log4j in Figures 7, 8, 9, and 10, the FIO microbenchmark latency and throughput results in Figures 11 and 12, and the performance breakdown study in Figure 14.

Contents

The artifact repository is organized into four directories, while the PipeBWrite kernel itself is provided as a separate Linux source tree, as described in Hosting.

- `scripts/` holds the setup and environment-check entry points, the full-suite runner, per-benchmark wrapper scripts, and a single configuration file, `scripts/ae_common.sh`, for machine-specific settings.
- `repos/` vendors the benchmark suites with pinned inputs, namely YCSB-cpp with RocksDB bindings, Kafka, and the FIO, command-line, and logging harnesses.
- `results/` collects one timestamped directory per run, containing a status report, a manifest of the host state, and per-experiment summaries.
- `output/` receives the generated data files and rendered figures.

Hosting

The artifact consists of two repositories cloned as sibling directories: the benchmark harness at `https://github.com/PipeBWrite/ae_artifact` on branch `AE` and the modified kernel at `https://github.com/PipeBWrite/linux`.

Requirements

Hardware requirements. An x86 server with a dedicated NVMe block device, which the benchmarks destructively reformat.

Software requirements. The software stack targets Ubuntu 24.04 and needs the usual kernel build toolchain, OpenJDK 17 with Maven for the Java workloads, fio, the RocksDB 10.2.1 and compression libraries linked by YCSB-cpp, and gnuplot for figure generation; the README lists the exact package set for `apt`.