The Dissertation Committee for Timothy P. Stamler
certifies that this is the approved version of the following dissertation:

Transparently Accelerating IO-Intensive Applications

Committee:

Simon Peter, Supervisor

Vijay Chidambaram

Chris Rossbach

Antoine Kaufmann

Transparently Accelerating IO-Intensive Applications

by

Timothy P. Stamler

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2022

This dissertation is dedicated to my grandmother, Susan Lewis, who passed away from cancer. I know she would be the most proud of me and this work out of anyone.

Acknowledgments

This thesis would not have been possible without a large supporting cast of friends, families, and mentors over the years that have guided me and created an environment in which I could thrive.

I'd first like to thank my advisor Simon Peter for guiding me for all these years through a number of academic endeavors. This dissertation would be absolutely, thoroughly impossible without the time he has spent over the years into instructing me and the research guidance he has provided on all the papers I've been a part of. I could not have asked for more from my graduate school experience and have learned just so much about operating systems, data centers, networks, and so much more all thanks to him. I'd also like to thank my previous teachers and professors that led me to this point. First Rudy Barlow, who originally stoked my interest in computer science. Bhagi Narahari who first introduced me to computer science research. Gabe Parmer and Tim Wood, who showed me the wonders of operating systems, networks, and distributed systems.

I'd next like to thank all of my research collaborators for all of their hard work on the TAS and zIO papers: Antoine Kaufmann, Naveen Sharma, Arvind Krishnamurthy, Thomas Anderson, Deukyeon Hwang, Amanda Raybuck, and Wei Zhang. I'd especially like to thank Antoine for the leadership he showed to

me and his help acclimating to the graduate school experience. Additionally, I know I put Deukyeon, Amanda, and Wei through a lot of trouble over the years and I am deeply appreciative of their help and support.

I'd also like to thank the members of my committee: Vijay Chidambaram, Chris Rossbach, and Antoine Kaufmann. Thank you for the time you've spent helping my form my disseration and all the feedback you have provided.

Through my school years I have met a lot of friends in UT CS: Tyler Hunt, Vance Miller, Jesse Thomason, and John Kallaugher. They have helped my navigate both the logistical and emotional challenges of graduate school. I'd also like to thank my group of friends outside of UT that have supported me through the process: Martin and Caroline Rosas, Victoria Hunt, and Will Plusnick. I thank you all and any others I don't have space to name for keeping me sane through my doctoral degree and the COVID-19 pandemic.

Finally, this would not have been possible without the love and support of my parents, Maggi and Paul. They provided me the structure and education that I needed to learn what I was good at and always supported me in my educational and career pursuits. Through this and sharing their experiences with me and helping me grow as an adult have been absolutely crucial to my graduate school process, even if I haven't always been able to express it to them fully. I would also like to thank my sisters Christa and Grace who have also supported me through the process. I hope that my experience over the last several years can be helpful and guiding to them. Last, and certainly not least, I would not be where I am today without the love and emotional support

of my wife Christina. She has been with me throughout the process, highs and lows, and I would not be the man I am today without her. I love her so much and I'm grateful to have her as a partner for the rest of my life.

Transparently Accelerating IO-Intensive Applications

Timothy P. Stamler, Ph.D.
The University of Texas at Austin, 2022

Supervisor: Simon Peter

As IO bandwidth continues to grow, processor speeds have stagnated. As such, the need to maximize the utility of our CPU cycles for IO-intensive applications is constantly growing. In my dissertation, I investigate processing bottlenecks for IO-intensive applications. I find that small and large IO cause different bottlenecks. Small IO, often caused by remote procedure calls (RPCs), cause IO stack bottlenecks. Large IO, caused by serving content such as web pages, audio, and video, causes bottlenecks due to overhead in IO buffer copying. I aim to accelerate both types of IO request without modification to the application, allowing for easier development and faster deployment. To mitigate these bottlenecks, I propose that IO-intensive applications can be accelerated in two ways, transparently to these applications: 1. by redesigning IO stacks to serve small IO with lower processing overhead via a fast-path that processes common cases of small IO, and 2. by eliminating unnecessary data copies of large IO by interposing on IO buffer copies, tracking modifications, and removing unnecessary copies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With computing continuing to trend towards big data, IO-intensive applications are increasingly important. Applications that run in the data center and applications that convey data across the Internet to data centers require an increasing amount of IO operations to fulfill their tasks, a trend that has been growing from as far back as the early '80s. Whether it is small requests as a result of remote procedure calls (RPCs), or larger content like pictures and videos being conveyed from data centers to clients, data is constantly being moved from application to application.

Although network and storage hardware continue to steadily improve performance [76], CPU speeds have begun to stagnate [77]. IO-intensive applications want to use as much of the IO bandwidth available to them, but find that hard to do without consuming more and more CPU cycles. This is why it is crucial to continue to accelerate the way we perform IO, inside of applications and the system stacks themselves.

Additionally, it is important to accelerate IO-intensive applications *transparently*. Solutions that require application code modification will require many development hours to fully debug and stabilize. Newly written data center applications will be slower to deploy at a large scale if they require

significant modification.

Not all IO requests are created equal, and there is no single solution for accelerating all of them. Handling a small (4 KB) RPC response compared to a request for a large (16 KB) video object trigger very different performance bottlenecks. By analyzing these cases, I find that small IO is more limited by system code execution and large IO is more limited by data movement. My dissertation seeks to analyze these cases and accelerate them transparently with unique solutions.

First, I analyze small IO requests, the most prevalent example of this being RPCs in the data center. Developers want simple semantics, which usually means running RPCs over TCP sockets. My key observation is that when running RPCs over TCP on Linux, packet processing dominates the CPU cycle count. For a key-value store, Linux spends 15.7k CPU cycles per request in TCP packet processing out of 16.8k total, almost 94%. To accelerate RPC-based applications, a TCP stack is needed that drastically reduces per-request overhead.

Second, I analyze serving large IO requests. Again, I study a key value store and find that as IO sizes increase, the relative amount of cycles spent in the IO stack decreases, and we benefit less from efficient IO stack code. Instead, data movement, specifically the copying of data, consumes an increasing number of CPU cycles. These copies can occur in the application and IO stack, and many of them exist for a simple reason: ease of programming. Many common libraries are built on top of interfaces that copy data because copying

data makes it easier to convey ownership of buffers and simplify development.

With this in mind, my dissertation proposes the following thesis statement: IO-intensive applications can be accelerated transparently by redesigning IO stacks with a dedicated fast-path for small IO and by eliminating unnecessary data copies of large IO by interposing on application and IO stack copies, tracking access, and removing unnecessary copies.

I present two solutions to this end. First, TCP acceleration as a service (TAS), a lightweight software TCP network fast-path optimized for common-case client-server RPCs and offered as a transparent user-level OS service to applications. Second, zIO, a transparent zero-copy IO mechanism for large IO-intensive applications.

My dissertation makes the following contributions:

- I present the design and implementation of TAS, a low latency, low-overhead TCP network fast-path. TAS is fully compatible with existing TCP peers and applications.
- I analyze the overheads of TAS and other state-of-the-art TCP stacks in Linux and IX, showing how they use modern processor architecture.
- I present an overhead breakdown of TAS, showing that TAS eliminates the performance and scalability problems of existing TCP stacks, especially for small IO.

- I evaluate TAS on a set of microbenchmarks and common data center server applications, such as a key-value store and a real-time analytics framework. TAS provides up to 57% lower tail latency and 90% better throughput compared to the state-of-the-art IX kernel bypass OS. IX does not provide sockets, which are heavy-weight, but TAS does. TAS still provides 30% higher throughput than IX when TAS provides POSIX sockets. TAS also scales to more TCP connections, providing 2.2x higher throughput than IX with 64K connections.

- An analysis of copying in IO-intensive applications. In particular, I study the number of copies made in popular IO-intensive applications. I follow with a case study of copies in the Redis key-value store, analyzing when and why copies are carried out and their overhead. Finally, using the Redis case study, I demonstrate that copies are a performance bottleneck for IO-intensive applications when leveraging optimized kernel-bypass IO stacks.

- I present zIO, a transparent zero-copy IO system for IO-intensive applications. zIO addresses the presented overheads due to copying. I show how to use zIO to eliminate application-level copies. I show how to eliminate IO stack API copies when combining zIO with kernel-bypass IO stacks. I show how to achieve optimistic input persistence by leveraging non-volatile memory (NVM).

- I 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. I also integrate zIO with the kernel-bypass IO stacks TAS [42] and Strata [44], enabling it to additionally eliminate copies performed by the IO stack APIs.

- I break down zIO's performance contributions with microbenchmarks and analyze the overheads of buffer tracking. I evaluate the performance benefit to IO-intensive applications, like Redis [45], Icecast [82], and MongoDB [16] and compare to Linux and kernel-bypass IO without copy elimination, where zIO improves performance by up to 1.8 and 2.5 , respectively. I 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.

The rest of the document is structured as follows: I first elaborate on some of the background and trends that have led us to more data-driven workloads $x2$. I then discuss in more detail the two bottlenecks ($x2.2.1$ and $x2.2.2$) that were identified in $x1$. Next I discuss the design and evaluation of TAS ($x3$) and zIO ($x4$). Finally I conclude with some related work $x5$ and some ideas for future work in the field($x6$).

Chapter 2

Background

In this section, I will discuss first the rise of IO-intensive applications and how critical they have become for modern computing. On the network side, there is an increasing need for communication between personal devices, base stations, gateways, and especially within data centers. On the storage side, a variety of different types of data, from photographs and videos to large-scale graphs, is being generated and collected each day. This data is transmitted to data center providers, causing more network overhead, and then also must be stored somewhere it can be efficiently accessed.

Additionally, I will discuss the challenges that an increasing amount of IO pose to systems researchers. In particular, I will look at how small-IO is bounded by the software overheads of TCP and how some have tried to address this problem. Next, I will also look at the performance of large-IO intensive applications and quantify the cost of moving large data objects.

2.1 Application IO Intensities are Increasing

In the past couple of decades, we have seen a significant shift in the types of compute tasks that we require for day-to-day life. An increasing

amount of IO throughput available coupled with stagnating CPU speed has led us to rethink how we architect systems to better utilize the resources available to us. It becomes increasingly important to better utilize the CPU cycles that are available to us to keep up with improving IO technologies.

In this section, I will examine how many applications have increased the amount of data they process from IO. I will first look at the networking side and then the storage side.

2.1.1 Increasing Network Speeds and Demands

Network cards and switches continue to push how quickly data can be transmitted. Google has given us a look into how increasingly fast network hardware has translated to increased traffic in their data centers [76]. Through 2014, they described a 100x increase in network traffic within their data centers over the previous decade.

In the paper, Google discusses a number of techniques they used to re-architect their data centers to handle this increase in network traffic. They look beyond the scope of typical computer networks and instead implement Clos topologies within their datacenters. They shift from expensive network technologies and opt instead for an abundance of cheaper, commodity network cards and switches. They rewrite their network control protocols to be more centralized and controllable. These decisions all stem from an exponentially increasing amount of network traffic and an expectation that this trend will continue.

Where is all of this network traffic going? With an abundance of network bandwidth at their disposal, data center application developers have shifted the way they develop software to be more reliant on microservices [26]. Applications started to be broken down from large monolithic code bases to potentially hundreds of small modules, deployed throughout the data center, and communicating with remote procedure calls. These remote procedure calls became the backbone for many applications and services within the data center.

One popular example of this kind of application is memcached [52]. Memcached is a simple, popular in-memory key-value store that was adopted and brought to data center scale by Facebook [56] (now Meta). They found it to be a convenient caching solution for objects in their data center that may or more not originate in storage. Of course, using memcached to cache objects may generate more network traffic whenever the cached data is accessed.

2.1.2 More Data Needs to be Persisted

In addition to increasing network traffic and the requirements that those impose, there is an increasing demand for data to be stored persistently. Naturally, as more people become connected to the Internet and make technology a larger portion of their lives, more data will be generated and collected by application providers and data center operators.

Just this year, Meta published some data about the amount of persistent data they handle and some techniques that they use to deliver this data

in their Owl system [25]. In this paper, they describe how now up to 800 petabytes of data is delivered from storage to clients every day.

While Owl describes how Meta handles large objects, a variety of small objects are still quite common as well, stemming again from microservices. Persistent object stores like LevelDB [29] and RocksDB [22] have become prevalent at their respective companies, Google and Facebook. RocksDB in particular has been further optimized for faster flash storage and distributed applications.

2.2 IO Processing Bottlenecks

As previously stated, both network and storage IO can be classified into small and large IO operations. These different operations have different characteristics and thus different processing bottlenecks that we can explore and optimize for. In the following sections, I will discuss how I have investigated and quantified these bottlenecks.

2.2.1 Small IO and IO Stack Overheads

As I have discussed, many data center applications are built on top of remote procedure calls (RPCs). These RPCs take the form of many small IO operations, particularly on the network side. Microservices constantly send requests and responses around the data center, all going through the network stack, and all requiring low latency and high throughput.

As network speeds rise, while CPU speeds stay stagnant, TCP packet processing efficiency is becoming ever more important. At the same time, they

rely on the lossless, in-order delivery properties provided by TCP. To provide this convenience, software TCP stacks consume an increasing fraction of CPU resources to process network packets.

TCP processing overheads have been known for decades. In 1993, Van Jacobson presented an implementation of TCP common-case receive processing within 30 processor instructions [36]. Common network stacks, such as Linux's, still use Van's performance improvements [1]. Despite these optimizations, a lot of CPU time goes into packet processing and TCP stack processing latencies are high.

I investigate TCP packet processing overhead in the context of modern processor architecture. I find that existing TCP stacks introduce overhead in various ways (and to varying degree): 1. By running in privileged mode on the same processor as the application, they induce system call overhead and pollute the application-shared L1, L2, and translation caches. 2. They spread per-connection state over several cache lines, causing false sharing and reducing cache efficiency; 3. They share state over all processor cores in the machine, resulting in cache coherence and locking overheads; 4. They execute the entire TCP state machine to completion for each packet, resulting in code with many branches that do not make efficient use of batching and prefetching opportunities.

To demonstrate the inefficiencies of kernel TCP stacks, I quantify the overheads of the Linux TCP stack. To do so, I instrument it using hardware performance counters, running a simple key-value store server benchmark. The

Module	Linux kc	%
Driver	1.82	9%
IP	2.63	13%
TCP	5.32	26%
Sockets	7.79	39%
Other	1.04	5%
App	1.53	8%
Total	20.14	100%

Table 2.1: CPU cycles per request

Counter	Linux
CPU cycles	1.5k/18.6k
Branch mispredicts	3/32
I-cache misses	48/646
DTLB misses	2/65
Cycles stalled on L1	0.5k/2.9k

Table 2.2: Per request user/stack overheads.

benchmark server serves 512 concurrent connections from several client machines that saturate the server network bandwidth with 64B key and 32B value requests for a small working set, half the size of the server's L3 cache (details of our experimental setup in *x*3.3). I execute the experiment for two minutes and measure for one minute after warming up for 30 seconds.

2.2.1.1 Linux Overheads

Table 2.1 shows the result. I find that Linux executes 20.14 kilocycles (kc) for an average request, of which only 1.53 kc (7%) are spent within the application, while the rest (93%) are spent within the Linux kernel. 87% of total cycles are spent in the network stack. For each request, Linux executes 12.5 kilo-instructions (ki), resulting in about 1.6 cycles per instruction (CPI), 6.4 above the ideal 0.25 CPI for the server's 4-way issue processor architecture. This results in high average per-request processing latency of 9 s. The reason for these inefficiencies is the computational complexity and high memory

footprint of a monolithic, in-kernel stack. Per-request privilege mode switches and software interrupts stall processor pipelines; software multiplexing, cross-module procedure calls, and security checks require additional instructions; large, scattered per-connection state increases memory footprint and causes stalls on cache and TLB misses; shared, scattered global state causes stalls on locks and cache coherence, inflated by coarse lock granularity and false sharing; covering all TCP packet processing cases in one monolithic block causes the code to have many branches, increasing instruction cache footprint and branch mispredictions.

I measure these inefficiencies with CPU performance counters. The results are shown in Table 2.2 and indicate the miss events per request. Each request spends about 3.4 kc on L1 data cache misses, incurs about 700 instruction cache misses, where an individual miss on the critical path takes at least 12 cycles to resolve, and also incurs 70 TLB misses at a cost of at least 14 cycles each.

As we can see, the Linux TCP stack is built for ease of use and compatability and not for processor efficiency. Especially in cases with many client connections and small message sizes, the overheads of handling TCP processing in the kernel are quite high.

2.2.1.2 Bypassing Linux Limits Transparency

Linux has overheads for TCP packet processing, so the simplest solution to improve TCP packet processing is to bypass Linux and allow some alternate

system to interact with the network instead.

The idea for giving access to the network to user space dates back to U-Net [78]. U-Net exposed typical Linux IO communications devices to user space in order for custom protocols to be implemented. The goal was to accelerate high performance computing clusters, but they do claim that efficient standard networking protocols like UDP and TCP can be implemented in such a way. U-Net implements a secure control plane and multiplexer in the kernel and avoids invoking the kernel in the critical path.

Arsenic [64] took kernel bypass solutions a step further by providing new hardware. that allows for the multiplexing of network interfaces in user space. This implementation allowed for a more general solution, where multiple applications could be given direct access to networking interfaces. The network card itself is then in charge of receiving packet filtering rules from the kernel to provide security.

The previous systems introduced the idea of kernel bypass, but the need for optimizing IO intensive applications became more pressing later on as CPU cycles became more limited. The Arrakis operating system [63] explored the effects of kernel network and storage processing on common data center applications. By separating control plane and common case data plane processing, and by leaving the control plane to the kernel and data plane in user space, Arrakis was able to attain much higher performance.

Moving TCP functionality to user space can reduce overheads, but some

work has also demonstrated how it can improve scalability. mTCP [39] seeks to move some socket and TCP functionality to a user space library in order to make packet processing more scalable to multiple cores. They achieve this by scheduling flows to cores and designing data structures to be more localized and cache-friendly. Additionally, to improve performance further, they batch packet operations and system call operations to their minimized kernel driver.

Limitations of kernel bypass. There is a security concern with kernel bypass solutions that allowing user space applications to have access to the networking stack and network driver libraries. This makes it more difficult to provide both privacy as well as performance isolation, as applications can choose to directly send or receive packets from the network card at any time. IX [12] seeks to address these concerns by taking a similar approach to Arrakis - splitting a common case data plane from the full control plane - but still running the network stack in a privileged CPU mode. This does incur some overheads for entering and exiting the privileged mode, but provides better protection with a streamlined networking stack.

Additionally, limitations can come in the form of inconvenient interfaces for applications, requiring application modification. This increases developer overhead and hinders widespread adoption. Solutions like mTCP seek to minimize the overhead of application modification but still can't provide a fully transparent interface.

Finally, some solutions still have performance limitations for entering

into and out of privileged modes. Whether via the kernel or a different privilege ring, as seen in IX, many systems still want to use hardware protection to prevent giving applications full access to network hardware. These overheads are usually alleviated with batching, but this can create tail latency concerns.

2.2.1.3 Conclusions

As I have discussed in this section, there is intrinsically overhead for handling the full TCP stack inside the kernel. Kernel bypass offers an intriguing solution for this, but introduces a new set of problems. Later in this dissertation (*x*3), I will describe how TAS can provide a kernel bypass TCP solution that provides high performance by fully excluding the kernel from the fast path without exposing the network to applications, all fully transparent to the application.

2.2.2 Large IO and The Cost of Using `memcpy`

Large IO has significantly different bottlenecks than previously discussed. While there are still TCP overheads, many more cycles are instead being spent elsewhere. I find that a large number of these cycles are spent in copying and moving data, both inside the application and the IO stack itself.

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 [59, 63] and

Application	Function	Copy call site App	Copy call site IO Stack
Redis [45]	SET	4	2
	GET	2	1
Icecast [82]	Broadcast to N listeners	0	1 + N
Ceph [79]	Write	1	2
	Read	0	2
Anna [81]	PUT	5	3
	GET	4	3
MongoDB [16]	Insert	3	2
	Disk sync	1	1
	Read	2	2
Tensor ow-serving [58]	Inference	2	1
Nebula Graph [34]	Insert vertex	5	2
	Store a vertex	4	3

Table 2.3: Number and call site of copies between input and output for various application functions.

zero-copy IO interfaces have also been developed (e.g., [3, 19, 20, 31, 40, 63]) to remove copies from APIs.

Despite these advances, data from IO is still copied. To show this, I study the prevalence of IO data copies in popular IO-intensive applications. I identify the call site of these copies and break down occurrences by whether copies are involved in an IO stack API or within application subsystems. My methodology involves a source code analysis of IO data flows through application subsystems from input to output. I identify what methods applications use to copy IO data and how copies are affected by the executed functionality and its parameters. I find that all applications investigated use C standard library functions, such as `memcpy` and `memmove`, to copy data. I use this insight to validate our source code analysis via an execution of the relevant application functions under a debugger set to break on these C library memory copy

APIs. For each application execution, I count the number of breakpoints hit on IO code paths between input and output and check that the count matches our source code observations.

Table 2.3 presents the number of copies made at the IO stack and within various IO-intensive applications, broken down by IO function. I am specifically interested in the copy of potentially large IO data, as small data copies do not significantly affect application performance. For example, the Anna [81] key-value store conducts up to 45 copies of keys during a PUT operation. I ignore these copies in the table.

While the number of copies varies between applications and functions, we can see that IO-intensive applications extensively copy data while processing it between input and output. We can also see that applications often make more internal copies of the data, rather than at the IO stack API. Redis [45] makes twice as many application-internal copies than at the IO stack for a SET request.

IO-intensive applications, like Anna [81], also often employ third-party libraries, including gRPC [28] and Protobuf [27], to serialize and deserialize data. I 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.

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, allowing for subsystem-internal use of the data without worry of corruption or deallocation of the memory backing the data from outside the subsystem. As an example, copies are used to simplify asynchronous IO. In POSIX systems, kernel IO stacks provide internal buffers to IO devices that are operated asynchronously from application execution. Applications request and copy data into local 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, such as alignment, padding, serialization and deserialization, and bucketization.

2.2.2.1 Copy overheads.

Unfortunately, copying is an imperfect tool. While copies provide the aforementioned benefits, they also introduce overhead. Using the Redis [45] key-value store as an example IO-intensive application, I study the overhead of copies for IO both using Linux kernel IO stacks and using kernel-bypass IO stacks for high-bandwidth IO devices. Copying overhead scales with the amount of the copied data. As IO devices, in particular for storage and networking, increase bandwidth, copies become the performance-limiting factor in IO-intensive applications.

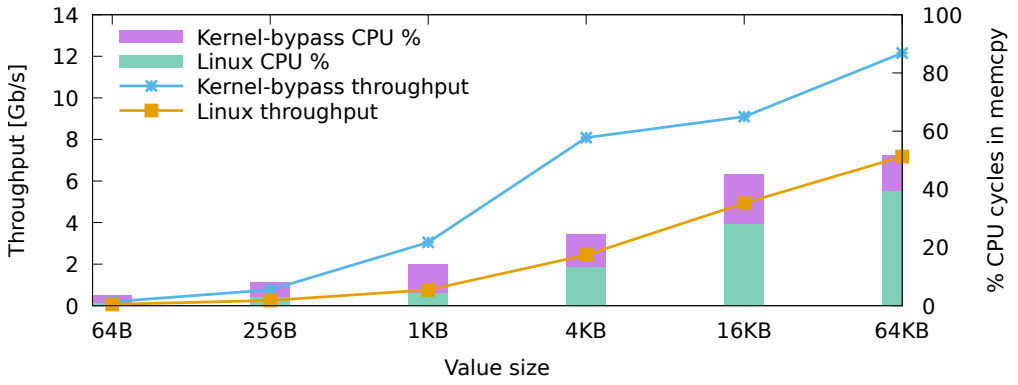

Figure 2.1: Redis SET throughput and fraction of CPU cycles in `memcpy` over value size, with and without kernel-bypass.

The results are presented in Figure 2.1. 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 be able to keep up with the demand and provide the necessary performance.

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 values per 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 heavy-weight Linux IO. 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%.

2.2.2.2 Existing Zero Copy APIs Limit Transparency

A limited solution proposed by SocksDirect [46] and also implemented in FreeBSD [14] and Solaris [17] 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. Further, transmit buffers must be kept until acknowledged, leaking memory if acknowledgments lag. The limited applicability, security concerns (including from malicious NICs [50]), and hardware requirements led the FreeBSD developers to abandon the transparent zero-copy socket API in FreeBSD 11.

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) [4]. 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.

More recently, netmap [71] is a zero-copy networking API within Linux to provide applications direct access to packet buffers. Netmap allows the kernel to read packet buffers into memory that is shared with the application and let the application choose whether or not these packets are run through the Linux TCP stack. Linux is still in charge of control plane operations, and the Linux TCP stack is still aware of data packets, but the application maintains the option to read packet data directly from the original buffers that are read from the NIC, eliminating the need to copy them. This solution requires significant application modification to be able to read and process packets directly from NIC buffers and there are typical security and performance concerns with making NIC interactions available to the application.

Limitations of alternative APIs Many of the zero-copy API solutions I have previously discussed have a common trend: they all place restrictions on how the application can interact with the IO stack. Whether it's modifying the API or placing stringent rules on what IO data buffers must look like, the previous systems had limited success. Additionally, none of these solutions have sought to eliminate copies within the applications, which I have demonstrated to be a significant source of overhead.

2.3 Summary

In this section, I have shown the importance of IO-intensive applications and some of the challenges we must overcome to accelerate them further. We have seen how other systems have tried to address these challenges and in what areas they have fallen short. In the following chapters, I will address these shortcomings and propose entirely new systems for both fast software TCP processing and fast data movement for large IO objects.

Chapter 3

TAS: TCP Acceleration as an OS Service

In this section, I will outline the design, implementation, and evaluation of TAS. I will demonstrate that TCP can be accelerated in software, in user space, and with full protection.

3.1 Streamlining TCP Functionality

In days past, at the time of TCP's origin as a computationally efficient transport protocol, TCP congestion control was designed to avoid the use of integer multiplication and division [35]. Although TCP as a whole has become quite complex with many moving parts, the common case data path remains relatively simple. For example, packets sent within the data center are never fragmented at the IP layer, packets are almost always delivered reliably and in order, and timeouts almost never fire [6]. Can we use this insight to eliminate the existing overheads?

With this in mind, and seeking to further optimize small TCP IO operations, I design TCP acceleration as a service (TAS), a lightweight software TCP network *fast-path* optimized for common-case client-server RPCs and offered as a separate OS service to applications. TAS interoperates with legacy

Linux TCP endpoints and can support a variety of congestion control protocols that are common in data centers, including TIMELY [53] and DCTCP [6].

Separating the TCP fast-path from a monolithic OS kernel and offering it as a separate OS service enables a number of streamlining opportunities. Like Van Jacobson, I realize that TCP packet processing can be separated into a common and an uncommon case. TAS implements the fast-path that handles common-case TCP packet processing and resource enforcement. A heavier stack (the *slow path*), in concert with the rest of the OS, processes less common duties, such as connection setup/teardown, congestion control, and timeouts. The TAS fast-path executes on a set of dedicated CPUs, holding the minimum state necessary for common-case packet processing in processor caches. While congestion control policy is implemented in the slow path, it is enforced by the fast path, allowing precise control over the allocation of network resources among competing flows by a trusted control plane. The fast path takes packets directly from (and directly delivers packets to) user-level packet queues. Unprivileged application library code implements the POSIX socket abstraction on top of these fast path queues, allowing TAS to operate transparent to applications.

Beyond streamlining, another benefit of separating TAS from the rest of the system is the opportunity to scale TAS independent of the applications using it. Current TCP stacks run in the context of the application threads using them, sharing the same CPUs. Network-intensive applications often spend more CPU cycles in the TCP stack than in the application. When

sharing CPUs, non-scalable applications limit TCP processing scalability, even if the TCP stack is perfectly scalable. Separation not only isolates TAS from cache and TLB pollution of the applications using it, but also allows TAS to scale independently of these applications.

3.2 TAS System Design

In this section, I describe the design and evaluation of TAS, a network stack optimized for small IO requests over TCP. I have already shown some of the overheads of the Linux TCP stack. In the following sections, I will demonstrate how I can address these overheads and provide fast, transparent access to TAS sockets with TAS with the following design principles:

- **E ciency:** Data center network bandwidth growth continues to outpace processor performance. TAS must deliver CPU efficient packet processing, especially for latency-sensitive small packet communication that is the common case behavior for data center applications and services.
- **Connection scalability:** As applications and services scale up to larger numbers of servers inside the data center, *incast*, where a single server handles a large number of connections, continues to grow. TAS must support this increasing number of connections.
- **Performance predictability:** Another consequence of this scale is that predictable performance is becoming as important as high common case performance for many applications. In large-scale systems, individual user requests can access thousands of backend servers [37, 56] causing one-in-a-thousand

request performance to determine common case performance.

- **Policy compliance:** Applications from different tenants must be prevented from intercepting and interfering with network communication from other tenants. Thus, TAS must be able to enforce policies such as bandwidth limits, memory isolation, firewalls, and congestion control.
- **Workload proportionality:** TAS should not use more CPUs than necessary to provide the required throughput for the application workloads running on the server. This requires TAS to scale its CPU usage up and down, depending on demand.

TAS has three components: Fast path, slow path, and untrusted per-application user-space stack. All components are connected via a series of shared memory queues, optimized for cache-efficient message passing [10]. The fast path is responsible for handling common case packet exchanges. It deposits valid received packet payload directly in user-space memory. On the send path, it fetches and encapsulates payload from user memory according to per-connection *rate or window limits* that are dynamically configured by the slow path. User-level TCP stacks provide the standard POSIX API to applications. Applications do not need to be modified, only (dynamically) relinked. For connection setup and teardown, each user-level stack interacts with the slow path. I describe each component in detail in this section.

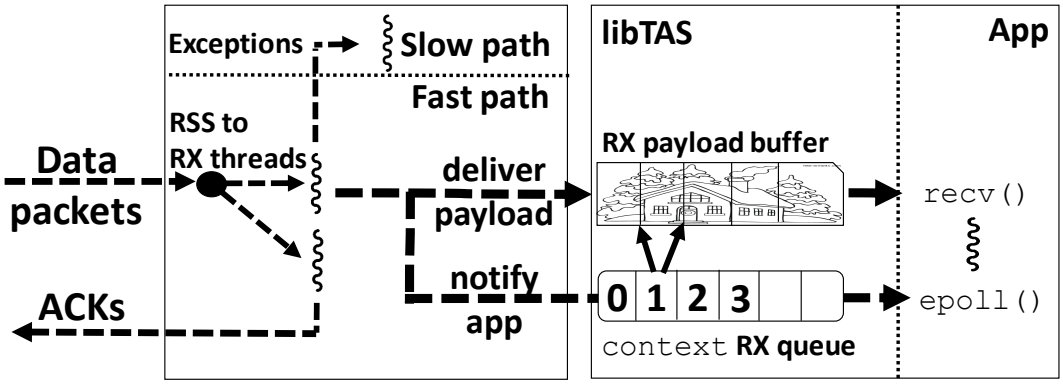

Figure 3.1: TAS receive flow.

3.2.1 Fast Path

To streamline RPC processing and improve performance, the fast path handles the minimum functionality required for the common-case exchange of RPC packets between untrusted user applications and the network in a data center. To do so, it processes protocol headers, sends TCP acknowledgements, enforces network congestion policy, and performs payload segmentation. To be fully functional, it must also detect exceptions, such as out-of-order arrivals, connection control events, and unknown or unusual packets (such as unusual IP and TCP header options). Data center applications typically pre-establish all required connections. Hence, the TAS fast path handles connection establishment and teardown as exceptions. Exceptions are generally forwarded to the slow path (but see *exceptions*, below).

Common-case receive (Figure 3.1). The TAS fast path receives TCP packets from the NIC on a dynamic number (cf. Section 3.2.4) of *receive threads* via

the NIC's receive-side scaling (RSS) functionality. TAS assumes that packets are commonly delivered in order by the network. This is true for data center networks today due to connection-stable multi-path routing [76]. With in order packets, the fast path can discard all network headers and directly insert the payload into a user-level, per-flow, circular *receive payload buffer* (identified by `rx_start|size` fields in per-flow state, with `rx_head|tail` identifying write/read positions), notifying an appropriate *receive context queue* (`context` field in per-flow state) by identifying the connection and number of bytes received. The application receives notifications when polling for them (e.g., via `epoll()`) and copies received data out of the payload buffer via socket calls (e.g., `recv()`).

When a payload buffer is full, the fast path simply drops the packet—an uncommon situation, as TCP controls the flow of packets into payload buffers via per-connection window size. If a context queue is full, the fast path will inform the user-level stack upon future packet arrivals when the queue is available again—context queues only fill when payload is queued at an application, providing that the application has work to do and will check the context queue soon. User-level stacks are free to define and configure contexts. I describe them in more detail in Section 3.2.3. Per-flow payload receive buffers simplify packet handling, flow control, and improve isolation. Calculating an accurate flow control window with shared buffers requires iteration over all connections sharing the buffer, imposing non-constant per-packet overhead. To avoid this overhead, I opt for per-flow payload buffers.

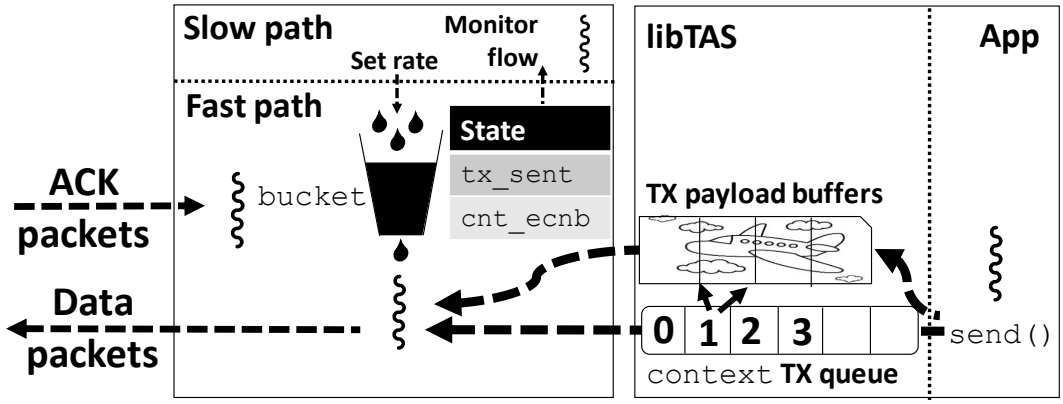

Figure 3.2: TAS transmit flow.

After depositing the payload of an in-order packet, the fast path automatically generates an acknowledgement packet and transmits it to the sender to update its TCP window. Handling TCP acknowledgements in the fast path is important for security. If user-space was given control over acknowledgements, as in many kernel-bypass solutions, it can use it to defeat TCP congestion control [73]. Fast path acknowledgements also provide correct explicit congestion notification (ECN) feedback, and accurate TCP timestamps for round-trip time (RTT) estimation. Finally, the fast path updates local per-connection state (e.g., `seq`, `ack`, and `window` fields).

Common-case send (Figure 3.2). User-level stacks send data on a flow by appending it to a flow's circular transmit buffer (e.g., when invoked by `send()`). Per-flow send buffers are required to alleviate head-of-line blocking under congestion and flow control. To inform the fast path, the stack issues a TX command on a context queue and wakes a waiting fast path thread. The fast

path fills a per-flow bucket (`bucket` field in per-flow state) with the amount of new data to send. Asynchronously, the fast path drains these buckets, depending on a slow path configured per-connection rate-limit or send window size and the receiver's TCP window, to enforce congestion and flow control. When data can be sent, the fast path fetches the appropriate amount from the transmit buffer, produces TCP segments and packet headers for the connection, and transmits. The fast path also uses TCP timestamps to provide the slow path with an accurate RTT estimate for congestion control and timeouts (cf. Section 3.2.2), among other relevant flow statistics (e.g., `tx_sent` and `cnt_ecnb` fields).

Transmit payload buffer space reclamation. Any payload that has been sent remains in the transmit buffer until acknowledged by the receiver. The fast path parses incoming acknowledgements, updates per-flow sequence and window state, frees acknowledged transmit payload buffer space, and informs user-space of reliably delivered packets by issuing a notification with the number of transmitted bytes for the corresponding flow on an RX context queue (not shown in figures). This requires constant time.

Per-flow state. To carry out its tasks, the fast path requires the per-flow state shown in Table 3.1. The opaque field is specified by and relayed to user-space to help it identify the corresponding connection. RX/TX buffer state is used for management of per-flow buffers in user-space. The slow path has access

Field	Bits	Description
opaque	64	Application-de ned ow identi er
context	16	RX/TX context queue number
bucket	24	Rate bucket number
rx|tx_start	128	RX/TX bu er start
rx|tx_size	64	RX/TX bu er size
rx|tx_head|tail	128	RX/TX bu er head/tail position
tx_sent	32	Sent bytes from tx_head
seq	32	Local TCP sequence number
ack	32	Peer TCP sequence number
window	16	Remote TCP receive window
dupack_cnt	4	Duplicate ACK count
local_port	16	Local port number
peer_ip|port|mac	96	Peer 3-tuple (for segmentation)
ooo_start|len	64	Out-of-order interval
cnt_ackb|ecnb	64	ACK'd and ECN marked bytes
cnt_frexmits	8	Fast re-transmits triggered count
rtt_est	32	RTT estimate

Table 3.1: Required per-flow fast path state (102 bytes).

to all fast path state via shared memory. In all, TAS requires 102 bytes of per-flow state. Current commodity server CPUs supply about 2MB of L2/3 data cache per core. This allows TAS to keep the state of more than 20,000 active flows per core in the fast path.

Exceptions (Figure 3.1). Unidentified connections, corrupted packets, out-of-order arrivals, and packets with unhandled flags, are exceptions. Exception packets are filtered and sent to the slow path for processing. The fast path detects out-of-order arrivals by matching arrivals against expected sequence numbers in the per-flow `seq` field. As an optimization, TAS handles two exceptions on the fast path:

1. When processing incoming acknowledgements the fast path counts duplicates and triggers fast recovery after three duplicates, by resetting the sender state as if those segments had not been sent. The fast path also increments a per-flow retransmit counter to inform the slow path to reduce the flow's rate limit.

2. The fast path tracks one interval of out-of-order data in the receive buffer (starting at `ooo_start` and of length `ooo_len`). The fast path accepts out-of-order segments of the same interval if they fit in the receive buffer. In that case, the fast path writes the payload to the corresponding position in the receive buffer. When an in-order segment fills the gap between existing stream and interval, the fast path notifies the user-level stack as if one big segment arrived, and resets its out-of-order state. Other out-of-order arrivals are dropped and the fast path generates an acknowledgement specifying the next expected sequence number to trigger fast retransmission.

3.2.2 Slow Path

The slow path implements all policy decisions and management mechanisms that have non-constant per packet overhead or are too expensive or stateful to process in the fast path. This includes congestion control policy, connection control, a user-space TCP stack registry, timeouts and other exceptional situations.

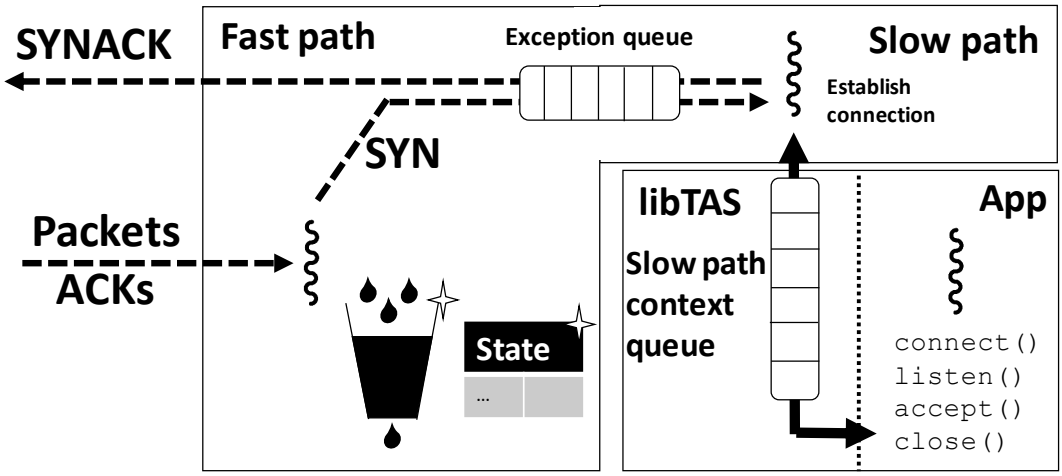

Figure 3.3: TAS slow path connection control.

Congestion control (Figure 3.2). TAS supports both rate and window-based congestion control. The fast path runs a control loop iteration for each flow every control interval (by default every 2 RTTs). The slow path retrieves per-connection congestion feedback from the fast path (`cnt_ackb|ecnb`, `cnt_frexmits`, and `rtt_est` fields), then runs a congestion control algorithm to calculate a new flow send rate or window size, and finally updates this information in the fast path via shared memory.

This provides a generic framework to implement different congestion control algorithms. I implement DCTCP [6] and TIMELY [53] (adapted for TCP by adding slow-start). I adapt DCTCP to operate on rates instead of windows by applying its control law (rate-decrease proportional to fraction of ECN marked bytes) to flow rates: during slow start TAS doubles the rate every control interval until there is an indication of congestion, and during additive increase TAS adds a configurable step size (10mbps by default) to the current

rate. To prevent rates from growing arbitrarily in the absence of congestion, TAS ensures at the beginning of the control loop that the rate is no more than 20% higher than the flow's send rate.

The choice for rate-based DCTCP for our prototype is deliberate. Rate-based congestion control is more stable with many flows. It smoothes bursts that otherwise occur due to abrupt changes in congestion window size and thus provides a fairer allocation of bandwidth among flows. The TAS rate-based DCTCP implementation is compatible with Linux peers.

Connection control (Figure 3.3). Connection control is complex. It includes port allocation, negotiation of TCP options, maintaining ARP tables, and IP routing. TAS thus handles it in the slow path. User-level TCP stacks issue a `new_flow` command on the slow path context queue to locally request new connections (triggered by a `connect()` call). If granted, the slow path establishes the connection by executing the TCP handshake and, if successful, installs the established flow's state in the fast path and allocates a rate/window bucket. Remote connection control requests are detected by the fast path and forwarded to the slow path, which then completes the handshake.

Servers can listen on a port by issuing a `listen` command to the slow path (triggered by `listen()` socket call). Incoming packets with a SYN flag are forwarded as exceptions to the slow path. The slow path informs user-space of incoming connections on registered ports by posting a notification in the slow path context queue. If the application decides to accept the connection (via

`accept()`), its TCP stack may issue the `accept` command to the slow path (via the slow path context queue), upon which the slow path establishes the flow by allocating flow state and bucket, and sending a SYNACK packet. To tear down a connection (e.g., upon `close()`), user-space issues `close`, upon which the slow path executes the appropriate handshake and removes the flow state from the fast path. Similarly, for remote teardowns, the slow path informs user-space via a `close` command.

Retransmission timeouts. TAS handles retransmission timeouts in the slow path. When collecting congestion statistics for a flow from the fast path, the kernel also checks for unacknowledged data (i.e. `tx_sent > 0`). If a flow has unacknowledged data with a constant sequence number for multiple control intervals (2 by default) the slow path instructs the fast path to start retransmitting by adding a command to the slow path context queue. In response to this command the fast path will reset the flow and start transmitting exactly as described above for fast retransmits.

TCP stack management. To associate new user-space TCP stacks with TAS, the slow path has to be informed via a special system call. If the request is granted, the slow path creates an initial pair of context queues that the user-space stack uses to create connection buffers, etc.

3.2.3 User-space TCP Stack

The user-space TCP stack presents the programming interface to the application. The default interface is POSIX sockets so applications can remain unmodified, but per-application modifications and extensions are possible, as the interface is at user-level [12, 49, 62]. For example, TAS also offers a low-level API that is similar to the IX networking API. The low-level API directly passes events from the context RX queue to the application and offers functions to add entries to the context TX queue. The TCP stack is responsible for managing connections and contexts. To fulfill our performance goal, common-case overhead of the TCP stack has to be minimal.

Context management. User-space stacks are responsible for defining and allocating contexts. Contexts are useful in various ways, but typically stacks allocate one context per application thread for scalability, as it allows cores to poll (e.g., `epoll()`) only a private context queue, rather than a number of shared payload buffers. Stacks allocate contexts via management commands to the slow path.

3.2.4 Workload Proportionality

TAS executes protocol processing on dedicated processor cores, and the number of cores needed depends on the workload. As a result, TAS has to dynamically adapt the number of processor cores used for processing to be proportional with the current system load. I implement this with three

separate mechanisms. On the fast path, I use hardware and software packet steering to direct packets to the correct cores, while the slow path monitors the CPU utilization and, as needed, adjusts steering tables to add or remove cores. Finally, the fast path blocks threads when no packets are received for a period of time (10 ms in our implementation). These cores can be woken up via kernel notifications (eventfd). This requires TAS to carefully handle packet re-assignments among queues during scale up/down events.

Fast path. When initializing, TAS creates threads for the configured maximum number of cores and assigns NIC queues and application queues for all cores. Because of the adaptive polling with notifications, cores that do not receive any packets automatically block and are de-scheduled.

I design the data path to handle packets arriving on the wrong TAS core, either from the NIC or the application, with a per-connection spinlock protecting the connection state. This avoids the need for expensive coordination and draining queues when adding or removing cores. Instead TAS simply asynchronously updates NIC and application packet steering to route packets to or away from a specific core. TAS eagerly updates the NIC RSS redirection table to steer incoming packets, and lazily update the routing for outgoing packets from applications. This allows it to be robust during scale up/down events.

Slow path. The slow path is responsible to decide when to add or remove cores by monitoring fast path CPU utilization. If it detects that in aggregate more than 1.25 cores are idle, it initiates the removal of a core. If, on the other hand, less than 0.2 cores are idle in aggregate, it adds a core. The specific thresholds are configuration parameters.

3.3 Evaluation

My evaluation seeks to answer the following questions:

How does TAS' throughput, latency, and connection scalability for remote procedure call operation compare to state-of-the-art software solutions in the common case? How in the case of short-lived connections? (§3.3.1)

Does the simplified fast-path TCP operation negatively affect performance under packet loss or congestion? (§3.3.2)

Do these improvements result in better end-to-end throughput and latency for data center applications? How do these workloads scale with the number of CPU cores? (§3.3.3, 3.3.4)

How does TAS perform at scale? Does the split of labor into slow and fast path affect congestion control fidelity with many connections and various round-trip times to remote machines? (§3.3.5)

Is TAS consuming CPU resources proportional to its workload? What is the impact on network throughput and latency when TAS changes its CPU resource use? (§3.3.6)

To answer these questions I first evaluate RPC performance on a number of systems using microbenchmarks. I then evaluate two data center application workloads: a typical, read-heavy, key-value store application and a real-time analytics framework. Finally, I validate our results at scale with an ns-3 simulation.

Testbed cluster. The evaluation cluster contains a 24-core (48 hyperthreads) Intel Xeon Platinum 8160 (Skylake) system at 2.1 GHz with 196 GB RAM, 33 MB aggregate cache, and an Intel XL710 40Gb Ethernet adapter. I use this system as *the server*. There are also six 6-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2 GHz with 18MB aggregate cache, which I use as clients. These systems have Intel X520 (82599-based) dual-port 10Gb Ethernet adapters with both ports connected to the switch. I run Ubuntu Linux 16.04 (kernel version 4.15) with DCTCP congestion control on all machines. I use an Arista 7050S-64 Ethernet switch, set up for DCTCP-style ECN marking at a threshold of 65 packets. The switch has 10G ports (connected to the clients) and 40G ports (connected to the server).

Baseline. I compare TAS performance to the Linux monolithic, in-kernel TCP stack (using `epoll`), to the mTCP kernel-bypass TCP stack [39], and to the IX protected kernel-bypass TCP stack [12]. Unless stated, our benchmarks do not mix peer systems. mTCP and IX do not provide the standard sockets API, requiring significant application modification [12, 39]. Unless stated, I use the same application binary for TAS and Linux.

Sender		Linux	TAS
Receiver	Linux	9.4Gbps	9.4Gbps
	TAS	9.4Gbps	9.4Gbps

Table 3.2: Compatibility between Linux and TAS: 100 bulk transfer flows from 1 sending machine to 1 receiving machine running the specified combination of network stacks.

Peer compatibility. I confirm that TAS interoperates with existing Linux TCP peers by comparing the aggregate throughput of 100 flows between two hosts among all combinations of Linux and TAS senders and receivers. Table 3.2 shows the result. Line rate was achieved in all cases.

3.3.1 Remote Procedure Call (RPC)

RPC is a demanding, but necessary mechanism for many server applications. RPCs are both latency and throughput sensitive. Scaling reliable RPCs to many connections has been a long-standing challenge due to the high overhead of software TCP packet processing [56, 70, 72]. To demonstrate the per-core efficiency benefits of TAS, I evaluate a simple event-based RPC echo server.

Connection scalability. For each benchmark run, I establish an increasing number of client connections to the server and measure RPC throughput over 1 minute. To do so, I use multi-threaded clients running on as many client machines as necessary to offer the required load. Each client thread leaves a single 64-byte RPC per connection in flight and waits for a response in a closed

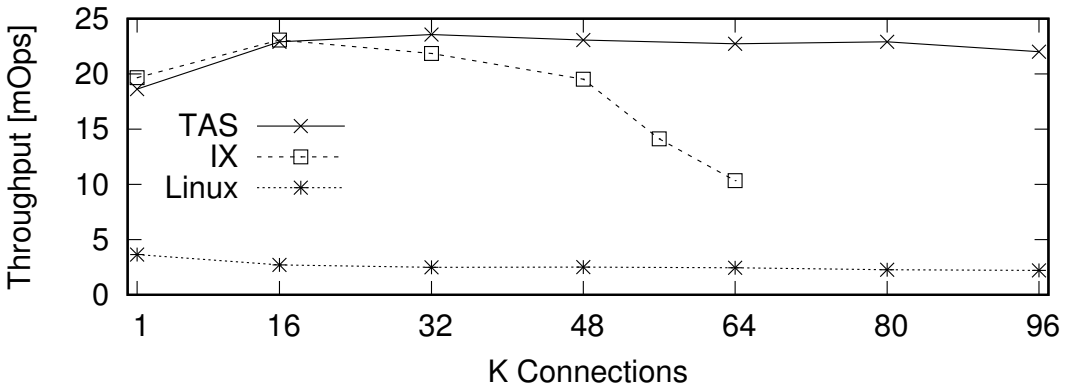

Figure 3.4: Connection scalability for RPC echo benchmark on 20 core server.

loop.

Figure 3.4 shows throughput as I vary the number of client connections. With 1k connections TAS shows a throughput of 5.1 Linux, and 0.95 IX. The improvement relative to Linux is because TAS streamlines processing and thus gains efficiency. After reaching saturation, throughput for both IX and Linux degrades as the number of connections increases, by 40% for Linux and up to 60% for IX. TAS on the other hand only degrades by up to 7% relative to peak throughput. This is because of TAS's minimal fast-path connection state and streamlined packet processing code allowing the CPU to prefetch state efficiently.

Short-lived connections. Separating packet processing into a common case fast path and a separate slow path reduces packet processing overheads in the common case. However, operations that involve slow path processing do incur additional overheads because of handoff overheads between the slow path

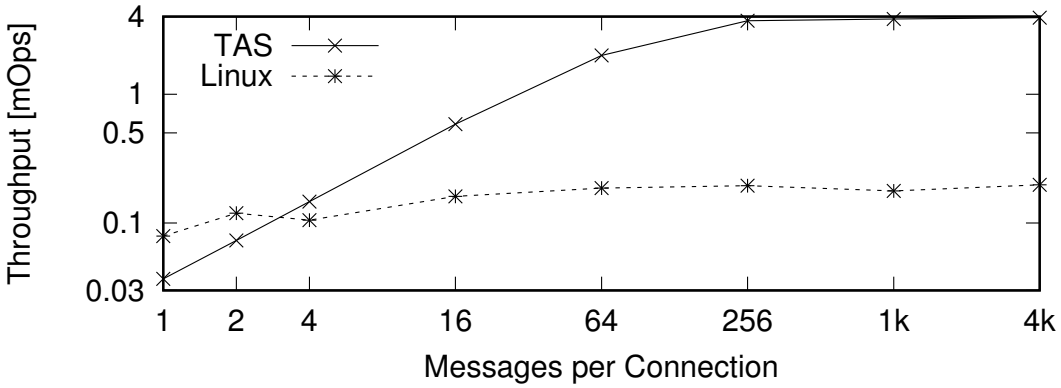

Figure 3.5: Throughput with short-lived connections.

and the fast path. The most heavy-weight such operations in TAS are connection setup and teardown, involving not just the slow path but also the application several times during each handshake. To quantify these overheads I measure throughput of 1,024 concurrent, short-lived connections in our RPC echo benchmark. I use one application core, and for TAS two fast-path cores and one partially used core for the slow-path. Figure 3.5 shows the results for varying numbers of RPCs before connections are torn down and re-established with Linux and TAS. With 4 or more RPCs per connection TAS outperforms Linux, and reaches 95% bandwidth utilization with 256 RPCs per connection.

Pipelined RPC. In cases without dependencies, RPCs can be pipelined on a single connection. These transfers can still be limited by TCP stack overheads, depending on RPC size. I compare pipelined RPC throughput for different sizes by running a single-threaded event-based server processing RPCs on 100 connections, partitioned equally over 4 client machines using 4 threads each.

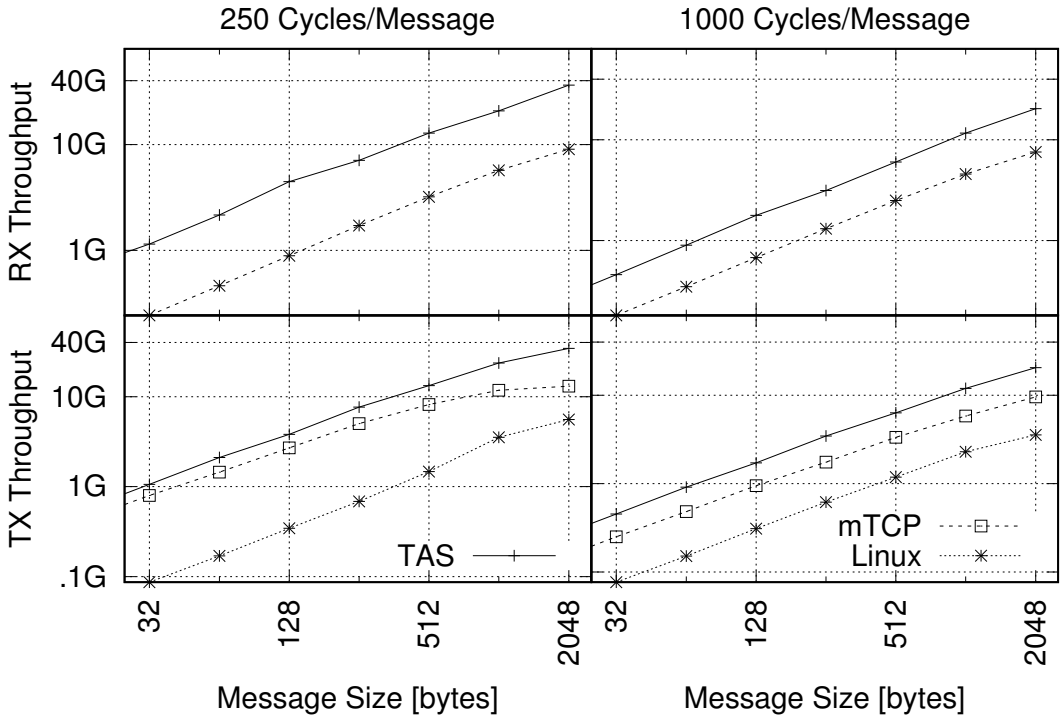

Figure 3.6: Pipelined RPC throughput, varying per-RPC delay and size, for a single-threaded server.

After each RPC the server waits for an artificial delay of 250 or 1000 cycles to simulate application processing. To break out improvements in receive and transmission overhead, I run separate benchmarks, one where the server only receives RPCs and one where it only sends.

Figure 3.6 shows the results. When receiving small (64B) RPCs, TAS provides up to 4.5 better throughput than Linux. TAS's improvement reduces to 4 as RPCs become larger. TAS reaches 40G line-rate with 2KB RPCs for 250 cycles of processing while Linux barely reaches 10G. For 1000 cycles of processing, no stack achieves line-rate and TAS provides a steady

throughput improvement around 2.5 regardless of RPC size. mTCP locks up in this experiment.

When sending small and moderate (256B) RPCs at 250 cycles processing time, TAS provides up to 12.4 Linux and 1.5 mTCP efficiency. For large (2KB) RPCs, TAS's advantage declines to 6.1 Linux, but improves to 2.6 mTCP. mTCP reaches scalability limitations beyond 512B RPCs, while Linux catches up as memory copying costs start to dominate. TAS again achieves 40G line-rate at 2KB RPC size, while Linux and mTCP do not reach beyond 10G. This shows that simplifications in common-case send processing, such as removing intermediate send queueing, can make a big difference.

This difference again diminishes as application-level processing grows to 1000 cycles. In this case, TAS provides a steady improvement of up to 5–6 Linux, regardless of RPC size. Compared to mTCP, TAS provides up to 2 improvement. TAS performs comparably to mTCP in both transmit cases, but does provide protection.

I conclude that TAS indeed provides better RPC latency and throughput when compared to both state-of-the-art in-kernel and kernel-bypass TCP stack solutions. Further, TAS provides throughput on par with and better latency than kernel-bypass stacks while retaining traditional OS safety guarantees. Thus TAS improve performances and efficiency of all networked data center applications relying on RPCs over TCP.

Linux TX Breakdown		
Function	Cycles	%
Sys Call Entry/Exit	1017	14%
msg Construction	705	10%
skb Alloc/Init	2245	31%
TCP	621	9%
IP	577	8%
Queueing	712	10%
Driver	91	1%
Other (Post-TX)	1308	18%
Total	7276	100%

Table 3.3: Unidirectional Benchmark `send` Cycles

Linux RX Breakdown		
Function	Cycles	%
Sys Call Entry/Exit	1023	46%
Socket Identification	407	19%
Message Copying	756	35%
`recv` Total	2186	100%
skb Alloc/Init	1673	70%
Driver	189	7%
Other	527	23%
`irq` Total	2389	100%

Table 3.4: Unidirectional Benchmark `recv` and `irq` Cycles

Unidirectional Processing Overhead Breakdown. In the previous pipelined RPC experiments we recall that with small, 64B messages TAS provided 4.5 better performance than Linux when receiving, but up to 12.4 better performance when sending. It is reasonable to ask why the observed speedup is asymmetric. In order to investigate the asymmetry, I modified the Linux kernel to embed timestamps into the packet payloads at various points in RX and TX processing, which I then read and average over a 30 second period. This timestamp embedding induces only a small overhead of less than 5%. The results can be found in Table 3.3 for the TX cycles and Table 3.4 for the RX cycles. We can see that there is more functionality involved in the Linux TX path. TX takes 7,276 cycles per message, while RX takes 4,575 cycles.

If we look closer at the RX path, we find that RX cycles are split between code that runs when the application calls `recv` and code that runs in interrupt handlers. Hence, the direct cost of each `send` call to the application is 7,276 cycles, while each `recv` call is only directly costing the application 2,186 cycles. The key difference between Linux and TAS in this case is that Linux interrupt handlers can run on any core available to the operating system, distributing the RX functionality over more compute bandwidth. TAS handles RX processing only on configured fast-path cores.

In order to further quantify the effects of distributing the interrupt requests, I ran the same unidirectional Linux RX benchmark as before, with 64B messages and 250 cycles of application processing, but I use the `irqbalance` tool to limit the number of cores that Linux uses to handle NIC interrupts

to a single core, which is equivalent to the evaluated TAS scenario. In this case, the performance of the RPC receive server drops by 29% from 0.62Gbps to 0.44Gbps. TAS now provides a 6.3 speedup. If the application thread is further required to run on the same core that is handling the interrupts, the performance drops by up to 58% in total, from 0.62Gbps to 0.26Gbps. This drops the performance to be more in-line with TX performance, measured to be 0.21Gbps. TAS now provides a 10.7 speedup.

I also break down TAS's RX and TX overheads by embedding timestamps at the start and end of TAS library code and TAS fast path code. On average, I find that TAS spends 413 cycles in the library for RX and 398 cycles in the library for TX. Inside the fast path, TAS spends 975 cycles for RX and 1,032 cycles for TX. Hence, aggregate RX and TX overheads for TAS are 1,388 and 1,430, respectively. These overheads are almost symmetric. Unlike Linux, TAS does not perform all of the TCP functionality for each packet. Certain slow operations, such as congestion control, are only done periodically and not for each packet. This reduces the TAS's TX overhead compared to Linux.

3.3.2 Packet Loss

Even in a data center environment, minimal (1%) packet loss can occur due to congestion and transmission errors. TAS uses a simplified recovery mechanism and we are interested in how packet loss affects TAS throughput in comparison to Linux. I quantify this effect in an experiment measuring throughput of 100 flows over a single link between two machines under different

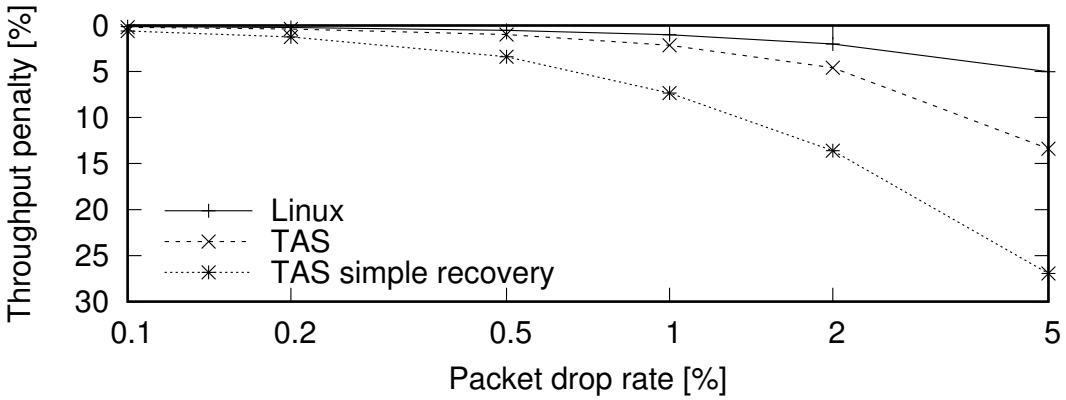

Figure 3.7: Throughput penalty with varying packet loss rate.

rates of induced packet loss between 0.1% and 5%. I compare TAS with receiver out-of-order processing (cf. Exceptions in Section 3.2.1) and without it (simple go-back-N).

Figure 3.7 shows the penalty relative to the throughput achieved without loss. We can see that TAS throughput is minimally affected (up to 1.5%) for loss rates up to 1%. For a loss rate of 5%, TAS incurs a throughput penalty of 13%. Overall, TAS's penalty is about 2 that of Linux. Linux keeps all received out-of-order segments and also issues selective acknowledgements, allowing it to recover more quickly. TAS only keeps one continuous interval of out-of-order data, requiring the sender to resend more in some cases. Without receiver out-of-order processing, the penalty increases by a factor of 3. I conclude that limited out-of-order processing has a benefit, but full out-of-order processing has minimal impact for the loss rates common in data centers.

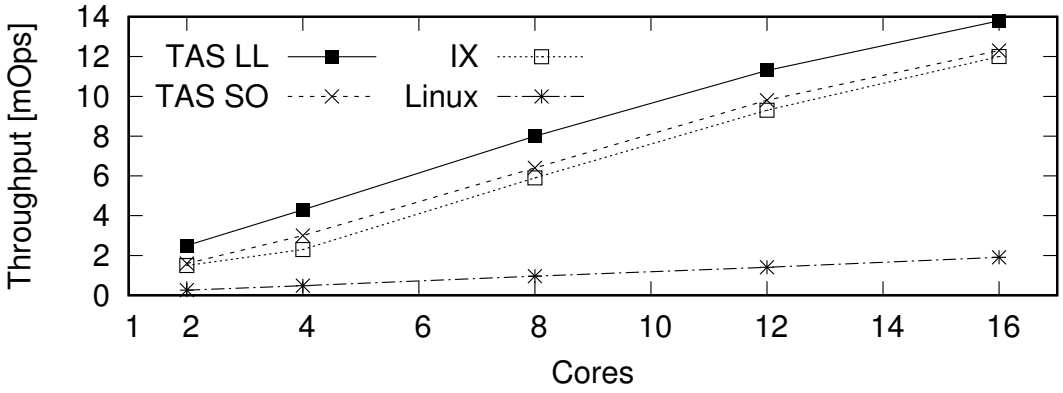

Figure 3.8: Key-value store throughput scalability.

3.3.3 Key-Value Store

Key-value stores strongly rely on RPCs. Due to the high TCP processing overhead, some cloud operators use UDP for reads and use TCP only for writing. In this section, I demonstrate that TAS is fast enough to be used for both reading and writing, simplifying application design. To do so, I evaluate a scalable key-value store, modeled after memcached [52]. I send it requests at a constant rate using a tool similar to the popular memslap benchmark. The workload consists of 100,000 key-value pairs of 32 byte keys and 64 byte values, with a skewed access distribution (zipf, $s = 0.9$). The workload contains 90% GET requests and 10% SET requests. Throughput was measured over 2 minutes after 1 minute of warm-up.

Throughput scalability. To conduct throughput benchmarks I run 5 client machines, each using 12 cores in total to generate requests directed at the server. For all cases I run the clients on TAS to maximize the throughput

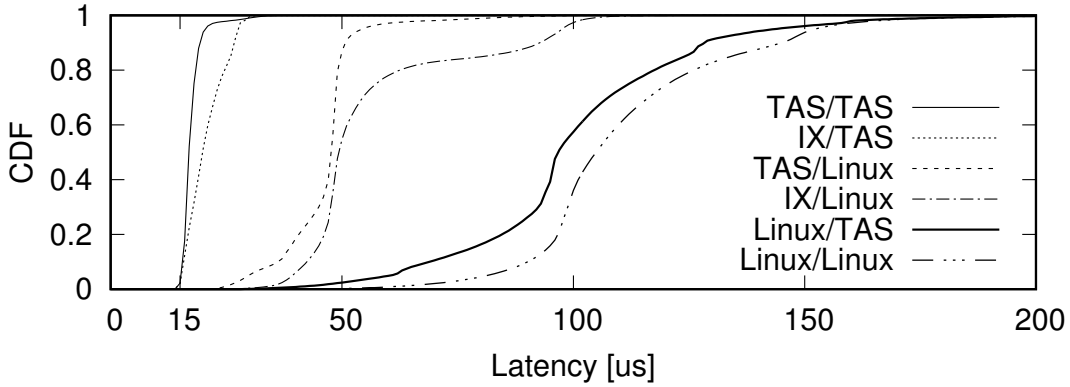

Figure 3.9: Key-value store latency CDF with different configurations (server stack / client stack).

Latency [s]	Median	90th	99th	Max
Linux	97	129	177	1319
IX	20	27	30	280
TAS	17	20	30	122

Table 3.5: Key-value store request latency in microseconds with TAS clients.

they can generate. I establish 32k connections with at most one request in flight per connection, while the clients adjust the offered load to maximize throughput without excessive queuing in the application receive buffers. I run the benchmark, varying the number of server cores available. Figure 3.8 shows the result, counting all server cores in use for the application and TCP stack. I also measure throughput for a version of the key-value store that uses the TAS low-level API (TAS LL), skipping the sockets compatibility layer (TAS SO). We can see that TAS LL outperforms Linux and IX in total throughput by up to 9.6 and 1.9 , respectively, and by up to 7.0 and 1.3 with TAS SO. Table 3.6 shows the split of cores between the key-value store and TAS. TAS SO requires up to 2 fewer cores for TCP processing, which are allocated to the application instead.

Latency. I also conduct end-to-end latency experiments under 15% bandwidth utilization, so that queues do not build excessively. This experiment uses a single application core (and one TAS fast-path core). To quantify both server-side and client-side effects, I repeat the experiment with TAS and Linux on the client side (IX does not support our client). Figure 3.9 shows the resulting latency distributions and Table 3.5 summarizes the results. When using TAS clients, we can see that TAS outperforms Linux and IX by a median 5.6 and 15%, respectively. Both IX and TAS demonstrate much better tail behavior than Linux, improving 99th percentile tail latency versus Linux by 5.9 . While 99th percentile latency of TAS and IX is identical, IX has a longer tail

Total Cores		2	4	8	12	16
Sockets	App	1	2	5	7	9
	TAS	1	2	3	5	7
Lowlevel	App	1	2	4	6	8
	TAS	1	2	4	6	8

Table 3.6: Core split for TAS in the key-value store throughput experiment from Figure 3.8.

than TAS, with a maximum latency 2.3 that of TAS. TAS also maintains lower latencies than IX in the median to 99th percentile range, with a 90th percentile improvement of 26%. I attain similar improvements when using a Linux client.

Non-scalable workloads. I evaluate TAS's performance for workloads with scalability bottlenecks by increasing the access skew to maximize contention on a single 4-byte key and value. Our key-value store uses locks to serialize key updates, causing it to scale badly in this case. This experiment uses the same client setup with 256 connections.

Table 3.7 shows throughput with varying numbers of aggregate cores used for TAS LL and SO, IX, and Linux. The TAS core numbers use 1 application core with 1-3 fast path cores. TAS scales to 4 cores and IX to 3 cores. In the limit TAS improves throughput by 1.6 relative to IX, and by 5.7 relative to Linux (1.1 and 3.9 with sockets). I conclude that TAS's ability to scale the network stack independently from the application can signficantly improve performance for applications with scalability bottlenecks (e.g.,

Throughput [mOps]	1 Core	2 C	3 C	4 C
TAS LL		2.4	3.8	4.6
TAS SO		2.4	3.1	3.1
IX	1.5	2.5	2.8	2.8
Linux	0.3	0.4	0.6	0.8

Table 3.7: Throughput for non-scalable key-value store workload, with a single 4-byte key and 4-byte value pair.

Memcached) or not designed to scale (e.g. Redis [45]).

I conclude that TAS can improve the performance of RPC-based client-server applications, such as key-value stores, even in cases where these applications have scalability bottlenecks. It exceeds state-of-the-art network stacks in both latency and throughput, both in median and the tail. TAS can simplify the design of RPC-based applications by allowing them to rely on the familiar TCP sockets interface.

3.3.4 Real-time Analytics

Real-time analytics platforms are useful tools to gain instantaneous, dynamic insight into vast datasets that change frequently. These systems must be able to produce answers within a short timespan and process millions of dataset changes per second. To do so, analytics platforms utilize data stream processing techniques: A set of *workers* run continuously on a cluster of machines; data *tuples* containing updates stream through them according to a dataflow processing graph, known as a *topology*. The system scales by replicating workers over multiple cores and spreading incoming tuples over the

replicas. To minimize loss, many implementations transmit tuples via the TCP protocol.

To direct tuples to worker cores on each machine, a demultiplexer thread is introduced that receives all incoming tuples and forwards them to the correct executor for processing. Similarly, outgoing tuples are first relayed to a multiplexer thread that batches tuples before sending them onto their destination connections for better performance.

Testbed setup. I evaluate the performance of the FlexStorm real-time analytics platform, obtained from the authors of [41], by running the same benchmark presented in [41]. Figure 3.10 and Table 3.8 show average achievable throughput and latency at peak load on this workload. Throughput is measured over a runtime of 20 seconds, shown raw and per core over the entire deployment. Per-tuple latency is broken down into time spent in processing, and in input and output queues, as measured at user-level, within FlexStorm. I deploy FlexStorm on 3 machines of our client cluster. I evenly distribute workers over the machines to balance the load.

Linux performance. Overhead introduced by the Linux kernel network stack limits FlexStorm performance. Even though per-tuple processing time is short, tuples spend several milliseconds in queues after reception and before emission. Queueing before emission is due to batching in the multiplexing thread, which batches up to 10 milliseconds of tuples before emission. Input queueing is

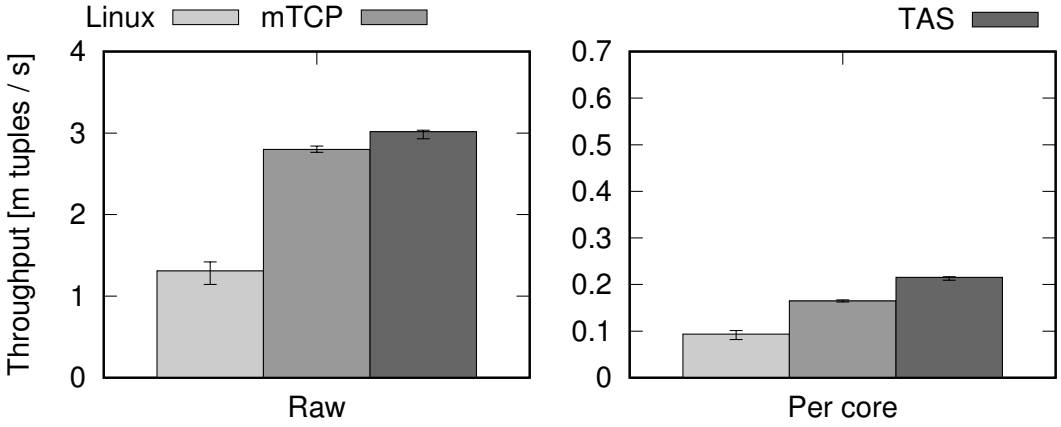

Figure 3.10: Average throughput on various FlexStorm configurations. Error bars show min/max over 20 runs.

minimal in FlexStorm as it is past the bottleneck of the Linux kernel and thus packets are queued at a lower rate than they are removed.

mTCP performance. Running all FlexStorm nodes on mTCP yields a 2.1 raw throughput improvement versus Linux, while utilizing an additional core per node to execute the mTCP user-level network stack. The per-core throughput improvement is thus lower, 1.8 . I could not run mTCP threads on application cores, as mTCP relies on the NIC's symmetric RSS hash to distribute packets to isolated per-thread stacks for scalability. This does not work for asymmetric applications, like FlexStorm, where the sets of receiving and sending threads are disjoint. The bottleneck is now the FlexStorm multiplexer thread. Input queuing delay has increased dramatically, while output queueing delay decreased only slightly. This is primarily because mTCP collects packets into large batches to minimize context switches among threads. Over-

	Input	Processing	Output	Total
Linux	6.96 s	0.37 s	20 ms	20 ms
mTCP	4 ms	0.33 s	14 ms	18 ms
TAS	7.47 s	0.36 s	8 ms	8 ms

Table 3.8: Average FlexStorm tuple processing time.

all, tuple processing latency has decreased only 10% versus Linux due to the much higher amount of batching in mTCP.

TAS performance. Running all FlexStorm nodes on TAS yields an 8% raw throughput improvement versus mTCP and the per-core throughput improvement is 26%. The improvement is only small as the bottleneck remains the multiplexer thread. Overall, tuple processing latency has decreased 56% versus mTCP. This is because TAS does not require any batching to achieve its performance.

While there are limited throughput improvements to using TAS due to application-level bottlenecks, I conclude that tuple processing latency can be improved tremendously compared to approaches that use batching, as fewer tuples are held in queues. This provides the opportunity for tighter real-time processing guarantees under higher workloads using the same equipment.

3.3.5 Congestion Control

I implemented DCTCP congestion control in TAS with the key difference that transmission is rate based, with rates updated periodically for all

flows by the kernel at a fixed pre-defined control interval . I investigate the impact of on congestion behavior via ns-3 simulations, comparing to vanilla DCTCP. First, I simulate a single 10Gbps link with an RTT of 100µs at 75% utilization with Pareto-distributed flow sizes and varying . Next, I simulate a large cluster of 2560 servers and a total of 112 switches that are configured in a 3-level FatTree topology with an oversubscription ratio of 1:4. All servers follow and on-off traffic pattern, sending flows to a random server in the data center at a rate such that the core link utilization is approximately 30%. Finally, I investigate congestion fairness experimentally with = 2 RTT (as measured for each flow) under incast.

Single link. Figure 3.11 shows average flow completion time (FCT) and average queue size with varying for the single 10Gbps link. The average FCT for TAS is very similar to that of DCTCP when is greater than the RTT. However, if is set too low, frequent fluctuations in congestion window cause slow convergence and long completion times. The average queue length is very similar to that of DCTCP and grows, but slowly, as increases beyond the RTT, due to delayed congestion window updates.

Large cluster. Figure 3.12 shows the average flow completion times for short and long flow sizes in the large cluster simulation with the control interval set to 100µs. The performance of TAS is similar to that of DCTCP in both cases. 100µs is a reasonable amount of time for the kernel to update

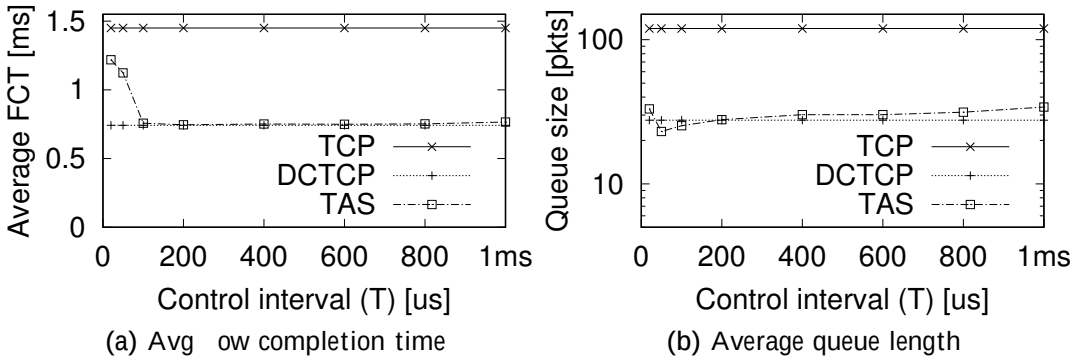

Figure 3.11: Simulation of a single 10Gbps link.

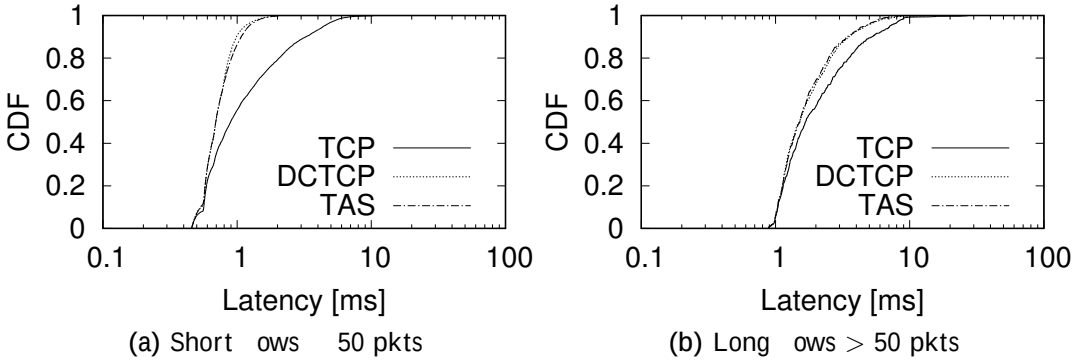

Figure 3.12: Flow completion times for large cluster simulation.

congestion windows for thousands of flows. Even with larger values of , queue size is only minimally affected and FCTs stay approximately identical. I thus conclude that our out-of-band approach works to provide DCTCP-compatible congestion behavior.

Tail-latency under incast. To evaluate performance under congestion, I measure tail latency under incast with 4 machines sending to a single receiver (operating at line rate) with different numbers of connections. I record the

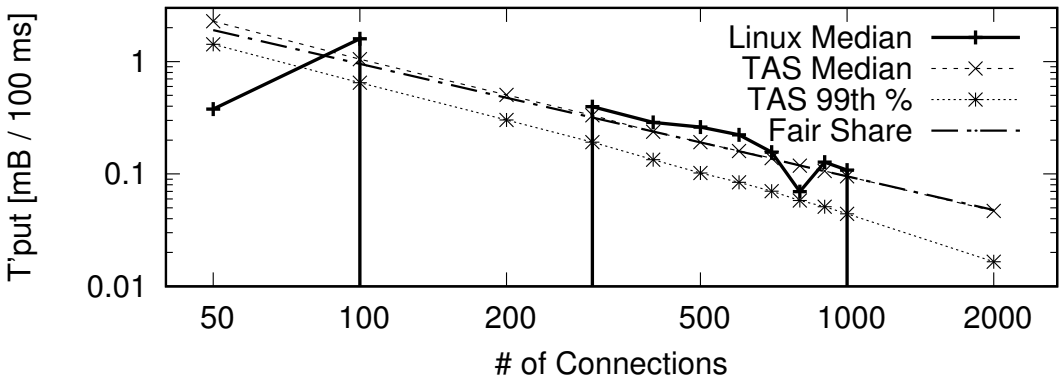

Figure 3.13: Distribution of connection rates under incast.

number of bytes received on each connection every 100ms on the receiver over the period of a minute, discarding a warmup 20 seconds. Figure 3.13 shows the median (and 99th percentile) throughput over the measured intervals and connections on Linux (using DCTCP) and TAS. For TAS, the tail falls within 1:6 and 2:8 of the median, while the median is close to each connection's fair share. Linux median (and tail—not shown) behavior fluctuates widely, showing significant starvation of flows in some cases.

Linux fairness is hurt in three interacting ways: (1) Linux window based congestion control creates bursts when windows abruptly widen and contract under congestion. (2) Window-based congestion control limits the control granularity for low-rtt links. (3) The Linux TCP stack architecture requires many shared queues that can overflow when flows are bursty, resulting in dropped packets without regard to fairness. Rate-based packet scheduling and per-flow queueing in TAS smoothes bursts and eliminates unfair packet drops at end hosts.

3.3.6 Workload Proportionality

Finally, I analyze a dynamic workload to evaluate how TAS adapts to workload changes and how this affects end-to-end performance. For this experiment, I re-use and instrument the key-value store server and vary the number of clients over time. At time 0 I start with one client machine, and add four additional client machines, one every 10 seconds, after an additional 10 seconds I remove the client machines again one by one. Figure 3.14 shows both the number of fast-path cores for TAS as well as the total server throughput. TAS starts out with just 1 core, and ramps up to 3 cores for the first client, and continues to add additional cores until reaching 9 cores, before incrementally removing cores again as the load reduces. Figure 3.15 shows request latency as measured by clients as for the transition from 3 to 4 clients and with it 7 to 9 cores. During the adjustment the latency temporarily spikes by about 15 s or 30% before quckly returning back to the previous level. I conclude that TAS is able to adapt to workload changes, acquiring and releasing processors as needed, without significantly impacting end-to-end latency or throughput.

3.4 Conclusion

The continuing increase in data center link bandwidth, coupled with a much slower improvement in CPU performance, is threatening the viability of kernel software TCP processing, pushing researchers to investigate alternative solutions. Any alternative has to ensure that it is safe, efficient, scalable, and flexible.

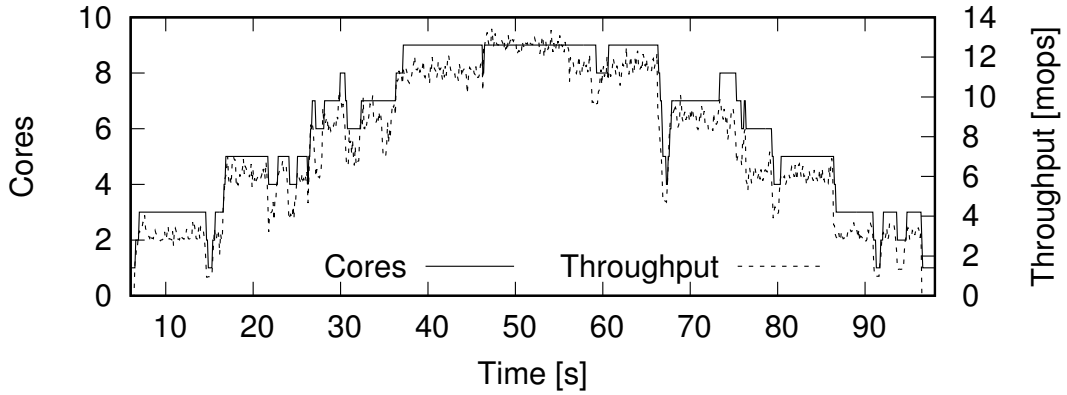

Figure 3.14: Number of TAS processor cores and end-to-end throughput as key-value store server load first increases and then decreases again.

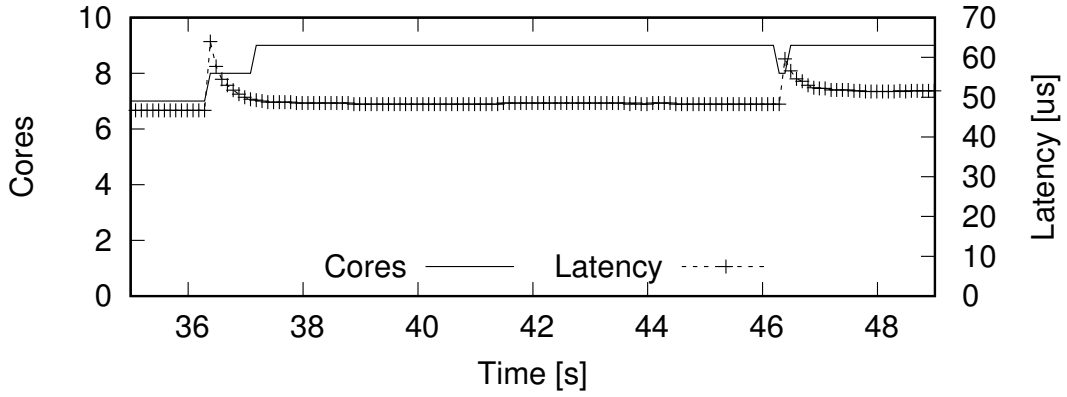

Figure 3.15: End-to-end request latency as TAS acquires additional processor cores in response to increasing load.

In this chapter, I presented TAS, TCP acceleration as a software service. TAS executes common-case TCP operation in an isolated fast path, while handling corner cases in a slow path. TAS achieves throughput up to 7x that of Linux and 1.3x thatof IX for common, unmodified cloud applications. Unlike kernel bypass, TAS enforces congestion control on untrusted applications, and achieves much higher levels of per-flow fairness than Linux. TAS scales to many cores and connections, providing up to 2.2x higher throughput than IX on 64K connections, while facilitating TCP protocol innovation.

By providing a software solution with full security and functionality, TAS seeks to easily drop-in and accelerate the network stacks of small IO-intensive applications.

Chapter 4

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

In Section 2.2.2, I quantify the overheads of data movement in large IO-intensive applications and I describe how existing zero-copy solutions are not transparent to applications. In this chapter, I show how we can create a solution to eliminating copies inside of these applications and IO-stack APIs while remaining fully transparent to the application code.

4.1 Minimizing Copies in the IO Path

As previously discussed, zero-copy IO stacks have been a research goal for a long time 2.2.2.2. Both zero-copy IO stacks and cross-stack APIs, seek to eliminate copies involved inside the IO stacks, as well as the interface used to interact with them. They do so by providing their own, special-purpose APIs. These introduce complexities, such as buffer ownership management and special API calls. They also introduce restrictions, such as requiring run-to-completion processing, disallowing in-place updates, and they may require knowledge of external IO properties. These complexities and restrictions are difficult for developers to navigate and knowledge about external IO properties is often difficult or impossible to attain. Finally, none of the existing zero-

copy APIs provide transparent copy elimination or eliminate copies that are made within the application.

With these limitations in mind, I designed zIO to transparently eliminate copies within the application itself, with the option of further removing copies with IO stacks. With zIO, the aforementioned restrictions for doing zero-copy IO are removed, while also targeting an even larger source of overhead: copies within the application. zIO's main idea is to track 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 the data that the application accesses. By tracking data and eliminating copies, zIO minimizes the overhead incurred by copies and transparently improves application performance.

zIO works under the assumption that IO-intensive applications often modify only a part of the data they process. Unmodified data may remain in its original location, while modified data continues to be copied. However, the challenge is that we do not know a priori what data will be modified. zIO optimistically assumes that most data will remain unmodified 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 modify 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 writable. Another challenge is to deal with unaligned memory areas. In this case, zIO performs the copy of unaligned sections (the *fringe*) of the area eagerly and leaves the remaining, core portion unmapped. For large IO, unaligned fringe sections are small compared to the core and copying them does not harm performance.

In addition to eliminating application copies, I also use zIO to eliminate copies across IO stack APIs. To do so, I use kernel-bypass IO stacks in addition to zIO. Kernel-bypass stacks often share memory with the application 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.

In following these choices, zIO seeks to eliminate the largest source of overhead—application copies—while further improving existing kernel bypass stacks to reduce their number of copies. In doing so, zIO improves performance for large IO-intensive applications.

In this section, we describe how we can accelerate large IO operations by reducing application and IO stack data movement. Specifically, zIO seeks to elide large copy operations. zIO replaces expensive copy operations with simpler unmappings and tracks data modifications to ensure correctness. The following is the design and evaluation of zIO.

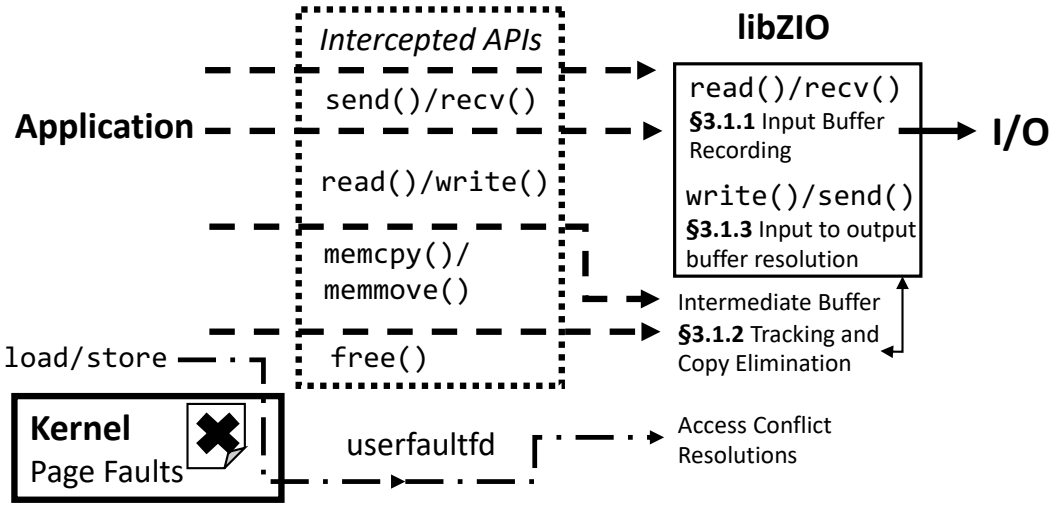

Figure 4.1: zIO overview.

4.2 zIO System 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 4.1), 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. I now describe zIO in three parts. First, I describe how zIO tracks data within the application to eliminate *application copies* (§4.2.1). Second, I describe how I extend zIO with kernel-bypass IO stacks to allow it to eliminate *IO stack copies* (§4.2.2). Third, I describe how to realize *optimistic input persistence* by mapping appropriate IO buffers into NVM (§4.2.3).

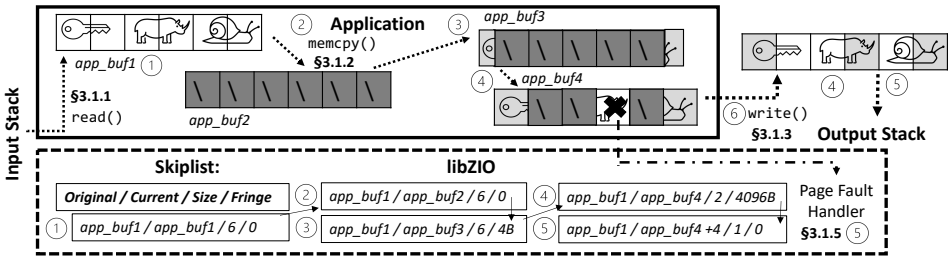

Figure 4.2: zIO application copy elimination example.

4.2.1 Application Copy Elimination

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

Figure 4.2 shows the mechanisms involved in this process via an example involving a key-value pair being read from an input stack, processed, and written to an output 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 (e.g., `memcpy/memmove` calls), zIO unmaps the destination buffer, forgoes the copy, and tracks the destination buffer as *intermediate* (②, ③, ④). 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 I detail each of these mechanisms, I 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, 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, I find that it often helps performance. The left and right buffer fringe often contain headers or 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 is frequently mutated, I choose a skiplist for probabilistic fast search and in-place updates. 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 last copy (cf. §4.2.1.7), and a free flag (cf. §4.2.1.4). The skiplist is sorted by intermediate buffer address.

4.2.1.1 Input buffer recording

When data is read from an IO stack, zIO intercepts these operations, and, according to policy (cf. §4.2.1.6), 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.

4.2.1.2 Copy tracking and elimination

zIO identifies copies within the application by interposing on standard library memory copy calls, such as `memcpy` and `memmove`. 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. §4.2.1.6), instead of executing the copy, zIO records in the skiplist the core 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, zIO first uses the core source buffer address to search through the skiplist to see if it falls within any existing intermediate buffers. If it does, zIO uses that buffer's original buffer location,

and, if this is the first time this original buffer is tracked, zIO also remaps the core original buffer read-only to prevent the application from modifying it. If zIO finds no intersecting intermediate buffer, then this data did not originate from IO (cf. §4.2.1.1) and the copy is executed instead, forgoing any tracking of this buffer. Finally, if the data originated from IO, zIO records 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 zIO does not need to record it). The left buffer fringe is necessary to resolve access conflicts (cf. §4.2.1.5). If the destination location is within a buffer 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 typically remain unmapped and are backed with physical memory 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.

4.2.1.3 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, non-shaded areas are sourced from original buffer locations). If the IO stack API supports gather IO, zIO leverages 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.

4.2.1.4 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 zIO has 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).

4.2.1.5 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 allocates and maps the faulting page (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 simply 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 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 (typically readable and writeable).

4.2.1.6 Tracking policy

I determine experimentally (cf. §4.4.1) that for data buffers smaller than 16KB, the overhead of tracking outweighs any performance benefits from eliminating copies. Hence, I 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. I determine this experimentally (§4.4.1) under a number of conditions. I find that if the ratio of bytes accessed by the application to bytes eliminated from copies exceeds 6%, zIO no longers see a performance benefit. After these thresholds, I stop eliding copies for these buffers.

4.2.1.7 Intermediate buffer garbage collection

zIO avoids tracking an arbitrary number of entries, as that will exhaust memory and reduce performance. For example, tracked intermediate buffers may be stored indefinitely in memory by the application, causing skiplist entries to accrue. Hence, skiplist entries are garbage collected periodically (once every second). For each skiplist entry to garbage collect, zIO must fill any intermediate buffers with consistent data. This is done by 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 as freed 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 4.2) 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.

4.2.2 IO Stack API Copy Elimination

I now describe how to use zIO with kernel-bypass IO stacks to add 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 readily intercept and process—rather than system calls. In particular, I chose the TAS [42] and Strata [44] 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 automatically intercepts memory copies into this log and can provide transparent copy elimination, provided that the source buffer already resided in NVM.

Input API copy elimination. Vanilla 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 a library-internal buffer to the application-provided buffer. zIO transparently tracks and eliminates this copy across the IO stack API (cf. §4.2.1). As the source buffers are IO stack-private, zIO does not need to protect the original data source by remapping it read-only. Instead, I modify the IO stacks to execute zIO's garbage collection protocol for any source data that needs to be freed or overwritten. To prevent this from happening frequently, the user can configure the IO stack input space 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. Vanilla POSIX file and socket output calls (e.g., `write` and `send`) require applications to provide a buffer that output data is copied out of. As with the input API calls, zIO already transparently eliminates stack-internal memory copies. As output buffers are stack-private, no unmapping is necessary. Instead, I modify the IO stacks to fetch the original buffer locations from zIO when the output data is processed. For example, when TAS send payload from the socket transmit buffer or when Strata "digests" [44] the update log. When zIO has to resolve copies due to misspeculation 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.

4.2.3 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, zIO simply has 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 mapped to NVM, zIO can eliminate and track any copies among the buffers, while ensuring persistence. I 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 in this shared memory. I can realize optimistic network receiver persistence simply by mapping the socket receive buffers into NVM. zIO will detect that original buffers are in NVM and eliminate copies end-to-end to the Strata update log, which also resides in process-local NVM.

4.3 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. §4.4.2) 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. My 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 I present IO stack API copy elimination with kernel-bypass stacks (§4.2.2), I 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. §2.2.2.2). For example, using Linux's zero-copy socket receive API, 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 §4.2.2. 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. I leave IO stack API copy elimination for the Linux kernel IO stacks for future work.

4.4 Evaluation

I first analyze zIO's performance with a number of evaluations based on a multi-threaded IO microbenchmark that can use network and storage stacks and vary relevant IO and copy parameters. I then evaluate zIO with IO-intensive applications, like Redis [45], MongoDB [16], and Icecast [82]. I compare zIO to Linux and kernel-bypass IO stacks without any copy optimizations.

My 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 (§4.4.1), IO size (§4.4.1), and number of IO threads (§4.4.1) 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? (§4.4.1)
- How do zIO performance improvements break down into its mechanisms? By how much can I improve IO performance when employing optimistic input persistence with NVM? (§4.4.2)
- What benefits to IO processing throughput and latency does zIO provide by eliminating copies within IO-intensive applications, such as Redis (§4.4.2), Icecast (§4.4.3), and MongoDB (§4.4.5)? In what situations might zIO hurt application performance (§4.4.4)?
- How does zIO perform compared to zero-copy IO APIs, such as memory mapped files and `sendfile`? (§4.4.4)

Experimental Setup I 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 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.

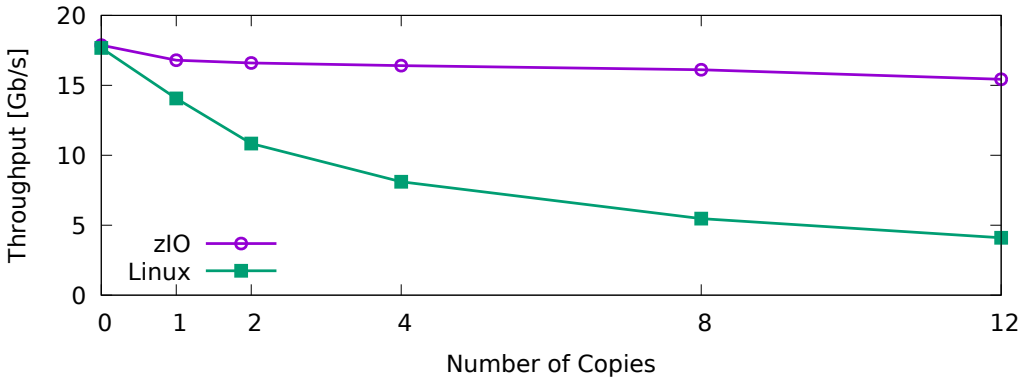

Figure 4.3: Linux throughput versus zIO application IO copy elimination (512KB message size).

4.4.1 Microbenchmarks

I 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 *x*2.2.2, but in place of Redis I 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 copies, I also vary other IO parameters, such as message size, fraction of IO data accessed, and number of echo server threads. I show the maximum throughput achieved over 3 runs for each configuration.

Number of Copies I first evaluate IO performance with a varying number of copies of the IO data made before it is echoed. I compare four scenarios: Vanilla Linux (Linux), Linux with zIO application copy elimination (zIO),

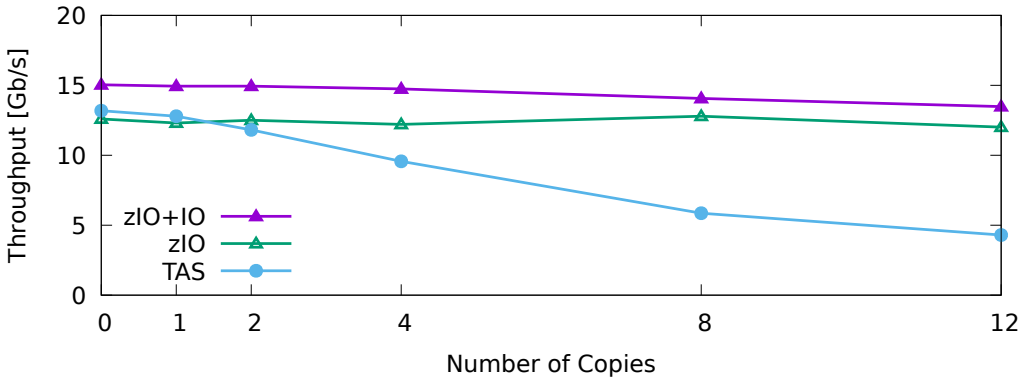

Figure 4.4: TAS throughput versus zIO application and IO stack API copy elimination (512KB message size).

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). I run this experiment with 512KB size messages, using a single server thread. For each run, I vary the number of times the request is copied before being echoed.

Figures 4.3 and 4.4 present the results. We can see that an increasing number of application IO data copies decreases the achieved throughput for Linux and kernel-bypass networking[1], 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

[1] I consistently observe TAS throughput to be lower than Linux with large message sizes. TAS is optimized for small messages and does not do the necessary batching to handle large messages e ciently.

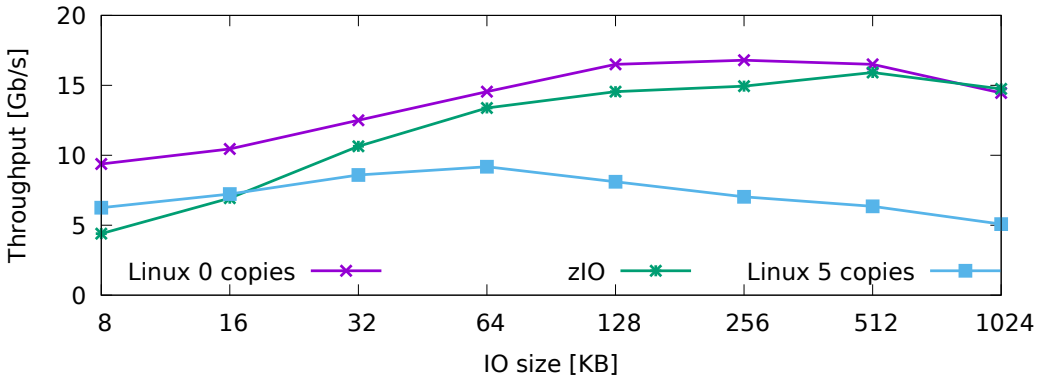

Figure 4.5: zIO throughput versus Linux with 0 and 5 copies.

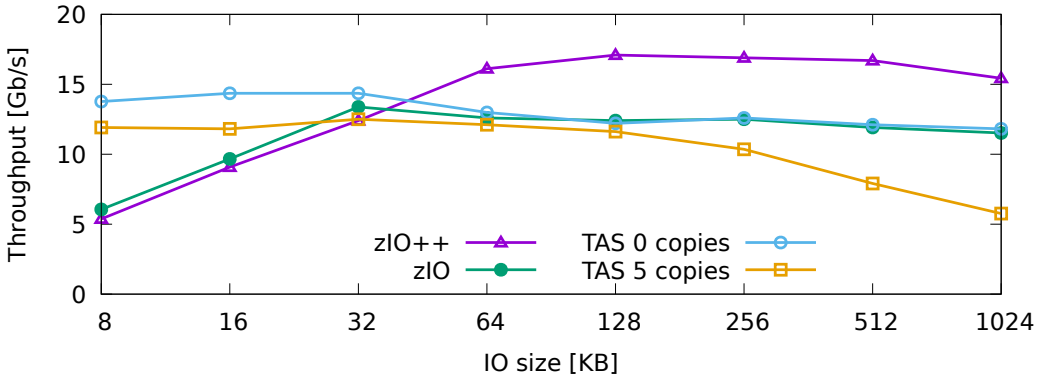

Figure 4.6: zIO throughput versus TAS with 0 and 5 copies.

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 elimination IO stack API copies.

IO Size I next investigate how IO size affects performance, using a single echo server thread. I vary the IO size from 8KB to 1MB and evaluate two extreme copy scenarios (cf. Table 2.3): 5 application copies and 0 application

copies.

Figures 4.5 and 4.6 present the results. 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[2]. 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 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 vanilla TAS.

However, zIO transparent copy elimination is no panacea, as the overhead of zIO tracking with small IO also limits throughput. For IO smaller than 16KB, zIO with Linux slows down throughput by up to 30%. For IO smaller than 32KB, zIO with TAS slows down throughput by up to 49%. Based on this measurement, I set zIO's tracking policy to avoid tracking IO buffers smaller than 16KB (cf. 4.2.1.6).

Scalability I evaluate two scalability aspects. zIO tracking scalability and the impact of page faults on scalability.

[2] TAS again struggles with larger IO sizes.

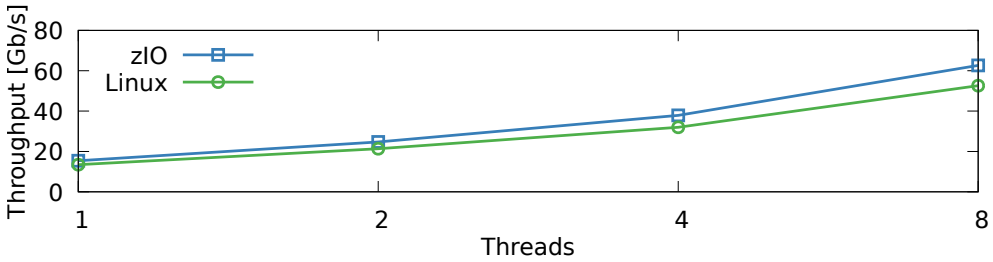

Figure 4.7: zIO scalability.

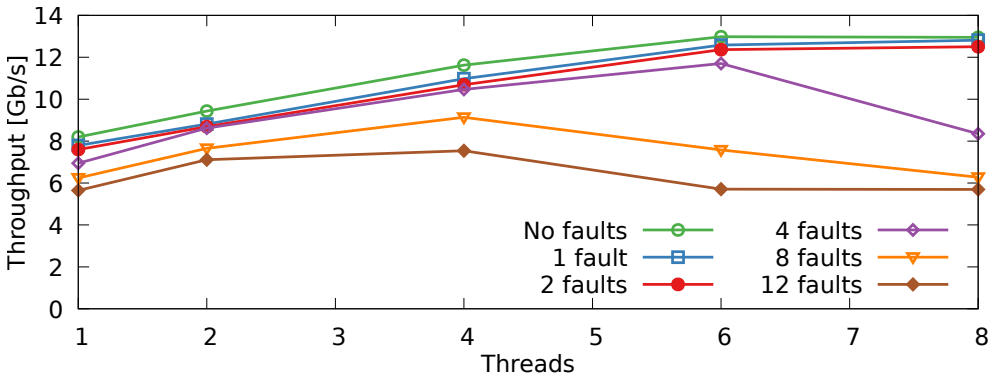

Figure 4.8: zIO scalability with page faults.

zIO tracking. I 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 4.7 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 TLB shootdowns are required. In theory, zIO's approach does not require TLB shootdowns for

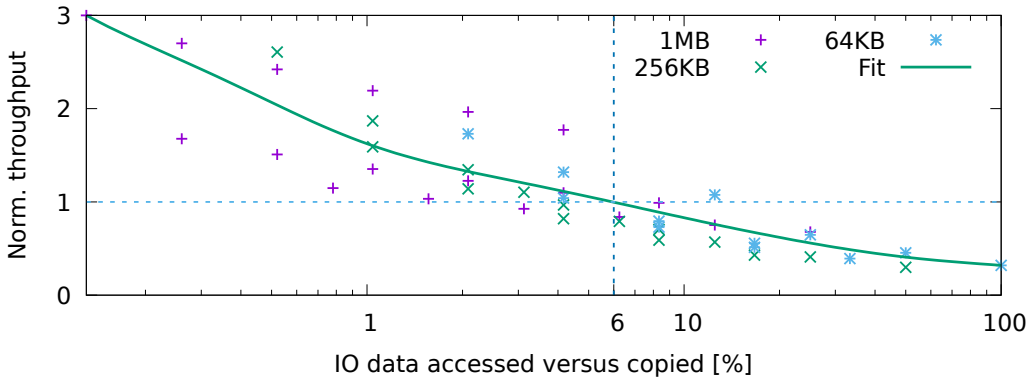

Figure 4.9: zIO throughput improvement under data access.

most IO-intensive applications. Pages are mapped on page fault and this information may be lazily synchronized among TLBs. If other cores access the same page, they 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, I 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. I supply the same IO buffer each time, requiring zIO to unmap it for each IO request.

The results can be found in Figure 4.7. 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, these TLB shootdowns can be avoided.

Mis-speculation To evaluate the impact of zIO mis-speculation on performance, I configure the echoserver to access a number of bytes in each IO request before echoing a response. I 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 4.9 presents the results as a scatter plot, where I compare zIO throughput improvement over vanilla Linux to the ratio of IO data accessed by the echoserver versus IO data elided in copies. We can see that IO size adds a size-dependent boost to zIO's throughput improvement (cf. Figure 4.5), but it does not interact with other parameters, such as IO data accessed and elided in copies. 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.

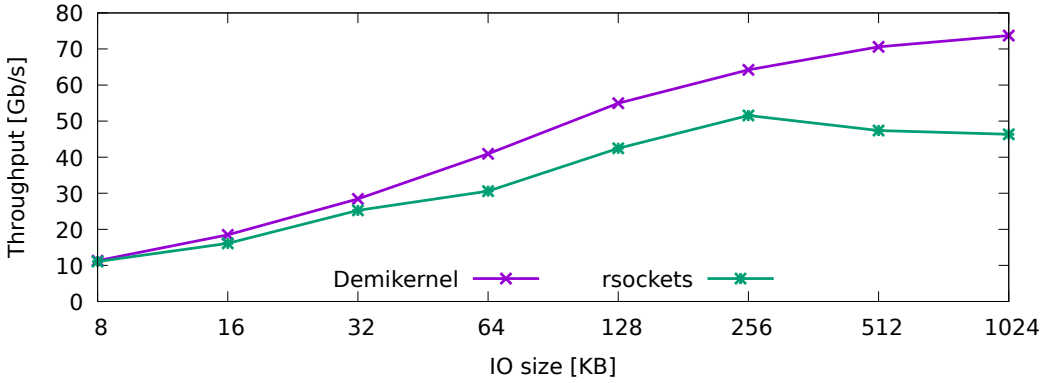

Figure 4.10: RDMA echoserver throughput with sockets versus Demikernel interface

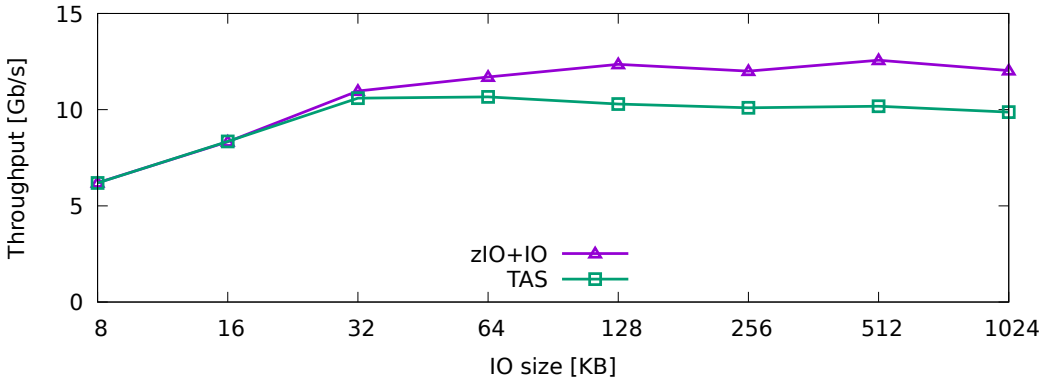

Figure 4.11: TAS echoserver throughput versus TAS with zIO+IO

Comparison to Demikernel. Another zero-copy IO interface with cross-stack capability is presented by Demikernel [84]. This makes Demikernel an interesting system for comparison. Unfortunately, I was unable to fairly evaluate Demikernel as a cross-stack API because of its storage stack functionality being limited to solid-state drives (SSDs). Since my zIO prototype with Strata currently only supports NVM, Demikernel's performance is severely limited by its choice of supported storage medium. Hence, I am omitting an evaluation of Demikernel as a cross-stack API.

Further, Demikernel's current prototype implementation also prevented me from fairly evaluating its network zero-copy API. Demikernel currently only supports message sizes smaller than the Ethernet maximum transfer unit (MTU) inside within the TCP stack. Even with jumbo frames, the limit of 9KB is smaller than the minimum message size threshold of 16KB where zIO starts to yield performance benefits. Hence, I could not execute this experiment, either.

Finally, I attempted to compare the Demikernel zero-copy API on top of an RDMA stack instead of TCP. I find that in this case, the underlying transport protocol is able to support larger message sizes, allowing us to compare RDMA with a sockets interface, `rsockets` [2], against RDMA with a Demikernel interface. This comparison will demonstrate an elimination of two IO Stack API copies.

I run a simplified echoserver from earlier to allow compatibility with Demikernel. This echoserver only runs a single thread with just a single RDMA

connection. The results can be found in Figure 4.10. We can see that as the message size increases, the performance benefit of using the Demikernel interface grows. Since the primary difference between the two cases is the IO Stack API, this performance boost comes from eliminating IO Stack API copies. We can see that with 8KB messages, there is only a 2% speedup, while at 1MB messages there is up to a 59% speedup.

I also run the same simplified echoserver as the previous experiment, in a similar configuration, but with TAS as the underlying transport stack instead of RDMA. Under the conditions of this echoserver (single thread, single connection), TAS will not perform close to RDMA. As a baseline comparison, `rsockets` with RDMA reaches a max throughput of 52.8Gbps as opposed to 10.9Gbps for TAS. The purpose here is not to compare TAS with RDMA, but to show zIO+IO eliminating IO Stack API copies in a similar configuration to the previous experiment. The results can be seen in Figure 4.11. As we can see, zIO+IO also starts with a smaller performance benefit, not providing a speedup until 32KB with a 4% speedup, then performing better up to 512KB with a 25% speedup. TAS still has some performance limitations as discussed earlier which limit the benefits of eliminating IO Stack API copies with zIO+IO. However, the Demikernel results demonstrate the potential for further performance benefit from eliminating IO Stack API copies with a different IO stack API.

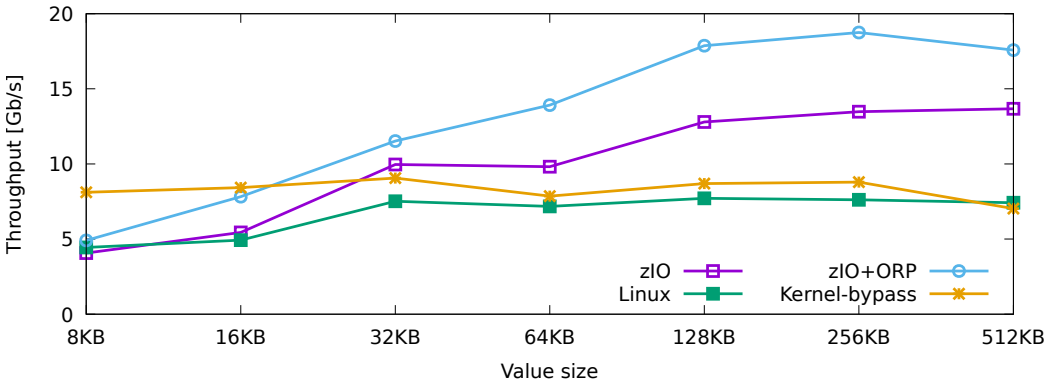

Figure 4.12: Redis throughput (100% SET).

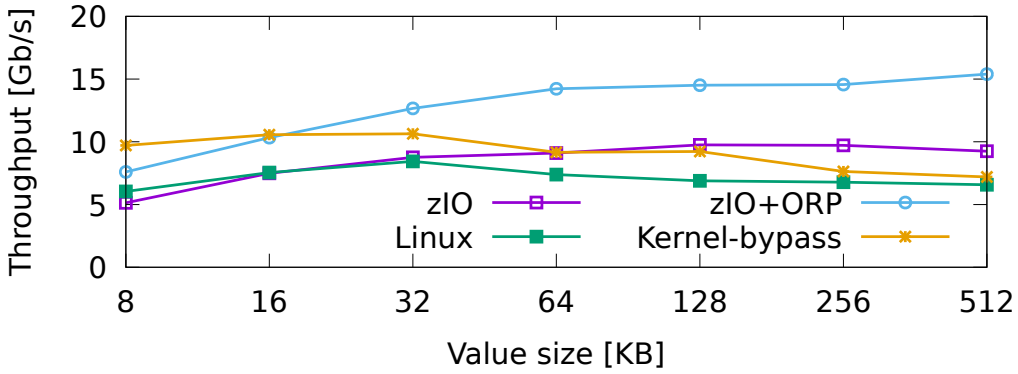

Figure 4.13: Redis throughput YCSB A.

4.4.2 Redis

I 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 §2.2.2. I evaluate two benchmark configurations: 1) 100% SET, and 2) YCSB Workload A, which has a distribution of 50% SETs and 50% GETs. I 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, I 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 objects, evaluated in §4.4.2.

I first look at 100% SET throughput. This case involves 2 IO operations (one to network and one to storage), as well as 4 application copies per request (cf. Table 2.3). The results can be found in Figure 4.12. 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.

I now look at at YCSB workload A, with 50% GET requests and 50% SET requests. These results can be found in Figure 4.13. 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,

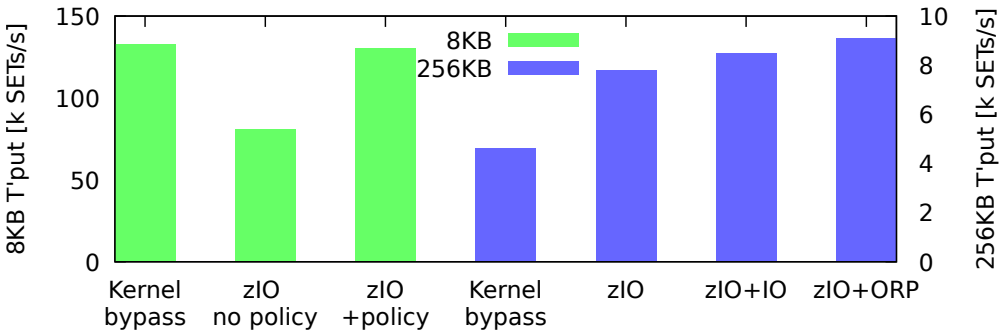

Figure 4.14: zIO performance breakdown.

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.

zIO Performance Breakdown I use the Redis 100% SET benchmark to break down the performance contributions of zIO. To do so, I evaluate zIO throughput with kernel-bypass IO, with different zIO optimizations enabled. These results can be found in Figure 4.14.

The first configuration uses 8KB SET requests. I evaluate zIO with and without its tracking policy, which applies a 16KB size threshold (*x*4.2.1.6). 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.

I 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%.

	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 4.1: Icecast throughput.

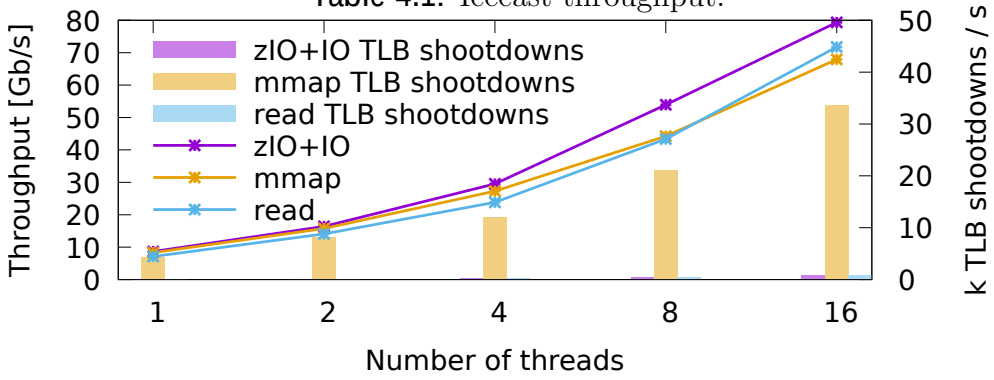

Figure 4.15: Icecast throughput scalability.

Adding optimistic receiver persistence in zIO+ORP finally improves performance by another 7%, for a combined improvement over vanilla kernel-bypass of 2 .

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

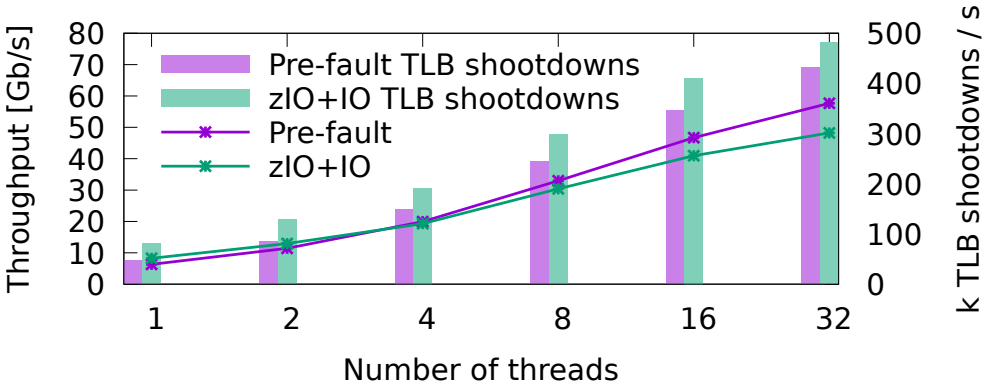

Figure 4.16: Icecast scalability with pre-faulted buffers.

4.4.3 Icecast

Icecast [82] 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. Table 2.3 shows that Icecast makes no application copies, but it uses the IO stack APIs. I evaluate both Icecast configurations, providing insight into network to network and storage to network performance. I use the kernel-bypass IO stacks for our evaluation, as they support IO stack API copy elimination. The results are shown in Table 4.1.

Storage to network. I configure Icecast to broadcast a 1-minute audio file to a number of clients. I 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 I set to 64KB. Listener clients request the audio stream via `curl-loader` [32], a HTTP benchmark tool. I connect enough

clients to saturate Icecast server throughput and measure audio throughput over a 30 second period. I 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, evaluate Icecast receiving data from a single client and broadcasting it to a number listeners. I measure the number of concurrent listeners that Icecast can broadcast to. In this configuration, Icecast uses a single relay thread to broadcast audio in 4KB chunks from the source to each connected listener. 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. However, Icecast is hard coded to broadcast 4KB at a time, which limits zIO+IO benefits.

4.4.4 Scalability

Icecast is a single-threaded application when serving local files to listeners. To evaluate IO-intensive application scalability with zIO, I 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 [7]. 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, I create a version of Icecast that maps a requested file into memory (cf. *x*2.2.2.2) 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.

I 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 4.15. 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, I 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 bu ers. I have already evaluated zIO scalability when buffers are touched, incurring page faults (*x*4.4.1). zIO can detect these cases and stop copy elision (*x*4.2.1.6). 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, I 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. I run these two configurations with a 512KB audio file, a 32KB chunk size, and an increasing number of threads. I measure throughput and TLB shootdowns for both cases. I present these results in Figure 4.16. 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.

4.4.5 MongoDB

I run MongoDB [16] on Linux, with and without zIO. I connect a client over the network running the YCSB [18] 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. I repeat this benchmark 5 times and report the average throughput for each configuration.

I 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. I 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 I modify MongoDB to skip checksum calculation, zIO is able to eliminate 2 out of 3 application copies (cf. Table 2.3). Similar to Redis, 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.

I also run MongoDB with the TAS kernel-bypass network stack, allowing me 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.

4.5 Conclusion

In this chapter, I presented 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. I implement zIO as a userspace library, supporting Linux kernel and kernel-bypass IO stacks. I evaluate zIO with IO-intensive applications, like Redis, Icecast, and MongoDB. zIO improves application throughput by up to 1.8x with Linux, as well as by up to 2.5x with kernel-bypass IO stacks with optimistic network receiver persistence.

zIO addresses the concerns of alternative zero-copy IO stacks by transparently interposing on applications and eliminating copies both within the IO stack API and *within applications themselves*. In doing so, zIO accelerates the performance of large IO-intensive applications.

Chapter 5

Related Work

There are a lot of modern systems that seek to address the problems that I have outlined in this thesis. In the following sections, I will discuss a number of these systems, both their advantages and drawbacks.

5.1 Optimized Network Stacks

Software TCP stack improvements. Many systems in addition to TAS have sought to improve the performance of TCP in software. Reducing software overhead often comes with some level of NIC assistance. Many of these systems also use batching to reduce overhead at some cost in latency; our focus is on reducing overhead for latency-sensitive RPCs where batching is less appropriate. Affinity-accept [61] and Fastsocket [47] use flow steering on the NIC to keep connections local to cores. Arrakis [62] and mTCP [39] use NIC virtualization to move the TCP stack into each application, eliminating kernel calls in the common case, at the cost of trusting the application to implement congestion control. StackMap [83] takes a hybrid approach, using the featureful Linux in-kernel TCP stack, but keeping packet buffers in user-space, eliminating copies between user and kernel space; this provides moderate speedups, but requires the application to be trusted and modified to StackMap's in-

terface. Sandstorm [49] co-designs the TCP stack using application-specific knowledge about packet payloads; this is an interesting avenue for future work. Megapipe [30] re-designs the kernel-application interface around communication channels; we use a similar idea in our design. IX [11] pushes this farther by also changing the socket interface; we aim to keep compatibility with existing applications. To improve load balancing, ZygOS [66] introduces an object steering layer that is similar to ours. Finally, CCP proposes separating congestion control policy from its enforcement [54]; our work can be seen as an implementation of that idea.

NIC-Software co-design. Earlier work on improving packet processing performance used new HW/SW interfaces to reduce the number of required PCIe transitions [13, 24], to scale rate limiting [67], and to enable kernel-bypass [23, 65, 78]. TCP Offload Engines [15, 21] and remote direct memory access (RDMA) [69] go a step further, entirely bypassing the remote CPU for their specific use-case. Scale-out NUMA [57] extends the RDMA approach by integrating a remote memory access controller with the processor cache hierarchy that automatically translates certain CPU memory accesses into remote memory operations. Portals [9] is similar to RDMA, but adds a set of offloadable memory and packet send operations triggered upon matching packet arrival. Out of these approaches, only kernel bypass has found broad market acceptance. One hindrance to widespread adoption of network stack offload is that hardware stack deployment is slower than software stack deployment, while

application demands and data center network deployments change rapidly. Hardware approaches are thus often not able to keep pace fast enough with the changing world around them. By providing an efficient software network stack, TAS side-steps this issue, while providing performance close to that of hardware solutions.

Since the publishing of TAS, there has been much more work on optimizing network stacks with hardware offload. Tonic [8] seeks to provide a solution more general than TCP offload. Tonic identifies patterns in network protocols and generalizes these to a protocol template that can be offloaded to network hardware. This allows for development of new network protocols directly on real hardware. FlexTOE [75] is another solution, this time focusing again on TCP offload. FlexTOE seeks to optimize TCP data path processing by parallelizing TCP operations, making them more efficiently processed by a many-core NIC hardware architecture. Again, while TAS may not provide quite as good performance as a hardware offload, it still provides good performance with flexibility and ease of use.

Optimized RPC Stacks In order to better serve specifically data center RPCs, there has been some work into minimized network protocols for ultra-low latency. This is demonstrated in the eRPC [40] paper, where the authors seek to absolutely minimize the amount of time it takes to handle RPCs with a purely software solution. This is an interesting comparison to TAS in that it avoids hardware offload, but takes it a step further by doing away with TCP

guarantees in order to minimize latency. This idea of absolutely minimizing latency is taken a step further in nanoPU [33]. In this case, the CPU, NIC and transport protocols are all co-designed to minimize the latency of small messages. This involves a new, modified CPU and network card that are designed to allow for network messages to be delivered directly to CPU registers. This, combined with a minimal, low-latency network protocol allows for nanosecond latencies for data center RPCs. This approach, while providing super low latency, requires too many hardware modifications to be practical, but provides an interesting case study.

Google Snap Google Snap [51] is a network stack used by Google in production that shares the same service-based structure as TAS. Google Snap provides a more functionality and compatibility with different Google services that are network stack adjacent. This is one of the primary goals of Google Snap: to provide a highly modular network stack so that different components can be rapidly developed and updated. Snap and TAS share a similar architecture - dedicated cores for network stack processing that communicate with application libraries in a microkernel fashion - but differ in their motivations. TAS seeks to optimize TCP data path processing while Snap seeks to provide high levels of modularization.

Summary While some work has been done to improve software TCP solutions, many researchers have chosen to reduce TCP CPU overheads by integrating hardware acceleration. Specialized hardware can be a limiting factor

to growth and adoption of these solutions, as it can be expensive and finding the correct balance between performance and usability can be difficult. Otherwise, again, many solutions require some kind of specialized API or application modification that is undesirable to most developers.

5.2 Zero Copy IO Stack APIs

Zero-copy networked storage Reflex [43] 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 [80] proposes DMA hardware with data transformation logic to offload serialization. Breakfast of Champions [68] 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 [49] 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.

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 [31] provides an API that combines the network stack with persistent data structures in NVM to avoid copies. PASTE builds on the Netmap [71] 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 [84] 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 [20].

Chapter 6

Conclusion

Given the evidence of this dissertation, it is clear that there is both a strong need for accelerating IO-intensive applications and variety of solutions to do so. As applications move to be more data-driven, it is becoming more important to improve network and storage speeds, and it is important to remember that different classes of applications, different classes of IO, and different classes of hardware will have different bottlenecks that must be specifically addressed.

In TAS, I improve TCP software performance by bypassing the kernel, separating common case data path processing from uncommon control operations, and dedicating cores to packet processing. This streamlines TCP packet processing but keeps it purely in software.

In zIO, I track large IO data buffers and speculatively eliminate copies. I use unmapped pages to track and apply data modifications. This speculation is usually successful, but incurs overhead for handling page faults and repeated mappings and unmappings.

In general, it is safe to say that many of the claims I make in the motivation for this dissertation will continue to be true and become more pressing.

We cannot count on CPUs to become more powerful in the future. Data demands are going to continue to increase. Hence, both TAS and zIO have room to grow and exciting new angles that can be taken to further improve performance. In the next section, I will discuss some areas where we can look for future performance improvements to transparent IO acceleration, both in software and hardware.

6.1 Outlook

Two promising improvements to transparent IO acceleration are in improving software scalability and utilizing new types of hardware beyond the CPU to achieve common application or system functionality. Here I will discuss how TAS and zIO could be improved in these two areas: in software and in hardware.

6.1.1 The Future of Zero Copy IO

The work presented in zIO demonstrates that many copies are unnecessary and can be eliminated, as well as the fact that we can do this completely transparently to the applications. However, there is still some room for improvement with how these mechanisms are realized.

6.1.1.1 A Hardware Tracking Solution

One of the biggest potential improvements to zIO is in the mechanism for tracking and handling data modifications. The current system of unmap-

ping buffers and handling page faults has overheads for handling the faults and passing them in and out of user space, but more importantly creates TLB shootdowns that negatively affect performance, especially on multi-core applications.

In creating zIO, I explored both software and hardware solutions. Alternative software solutions included using tools like valgrind [55] and Intel's pin tool [48] to transparently instrument application code to check for write operations to tracked memory regions. This, in conjuction with x86 array bound check instructions could give us information about when specific memory regions are accessed, but I found there was too much overhead to interpose and do these checks on every application store instruction. Additionally, memory protection keys [60] could provide a useful interface for zIO to use but currently lack the functionality we want. For example, they are limited to 16 keys, which is not enough to handle the expected number of tracked intermediate buffers. Most importantly, they are currently not implemented in a way that optimizes for frequently changing keys. Memory protection keys currently trigger a system-wide TLB shootdown when adding or removing a new protected region—worse than just unmapping or remapping a buffer.

Existing hardware solutions provide some of the features we might want for an improved zIO tracking library, but are clearly not implemented to suit our purpose. A new hardware mechanism that is tailor-made for tracking and collecting accesses to large data buffers to drastically improve the overheads of our current zIO software solution would be required. An in-depth analysis of

the way that memory protection keys are implemented would be a start. The new mechanism should deliver a specific kind of fault, or better yet propagate any writes to a page to a different page (the original copy of the data). Additionally, we would need to be able to specify many protected regions in a way that does not generate TLB shootdowns.

6.1.1.2 Eliminate More Copies in Software

Software improvements for zIO could fall into two categories: reducing the overhead of tracking and eliminating copies, and expanding the scope of the copies that we can eliminate.

First, zIO has some overheads, previously described, that could be addressed with some software improvement. One example of this would be utilizing page alignments to improve zIO tracking efficiency and lower the number of page faults. By default, zIO is agnostic to application and memory manager page alignments and provides some simple mechanisms (fringe tracking and copying) to handle misalignment. Simply copying the fringe still provides a performance benefit for the applications we run, but can cause additional page faults when buffer alignments change. It also forces us to fully unmap buffers instead of mapping read-only, which causes additional page faults in applications that solely read intermediate IO buffers.

I have experimented with modified memory managers, specifically jemalloc [38], to always force page alignment of allocations of certain size. This is typically already done for buffers of 1MB and larger, but we lower this

threshold to the zIO size threshold (16KB). By doing this, I found in Redis that dynamically allocated buffers would all share the same alignment. This alignment could allow us to map dynamic buffers as read-only instead of unmapped, but I did not pursue this further for two reasons: first, most of Redis' buffers ended up being statically allocated, which would be harder to align, and second, even if we map the dynamically allocated buffers read-only, Redis does not access them at all and would not see a performance benefit. However, this approach could still be generally applied and may work for applications that read large quantities of aligned buffers (for example, MongoDB checksumming).

Second, while zIO should eliminate most application IO copies, it does not encompass all copies. All memcpy calls have a source and destination, and thus all chains of memcpys have some original source and final destination. At the moment, those sources and sinks are limited to IO read and write operations as those are both common and easy to transparently interpose on. I focused on these options as they were both the most common cases and easy to interpose on. However, we could expand the scope of zIO to encompass more.

I will primarily consider alternative data sources. Large data buffers could be generated in memory for a machine learning workload, or data could be stored in an in-memory key-value store in a way that zIO cannot track—for example, data that is read from network or storage and becomes cold, so it is evicted by zIO, or data that is pre-loaded from a file in a way that zIO doesn't

properly handle. In these cases, the data could still be copied and eventually end up at an IO stack, but we don't have a way to transparently identify the source of the memcpy chain. There are two easy but likely suboptimal solutions: first, offer a non-transparent call into the zIO library to specify a buffer to be tracked. This option would require application modification to fully realize, which is something we want to avoid for many previously discussed reasons. The second option would be speculatively start zIO tracking on all memcpy calls for buffers that meet the size threshold. This would drastically expand the scope of zIO to include any copy in the process of a certain size. While I have not explored this option, I believe this would likely conflict with one of the core observations of the zIO motivation: that most application data buffers are not touched but simply copied from IO stack to IO stack. By drastically expanding the scope of zIO, we risk unmapping buffers that do need to be accessed, which could create a large number of page faults and prevent copy optimization. However, if we have a more efficient solution for tracking data and handling page faults, such as the previously discussed hardware solution, this might be a manageable overhead for some applications.

6.1.2 The Future of Network Intensive Applications

Of course, if we can't make CPUs faster, we can try to just add more CPUs. Doing so works well for workloads that are already highly-parallelizable like packet processing and machine learning, but what about operating systems? Having a centralized operating system becomes harder with more

CPUs and this path has been explored thoroughly in research systems like Barrelfish [10]. However, it you can only optimize so far before it becomes untenable to try and run a traditional operating system. Disaggregated operating systems like [74] propose a new way of providing hardware resources to applications that could be more scalable, but introduces a number of new challenges.

Especially for network stack code, there are a number of new ways that silicon is being utilized to provide application and system functionality without consuming CPU cycles. Programmable network cards, FPGAs, and ASICs are all being used to improve network stack performance and have been discussed to different extents already. In general, there is a difficult balance between programmability, performance, and commoditization of the hardware that could be explored further. Systems like FlexTOE [75] have demonstrated the value of reworking TCP operations to be more parallelizeable and thus more adaptable to hardware acceleration. In doing this, TCP functionality can be moved from the CPU, running something like TAS, and instead be performed on the network card. Further, we can utilize programmable hardware that is already readily available in programmable switches. Many programmable switches use re-configurable match and action tables, which would require implementing TCP in a different way, but could continue to offload some processing from the CPU.

Bibliography

[1] https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_input.c#L5302.

[2] rsocket(7) - linux man page. https://linux.die.net/man/7/rsocket.

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

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

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

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review*, 40(4):63–74, August 2010.

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

[8] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *17th USENIX Symposium on Net-*

worked Systems Design and Implementation (NSDI 20), pages 93–109, Santa Clara, CA, February 2020. USENIX Association.

[9] Brian W. Barrett, Ron Brightwell, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabee, and Trammell Hudson. *The Portals 4.0.1 Network Programming Interface*. Sandia National Laboratories, sand2013-3181 edition, April 2013.

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. SOSP '09.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. OSDI '14.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, 2014.

[13] Nathan L. Binkert, Ali G. Saidi, and Steven K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. ASPLOS '06.

[14] 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.

[15] Chelsio Communications. T6 ASIC: High performance, dual port uni ed wire 1/10/25/40/50/100Gb Ethernet controller. https://www.chelsio.com/wp-content/uploads/resources/Chelsio-Terminator-6-Brief.pdf , 2017.

[16] Kristina Chodorow. MongoDB: the de nitive guide: powerful and scalable data storage " O'Reilly Media, Inc.", 2013.

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

[18] 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, SoCC '10, pages 143{154, Indianapolis, Indiana, 2010. ACM.

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

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

[21] Andy Currid. TCP o oad to the rescue: Getting a toehold on TCP o oad engines|and why we need them. Queue, 2(3):58{65, May 2004.

[22] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. ACM Trans. Storage, 17(4), oct 2021.

[23] Peter Druschel, Larry Peterson, and Bruce Davie. Experiences with a high-speed network adaptor: A software perspective. SIGCOMM '94.

[24] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. USENIX ATC '13.

[25] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and exibility in distribution of hot content. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) pages 1{15, Carlsbad, CA, July 2022. USENIX Association.

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. ASPLOS '19, page 3{18, New York, NY, USA, 2019. Association for Computing Machinery.

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

[28] Google. grpc, 2016https://grpc.io .

[29] Google. LevelDB, 2022https://github.com/google/leveldb .

[30] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 12, pages 135{148, Hollywood, CA, October 2012. USENIX Association.

[31] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. Paste: A network programming interface for non-volatile main memory. InProceedings of the 15th USENIX Conference on Networked Systems Design and Implementation, NSDI'18, page 17{33, USA, 2018. USENIX Association.

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

[33] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 239{256. USENIX Association, July 2021.

[34] Vesoft inc. Nebula graph, 2019https://nebula-graph.io/ .

[35] V. Jacobson. Congestion avoidance and control. 18(4):314{329, August 1988.

[36] Van Jacobson. Tcp in 30 instructions. http://www.pdl.cmu.edu/mailinglists/ips/mail/msg00133.html .

[37] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response work ows. SIGCOMM '13.

[38] jemalloc. jemalloc, 2021. https://jemalloc.net/ .

[39] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI'14, pages 489{502, 2014.

[40] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI'19, pages 1{16, 2019.

[41] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. ASPLOS '16.

[42] 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 Fourteenth EuroSys Conference 2019, EuroSys'19, pages 1–16, 2019.

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

[44] 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, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Redis Labs. Redis, 2021. https://redis.io/ .

[46] 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, SIGCOMM '19, page 90–103, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel TCP design and implementation for short-lived connections. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, page 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geo Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. SIGPLAN Not., 40(6):190{200, jun 2005.

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

[50] 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.

[51] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP'19, pages 399{413, 2019.

[52] memcached. Memcached, 2020. https://memcached.org/ .

[53] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall,

and David Zats. TIMELY: RTT-based congestion control for the datacenter. ACM SIGCOMM Computer Communication Review, 45(4):537{550, 2015.

[54] Akshay Narayan, Frank J. Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. The Case for Moving Congestion Control Out of the Datapath. In Sixteenth ACM Workshop on Hot Topics in Networks (HotNets), Palo Alto, CA, November 2017.

[55] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not., 42(6):89{100, jun 2007.

[56] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Sta ord, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. NSDI '13.

[57] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsa , and Boris Grot. Scale-out numa. SIGPLAN Not., 49(4):3{18, feb 2014.

[58] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensor ow-serving: Flexible, high-performance ml servingarXiv preprint arXiv:1712.06139, 2017.

[59] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A uni ed i/o bu ering and caching system. ACM Trans. Comput. Syst., 18(1):37{66, February 2000.

[60] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software abstraction for intel memory protection keys (intel mpk). In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '19, page 241{254, USA, 2019. USENIX Association.

[61] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. Eurosys '12.

[62] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, 2014.

[63] 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.

[64] I. Pratt and K. Fraser. Arsenic: a user-accessible gigabit ethernet interface. In Proceedings IEEE INFOCOM 2001. Conference on Com-

puter Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213), volume 1, pages 67–76 vol.1, 2001.

[65] Ian Pratt and Keir Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. INFOCOM '01.

[66] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[67] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for end-host rate limiting. NSDI '14.

[68] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with nic scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 199–205, New York, NY, USA, 2021. Association for Computing Machinery.

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

[70] Rick Reed. Scaling to millions of simultaneous connections, March 2012. `http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf`.

[71] Luigi Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9, USA, 2012. USENIX Association.

[72] Mihai Rotaru. Scaling to 12 million concurrent connections: How MigratoryData did it, October 2013. https://mrotaru.wordpress.com/2013/10/10/scaling-to-12-million-concurrent-connections-how-migratorydata-did-it/.

[73] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. 29(5):71–78, October 1999.

[74] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[75] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.

[76] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie

Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM Computer Communication Review*, volume 59, pages 88–97, 2015.

[77] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[78] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 40–53, New York, NY, USA, 1995. Association for Computing Machinery.

[79] 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*, OSDI '06, page 307–320, USA, 2006. USENIX Association.

[80] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

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

[82] xiph. Icecast, 2021. `https://icecast.org/`.

[83] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. Stackmap: Low-latency networking with the os stack and dedicated nics. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 43–56, Berkeley, CA, USA, 2016. USENIX Association.

[84] 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 ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.