zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO

Timothy Stamler, Deukyeon Hwang, and Amanda Raybuck, *UT Austin;*
Wei Zhang, *Microsoft;* Simon Peter, *University of Washington*

https://www.usenix.org/conference/osdi22/presentation/stamler

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO

Tim Stamler[1] Deukyeon Hwang[1] Amanda Raybuck[1] Wei Zhang[2] Simon Peter[3]
[1]UT Austin [2]Microsoft [3]University of Washington

Abstract

We present zIO, a transparent zero-copy IO mechanism for unmodified IO-intensive applications. zIO tracks IO data through the application, eliminating copies that are unnecessary while maintaining data consistency.

Applications often modify only a part of the data they process. zIO leverages this insight and interposes on IO stack and standard library memory copy calls to track IO data and eliminate unnecessary copies. Instead, intermediate data locations are unmapped, allowing zIO to intercept and resolve any access via page faults to maintain data consistency. To avoid harming application performance in situations where data tracking overhead is high, zIO's tracking policy decides on a per IO basis when to eliminate copies. Further, we demonstrate how to use zIO to achieve *optimistic network receiver persistence* for applications storing data from the network in non-volatile memory (NVM). By mapping socket receive buffers in NVM and leveraging kernel-bypass IO, we can rely on zIO to transparently eliminate all copies from the network, through the application, to storage.

We implement zIO as a user-space library. On top of kernel IO stacks, zIO eliminates application-level IO copies. We also integrate zIO with kernel-bypass IO stacks, where it can additionally eliminate copies incurred by the IO stack APIs and enable optimistic network receiver persistence. We evaluate zIO with IO-intensive applications, such as Redis, Icecast, and MongoDB. zIO improves application throughput by up to 1.8× with Linux and by up to 2.5× with kernel-bypass IO stacks and optimistic network receiver persistence. Compared to common uses of zero-copy IO stack APIs, such as memory mapped files, zIO can improve performance by up to 17% due to reduced TLB shootdown overhead.

1 Introduction

Zero-copy IO has been a long-standing performance goal. Copies introduce memory and CPU overhead, limiting the performance of IO-intensive applications. IO data copies are performed within IO stacks, by their application programming interfaces (APIs), and within applications. Existing work has focused on eliminating copies within IO stacks [27, 28] and within IO stack APIs by developing zero-copy IO APIs [1, 11, 12, 15, 17, 28, 32], including some that strive for transparency [8, 9, 22].

Despite these advances, data from IO is still copied. We find that IO-intensive applications perform up to 8 copies of request data for each IO request (cf. §2.1). Many of these copies occur among subsystems within the applications themselves (*application copies*). Only a fraction is performed at the IO stack API (*IO copies*—for example, many standard POSIX socket and file IO system calls copy data between system and user-provided buffers).

A reason for the continued adoption of copies is that they simplify development. Copies are used as a robust mechanism to pass ownership of data among independent subsystems. A data buffer local to a subsystem cannot be touched by a caller of the subsystem, allowing for subsystem-internal use of the data without worry of corruption or deallocation of the memory backing the data from the outside. For example, copies are used to simplify asynchronous IO. POSIX allows kernel IO stacks to provide internal buffers to IO devices that operate asynchronously. Applications request and copy IO data into user-space buffers, allowing applications synchronous processing of a single buffer at a time, while the IO stack recycles its internal buffers for further asynchronous IO. Finally, applications use copies to simplify data handling, for example to perform alignment, padding, serialization and deserialization, as well as bucketization (cf. §2.2).

Unfortunately, copying is an imperfect tool. While copies provide the aforementioned benefits, they also introduce overhead. Using the Redis [21] key-value store as an example IO-intensive application, we study the overhead of copies for IO, both using Linux kernel IO stacks (§2.2) and using kernel-bypass IO stacks for high-bandwidth IO devices (§2.3). IO copying overhead scales with the amount of copied IO data. As IO devices, in particular for storage and networking, increase bandwidth, copies become the performance-limiting factor in IO-intensive applications [7].

The question we ask is: can we attain the benefits of simple development offered by copying, while alleviating its increasing overheads? As we have seen, application developers opt for copies regardless of the availability of zero-copy IO APIs. We find that zero-copy APIs require application modification, increase code complexity, and are not widely supported (§2.4). Hence, we strive for a solution that allows application developers the freedom to program with copies and to use any IO API, while *transparently* eliminating copies where it makes sense, without requiring application modification.

We present zIO, a transparent zero-copy IO mechanism for IO-intensive applications. IO-intensive applications act between IO stacks, examining and potentially transforming input data before output. zIO tracks data that is read by applications from IO stacks to its final destination (typically another

IO stack, but the data may also be held in memory). In the process, zIO eliminates copies that are unnecessary while maintaining consistency for data that the application accesses. By tracking data and eliminating copies, zIO minimizes the overhead incurred by copies, improving application performance.

zIO works under the assumption that IO-intensive applications often touch only a part of the data they process. Untouched data may remain in its original place, while touched data continues to be copied. However, the challenge is that we do not know a priori what data will be touched. zIO optimistically assumes that most data will remain untouched and, by interposing on IO system calls and C standard library calls like memcpy and memmove, eliminates copies, instead simply marking the target memory area as *intermediate*. zIO can do so transitively for entire copy chains. To maintain consistency, each target area remains unmapped. If the application attempts to touch any intermediate memory area, zIO intercepts the access via a page fault. In this case, zIO performs the copy for touched pages and remaps them. Another challenge is to deal with unaligned memory areas. In this case, zIO performs the copy of unaligned sections of the area and leaves only page-aligned portions unmapped. Unaligned sections are small and copying them does not harm performance.

To avoid harming application performance due to data tracking overhead, zIO dynamically decides on a per IO basis when to track and when to copy (via its *tracking policy*). If the size of an IO buffer is smaller than 16KB, zIO copies the buffer. zIO also tracks the average number of page faults and eliminated copied bytes per buffer. If the ratio of bytes accessed to bytes eliminated from copies exceeds 6%, we impose too much overhead handling page faults to improve application performance and zIO copies the buffer instead.

In addition to eliminating application copies, we also use zIO to eliminate copies across IO stack APIs. To do so, we use kernel-bypass IO stacks in addition to zIO. Kernel-bypass stacks use shared memory to implement their APIs, allowing zIO to track IO as it arrives from the IO devices and eliminate copies, even across the IO stack API. We implement these changes in the TAS [18] network stack and the Strata [20] file system. We discuss how to apply these principles to any IO stack in (§3.4).

By leveraging non-volatile memory (NVM), we achieve a further optimization: *optimistic input persistence*. If input received from an IO stack is persisted in NVM via a storage stack by applications, optimistic input persistence enables end-to-end transparent elimination of copies through to storage. To do so without violating application data persistence requirements, we extend zIO to identify NVM mappings. Data copies to NVM may be eliminated if the original data already resides in NVM. Otherwise, a copy is necessary to enforce persistence. Using this technique, we demonstrate how to achieve *optimistic network receiver persistence* by mapping socket receive buffers in NVM and relying on zIO to transparently eliminate all copies through to the file system.

We make the following contributions:

- An analysis of copying in IO-intensive applications (§2). We study the number of copies made in popular IO-intensive applications and find that copies are common, in particular within applications themselves. We conduct a case study of copies in the Redis key-value store, analyzing when and why copies are carried out. Finally, using the Redis case study, we demonstrate that copies are a performance bottleneck for IO-intensive applications, especially when leveraging optimized kernel-bypass IO stacks.
- We present zIO, a transparent zero-copy IO system for IO-intensive applications. zIO addresses the presented overheads due to copying. We show how to use zIO to eliminate application-level copies. We show how to eliminate IO stack API copies when combining zIO with kernel-bypass IO stacks. We show how to achieve optimistic input persistence by leveraging NVM.
- We implement zIO as a user-space library. When executing on top of the Linux kernel network and storage stacks, zIO successfully eliminates application copies of IO buffers. We also integrate zIO with the kernel-bypass IO stacks TAS [18] and Strata [20], enabling it to additionally eliminate copies performed by the IO stack APIs.
- We break down zIO's performance contributions with microbenchmarks and analyze the overheads of buffer tracking. We evaluate the performance benefit to IO-intensive applications, like Redis [21], Icecast [37], and MongoDB [25] and compare to Linux and kernel-bypass IO without copy elimination, where zIO improves performance by up to 1.8× and 2.5×, respectively. We also compare zIO's performance to common uses of zero-copy IO stack APIs, such as memory mapped files, where zIO can improve performance by up to 17% due to reduced TLB shootdown overhead.

2 Background

IO-intensive applications often make several copies of IO data while processing it. We survey the prevalence of these copies in IO-intensive applications (§2.1). To learn how copies are used for IO, we study one of these applications, Redis, and investigate how it uses copies to do IO processing (§2.2). Looking forward, we investigate how copies can become a limiting factor to IO performance (§2.3). Zero-copy APIs are a potential alternative to IO copies. We study their intended use and the tradeoffs they make (§2.4).

2.1 Copies in IO-Intensive Applications

We study the prevalence of IO data copies in popular IO-intensive applications. We identify the call site of these copies and break down occurrences into copies that are involved in an IO stack API call and copies occurring within application subsystems. Our methodology involves a source code analysis of IO data flows through application subsystems from input to output. We identify what methods applications use

		Copy call site	
Application	**Operation**	**App**	**IO Stack**
Redis [21]	SET	4	2
	GET	2	1
Icecast [37]	Cast to N clients	0	1 + N
Ceph [34]	Write	1	2
	Read	0	2
Anna [36]	PUT	5	3
	GET	4	3
MongoDB [25]	Insert	3	2
	Disk sync	1	1
	Read	2	2
Tensorflow-serving [26]	Inference	2	1
Nebula Graph [33]	Insert vertex	5	2
	Store a vertex	4	3

Table 1. Number and call site of copies between input and output for various application operations.

to copy IO data and how copies are affected by the executed functionality and its parameters. We find that all applications investigated use C standard library functions, such as `memcpy` and `memmove`, to copy data. We use this insight to validate our source code analysis via an execution of the relevant application operations under a debugger set to break on these C library memory copy APIs. For each application operation, we count the number of breakpoints hit on IO code paths between input and output and check that the count matches that of our source code analysis.

Table 1 presents the number of copies made at the IO stack and within various IO-intensive applications, broken down by operation. We are specifically interested in the copy of potentially large IO data, as small data copies do not significantly impact application performance. For example, the Anna [36] key-value store conducts up to 45 copies of keys during a PUT operation. We ignore these copies in the table.

While the number of copies varies among applications and operations, we can see that IO-intensive applications extensively copy IO data between input and output. We can also see that applications often make more internal copies of IO data than at the IO stack API. For example, Redis [21] makes twice as many application-internal copies than at the IO stack for a SET request. IO-intensive applications also often employ third-party libraries. For example, the Anna [36] key-value store uses gRPC [14] and Protobuf [13] to serialize and deserialize data. We observe that these libraries incur up to 3 per-IO data copies for this task, leading Anna to make up to 5 internal IO copies. This indicates that zero-copy IO APIs are only going to eliminate a fraction of the overhead due to copies. Application-internal copies, including in third-party libraries, often constitute a similar or even larger fraction of copy-induced CPU overhead.

2.2 Copy Case Study: Redis

To better understand these IO data copies, we study the Redis SET request. Redis [21] is a popular key-value store providing a rich RPC-based network API to an in-memory store, persisted via snapshots or operation logging. We configure Redis to log SET operations to study a use-case that is equally network and storage IO intensive. In our study, each SET request provides a new value, identified by a 32 byte key. We run a single-threaded Redis server instance on the evaluation platform described in Section 5. We use `redis-benchmark` [21] to attach 64 clients over a 100G network, enough to saturate the server. We configure Redis to use an append-only file to persist data without delay. This configuration provides strong crash consistency—every operation is persisted before it is acknowledged. We evaluate the number of memory copies that Redis performs per SET request and we study these copies.

#	Source	Destination	Call site
IO_1	Socket buffer	`c.socket_buf`	`readQueryFromClient`
A_1	`c.socket_buf`	`c.socket_buf`	`processInputBuffer`
A_2	`c.socket_buf`	`hash_node`	`dbAdd`
A_3	`c.socket_buf`	`c.write_to_aof`	`feedAppendOnlyFile`
A_4	`c.write_to_aof`	`aof_buf`	`flushAppendOnlyFile`
IO_2	`aof_buf`	Append-only file	`flushAppendOnlyFile`

Table 2. Copies in Redis SET request. IO_i are IO stack copies, A_j are application copies. `c` is a per-client structure.

As reported in Table 1, we find that Redis copies request data 6 times for each SET request. We list these copies and their call sites in Table 2. As we can see, Redis performs copies to read and deserialize the SET request and to store the request both in an in-memory hash table and in the append-only file. After reading a number of kilobytes from the network socket to an input buffer (copy IO_1), Redis identifies the next request within the input buffer and removes its headers from the buffer (copy A_1). If the identified SET request is admissible, Redis creates a copy of the key and value data to store in its in-memory hash table (copy A_2). Redis then reformats the request so it can be logged to its append-only file and appends the request to a per-client log (copy A_3). Redis uses per-client logs to support *group commit*—Redis can process a number of pending client requests in-memory and then persist and acknowledge these requests in a batch, eliminating storage stack overheads incurred for small IO. To do so, Redis first combines all pending per-client logs into a single log stream (copy A_4) and then writes the log stream to the append-only file (copy IO_2).

All of these copies could have been avoided. However, it would have required the Redis developers to design a complex set of coordinated, reference counted buffer descriptors that can track each request and its data in each source buffer (in this case, a network socket buffer). Reference counts provide use-after-free protection. Use-after-free [38] is an error condition where one part of an application or IO stack frees an allocated IO data buffer and re-uses it for other purposes while another part of the application or an IO device still uses the data. Use-after-free protection requires complex ownership tracking, including APIs to convey ownership transfer. Further, fine-grained memory management is required, including the ability to free fragments of a previously allocated memory

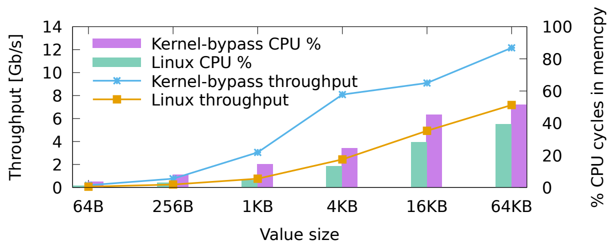

Figure 1. Redis SET throughput and fraction of CPU cycles in `memcpy` over value size, with and without kernel-bypass.

buffer. For example, the headers of incoming SET requests can be freed after each request is processed, while the keys and values remain in their original buffers for as long as they are stored in the key-value store. This creates buffer fragmentation that is difficult to resolve via memory management alone, requiring further APIs to defragment buffers over time. All of these APIs are complex and it is often impossible to support them in an application when third-party libraries or IO stacks are used that do not support the APIs.

2.3 When is IO Performance Copy-Limited?

As hardware IO bandwidth continues to increase and IO stacks become lighter-weight to keep up with increasing application demand for bandwidth, copies start to limit IO performance. In light of these trends, we investigate the performance impact of copying for Redis SET requests over increasing value sizes, while using heavy-weight in-kernel and light-weight kernel-bypass IO stacks. We use the same Redis configuration described in Section 2.2, evaluating the Linux network stack and the ext4 file system, as well as TAS [18] and Strata [20] for kernel-bypass. As we vary the value size, we measure throughput and the fraction of CPU cycles spent in data copies per request with Linux `perf`.

The results are presented in Figure 1. We can see that larger value sizes imply higher per-core throughput. Also, kernel-bypass IO improves throughput by up to 4×. This is intuitive. Kernel-bypass IO is lighter-weight than in-kernel IO, while larger IO granularity amortizes IO stack overheads. As hardware IO bandwidth continues to increase, it is likely that applications will employ larger IO sizes to leverage the available bandwidth. At the same time, IO stacks will become lighter-weight to provide the necessary performance to keep up with the increasing IO speeds.

We can also see that larger value (and thus IO) sizes cause a noticeable increase of per-request CPU cycles spent in memory copies. We already know that Redis makes 6 copies of IO data for each SET request. As value sizes increase, the amount of CPU cycles spent copying them must naturally also increase. Even moderate value sizes of 64KB cause 39% of per-request CPU cycles to be spent in memory copies using the heavy-weight Linux kernel IO stacks. The lighter-weight kernel-bypass IO causes an even larger fraction of up to 52% of per-request CPU cycles to be spent in memory copies, owing to a reduction of per-request CPU cycles spent in IO stack processing. For even larger value sizes of 512KB, CPU cycles spent in copying reaches 60%.

2.4 Limitations of Existing Zero-Copy IO APIs

Various zero-copy APIs have been proposed to limit the number of copies involved in IO-intensive applications. Zero-copy IO APIs fall into two categories. (1) Single-stack APIs, and (2) cross-stack APIs. Single-stack APIs eliminate copies for a particular IO API, such as the network sockets system call API. Cross-stack APIs eliminate copies across IO APIs. For example, across network and storage APIs. We study the tradeoffs made by each category in this section.

Single-stack APIs. Single-stack APIs provide zero-copy IO for single IO stacks. The API is specific to the IO stack and is often provided in the form of new parameters or tweaks to a familiar IO API that enable zero-copy, typically along with a set of invocation and environment requirements that have to be met by the application developer for the API to function. We describe a number of storage and networking zero-copy APIs here, including memory mapped files, Linux FreeBSD, and Solaris zero-copy networking, remote direct memory access (RDMA), and the Arrakis [28] zero-copy networking API.

Memory mapping files is one of the oldest zero-copy storage IO APIs. Applications map (parts of) files into their virtual address space, which the OS implements by loading the file into the page cache and providing direct application access to the relevant pages. Page cache entries may be directly written to disk, without further copies. More recently, applications may also map non-volatile memory (NVM) directly into virtual memory, referred to as direct access (DAX) [2]. Memory mapped files restrict some file IO. For example, memory mapped files cannot be appended to. Instead, an application developer has to determine the file size in advance and `truncate` the file to the desired length before memory mapping it. Further, the interface does not allow applications to make atomic modifications to file data without copying data to their own buffers first.

Linux provides two networking zero-copy APIs [11, 12] for TCP sockets. A zero-copy `send` will lock a given application buffer into memory and start the transmission. If transmission is not complete by the time `send` returns, the application must take care not to touch the buffer. The zero-copy mechanism will place a notification message in the error queue associated with the socket, which has to be monitored by the application. When an "error" packet appears, it can be examined to determine the status of the operation, including whether the transmission succeeded and whether it was able to run in zero-copy mode.

For zero-copy receive, Linux allows to memory map a TCP socket. If several network conditions are met, including the next incoming data chunk being page-sized and page-aligned, the socket buffer containing the incoming chunk will be mapped into the calling process's address space, where it can be accessed directly. When the incoming data has been processed, the application calls `munmap` to release the pages and free the buffer for another incoming packet. The mechanism only works if the application developer has knowledge of exactly what each incoming packet will look like.

RDMA [30] provides zero-copy IO either by directly reading/writing remote memory or by pre-registering buffers with the network card for receive and transmit operation. Similarly, Arrakis [28] provides a zero-copy IO network socket interface that returns buffers and consumes them, rather than letting the application specify its own buffers. All of these interfaces introduce the same complexities of buffer ownership management and knowledge of network conditions. These conditions are often difficult to meet and the additional buffer management burden is cumbersome for many developers.

A limited solution proposed by SocksDirect [22] and also implemented in FreeBSD [8] and Solaris [9] to *transparently* avoid copies in the network sockets API is to simply remap the pages carrying IO data from the network stack to the application-provided buffer location, instead of copying the data. This works in cases where both buffers are page-aligned and it requires the NIC to be able to isolate packet payloads and place them into page-aligned buffers. To isolate payloads, SocksDirect requires RDMA, while Solaris requires ATM. FreeBSD supports traditional Ethernet NICs, but requires that the maximum transfer unit is configured to be greater than the hardware page size, which may be undesirable or difficult. Unfortunately, applications often misalign IO buffers, even if memory allocators return aligned memory. For example, when headers are inserted into a buffer and IO is read to a location after the header. Our investigation into Redis shows that only about 40% of IO data can be remapped using this approach. Further, transmit buffers must be kept until acknowledged, leaking memory if acknowledgments lag. The limited applicability, security concerns (including from malicious NICs [24]), and hardware requirements led the FreeBSD developers to abandon the transparent zero-copy socket API in FreeBSD 11.

Cross-stack APIs. A variety of cross-stack APIs attempt to eliminate copies across IO stacks, in particular the networking and storage stacks. To do so, they offer new and often higher-level APIs that the application developer must use. These new APIs avoid copies. We describe three example cross-stack APIs here, the Linux `sendfile` family of system calls, PASTE, and Demikernel.

The Linux `sendfile` system call (and cousins `splice` for pipes and `copy_file_range` for files) transmits data from the storage stack via the network stack without user-level copies. The API is restricted to network and storage IO and does not permit the application developer to inspect data before transmission. To add any application data, such as headers, the developer must use the `TCP_CORK` option, requiring them to add the necessary data within a 200 millisecond time window. `sendfile` does not allow sending or receiving from/to user memory. The API is used to send static files across the network but is increasingly obsolete with the prevalence of dynamic in-memory content.

PASTE [15] provides an API that combines the network stack with persistent data structures in NVM to avoid copies. PASTE builds on the Netmap [31] kernel framework to place packets from the network interface card (NIC) directly in NVM. Developers can refer to these packets from application-specific persistent data structures. However, PASTE operates at the packet level and requires developers to track network connections and decode byte streams to find relevant data to persist. PASTE also requires the developer to implement a copy-on-write scheme to efficiently return packet buffer space to the NIC after use. Due to the complexity of its API, PASTE's intended use is constrained to run-to-completion processing of requests that fit in individual network packets.

Demikernel [38] eliminates copies between kernel-bypass networking stacks, like DPDK and RDMA, and kernel-bypass storage stacks, like SPDK. The Demikernel memory manager allocates memory to applications from DPDK's memory pool and it registers that memory with RDMA. This allows Demikernel applications to receive data over the network and to store it without any copies. Demikernel offers a queue-oriented interface, PDPIX, which replaces datapath IO calls with pushes and pops to and from queues that may return tokens if data is unavailable. Demikernel's interface requires application developers to implement run-to-completion IO processing. This simplifies zero-copy IO for Demikernel, but it limits the application developer. The Demikernel interface does not support making in-place updates to IO data or allow developers to schedule input and output beyond handling each input request to completion, and it cannot eliminate any further copies an application might make internally to process input.

Summary. Both categories of zero-copy IO stacks seek to eliminate copies involved in IO stack APIs. However, in doing so they introduce complexities, such as buffer ownership management involving special API calls. They also introduce restrictions, such as requiring run-to-completion processing, buffer alignment, or disallowing in-place updates. Finally, they may enforce external IO properties, such as packet layout and MTU size. These complexities and restrictions are difficult for developers to navigate and external IO properties are often difficult or impossible to enforce. Further, **none of the existing zero-copy APIs provide transparent copy elimination across IO stacks or eliminate copies that are made within the application**. For these reasons, both application and kernel developers forgo zero-copy APIs, as they often struggle to outperform APIs that involve copies and are deemed not worth the complexity they introduce [12].

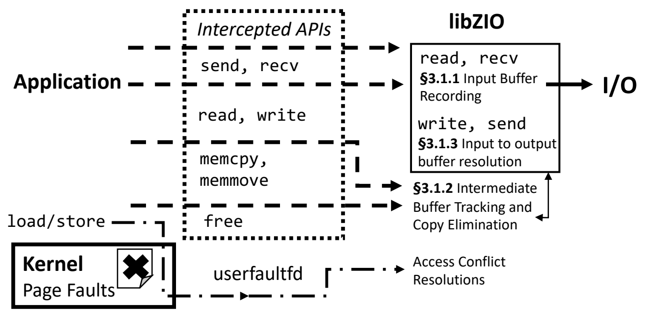
Intercepted APIs
libZIO
Application
send, recv
read, write
memcpy, memmove
free
load/store
Kernel Page Faults
userfaultfd
read, recv
§3.1.1 Input Buffer Recording
write, send
§3.1.3 Input to output buffer resolution
I/O
§3.1.2 Intermediate Buffer Tracking and Copy Elimination
Access Conflict Resolutions

Figure 2. zIO overview.

3 zIO Design

zIO is a user-level library (libzIO) that may be dynamically and transparently linked to applications. zIO intercepts a number of C standard library and IO system calls (shown in Figure 2), including memory copy and management, and socket and file IO. zIO leverages userfaultfd [5] to intercept page faults, which may be caused by applications touching intermediate memory buffers. We now describe zIO in three parts. First, we describe how zIO tracks data within the application to eliminate *application copies* (§3.1). Second, we describe how we extend zIO with kernel-bypass IO stacks to allow it to eliminate *IO stack copies* (§3.2). Third, we describe how to realize *optimistic input persistence* by mapping appropriate IO buffers into NVM (§3.3).

3.1 Application Copy Elimination

To eliminate application copies, zIO tracks IO data buffer locations transitively through the application. zIO intercepts any copies of IO buffers and optimistically forgoes them. To provide data consistency in the face of the application accessing any intermediate, uncopied buffer locations, zIO leverages page faults to detect and resolve these accesses.

Figure 3 shows the mechanisms involved in this process via an example involving a key-value pair being read from an input IO stack, processed by the application, and written to an output IO stack. On input (e.g., IO stack `read/recv` calls), the provided location of the application buffer is recorded by zIO (①). For the purpose of application copy elimination, this is the original location of the IO data. zIO uses this information to track and eliminate any application-level copies of this data. Upon memory copy of any tracked data (`memcpy/memmove` calls), zIO unmaps the destination buffer, forgoes the copy, and tracks the destination buffer as *intermediate* (②). Some buffer locations may not be page aligned, in which case, *buffer fringes* have to be copied (`app_buf3` in Figure 3 is unaligned, causing copies in ③ and ④, where it is used as destination and source buffer, respectively). To provide consistency when applications access intermediate buffers, zIO leverages page faults. If a page fault to any intermediate buffer occurs, zIO finds the original buffer location to resolve the page fault with the appropriate data by lazily copying faulted pages (⑤). Finally, when tracked data is written to another IO stack (e.g., `send/write` calls), zIO intercepts the call and provides the original buffer instead of the application-provided intermediate buffer, but including any intermediate data updates (⑥). Before we detail each of these mechanisms, we describe zIO's tracking granularity and data structure.

Page granularity copy elimination. To be able to provide data consistency via page faults, zIO eliminates copies only at page granularity. However, buffers may reside at any address in virtual memory. To resolve this issue, zIO will only eliminate the part of a copy that lies within page boundaries of the provided buffer (i.e., unaligned buffer start addresses are rounded up to the page boundary, while unaligned buffer end addresses are rounded down)—the *core buffer*. The *left and right buffer fringe*—the beginning and end of an application buffer that is beyond the core buffer page boundaries, respectively—is always copied. While this approach involves small copies for unaligned buffers, we find that it often helps performance. The left and right buffer fringe often contain headers and footers that applications are more likely to access than the core.

Intermediate buffer tracking via skiplists. zIO records the locations of all application data buffers containing IO data. As buffer tracking has to incur minimal overhead and records are frequently mutated, we choose a skiplist for probabilistic fast search and insertion. Each entry in the skiplist keeps track of the original buffer address, a corresponding core intermediate buffer address, the length of the core intermediate buffer as a number of base pages, the size of the left intermediate buffer fringe in bytes, a timestamp of the last copy (cf. §3.1.7), and a free flag (cf. §3.1.4, not shown in Figure 3). The skiplist is sorted by intermediate buffer address. We evaluate the performance of buffer tracking via skiplists in §5.2.1.

3.1.1 Input buffer recording. When data is read from an IO stack via a function or system call, zIO intercepts these operations. We have implemented intercepts for all common POSIX network and file system calls. According to its policy (cf. §3.1.6), zIO records the application-provided destination buffer as the original buffer, along with an identity core intermediate buffer (①). This record filters IO buffers for copy tracking—zIO only eliminates copies for data originally read from an IO stack.

3.1.2 Copy tracking and elimination. zIO identifies copies within the application by interposing on the standard library memory copy calls `memcpy` and `memmove`[1]. These calls take a source and destination buffer address, as well as a size (in bytes) to copy. On each call, according to policy (cf. §3.1.6), instead of executing the copy, we record in the skiplist the core

[1]Variations of these calls use `memcpy` and `memmove` in our standard C library. For other C libraries, variations may need to be explicitly intercepted.

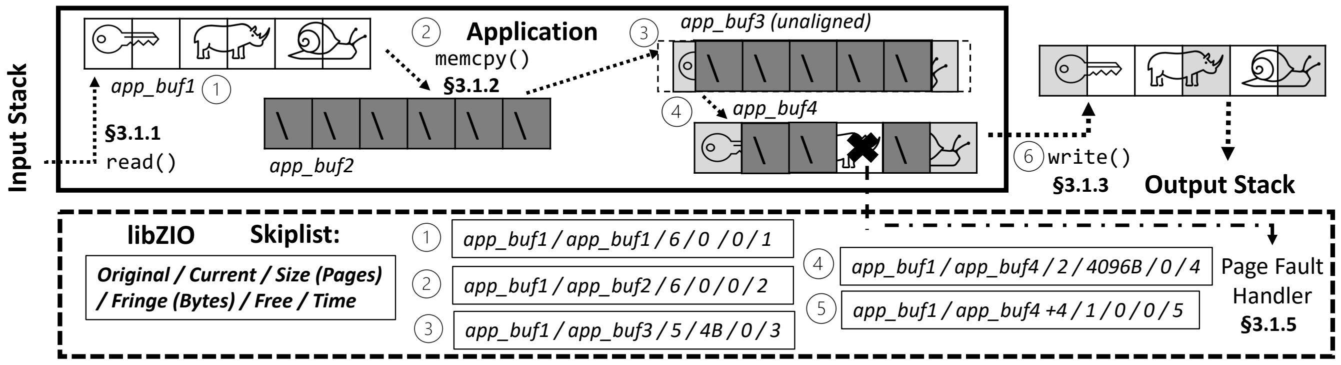
Input Stack
read()
§3.1.1
app_buf1
①
Application
② memcpy()
§3.1.2
app_buf2
③ app_buf3 (unaligned)
④ app_buf4
⑥ write()
§3.1.3
Output Stack
libZIO
Skiplist:
Original / Current / Size (Pages) / Fringe (Bytes) / Free / Time
① app_buf1 / app_buf1 / 6 / 0 / 0 / 1
② app_buf1 / app_buf2 / 6 / 0 / 0 / 2
③ app_buf1 / app_buf3 / 5 / 4B / 0 / 3
④ app_buf1 / app_buf4 / 2 / 4096B / 0 / 4
⑤ app_buf1 / app_buf4 +4 / 1 / 0 / 0 / 5
Page Fault Handler
§3.1.5

Figure 3. zIO application copy elimination example with an IO buffer spanning 6 pages. Original buffer pages are unshaded. Shaded pages are copied. Dark shaded pages are unmapped.

destination buffer and its size as intermediate location and size (e.g., ②, where app_buf1 and app_buf2 do not have a fringe).

To determine the original buffer location, we first use the core source buffer address to search through the skiplist to see if it falls within any existing intermediate buffers. If it does, we use that buffer's original buffer location, and, if this is the first time this original buffer is copied, zIO also remaps the core original buffer read-only to detect any application modification to it. If we find no intersecting intermediate buffer, then this data did not originate from IO (cf. §3.1.1) and we execute the copy, forgoing any tracking of this buffer. Finally, if the data originated from IO, we record the size of the left intermediate buffer fringe (③, where app_buf3 is unaligned and has a left fringe of 4 bytes—it also has a right fringe, but we do not need to record it). The left buffer fringe is necessary to resolve access conflicts (cf. §3.1.5). If the destination location is within a buffer that is already tracked in the skiplist, the skiplist entry is updated with the new buffer information.

zIO unmaps the core intermediate buffer and registers it with userfaultfd to intercept application access. The union of left and right buffer fringes of original and intermediate buffer is copied. For example, if the source buffer has a left fringe of 4 bytes and the target buffer has no left fringe, then the left fringe of the target buffer becomes 4KB, as the original left fringe taints the first 4 bytes of what could have been a core page of the target buffer, making the entire page part of the fringe (④, where app_buf3 has a left fringe of 4 bytes, app_buf4 acquires a left fringe of 4KB).

The cost of unmapping intermediate buffers is often avoided or amortized. For example, buffers that are allocated on the heap are backed with physical memory and mapped only on first access (this *lazy memory allocation* is the default in Linux, for example). zIO can simply register these unmapped buffers with userfaultfd. Statically allocated buffers are often reused across requests instead of freed and reallocated. These buffers remain unmapped across requests if they are not otherwise accessed by the application. Upon reuse, zIO simply updates the skiplist when new IO data is processed.

3.1.3 Input to output buffer resolution.

Whenever data is written to an IO stack, zIO interposes on the IO stack API and searches the skiplist to see if the core buffer being written intersects with any intermediate buffers tracked in the skiplist. If a match is found, zIO modifies the write operation to use any original buffer addresses recorded in the skiplist. This may result in a single IO stack source buffer location being transformed into multiple buffer locations (⑥, where shaded areas of the output are copied, unshaded areas are sourced from original buffer locations). If the IO stack API supports gather IO, we leverage that API to refer to the appropriate buffer pages when generating the output IO call. If the IO stack does not support gather IO, zIO breaks up the output call into multiple calls that refer to each individual buffer.

3.1.4 Freeing buffers.

Finally, zIO interposes on free. This interposition allows zIO to look up and delete skiplist entries that are potentially no longer needed. If an intermediate buffer is freed that means we have successfully eliminated a copy; the contents of the buffer were not touched and the application has specified that it no longer needs it. At this point, the skiplist entry can be deleted and the memory region unregistered from userfaultfd. If an original buffer is being freed, the skiplist entry is only marked as freed to prevent use-after-free violations. Buffers marked as freed are deleted upon garbage collection (see below).

3.1.5 Access conflict resolution.

When a core intermediate buffer is touched by the application, it will trigger a page fault. zIO looks up the faulting page number in the skiplist. zIO then maps the faulting page, potentially allocating it (lazy memory allocation), and copies the data from the original buffer, as recorded in the skiplist entry. zIO uses the left buffer fringe size to determine the byte offset of the intermediate core buffer, which is used as an offset into the recorded original buffer to determine the copy source address.

If a page at the beginning or end of a buffer is faulted in, it is removed from the tracked buffer core and thus copied going forward. If a page is faulted in the middle of a buffer, a new skiplist entry must be created for the second section of the buffer, if it meets the appropriate size threshold (⑤). The original buffer is effectively split; the buffer before the faulting page is considered part of the originally tracked buffer and the buffer after the faulting page is a newly tracked buffer.

If a core original buffer is modified by the application, it also triggers a page fault. In this case, zIO walks the skiplist to determine any intermediate buffers derived from the core original buffer page. zIO copies the faulting original buffer page to the relevant intermediate buffers and resets the access permissions to the relevant original and intermediate pages.

3.1.6 Tracking policy. We determine experimentally (cf. §5.1.2) that for data buffers smaller than 16KB, the overhead of tracking outweighs any performance benefits from eliminating copies. Hence, we configure zIO to track and elide only sufficiently large copies (core buffer sizes of 16KB or larger).

There is also an overhead for handling page faults. We determine this experimentally (§5.1.4) under a number of conditions. For example, we find that if the ratio of bytes accessed by the application to bytes eliminated from copies exceeds 6%, we no longer see a performance benefit with a single application thread. This number can change with a different number of threads and is fully explored in §5.1.3. After these thresholds, we stop eliding copies for these buffers.

3.1.7 Intermediate buffer garbage collection. zIO avoids tracking an arbitrary number of entries to prevent memory exhaustion and skiplist performance reduction. For example, tracked intermediate buffers may be kept indefinitely in memory by the application, causing skiplist entries to accrue. Hence, skiplist entries are garbage collected periodically (once every second in our prototype). For each collected skiplist entry, we must fill any intermediate buffers with consistent data. This is done via the same process as conflict resolution. The region is mapped and the data is copied from its original location at the appropriate offset. Buffers marked free can be freed immediately.

zIO's garbage collection policy collects intermediate buffers that have been least recently used in copies. A timestamp on each skiplist entry (not shown in Figure 3) keeps track of the last time the entry was involved in a copy. If the skiplist grows beyond a threshold, zIO collects the least recently used entries.

3.2 IO Stack API Copy Elimination

Simply linking zIO when kernel-bypass IO stacks are used already provides transparent zero-copy IO. However, we can achieve further performance benefits by modifying these IO stacks to integrate with zIO more tightly. We now describe how we integrate zIO with kernel-bypass IO stacks to optimize IO stack API copy elimination.

Kernel-bypass IO stacks are a good fit for zIO, as they communicate with the application via shared library calls and shared memory—mechanisms that zIO can transparently process at user-level—rather than system calls. We choose the TAS [18] and Strata [20] kernel-bypass network and storage stacks, which are state-of-the-art. Strata, in particular, is a good fit, as it uses a per-process operation log in NVM, mapped into userspace, to persist file writes. zIO transparently intercepts Strata's memory copies into this log and can provide transparent copy elimination, provided that the original buffer already resides in NVM.

Input API copy elimination. POSIX file and socket input calls (e.g., `read` and `recv`) require applications to provide a buffer that input data is copied into. In TAS and Strata, these library calls internally call `memcpy` to copy from an IO stack internal buffer to the application-provided buffer. zIO transparently tracks and eliminates this copy across the IO stack API (cf. §3.1). As the source buffers are IO stack-private, we do not need to protect the original source data buffer by remapping it read-only. Instead, we modify the IO stacks to execute zIO's garbage collection protocol for any tracked buffers that the IO stack intends to free or overwrite. To prevent this from happening frequently, we can configure the IO stack internal buffers to be sufficiently large. For example, socket receive buffers can be resized to hold at least the expected size of input data per IO request.

Output API copy elimination. POSIX file and socket output calls (e.g., `write` and `send`) require applications to provide a source buffer that output data is copied from. As with the input API calls, zIO already transparently eliminates stack-internal memory copies. As output buffers are IO stack-private, no unmapping is necessary. Instead, we modify the IO stacks to fetch the original buffer locations from zIO when the output data is processed. For example, when TAS sends payload from the socket transmit buffer or when Strata "digests" [20] the update log. When zIO has to resolve copies due to mis-speculation or garbage collection, the relevant output buffer fields are simply filled in with the appropriate data. When the IO stacks ask zIO for original buffer locations, filled output buffers will not be marked as intermediate.

3.3 Optimistic Input Persistence

To realize optimistic input persistence for end-to-end IO copy elimination when data is persisted in NVM by a storage stack, we simply have to ensure that the original data already resides in NVM. zIO automatically detects the type of memory backing a virtual memory mapping. If original and intermediate buffers are backed by NVM, zIO can eliminate and track any copies among the buffers, while ensuring persistence. We describe here how we use this feature to realize *optimistic network receiver persistence*, where incoming data from the network does not need to be copied to storage.

Optimistic network receiver persistence. TAS uses shared memory for socket receive buffers between its TCP fast-path process and processes linking the kernel-bypass libTAS library. The fast-path writes incoming payload directly into socket receive buffers residing in this shared memory. We can realize optimistic network receiver persistence simply by mapping the socket receive buffers into NVM. zIO will detect that original buffers are backed by NVM and eliminate copies end-to-end to the Strata update log, which also resides in process-local NVM.

3.4 Discussion

Huge pages. Huge pages (pages larger than the system's base page size) are desirable for improved memory address translation performance. However, tracking IO buffers requires fine-grained page protection, as tracked buffers may be smaller than the huge page size. In this case, zIO's fine-grained page mapping requests force the OS to break huge pages into base page mappings. Indeed, an investigation of the Redis YCSB benchmark with 512KB value size (cf. §5.2) shows that Linux with transparent huge page (THP) support maps 40% of Redis' working set with huge pages when zIO is not used, while mapping only 35% of the working set with huge pages when zIO is used.

Unfortunately, if the application stores IO buffers in reserved huge page memory using Linux's `hugetlbfs` mechanism, fine-grained page protection is disallowed and zIO can only track buffers at huge page granularity. Note that Linux could technically allow fine-grained protection for reserved huge page memory, while still allocating memory at huge page granularity. This would be compatible with zIO.

Luckily, transparent zero-copy and huge pages do not need to be at odds. zIO operates on the assumption that tracked IO buffers are seldom touched by applications. Hence, leveraging fine-grained page protection for tracking IO buffers does not impact application performance in the common case, as these mappings are seldom exercised. On mis-speculation, zIO's policy reverts to copying IO buffers and the OS may again map them with huge pages. This may happen transparently when THP support is enabled in the OS. Our application benchmarks run with THP, showing that transparent zero-copy IO still outperforms any potential slow-down from fine-grained page protection.

Linux kernel IO stack API copy elimination. While we present IO stack API copy elimination with kernel-bypass stacks (§3.2), we believe it is possible to provide IO stack API copy elimination for the Linux kernel IO stacks in certain cases by leveraging Linux's zero-copy IO APIs (cf. §2.4). For example, using Linux's zero-copy socket receive API (cf. §2.4), zIO can memory map kernel TCP socket receive buffers into user-private memory when sockets are created. It can then intercept application `recv` calls and track the target application buffer as an intermediate buffer, with the private socket buffer mapping as the original. This eliminates the IO stack API copy for `recv`, similar to our integration with TAS, as described in §3.2. Network receiver persistence may also be realizable, albeit with kernel modifications, by mapping socket buffers into a file stored in NVM and then using the `FICLONERANGE` ioctl to remap core data buffers to their final destination upon input to output resolution to a file. We leave IO stack API copy elimination for the Linux kernel IO stacks for future work.

4 Implementation

Our zIO implementation consists of two key components. The first component is tracking data through an application and eliminating copies along the way. The second component is closely integrating this tracking with the kernel-bypass network and storage stacks TAS and Strata, respectively, to provide transparent zero-copy across IO stack APIs, as well as optimistic input persistence.

Application copy elimination. This component of zIO is written in 1,608 lines of C code and is dynamically loaded with `LD_PRELOAD`.

IO stack API copy elimination. To integrate zIO with TAS and Strata to provide IO stack API copy elimination, we modify 184 lines of code in TAS and 66 lines of code in Strata.

5 Evaluation

We analyze zIO's performance via a number of experiments based on a multi-threaded IO microbenchmark, using network and storage stacks, and varying relevant IO and copy parameters. We also evaluate zIO with the IO-intensive applications Redis [21], MongoDB [25], and Icecast [37]. We compare zIO to Linux and kernel-bypass IO stacks without any copy optimizations.

Our evaluation answers the following questions:

- What is the impact of copies on IO performance? What benefits to IO processing throughput does zIO provide by transparently eliminating copies? How do the number of copies per IO (§5.1.1), IO size (§5.1.2), and number of IO threads (§5.1.3) affect the observed performance?
- What are the overheads zIO introduces by tracking data? How do overheads increase as applications touch the data they copy, causing zIO to mis-speculate? How effective is zIO's tracking policy in avoiding mis-speculation? (§5.1.4)
- How do zIO performance improvements break down into its mechanisms? By how much can we improve IO performance when employing optimistic input persistence with NVM? (§5.2.1)
- What benefits to IO processing throughput and latency does zIO provide by eliminating copies within IO-intensive applications, such as Redis (§5.2), Icecast (§5.3), and MongoDB (§5.4)? In what situations might zIO hurt application performance (§5.3.1)?
- How does zIO perform compared to zero-copy IO APIs, such as memory mapped files and `sendfile`? (§5.3.1)

Evaluation platform. We run our evaluation on a single socket of a dual-socket Intel Cascade Lake-SP system running at 2.2GHz with 24 cores per socket and a 100 GbE ConnectX-5

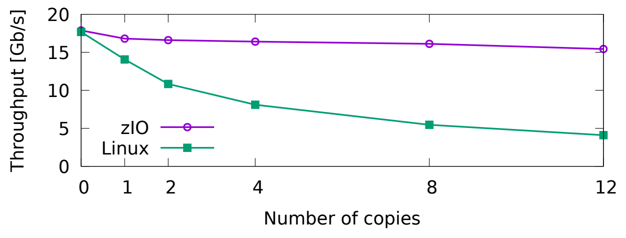

Figure 4. Linux throughput versus zIO application IO copy elimination (512KB IO size).

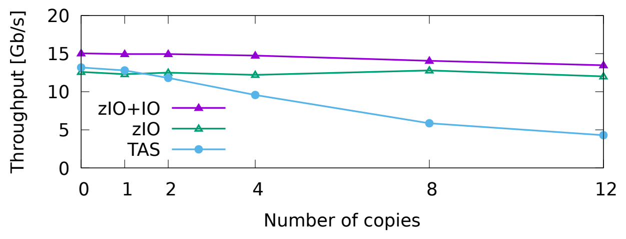

Figure 5. TAS throughput versus zIO application and IO stack API copy elimination (512KB IO size).

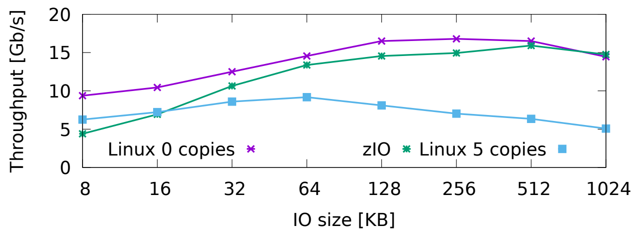

Figure 6. zIO throughput versus Linux with 0 and 5 copies.

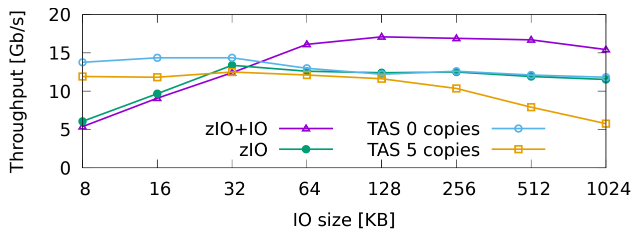

Figure 7. zIO throughput versus TAS with 0 and 5 copies.

NIC. Each socket has 192 GB of DDR4 DRAM and 3 TB of Intel Optane DC NVM. To leverage all 6 memory channels per processor, there are 6 DIMMs of DRAM and NVM per socket. The machine runs Fedora 27 with Linux kernel version 5.10.0. We use the latest master branches of TAS [3] and Strata [4].

5.1 Microbenchmarks

We quantify the overhead introduced by copies of IO data and the benefit that zIO provides for various IO parameters via a simple echo server benchmark. Our evaluation setup is the same as in §2.2, but in place of Redis we run a simple TCP echo server that echoes client messages back to the sender. To simulate IO-intensive application processing, our echo server can make a configurable number of copies to the IO data. Beyond the number of copies, we also vary other IO parameters, such as IO size, fraction of IO data accessed, and number of echo server threads. We report the average echo server throughput, measured at the client, over 3 runs for each configuration, using the steady-state throughput of each run.

5.1.1 Number of Copies. We first evaluate IO performance with a varying number of copies of the IO data made before it is echoed. We compare four scenarios: Vanilla Linux (Linux), Linux with zIO application copy elimination (zIO), vanilla kernel-bypass (TAS), kernel-bypass with zIO application copy elimination (zIO), and kernel-bypass with zIO application and IO stack API copy elimination (zIO+IO). We run this experiment with 512KB IO, using a single server thread. For each run, we vary the number of times the request is copied before being echoed.

Figures 4 and 5 present the results. We can see that an increasing number of application IO copies decreases the achieved throughput for Linux and TAS networking[2], due to the involved copying overhead. Kernel-bypass maintains high throughput with more copies than Linux, as more CPU cycles are available for copies due to the lighter-weight kernel-bypass network stack. zIO maintains performance close to the configuration without copies for both stacks, showing that it successfully eliminates these copies with negligible overhead. With 12 copies, zIO improves throughput by 3.8× with Linux and by 2.8× with TAS. Finally, zIO+IO improves throughput by up to 21% versus zIO by additionally eliminating IO stack API copies.

5.1.2 IO Size. We next investigate how IO size affects performance, using a single echo server thread. To evaluate the overhead of tracking small IO, we disable zIO's IO size threshold for this benchmark, causing zIO to always track buffers and eliminate copies. We vary the IO size from 8KB to 1MB and evaluate two extreme copy scenarios (cf. Table 1): 5 application copies and 0 application copies. Figures 6 and 7 present the results.

zIO benefits large IO. Firstly, we can see that Linux has poor performance with small IO, but performance improves as IO size increases. TAS performs better with smaller IO size. This is expected, as kernel-bypass stacks are light-weight. When copies are involved, both Linux and TAS perform worse, in particular as IO size increases. This is also expected, as

[2]We consistently observe TAS throughput to be lower than Linux with large IO sizes. TAS is optimized for small IO and does not do the necessary batching to handle large IO efficiently.

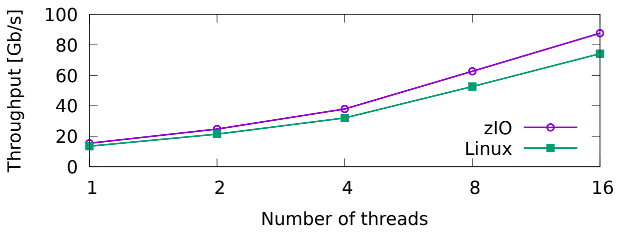

Figure 8. zIO scalability.

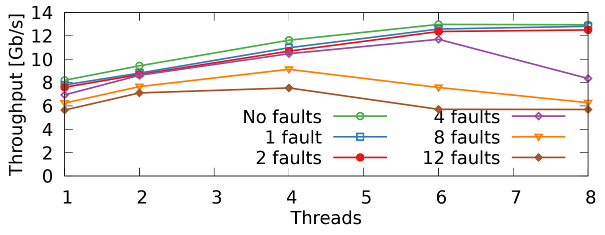

Figure 9. zIO scalability with page faults.

larger copies require more CPU time. zIO improves throughput by up to 2.9× with Linux and up to 2× with TAS as IO size increases, reaching zero-copy performance with IO sizes larger than 512KB for Linux and 32KB for TAS. zIO+IO improves throughput further, by up to 40% versus zIO, for a combined improvement of up to 2.7× versus TAS.

Limits of zIO with small IO. zIO transparent copy elimination is no panacea, as the overhead of zIO tracking with small IO limits throughput. For IO smaller than 16KB, zIO reduces throughput by up to 30% versus Linux. For IO smaller than 32KB, zIO reduces throughput by up to 49% versus TAS. IO sizes smaller than 8KB would incur even further throughput reduction. Based on this measurement, we set zIO's tracking policy to avoid tracking IO buffers smaller than 16KB (cf. 3.1.6).

5.1.3 Scalability.

We evaluate two scalability aspects. zIO tracking scalability and the impact of page faults.

zIO tracking. We configure the echo server to make 1 application copy of each 512KB IO buffer and vary the number of server threads. Each thread handles a private pool of clients and uses private IO buffers. Figure 8 shows that zIO improves throughput scalability over Linux by up to 19% due to copy elimination. Copies pollute the CPU caches, causing Linux's scalability to be impacted.

Page faults. Page faults can affect scalability when faulting pages are mapped, requiring TLB shootdowns. In theory, information about newly mapped pages may be lazily synchronized among TLBs, avoiding TLB shootdowns. Other cores accessing the same unmapped page simply fault on the stale TLB information, synchronizing the TLB at this moment. Most IO-intensive applications use thread-private IO buffers and access across cores is rare. Unfortunately, Linux does not support lazy mapping of pages. Hence, page faults do affect scalability.

To show this effect, we configure the echo server to access a number of pages of each 512KB IO buffer, without application copies, and vary the number of server threads. We supply the same IO buffer each time, requiring zIO to unmap it for each IO request. The results can be found in Figure 9. We can see that, up to 2 page faults, server throughput still scales well. Increasing the number of page faults per IO beyond this point starts limiting server throughput due to TLB shootdowns. With Linux modifications, many of these TLB shootdowns could be avoided.

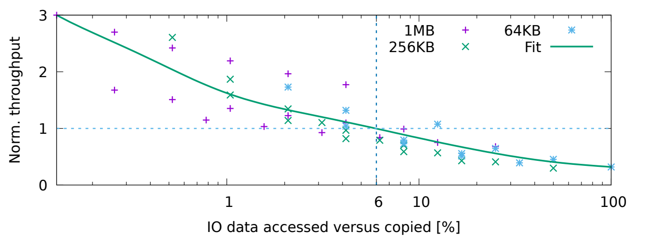

Figure 10. zIO throughput improvement under data access.

5.1.4 Mis-speculation.

To evaluate the impact of zIO mis-speculation on performance, we configure the echo server to access a number of bytes in each IO request before echoing a response. We run this experiment under a variety of IO sizes (64KB, 256KB, and 1MB) and copies (1, 3, and 6), using the Linux network stack.

Figure 10 presents the results as a scatter plot, where we compare zIO throughput improvement over vanilla Linux to the ratio of IO bytes accessed versus elided in copies. This ratio clearly limits zIO's throughput improvements. Applications accessing copied IO data means that zIO mis-speculated. zIO has to resolve the elided copies for the accessed data, which incurs a performance penalty. Less IO data accessed implies better performance improvements. At the same time, more IO data elided in copies also implies better performance improvements and creates headroom for mis-speculation. Fitting a Bezier curve to the scatter plot shows that zIO improves throughput when the ratio of data bytes accessed by an application versus data bytes elided in copies is less than 6%. Above 6%, overheads created by zIO mis-speculation decrease throughput. As an example, for an input buffer of size 200KB that is copied twice, the application may incur up to 6 page faults before output to still yield a speed-up. If the same buffer is copied 4 times, up to 12 page faults are permissible.

5.2 Redis

We evaluate how zIO improves Redis throughput with Linux and kernel-bypass IO stacks (TAS and Strata). Our benchmark setup is identical to the one presented in §2.2. We evaluate two

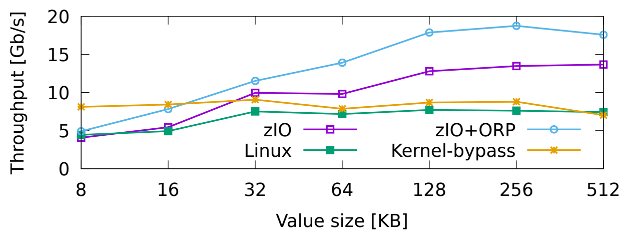

Figure 11. Redis throughput (100% SET).

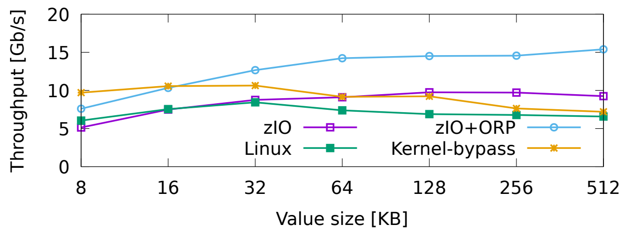

Figure 12. Redis throughput YCSB A (50% SET, 50% GET).

benchmark configurations: 1) 100% SET, and 2) YCSB Workload A, which has a distribution of 50% SETs and 50% GETs. We vary the value size over independent runs for each of these configurations. In addition to zIO's improvement over vanilla Linux with application copy elimination, we investigate the performance of zIO with additional optimistic receiver persistence and IO stack API copy elimination (zIO+ORP) over kernel-bypass IO stacks. The zIO size threshold is disabled for these experiments; enabling it would allow zIO to match vanilla IO stack performance for smaller values, evaluated in §5.2.1.

We first look at 100% SET throughput. This case involves 2 IO copies (one from the network and one to storage), as well as 4 IO application copies per request (cf. Table 1). The results can be found in Figure 11. zIO with Linux eliminates all application copies, which allows for a throughput improvement of up to 1.8×, especially for larger values. zIO+ORP with kernel-bypass IO stacks improves performance by up to 2.5×, as the IO paths consume noticeably less CPU time.

We now look at YCSB workload A, with 50% GET requests and 50% SET requests. These results can be found in Figure 12. As the 50% GET requests require fewer application copies, zIO with Linux provides less of a performance improvement than in the first benchmark, up to 1.3×. However, GET requests provide an opportunity for zIO+ORP to eliminate IO stack API copies, maintaining a speedup of up to 2× over vanilla kernel-bypass.

5.2.1 zIO Performance Breakdown.

We use the Redis 100% SET benchmark to break down the performance contributions of zIO. To do so, we evaluate zIO throughput with kernel-bypass IO in two IO size configurations, progressively

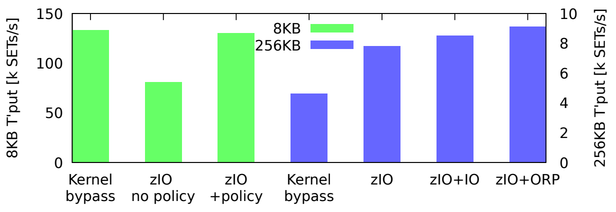

Figure 13. zIO performance breakdown.

	Storage to net **Throughput**	Net to net **Listeners**
Kernel-bypass	0.89 GB/s (1.00×)	812 (1.00×)
zIO+IO	1.08 GB/s (1.25×)	944 (1.16×)

Table 3. Icecast throughput.

enabling different zIO optimizations. These results can be found in Figure 13.

The first configuration uses 8KB SET requests. We evaluate zIO with and without its tracking policy, which applies a 16KB size threshold (§3.1.6). We can see a drastic slowdown of 40% when zIO does not apply this policy, due to the overhead of tracking small IO. Enabling zIO's policy instead copies the IO buffers and attains a negligible slowdown of 2% versus vanilla kernel-bypass.

We further evaluate a configuration with 256KB SET requests. When eliminating application copies, zIO provides a speedup of 1.7×. When adding IO stack API copy elimination, zIO+IO improves performance by another 9%. Adding optimistic receiver persistence in zIO+ORP finally improves performance by another 7%, for a combined improvement over vanilla kernel-bypass of 2×.

Intermediate buffer tracking overhead. We investigate the overhead of buffer tracking via zIO's skiplist. For the same 100% SET request Redis configuration, we find an average of 5 skiplist entries per client connection. With 64 clients, we measured a maximum of 640 entries in the skiplist over the duration of the benchmark. For this scenario, we measure the average skiplist operation latency for lookup and insert to be 190ns. This confirms that intermediate buffer tracking via skiplists is lightweight.

5.3 Icecast

Icecast [37] is an audio broadcasting service. Icecast can stream audio from a source client to a number of listener clients or read data from a local file and serve it to a number of listener clients via HTTP. Table 1 shows that Icecast makes no application copies, but it uses the IO stack APIs. We evaluate both Icecast configurations, providing insight into network to network and storage to network performance. We use the kernel-bypass IO stacks for our evaluation, as they support IO stack API copy elimination. The results are shown in Table 3.

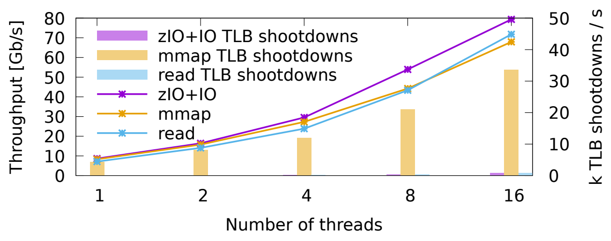

Figure 14. Icecast throughput scalability.

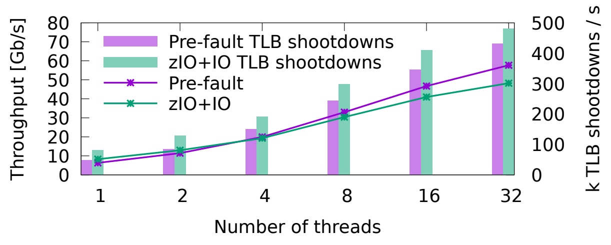

Figure 15. Icecast scalability with pre-faulted buffers.

Storage to network. We configure Icecast to broadcast a 1.1MB audio file to a number of listener clients. We evaluate the amount of audio that a single Icecast file-serving thread can deliver. Icecast reads and sends the audio file in a configurable chunk size, which we set to 64KB. Listener clients request the audio stream via `curl-loader` [16], a HTTP benchmark tool. We connect enough clients to saturate Icecast server throughput and measure throughput over a 30 second period. We can see that zIO+IO improves Icecast maximum throughput by 1.25× by eliminating IO stack API copies, freeing CPU cycles for audio streaming.

Network to network. Next, we evaluate Icecast receiving data from a single client and broadcasting it to a number listeners via a single relay thread. We configure Icecast to relay 64KB at a time and measure the number of concurrent listeners that Icecast can broadcast to. We see a zIO+IO improvement of 1.16× by eliminating IO stack copies. Icecast uses a static buffer for relay, which remains unmapped across IO chunks. This allows zIO+IO to eliminate IO stack copies with minimal overhead.

5.3.1 Scalability. Icecast is a single-threaded application when serving local files to listeners. To evaluate IO-intensive application scalability with zIO, we modify Icecast to create a thread-pool, where each thread can handle listener client HTTP requests from storage via a thread-local IO buffer. This configuration makes Icecast behave like a web server, such as Apache [6]. This version of Icecast is using the `read` system call to read from each file (read).

zIO scalability versus zero-copy IO interfaces. Web servers (like Apache) often use zero-copy IO interfaces to accelerate service, such as memory mapped files and the `sendfile` system call. To compare application performance with a zero-copy IO interface to that of zIO's transparent zero-copy IO, we create a version of Icecast that maps a requested file into memory (cf. §2.4) and sends data to the clients from the memory-mapped file via the socket `send` call (mmap). Memory mapping each requested file eliminates an IO stack copy on input, but also incurs a TLB shootdown. Common usage (cf. Apache) of the mmap API unmaps each file after serving it, incurring another TLB shootdown. zIO+IO can eliminate copies without having to incur TLB shootdowns if buffers are re-used and remain untouched in the common case.

We evaluate these configurations with a 512KB audio file with an increasing number of threads and measure throughput, as well as the number of TLB shootdowns. These results are found in Figure 14. We can see that zIO+IO consistently performs the best, as it does not incur TLB shootdowns. For a small number of threads, memory mapping input files performs similarly to zIO+IO. However, as the number of threads increases, the number and cost of performing TLB shootdowns increases, which negatively affects mmap performance. The number of TLB shootdowns when using `read` and zIO+IO are negligible, as no memory mapping calls happen in the common case. zIO outperforms memory mapping of input files by up to 17%.

Finally, we evaluate versions of Icecast using the Linux `sendfile` API to transmit files to listeners. The first version uses mmap to memory map each file to validate its header before using `sendfile` to transmit it. The second version uses the `read` system call to read the file's header. These versions cannot use the kernel-bypass IO stacks, as `sendfile` is kernel-specific, and `read+sendfile` performs up to 7% worse than zIO+IO, while `mmap+sendfile` performs up to 30% worse than zIO+IO. The scalability trend of `read+sendfile` follows that of zIO+IO, while `mmap+sendfile` scales similarly to mmap.

zIO scalability with pre-faulted buffers. We have already evaluated zIO scalability when buffers are touched, incurring page faults (§5.1.3). zIO can detect these cases and stop copy elision (§3.1.6). However, if the application causes page faults before buffers are tracked by zIO, for example by pre-faulting mapped memory (cf. `MAP_POPULATE` flag for `mmap`) before using it to buffer IO, then zIO can incur TLB shootdowns by unmapping these buffers for tracking.

To evaluate this scenario, we modify Icecast to pre-fault the IO buffer before reading into it via `read` and unmapping it after it was sent over the network (pre-fault). This forces zIO+IO to unmap the IO buffer to track potential access. We run these two configurations with a 512KB audio file, a 32KB chunk size, and an increasing number of threads. We measure throughput and TLB shootdowns for both cases. We present these results in Figure 15. With a small number of threads, zIO+IO outperforms pre-fault, as it still eliminates copies in the IO stack API. However, as the number of threads increases, performance is affected by the additional TLB shootdown overhead and zIO+IO performance degrades. Note that pre-faulting memory causes TLB shootdowns by itself and the scalability of this scenario is already limited.

5.4 MongoDB

We run MongoDB [25] on Linux, with and without zIO. We connect a client over the network running the YCSB [10] load phase and measure request throughput with 1MB values, divided into 10 fields. The YCSB load phase is a workload with 100% inserts of a uniform random distribution. We repeat this benchmark 5 times and report the average throughput for each configuration.

We find that zIO is not able to provide a performance benefit for this workload, with a performance of 191 requests/s compared to 194 for Linux without zIO. zIO is disabling all optimizations due to a large number of page faults. We find that the page faults are generated by MongoDB reading each inserted value in its entirety to calculate a checksum before writing it to the file system.

If we modify MongoDB to skip checksum calculation, zIO is able to eliminate 2 out of 3 application copies (cf. Table 1). Similar to Redis (cf. Table 2), MongoDB copies the inserted value first into an in-memory B-tree (similar to Redis' copy A_2) and then into a log (copy A_3). Finally, MongoDB reallocates the IO buffer, causing a copy, before inserting it into an on-disk index. All three copies are initially elided by zIO, the file system writes complete and their buffers are freed. However, the next IO request re-uses the original IO buffer, forcing zIO to execute the elided copy of the previous buffer to the B-tree data structure. zIO achieves a throughput of 222 requests/s, a 6% improvement over Linux' throughput of 209 requests/s.

We also run MongoDB with the TAS kernel-bypass network stack, allowing us to use zIO+IO to elide an IO stack API copy in `recvmsg` that MongoDB uses to read data from the network. Doing so additionally implies that original buffer reuse, which is now internal to the IO stack and directly communicated to zIO+IO, is lighter weight, as it is not initiated via a page fault. TAS without zIO+IO achieves a throughput of 191 requests/s, while TAS with zIO+IO achieves a throughput of 229 requests/s, a 19% performance improvement.

6 Related Work

In this section, we cover related work beyond the zero-copy IO APIs studied in §2.4.

Zero-copy networked storage. Reflex [19] is a networked storage system designed to provide fast access to remote flash devices. Reflex gains performance by eliminating software copies between network interface cards and flash storage. Unlike zIO, ReFlex does not focus on eliminating application-level or IO stack API copies.

Hardware-accelerated serialization. Recent work has looked at accelerating serialization with help from hardware. Zerializer [35] proposes DMA hardware with data transformation logic to offload serialization. Breakfast of Champions [29] proposes using existing scatter-gather capabilities of NICs to offload serialization. Unlike these works, zIO provides zero-copy without assuming specialized hardware and can eliminate application copies beyond those needed for serialization.

Custom user-level IO stacks. Sandstorm [23] addresses the idea of specially tailoring user-level IO stacks to meet the specific needs of applications to maximize performance, including zero-copy. However, similar to cross-stack APIs, these customizations are not transparent. Either the IO stack has to be modified to work with the application, the application has to be modified to use new APIs, or both. zIO offers transparent cross-stack zero-copy.

7 Conclusion

We present zIO, a transparent zero-copy IO mechanism for unmodified IO-intensive applications. zIO tracks IO data through the application, eliminating copies that are unnecessary while maintaining data consistency. We implement zIO as a user-space library, supporting Linux kernel and kernel-bypass IO stacks. We evaluate zIO with IO-intensive applications, like Redis, Icecast, and MongoDB. zIO improves application throughput by up to 1.8× with Linux, as well as by up to 2.5× with kernel-bypass IO stacks with optimistic network receiver persistence.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Dan Tsafrir, for their helpful comments and feedback. This work was supported by NSF grants 2226057, 2227066, and 2227132.

References

[1] sendfile(2)—linux manual page. https://man7.org/linux/man-pages/man2/sendfile.2.html.

[2] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, September 2014.

[3] https://github.com/tcp-acceleration-service/tas, 2020. Commit d3926baf6ad65211dc724206a8420715eb5ab645.

[4] https://github.com/ut-osa/strata, 2020. Commit f368da4cefe874e1b31a19df7c6436b48f489381.

[5] userfaultfd(2). http://man7.org/linux/man-pages/man2/userfaultfd.2.html, February 2020.

[6] Apache. Apache HTTP Server, 2022. https://httpd.apache.org/.

[7] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 65–77, 2021.

[8] J.S. Chase, A.J. Gallatin, and K.G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[9] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference*, January 1996.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[11] Jonathan Corbet. Zero-copy networking, 2017. https://lwn.net/Articles/726917/.

[12] Jonathan Corbet. Zero-copy TCP receive, 2018. https://lwn.net/Articles/752188/.

[13] Google. Protocol buffers, 2008. https://developers.google.com/protocol-buffers.

[14] Google. gRPC, 2016. https://grpc.io.

[15] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, pages 17–33, 2018.

[16] Robert Iakobashvili and Michael Moser. curl-loader, 2007. http://curl-loader.sourceforge.net/index.html.

[17] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–16, 2019.

[18] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the 14th EuroSys Conference*, pages 1–16, 2019.

[19] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. *SIGARCH Comput. Archit. News*, 45(1):345–359, April 2017.

[20] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[21] Redis Labs. Redis, 2022. https://redis.io/.

[22] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103, 2019.

[23] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, August 2014.

[24] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–262, 2016.

[25] MongoDB. MongoDB, 2022. https://www.mongodb.com/.

[26] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. TensorFlow-Serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS*, 2017.

[27] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, February 2000.

[28] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4):1–30, 2015.

[29] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the 23rd USENIX Conference on Hot Topics in Operating Systems*, pages 199–205, 2021.

[30] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[31] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[32] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: a virtual memory management technique for zero-copy communication. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 308–314, 1998.

[33] Vesoft, Inc. Nebula graph, 2019. https://nebula-graph.io/.

[34] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.

[35] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the 23rd USENIX Conference on Hot Topics in Operating Systems*, pages 206–212, 2021.

[36] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment*, 12(6):624–638, 2019.

[37] xiph. Icecast, 2022. https://icecast.org/.

[38] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.