StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata, *Keio University;* Michio Honda, Douglas Santry, and Lars Eggert, *NetApp*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIC ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata[†][1], Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.*, Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

[†]Most of the research was done during an internship at NetApp.

Another group of recent proposals [16, 22, 46] attempts to reduce the overheads associated with using the kernel stack by optimizing the socket API in a variety of ways. Unlike kernel-bypass approaches, which dedicate network interfaces (NICs) to individual applications, they continue to allow different applications to share NICs. This limits the benefit of this set of proposals, due to the continued need for kernel mechanisms to arbitrate NIC access, and motivates our StackMap proposal.

On today's multi-core and multi-NIC servers, it is common practice to dedicate individual cores and individual (virtual) interfaces to individual applications; even more so for distributed scale-out applications that use all system resources across many systems.

This paper presents StackMap, a new OS network service that dedicates NICs to individual applications (similar to kernel-bypass approaches) but continues to use the full-fledged kernel TCP/IP stack (similar to API-optimizing approaches). StackMap establishes low-latency, zero-copy data paths from dedicated NICs, through the kernel TCP/IP implementation, across an extended version of the netmap API into StackMap-aware applications. The extended netmap API features operations for explicit execution of packet I/O and TCP/IP processing, event multiplexing and application-data I/O, which is tightly coupled with an abstraction of the NIC packet buffers. Alongside the data path, StackMap also retains the regular socket API as a control path, which allows sharing of the OS network stack properly with regular applications.

StackMap outperforms Linux by 4 to 80 % in average latency, 2 to 70 % in 99th-percentile latency and 4 to 391 % in throughput. StackMap also improves memcached throughput by 42 to 133 %, and average latency by 30 to 58 %. Up to six CPU cores, StackMap even outperforms memcached on Seastar, which is a highly-optimized, user-level TCP/IP stack for DPDK.

The StackMap architecture resembles existing kernel-bypass proposals [4, 6] and borrows common techniques from them, such as batching, lightweight buffer management and the introduction of new APIs. However, applying these techniques to an OS network stack requires a significant design effort, because of its shared nature. Additionally, the kernel TCP/IP implementation depends heavily on other components of the network stack and other kernel subsystems, such as the memory allocator, packet I/O subsystem and socket API layer. StackMap carefully decomposes the OS network stack into its TCP/IP implementation and other components, and optimizes the latter for transactional workloads over dedicated NICs.

StackMap inherits the complete set of the rich TCP/IP protocol features from the OS stack. A latency analysis of the Linux kernel implementation shows that TCP/IP processing for both the transmit and receive directions consumes less than 0.8 µs, respectively. This is less than the half the overall request handling latency of 3.75 µs (see Section 3.1).

In summary, we make three major contributions: (1) a latency analysis of the Linux kernel TCP/IP implementation, (2) the design and implementation of StackMap, a new low-latency OS networking service, which utilizes dedicated NICs together with the kernel TCP/IP implementation, and (3) a performance evaluation of StackMap and a comparison against a kernel-bypass approach.

The remainder of this paper is organized as follows: Section 2 discusses the most relevant prior work, much of which StackMap draws motivation from. Section 3 analyses latencies of the Linux network stack, in order to validate the feasibility of the StackMap approach and identify problems we must address. Section 4 describes the design and implementation of StackMap. Section 5 evaluates StackMap. Section 6 discusses the inherent limitations posed by the StackMap architecture as well as those only present in the current prototype implementation. The paper concludes with Section 7.

2 Motivation and Related Work

Inefficiencies in OS network stacks are a well-studied topic. This section briefly discusses the most relevant prior work in the field from which StackMap draws motivation, and summarizes other related work.

2.1 Kernel-Bypass Networking

An OS typically stores network payloads in kernel-space buffers, accompanied by a sizable amount of metadata (*e.g.*, pointers to NICs, socket and protocol headers). These buffers are dynamically allocated and managed by reference counts, so that producers (*e.g.*, a NIC driver in interrupt context) and consumers (*e.g.*, `read()` in syscall context) can operate on them [55].

Around 2010–2012, researchers developed approaches to use static, pre-allocated buffers with a minimum amount of metadata for common, simple packet operations, such as packet forwarding after IPv4 longest-prefix matching [62]. A key architectural feature of these systems [21, 54] is to perform all packet processing in user space[1], by moving packets directly to and from the NIC, bypassing the OS stack. These systems also extensively exploit batching, *e.g.*, for syscalls and device access, which is preceded by a write barrier. The outcome of these research proposals is general frameworks for fast, user-space packet I/O, such as netmap [54] and DPDK [26].

[1]Click [43], proposed in 1999, has an option to run in user-space, but as a low-performance alternative to the in-kernel default, due to using the traditional packet I/O API or Berkeley Packet Filter. A netmap-enabled version of Click from 2012 outperforms the in-kernel version [52].

In 2013–2014, “kernel-bypass” TCP stacks based on these user-space networking frameworks emerged, driven by a desire to further improve the performance of transactional workloads. UTCP [24] and Sandstorm [37] build on netmap; mTCP [29] on PacketShader; Seastar [6], IX [4] and UNS [27] on DPDK. They either re-implement TCP/IP almost from scratch or rely on existing—often similarly limited or outdated—user-space stacks such as lwIP [12], mostly because of assumed inefficiencies in OS stacks and reported inefficiencies of running more modern OS stack code in user-space via shims [31, 60].

Further, these kernel-bypass stacks introduce new APIs to avoid inefficiencies in the socket API, shared-nothing designs to avoid locks, active NIC polling to avoid handling interrupts, network stack processing in application threads to avoid context switches and synchronization, and/or direct access to NIC packet buffers. These techniques become feasible, because kernel-bypass approaches dedicate NICs or packet I/O primitives, such as a NIC rings, to application threads. In other words, they do not support sharing NICs with applications that use the OS stack. Each thread therefore executes its own network stack instance, and handles all packet I/O, network protocol processing in addition to executing the application logic.

However, such techniques are not inherently limited to user-space use. In 2015, mSwitch [23] demonstrated that they can be incorporated into kernel code (with small modifications) and result in similar benefits: mSwitch speeds up the Open vSwitch data path by a factor of three. It retains the OS packet representation, but simplifies allocation and deallocation procedures by performing all packet forwarding operations within the scope of a single function. StackMap borrows the idea of using acceleration techniques first used for kernel-bypass networking inside the kernel from mSwitch.

2.2 Network API Enhancements

Server applications typically execute an event loop to monitor file descriptors, including network connections. When an event multiplexing syscall such as `epoll_wait()` returns “ready” file descriptors, the application iterates over them, *e.g.*, to read requests, send responses, or to accept new connections.

In 2012, MegaPipe [22] improved on this common pattern by introducing a new API that featured two techniques: First, it batches syscall processing across multiple file descriptors (similarly to FlexSC [59]) to amortize their cost over a larger number of bytes. This is particularly effective for transactional workloads with small messages. Second, it introduces new *lightweight sockets*, which relax some semantics of regular sockets that limit multi-core scalability, such as assigning the lowest available integer to number a new descriptor (which requires a global lock for the entire process). Not surprisingly, this approach provides a smaller performance improvement compared to the kernel-bypass approaches that dedicate NICs to applications [29].

2.3 TCP Maintenance

Originally, TCP was a relatively simple protocol [48] with few extensions [28, 38], and the implementation and maintenance cost was very manageable. In addition to implementations by general-purpose OS vendors, many more specialized TCP implementations were undertaken to support purpose-specific appliances, such as middleboxes [25, 40], storage servers [15] and embedded systems [34].

Over the years, maintaining a TCP implementation has become much more challenging. TCP is being improved at a more rapid pace than ever, in terms of performance [1, 3, 39, 49, 57], security [5, 13, 32, 44, 51], as well as more substantial extensions such as Multipath TCP [18]. In addition to the sheer number of TCP improvements that stacks need to implement in order to remain competitive in terms of performance, security and features, the improvements themselves are becoming more complex. They need to take the realities of the modern Internet into account, such as the need for correct operation in the presence of a wide variety of middleboxes or the scarcity of available option space in the TCP header [5, 49, 50].

Consequently, the set of “modern” TCP stacks that offer the best currently achievable performance, security and features has been shrinking, and at the moment consists mostly of the large general-purpose OS stacks with a sizable developer base and interest. Many other TCP implementations have fallen behind, and it is very uncertain whether they will ever catch up. This is especially regrettable for stacks that underlie many deployed middleboxes, because they in turn limit future Internet evolution [25, 40]. The situation is unfortunately similar for many of the recent kernel-bypass stacks (see Section 2.1), none of which shows signs of very active maintenance. Networking products that wish to take advantage of the performance benefits of kernel-bypass solutions thus run the risk of adopting a stack that already is not competitive in terms of features, and may not not remain competitive in terms of performance in the future.

StackMap mitigates this risk, by offering similar performance benefits to kernel-bypass approaches while using the OS stack, which has proven to see active maintenance and regular updates.

2.4 Other Related Work

Some other pieces of relevant related work exist, in addition to the general areas discussed above.

In the area of latency analyses of network stacks, [33] analyses the latency breakdown of memcached, in order to understand how to fulfill the quality-of-service objectives of multiple workloads. Their measurements show much higher processing delays than ours, and their focus is not on improving the OS network service. Also, [4] reports a one-way latency of 24 µs with Linux TCP/IP, which is twice what we measure in Section 3. Our results are similar to those reported in [47], taking into account that they use UDP instead of TCP.

In the area of API extensions, many UNIX variants (including Linux) implement a `sendfile()` syscall that transmits a regular file from the buffer cache directly into a TCP socket without a memory copy or multiple context switches. The Linux `sendmmsg()` and `recvmmsg()` syscalls support passing multiple messages to and from the kernel at a time. However, they do not allow batching across different descriptors. In late 2015, the Linux kernel connection multiplexer (KCM) [7] was proposed. It enables applications to use message-based interfaces, such as `sendmmsg()`, over TCP and allows syscall batching across multiple TCP connections. Linux busy poll sockets [8] permit to directly poll a network device to avoid interrupts when receiving packets. Windows IOCP [42], the Linux `epoll` and BSD `kqueue` families of syscalls are event multiplexing APIs. All of these approaches are limited by needing to remain compatible with the semantics established by the socket API and to arbitrate NIC access, which incurs significant overheads (see Section 3).

Kernel-bypass network stacks introduce other techniques to optimize some TCP code paths, such as sorting TCP connections by timeout order or pre-allocating TCP protocol control blocks for fast connection setup. None of these techniques are inherently limited to kernel-bypass approaches, and StackMap will immediately gain their benefits once the kernel stack implements them. The same is true for Fastsocket [35], a recent optimization of the Linux stack for multi-core scalability—when Fastsocket functionality is present in the kernel, applications using StackMap will immediately gain its benefits.

3 Design Space Analysis

Unless a network hop becomes the bottleneck, the end-to-end latency of a transactional workload depends on two main factors: (1) the processing delays of the network stack and the application, and (2) the queueing latency, particularly in the presence of concurrent requests. This section analyzes these latency factors for the Linux kernel network stack, in order to determine the feasibility of using the kernel TCP/IP implementation for low-latency networking and to identify any challenges such an architecture must address.

Layer	Component	Time [µs]
Kernel	Driver RX	0.60
	Ethernet & IPv4 RX	0.19
	TCP RX	0.53
	Socket enqueue	0.06
Application	`epoll_wait()` syscall	0.15
	`read()` syscall	0.33
	Generate “OK” reply	0.48
	`write()` syscall	0.22
Kernel	TCP TX	0.70
	IPv4 & Ethernet TX	0.06
	Driver TX	0.43
Total		3.75

Table 1: Request processing overheads at a server.

3.1 TCP/IP Processing Cost

We start by analyzing a single, short-message request-response exchange (96 B “GET”, 127 B “OK”) between two Linux machines connected with 10 G Ethernet NICs (see Section 5 for configuration details). We use Systemtap [14] to measure processing delays in particular components of the server.

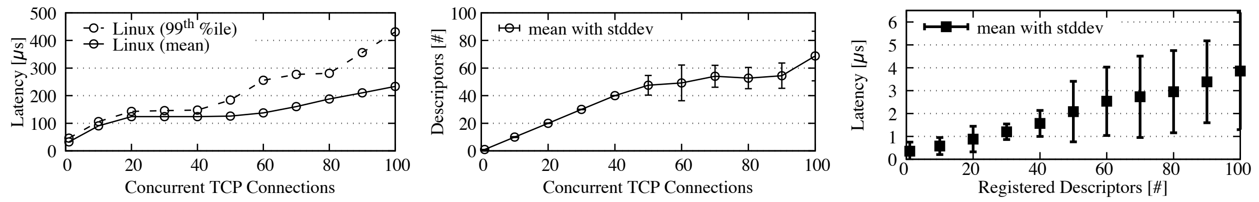
Table 1 shows the processing overheads measured at the various layers during this experiment. The key insight is that TCP/IP processing takes 0.72 µs on receive (Ethernet & IPv4 RX and TCP RX) and 0.76 µs on transmit (TCP TX and IPv4 & Ethernet TX). The combined overhead of 1.48 µs is not a large factor of the overall processing delay of 3.75 µs, and of the end-to-end one-way latency of 9.75 µs (half of the round-trip latency reported by `wrk`).

The significant difference of 6 µs between the end-to-end one-way latency and the processing delay of the server is due to link, PCIe bus and switch latencies (1.15 µs combined, one-way), and some indirection between the hardware and software, which is unavoidable even for kernel-bypass TCP/IPs. We confirm this finding by running a netmap-based ping-pong application between the same machines and NICs, which avoids most of the network-stack and application processing. The result is a one-way latency of 5.77 µs, which is reasonably similar to the 6 µs measured before.

Data copies do not appear to cause major overheads for short-message transactions. In this experiment, copying data only takes 0.01 and 0.06 µs for 127 and 1408 B of data, respectively (not shown in Table 1).

3.2 Latencies for Concurrent Connections

A busy server continually serves a large number of requests on many different TCP connections; with clients using potentially multiple parallel connections to the server to avoid head-of-line blocking. For new data arriving on connec-



(a) End-to-end latency. (b) Descriptors returned by `epoll_wait()`. (c) Processing delay of `epoll_wait()`.

Figure 1: Latency analysis of concurrent TCP connections.

tions (*i.e.*, new client requests), the kernel needs to mark the corresponding file descriptor as "ready". Applications typically run an event loop on each CPU core around an event multiplexing syscall such as `epoll_wait()` to identify any "ready" connections and serve the new batch of requests that has arrived on them. During application processing of one such batch of requests, queuing delays can occur in two ways: First, an application iterates over ready descriptors one after the other, serving their requests. When many requests arrive on many connections, requests on connections iterated over at the end of the cycle incur significant delays. Second, new requests arriving while an application is processing the current batch are queued in the kernel until the next cycle. This behavior increases both the mean and tail end-to-end latencies during such an event processing cycle, proportional to the number of descriptors processed during the cycle.

In order to demonstrate this effect and quantify its latency impact, we repeat the measurement from Section 3.1, but generate concurrent requests on several TCP connections. Figure 1a plots the mean (solid lines) and 99th-percentile (dashed lines) end-to-end latencies, as measured by the client. Figure 1b shows the measured mean number of descriptors returned by `epoll_wait()` at the server (with standard deviations). There is a clear correlation between the end-to-end latencies and the number of file descriptors returned, as the number of concurrent connections increases.

Concurrent connections also increase the processing delay of the `epoll_wait()` syscall, which iterates over all the descriptors registered in the kernel. Figure 1c quantifies this cost for different numbers of registered descriptors, using Systemtap to measure time spent in `sp_send_events_proc()`. The cost to iterate over ready descriptors is negligible when the number of registered descriptors is small. However, it reaches about 1 µs for 20 descriptors, and almost 4 µs for 100 descriptors, which is substantial.

Note that the processing delay for one registered descriptor in Figure 1c is slightly higher than that reported for `epoll_wait()` in Table 1, because the numbers reported there subtract the Systemtap overhead from the result (estimated by measuring the extremely cheap (`tcp_rcv_space_adjust()` function).

3.3 Takeaway

It is clear that compatibility with the socket API comes at a significant cost, including the overheads of `read()` and `write()` (Table 1) as well as `epoll_wait()` (Figure 1c). Packet I/O also introduces significant overheads (Driver TX and RX in Table 1), and more performance is lost due to the inability of batching transmissions over multiple TCP connections. With concurrent TCP connections, the overheads associated with request processing result in long average and tail latencies due to queueing delays, on the order of tens to hundreds of µs (Figure 1a).

4 StackMap Design

The discussion in the last two sections leads us to three starting points. First, there are a number of existing techniques to improve network stack efficiency. StackMap should incorporate as many of them as possible. Second, it must use an actively-maintained, modern TCP/IP implementation, *i.e.*, one of the main server OS stacks. And StackMap must use this stack in a way that lets it immediately benefit from future improvements to that code, without the need to manually port source code changes. This is important, so that applications using StackMap are not stuck with an outdated stack. Finally, while TCP/IP protocol processing in an OS stack is relatively cheap, StackMap must improve other overheads, most notably ones related to the API and packet I/O, in order to significantly reduce queueing latency in the presence of concurrent TCP connections.

4.1 StackMap Design Principles

StackMap dedicates NICs to privileged applications through a new API. We believe this is a reasonable principle for today's high-performance systems, and the same approach is already followed by kernel-bypass approaches (*netmap, DPDK*). However, unlike such kernel-bypass approaches, StackMap also "maps" the dedicated NICs into the kernel TCP/IP stack. This key differentiator results in key benefits, because many overheads of the traditional socket API and buffer management can be avoided.

A second design principle is retaining isolation: although StackMap assumes that applications are privileged (they see all traffic on dedicated NICs), it must still protect the OS and any other applications when a privileged StackMap application crashes. StackMap inherits most of its protection mechanisms from netmap, which it is based on, including protection of NIC registers and exclusive ring buffer operations between the kernel and user space. We discuss the limitations posed by our current prototype in Section 6.

A third design principle is backwards compatibility: when a NIC is dedicated to a StackMap application, regular applications must remain able to use other NICs. StackMap achieves this by retaining part of the socket API for control plane operations. Since today's commodity OSes mostly use monolithic kernels, StackMap must share a single network stack instance across all NICs, whether they are dedicated to privileged applications or shared by regular applications. One implication of this design principle is that it makes a complete shared-nothing design difficult, *i.e.*, some coordination remains required. However, Section 5 shows that this coordination overhead is small. Additionally, the OS stack is increasingly being disaggregated into shared objects, such as accept queues, and StackMap will benefit from such improvements directly, further improving future performance.

4.2 StackMap Architecture

The StackMap design centers around combining a fast packet I/O framework with the OS TCP/IP stack, to give application fast message-oriented communication over TCP connections, which has been crucial for the applications like memcached, web servers and content delivery network (CDN) servers, to name a few [7, 22]. Thus, in addition to dedicating NICs to privileged applications, StackMap must also enable the kernel stack to apply its regular TCP/IP processing to those NICs. To this end, StackMap extends the netmap framework to allow it to efficiently integrate with the OS stack.

DPDK is not a suitable basis for StackMap, because it executes its NIC drivers entirely in user space. It is difficult to efficiently have such user-space NIC drivers call into the kernel network stack.

Although netmap already supports communicating with the OS stack [54], its current method has significant overheads, because it is unoptimized and only focuses on applications that use the socket API, which as we have shown to have undesirable overheads.

Figure 2 illustrates the StackMap architecture. StackMap (i) mediates traffic between a dedicated NIC and a privileged application through a slightly extended version of the netmap API; (ii) uses the kernel TCP/IP stack to process incoming TCP packets and send outgoing

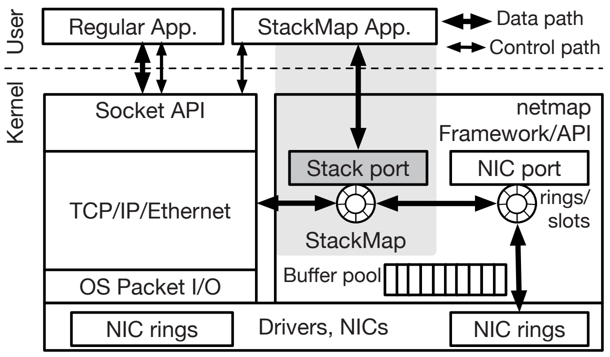


Figure 2: StackMap architecture overview.

application data; (iii) and uses the regular socket API for control, to both share the protocol/port-number space with regular applications and to present a consistent picture of clients and connections to the kernel.

For the data path API, the goal of StackMap is to combine the efficiency, generality and security of the netmap packet I/O framework with the full-fledged kernel TCP/IP implementation, to give applications a way to send and receive messages over TCP with much lower overheads compared to the socket API.

These performance benefits will require an application to more carefully manage and segment the data it is handling. However, we believe that this is an acceptable trade-off, at least for transactional applications that care about message latencies rather than bulk throughput. In addition, because StackMap retains the socket API for control purposes, an application can also still use the standard data plane syscalls *e.g.*, `read()` and `write()` (with the usual associated overheads).

4.3 Netmap Overview

Netmap [54] maintains several *pools* of uniquely-indexed, pre-allocated *packet buffers* inside the kernel. Some of these buffers are referenced by *slots*, which are contained in *rings*. A set of rings forms a *port*. A NIC port maps its rings to NIC hardware rings for direct packet buffer access (Figure 2). A *pipe* port provides a zero-copy point-to-point IPC channel [53]; a *VALE* port is a virtual NIC of a VALE/mSwitch [23, 56] software switch instance (not shown in Figure 2).

The netmap *API* provides a common interface to all types of ports. It defines methods to manipulate rings and uses `poll()` and `ioctl()` syscalls for synchronization with the kernel, whose backend performs port-specific operations, *e.g.*, device I/O for NIC ports or packet forwarding for VALE ports. Netmap buffers that are part of the same pool are interchangeable between slots, even across different rings or ports, which enables zero-copy operations.

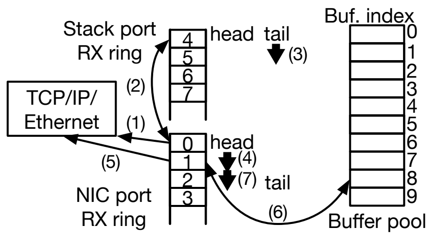


Figure 3: TCP in-order and out-of-order reception. Initially, buffers 0–7 are linked to either of the rings, and 8–9 are extra buffers.

When a netmap client (*i.e.*, an application) registers a port, netmap exposes the corresponding rings by mapping them into its address space. Rings can be manipulated via *head* and *tail* pointers, which indicate the beginning and end of the user-owned region. A client fills a TX ring with new packets from head and advances it accordingly. On the next synchronization syscall, netmap transmits packets from the original head to the current head. During a packet reception syscall, netmap places new packets onto the ring between head and tail; the client then consumes these packets by traversing the ring from head to tail, again advancing the head accordingly.

4.4 StackMap Data Path

StackMap implements a new type of netmap port, called a *stack port*. A stack port runs in cooperation with a NIC port, and allocates packet buffers for its rings from the same pool. StackMap also uses buffers that are not linked to any slot in the same pool as the NIC and stack port. These buffers allow StackMap to retain packets in netmap buffers, but outside NIC or the stack port rings, which prevents them from being processed by ring operations. This is useful, for example, to store packets that may require retransmission or to maintain data that was received out-of-order. To utilize multiple CPU cores efficiently, for each core, one stack port ring and NIC port ring should be configured.

A ring belonging to a stack port can handle multiple TCP connections, which are exposed to the application through regular descriptors. This is essential for syscall *and* packet I/O batching over these connections. Thus, an application (on TX) or the kernel (on RX) indicates a corresponding connection or file descriptor for each slot. We later explain how this can be used by an application to process an entire RX ring for a particular descriptor, rather than looking up it for every packet.

Stack ports implement their own netmap syscall backend, so they can provide the netmap API to an application. Their TX and RX backends start StackMap data path processing. When a StackMap application performs a syscall for RX, the stack port backend first brings new packets into an RX ring of its NIC port, then instructs the OS stack to run them through its RX path. StackMap then moves any packets that the OS stack identifies as in-order TCP segments for their respective connections into an RX ring of its stack port. Out-of-order TCP segments are moved into the extra buffer space; They are delivered to the stack port RX ring when the out-of-order hole is filled. All buffer movements are performed by swapping indices, *i.e.*, zero-copy. Figure 3 illustrates these steps using two packets, where the second one arrives out-of-order. Step (1) and (5) process a packet in the TCP/IP stack. StackMap acts as a netmap "client" for a NIC port, thereby advancing the head pointer of its RX ring to consume packets. Conversely, StackMap acts as a netmap "backend" for a stack port. It thus advances the tail pointer of its RX ring when it puts new data into buffers.

The TX path is the opposite. When an application wishes to transmit new data located in buffers of a stack port TX ring, StackMap pushes these buffers through the TX path of the OS stack (see Table 1 for the overheads of those steps; in TCP TX StackMap skips packet buffer allocation, and so is somewhat faster). StackMap then intercepts the same packets after the Ethernet TX processing and moves the buffers into a TX ring of the NIC port (again zero-copy by swapping buffer indices). StackMap then advances the head of the NIC port ring and the old head of the stack port ring and triggers the NIC port for transmission. Since the OS stack would normally keep these TCP packets in its retransmission queue, StackMap unlinks the packet buffers after transmission.

If the size of data in a given TCP connection exceeds the available window of the respective connection *i.e.,* the minimum of the advertised receive window and the congestion window computed by the TCP stack, StackMap swaps the excess buffers out of the stack port TX ring, in order to avoid stalling the transmission on other connections. Any such buffers are moved back to the NIC port ring during the next operation.

To pass netmap buffers to the OS stack, a stack port pre-allocates a *persistent* `sk_buff` for each netmap buffer, which is the internal OS packet representation structure. This approach allows StackMap to avoid dynamic allocation and deallocation of `sk_buffs`, which has been shown to significantly reduce packet processing overheads [23].

In addition to being clocked by the stream of inbound acknowledgments (ACKs), the TCP protocol also has some inherent timers, *e.g.*, the retransmission timeout (RTO). StackMap processes any pending TCP timer events only on the TX or RX syscalls. This preserves the synchronization model of the netmap API between the kernel and user space, protecting buffers from concurrent access by a timer handler and user space code.

```
struct sockaddr_in sin = { AF_INET, "10.0.0.1", INADDR_ANY };
int sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(sd, &sin);
// prefix "stack" opens stack port for given interface
struct nm_desc *nmd = nm_open("stack:ix0");
connect(sd, dst_addr); /* non-blocking */
// transmit using ring 0 only, for this example
struct netmap_ring *ring = NETMAP_TXRING(nmd->nifp, 0);
uint32_t cur = ring->cur;
while (app_has_data && cur != ring->tail) {
    struct netmap_slot *slot = &ring->slot[cur];
    char *buf = STACKMAP_BUF(ring, slot->buf_index);
    // place payload in buf, then
    slot->fd = sd;
    cur = nm_ring_next(ring, cur);
}
ring->head = ring->cur = cur;
ioctl(nmd->fd, NIOCTXSYNC);
```

Figure 4: Initiating a connection and sending data.

4.5 StackMap API

In order to share the kernel TCP/IP stack with regular applications, StackMap retains the socket API for control, including, *e.g.*, the `socket()`, `bind()`, `listen()` and `accept()` syscalls. To reduce connection setup costs, StackMap optionally can perform `accept()` in the kernel before returning to user space, similar to MegaPipe [22].

The StackMap data path API has been designed to resemble the netmap API, except for a few extensions. One is the `STACKMAP_BUF(slot_idx, ring)` macro, which extends the netmap `NETMAP_BUF(slot_idx, ring)` macro and returns a pointer to the beginning of the payload data in a packet buffer. `STACKMAP_BUF` allows an application to easily write and read payload data to and from buffers, skipping the packet headers (which on TX are filled in by the kernel).

On TX, the application must indicate the descriptor for each slot to be transmitted, so that the OS stack can identify the respective TCP connections. On RX, the OS stack marks the slots accordingly, so that the application can identify which connection the data belongs to. Figure 4 illustrates use of the StackMap API for opening a new TCP connection and sending data in C-like pseudo code.

On RX, an application can consume data by simply traversing the RX ring of a stack port. However, this simple approach often does not integrate naturally into existing applications, because they are written to iterate over descriptors or connections, rather than iterating over data in packet arrival order. Unfortunately, using the `epoll_wait()` syscall is not an option because of significant overheads shown in Section 3.2.

StackMap thus introduces a new API that allows applications to consume data more naturally, ordered by descriptor. It is based on constructing a list of ready file descriptors during network stack processing, as well as grouping buffers for each descriptor, and by exploiting the opportunity that the application *synchronously* calls into the kernel network stack.

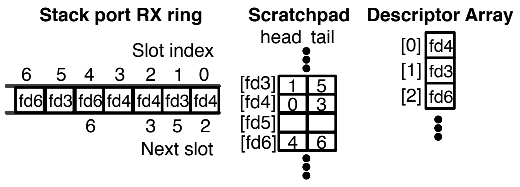


Figure 5: Algorithm to build an array of ready descriptors.

Figure 5 illustrates this algorithm with an example. Here, the OS stack has identified packets 0, 2 and 3 to belong to file descriptor 4 (`fd4`), packets 1 and 5 belong to `fd3`, and packets 4 and 6 to `fd6`. Note that descriptors are guaranteed to be unique per process. Each slot maintains a "next slot" index that points to the next slot of the same descriptor. StackMap uses a *scratchpad* table indexed by descriptor to maintain the head and tail slot index for each. The tail is used by the OS stack to append new data to a particular descriptor, by setting the "next slot" of the last packet to this descriptor, without having to traverse the ring. The head is used by the application to find the first packet for a particular descriptor, also without having to traverse the ring. This algorithm is inspired by mSwitch [23], and we expect similarly high performance and scalability.

The scratchpad is a process-wide data structure which requires 32 B (for two 16 B buffer indices) per entry, and usually holds 1024 entries (the default per-process maximum number of descriptors in Linux). We do not consider the resulting size of 32 KB problematic on today's systems even if it was extended by one or two orders of magnitude, but it would be possible to reduce this further by dynamically managing the scratchpad (which would incur some modest overhead).

When the first data for a particular descriptor is appended, StackMap also places the descriptor into a *descriptor array* (see Figure 5) that is exposed to the application. The application uses this array very similarly to how it would use the array returned by `epoll_wait()`, but without incurring the overhead of that syscall. Figure 6 illustrates how an application receives data and traverses the RX ring by descriptor.

The current API allows an application to traverse an RX ring both in packet-arrival order (*i.e.*, without using the descriptor array) and in descriptor order. In the future, StackMap may sort buffers in descriptor order when moving them into the stack port RX ring. This would remove the ability to traverse in packet order, but greatly simplifies the API and eliminates the need for exporting the descriptor array and scratchpad to user space.

4.6 StackMap Implementation

In order to validate the StackMap architecture, we implemented it in the Linux 4.2 kernel with netmap support.

```
// this example reuses (and omits) some of the definitions from Figure 4
ioctl(nmd->fd, NIOCRXSYNC);
uint32_t cur = ring->cur;
uint32_t count = 0;
for (uint32_t i = 0; i < nfds; i++) {
   const int fd = fdarray[i];
   for (uint32_t slot_idx = scratchpad[fd]; slot_idx != FD_NULL) {
      struct netmap_slot *slot = &ring->slots[slot_idx];
      char *buf = STACKMAP_BUF(slot->buf, ring);
      // consume data in buf
      slot_idx = slot->next_slot;
      ++count;
   }
}
// we have consumed all the data
cur = cur + count;
if (cur > ring->num_slots)
   cur -= ring->num_slots;
ring->cur = ring->head = cur;
```

Figure 6: Traversing the RX ring by descriptor on receive.

To pass packets from a NIC port RX ring to the OS stack, StackMap calls `netif_receive_skb()`, and intercepts packets after this function. To pass packets from a stack port TX ring to the OS stack, StackMap calls `__tcp_push_pending_frames()` after executing some code duplicated from static functions in `tcp.c`. Packetized data is intercepted at the driver transmit routine (`ndo_start_xmit()` callback). Also, StackMap modifies the `sk_buff` destructor, TCP timer handlers and connection setup routines in Linux, and implements StackMap-specific extensions to netmap, such as `sk_buff` pre-allocation and APIs.

All in all, StackMap modifies 56 LoC (lines of code) in the netmap code, and adds 2269 LoC in a new file. In the Linux kernel, 221 LoC are added and 7 LoC are removed across 14 existing and 2 new files.

5 Experimental Evaluation

This section presents the results of a series of experiments that analyze the latency and throughput that StackMap achieves in comparison to the kernel stack used via the socket API, in order to validate our design decisions. As discussed in Section 3, StackMap focuses on improving transactional workloads with small messages and many concurrent TCP connections.

Sections 5.2 and 5.3 measure by how much StackMap reduces processing delays and queueing latencies. The experiments use a minimal HTTP server, to highlight the performance differences between StackMap and the Linux kernel.

Sections 5.4 and 5.5 measure how well StackMap performs for a realistic application (memcached), and how it competes against a Seastar [6], a highly optimized, production-quality user-space TCP/IP stack for DPDK.

5.1 Testbed Setup

The experiments use two identical Fujitsu PRIMERGY RX300 servers equipped with a 10-core Intel Xeon E5-2680 v2 CPU clocked at 2.8 GHz (3.6 GHz with Turbo Boost) and 64 GB of RAM. One machine acts as the server, the other as the client; they are connected via an Arista DCS-7050QX switch and 10 Gbit Ethernet NICs using the Intel 82599ES chipset. The multi-core experiments in Section 5.5 use two additional, similar machines connected to the same switch, to saturate the server.

The server machine runs either an unmodified Linux 4.2 kernel, our StackMap implementation (see Section 4.6) or the Seastar user-level TCP/IP stack. For all experiments involving HTTP workloads, the server executes a minimal HTTP server that uses either the socket or StackMap APIs. In the former case, the HTTP server runs an `epoll_wait()` event loop and iterates over the returned descriptors. For each returned descriptor, it (1) fetches events using `epoll_wait()`, (2) `read()`s the client request, (3) matches the first four bytes against "GET␣", (4) copies a pre-generated, static HTTP response into a response buffer and (5) `write()`s it to the descriptor. In the latter case (using the StackMap API), the HTTP server runs an `ioctl()` event loop, as described in Section 4.5 with the same application logic *i.e.*, (3) and (4). For all experiments involving memcached workloads, the server runs a memcached instance. For Linux, we use memcached [41] version 1.4.24, and for StackMap, we ported the same version of it, which required 1151 LoC of modifications.

The client always runs a standard Linux 4.2 kernel. To saturate the HTTP server, it runs the `wrk` [19] HTTP benchmark tool. `wrk` initiates a given number of TCP connections and continuously sends HTTP GETs over them, measuring the time until a corresponding HTTP OK is received. Each connection has a single outstanding HTTP GET at any given time (`wrk` does not pipeline); open connections are reused for future requests. In the experiments involving memcached, the client executes the `memaslap` [36] tool.

Except for Section 5.5, the server uses only a single CPU core to serve requests, because in these first experiments, we are interested in how well StackMap solves the problems described in Section 3. The client uses all its 10 CPU cores with receive-side-scaling (RSS) to efficiently steer traffic towards them. Unless otherwise stated, the experiments enable all hardware/software offload facilities for the experiments that use the socket API. For StackMap, such offloads are *disabled*, because netmap at the moment does not support them. Once such support is added to netmap, we expect StackMap to directly benefit from these improvements.

Configuration	64 B	512 B	1280 B	memcached
Linux	32.7	46.4	68.4	52
	$\sigma = 5.0$	$\sigma = 3.7$	$\sigma = 4.1$	$\sigma = 9.7$
Linux-NIM	19.5	21.5	23.7	26
	$\sigma = 2.5$	$\sigma = 2.8$	$\sigma = 3.2$	$\sigma = 7.5$
StackMap	18.6	20.6	22.7	23
	$\sigma = 2.8$	$\sigma = 2.9$	$\sigma = 3.2$	$\sigma = 5.9$

Table 2: Mean roundtrip latencies in µs with standard deviations σ for different response message sizes in a granularity `wrk` or `memaslap` reports.

5.2 Processing Delay

The first experiment highlights the baseline latency of StackMap, *i.e.*, for a single message exchange without concurrency and therefore without any queueing.

Table 2 shows the mean latencies and their standard deviations for single request-response exchanges as measured by `wrk`, as well as with the `memaslap` memcached client. With `wrk`, the request message is always 96 B, and the response size varies between 64 B, 512 B and 1280 B, plus a 63 to 65 B HTTP header in each case. The memcached workload is described in Section 5.4. For the regular Linux stack, Table 2 reports two measurements. "Linux", where the NIC interrupt moderation period has been set to 1 µs (which is the Linux default), and "Linux-NIM", where interrupt moderation has been disabled. The "Linux-NIM" 64 B response size measurements were also used as the basis for the latency drill-down in Table 1 in Section 3.1.

StackMap achieves latencies that are better than Linux-NIM by 0.9 to 1 µs, which may seem minor. However, this is in fact a significant result, because both StackMap and Linux share most of the network protocol logic (Ethernet & IPv4 RX, TCP RX, TCP TX and IPv4 & Ethernet TX in Table 1) as well as the application logic (to generate "OK" replies). The StackMap latency improvement is a result of replacing the driver RX and TX operations with netmap, eliminating the socket enqueue and `epoll_wait()` operations, and replacing `read()` and `write()` with shared memory accesses by bypassing the `vfs` layer. Since these replaced or eliminated parts take 1.79 µs, this result in fact means that StackMap eliminates half of this processing overhead.

Note that for this experiment, busy-waiting on the NIC should not contribute to the latency reduction. Since Linux-NIM busy-waits on `epoll_wait()` to prevent its thread from sleeping and to avoid the thread wakeup latency, handling an interrupt (entirely done on the kernel stack of the current thread) is very cheap. We confirmed this by running netmap between the same machines, using busy-wait on either the NIC or a `epoll_wait()` descriptor. The round-trip latencies are 11.70 and 11.53 µs, respectively. This demonstrates that noticing a new packet by handling an interrupt in an active thread is slightly cheaper than through an explicit device access.

Also note that for Linux-NIM and StackMap, the latency differences between different message sizes do not result from data copies at the server or client, but are due to two 10 Gbit Ethernet and four 16 Gbit PCIe bus traversals for each of the packets, which approximately translates into 4 µs for a 1.2 KB size difference.

The "Linux" configuration is used in the rest of this section, and was used for experiments in Section 3.2. While this configuration exhibits higher latencies for a small number of concurrent connections, it achieves 28 to 78 % higher throughput and 22 to 44 % lower average latencies than Linux-NIM with 40 or more concurrent connections (not shown in the graphs).

5.3 Queueing Latency

This section evaluates by how much StackMap can reduce transaction latencies in the presence of concurrent connections (*i.e.*, transactions) compared to a system that uses the regular socket API and packet I/O methods (which were identified as inefficient in Sections 3.1 and 3.2). Recall that larger numbers of registered descriptors increase latencies because of queueing, particularly tail latencies.

The top row of graphs in Figure 7 compares the mean and 99th-percentile latencies of StackMap and Linux. The middle row compares the mean numbers of ready descriptors returned during each event processing cycle (`ioctl()` in StackMap and `epoll_wait()` in Linux), together with their standard deviations. The bottom row shows aggregate throughputs across all connections, to validate that the improved latency does not stem from a reduction in throughput. Each column shows results for the same response size (64, 512 and 1280 B).

The results in Figure 7 match our expectations. Because StackMap reduces per-message processing delays, it can serve the same number of descriptors at a lower latency than Linux. This faster turnaround time leads to fewer descriptors that need to queue for the next round of event processing, as reflected by the lower numbers of returned ready descriptors for StackMap (middle row). As a result, StackMap increasingly outperforms Linux as the number of concurrent connections increases.

5.4 Memcached Performance

After validating that StackMap outperforms Linux for a simple application, this section evaluates how StackMap performs for a realistic application. We also compare StackMap against memcached on Seastar [6], a highly-optimized user-space TCP/IP stack for DPDK.

The experiment uses a default workload of `memaslap`, which comprises of 10 % "set" and 90 % "get" operations

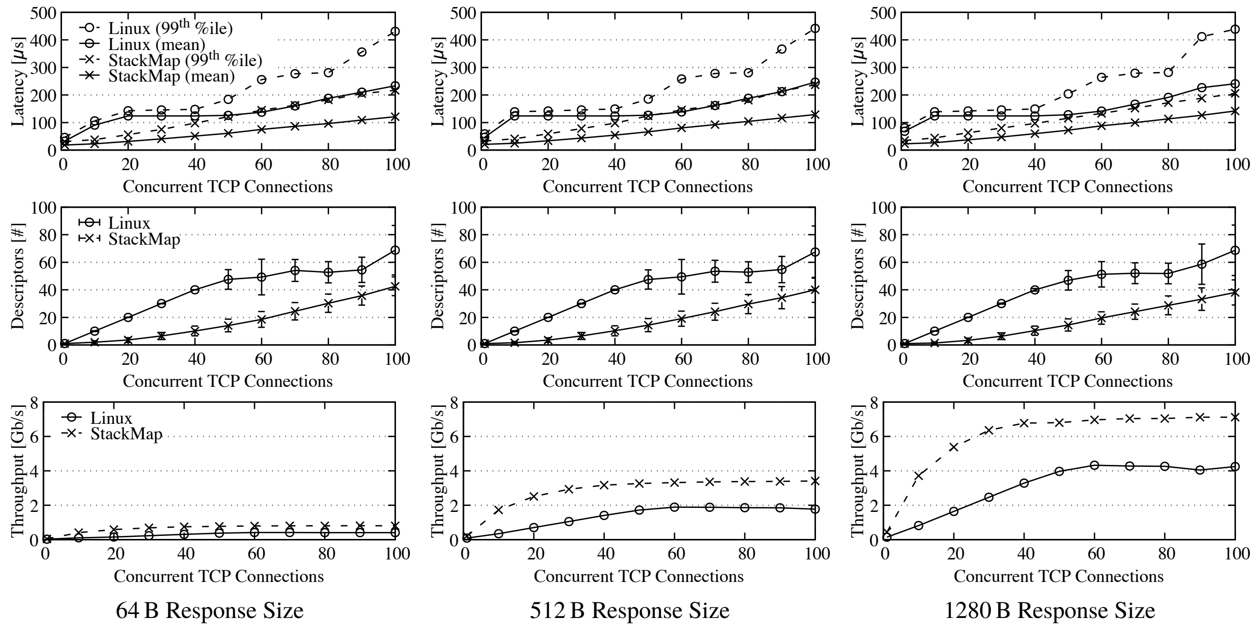


Figure 7: Mean and 99th percentile round-trip latencies (top row), mean number of ready descriptors in each event processing cycle (middle row) and throughputs (bottom row), with the number of concurrent TCP connections (horizontal axis) for different response sizes (64, 512 and 1280 B).

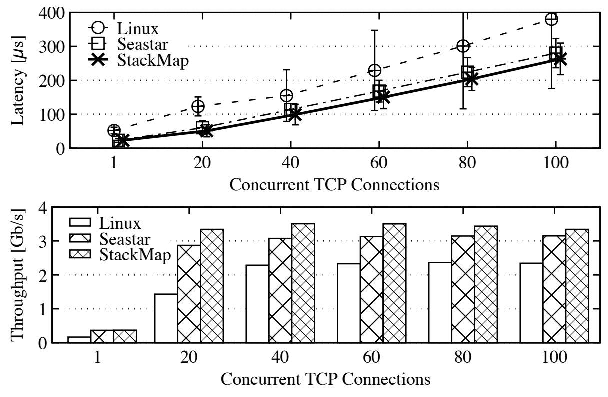


Figure 8: Memcached latency and throughput results.

on 1024 B objects. While still simple, memcached has a slightly more complex application logic than the simple HTTP server we used for the previous benchmarks, and therefore exhibits higher processing delays (see Table 2).

Figure 8 shows mean latencies with standard deviations, as well as aggregate throughputs. StackMap achieves significantly higher throughputs and lower latencies than Linux, as well as a much smaller latency variance. This is similar to observations earlier in this section. Surprisingly, StackMap also slightly outperforms Seastar.

5.5 Memcached Multicore Scalability

Finally, we evaluate multi-core scalability with StackMap, again using memcached as an application, and compares the results against Linux and Seastar. The object size for this experiment is 64 B to prevent the network from becoming saturated. In order to investigate the overheads and scalability of memcached itself, we also compare against the simple HTTP server used for our other measurements, configured to also serve 64 B messages.

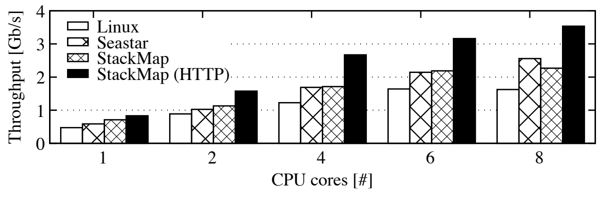


Figure 9: Memcached multi-core throughput.

Figure 9 shows aggregate throughputs when using a different number of CPU cores to serve the workload. Up to six cores used, the relative performance differences between Linux, Seastar and StackMap remain similar, with Linux being slowest, and StackMap slightly outperforming Seastar. However, Seastar begins to outperform the others at eight cores. StackMap and Linux retain their relative performance difference. This is to be expected, because the current StackMap implementation does not yet optimize locking when the Linux TCP/IP operates in StackMap mode, nor has memcached been modified to remove such locks when used with StackMap. In contrast, Seastar adopts a shared-nothing design, and memcached on top of Seastar also has been highly optimized for multi-core scalability. The inclusion of the simple HTTP server results for StackMap attempt to illustrate its scalability potential (and also illustrate the need for future improvements to StackMap).

6 Limitations and Future Work

This section briefly discusses limitations of the StackMap architecture and prototype implementation, as well as potential future improvements.

6.1 Misbehaving Applications

The OS packet representation structure (`sk_buff` or `mbuf`) consists of metadata and a pointer to a data buffer, whereas StackMap exposes only data buffers to its applications. However, Linux also stores some metadata at the end of a data buffer (`skb_shared_info`), which includes information to share a data buffer between multiple `sk_buff`s and pointers to additional data buffers [9].

Thus, system memory could leak—or the system may crash—if a misbehaving StackMap application modifies this metadata. One possible solution is to link a netmap buffer as an additional buffer, because Linux stores no metadata for those, and to only use a primary data buffer to store `skb_shared_info`. A similar method would also work for FreeBSD, using external storage associated with an `mbuf`. We will explore such solutions in the future.

StackMap also increases the risk of the network stack malfunction when a StackMap application misuses the netmap API or it modifies buffers owned by the kernel (indicated by the head and tail pointers [53]). In the original netmap, kernel-owned buffers are only touched by the netmap backend, which is robust against the case where an application modifies data out of turn. However, with StackMap, these kernel-owned buffers are also processed and referred to by the kernel TCP/IP stack. Therefore, if such buffers are modified by the application out of turn (*e.g.*, modifying the sequence number of a sent packet that is referred to from the retransmission queue), the TCP/IP implementation could fall into inconsistent state.

Possible solutions to this issue include providing a wrapper API to prevent the application from accessing the kernel-owned buffers, making the kernel create a private copy of data and making the kernel its buffers read-only. Since any of these mitigation methods comes at some cost, we leave their investigation as future work.

6.2 Support for Other Protocols

Although the StackMap prototype implementation supports only TCP and IPv4, the StackMap architecture is not limited to these protocols. However, UDP-based applications can already batch syscalls using `sendmmsg()` and `recvmmsg()`, may not need an event multiplexing API like `epoll_wait()` and can exploit batching during packet I/O for transmission of messages to different clients [30]. Therefore, the performance benefits that the StackMap architecture could bring to new UDP-based protocols such as QUIC [20] need to be investigated.

6.3 System Configuration

In order to achieve the best network performance, system administrators should configure their systems such that traffic to and from regular applications are routed via NICs that are not used by any StackMap application. Not doing so does not crash the system, but regular applications could see unexpected packet delays, because moving packets in and out of the NICs is triggered by the StackMap application, and not the normal kernel methods. Nevertheless, in many of today's production systems, such configuration is already regularly performed, and so does not complicate StackMap deployment.

7 Conclusion

Our goal in this paper has been to address the latency problems of transactional workloads over TCP, which consist of small messages sent over a large number of concurrent connections. We demonstrated that the kernel TCP/IP implementation is reasonably fast, but showed that the socket API and the traditional packet I/O methods increase transaction latencies and limit throughputs. StackMap, a new interface to the OS TCP/IP service that exploits the opportunities afforded by dedicating NICs to applications, addresses these performance issues. The StackMap design challenges included combining the full-featured TCP/IP implementation in the kernel with netmap, which in addition to fast packet I/O methods provides protection of the system and NICs, and a sophisticated API.

A key takeaway is that an integration of most of the techniques introduced by high-performance kernel-bypass TCP/IPs—including new APIs, syscall and I/O batching, lightweight buffer management and direct packet buffer access—can be leveraged smoothly into the OS stack, and help it achieve comparable performance. The key advantage of this approach is that StackMap allows applications to enjoy modern TCP/IP features and benefit from the active maintenance that the OS stack is seeing, which kernel-bypass TCP/IPs lack.

8 Acknowledgments

This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". *Proc. ACM SIGCOMM*. 2010.

[2] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. "pFabric: Minimal Near-optimal Datacenter Transport". *Proc. ACM SIGCOMM*. 2013.

[3] M. Allman, H. Balakrishnan, and S. Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC 3042. Jan. 2001.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency". *Proc. USENIX OSDI*. 2014.

[5] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. "The Case for Ubiquitous Transport-level Encryption". *Proc. USENIX Security*. 2010.

[6] Cloudius Systems. *Seastar*. http://www.seastar-project.org/.

[7] J. Corbet. *The kernel connection multiplexer*. https://lwn.net/Articles/657999/. Sep. 21, 2015.

[8] J. Cummings and E. Tamir. *Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll*. Intel White Paper. 2013.

[9] David S. Miller. *How SKBs work*. http://vger.kernel.org/~davem/skb_data.html.

[10] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01. Feb. 25, 2013.

[11] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. "Proportional Rate Reduction for TCP". *Proc. ACM IMC*. 2011.

[12] A. Dunkels. "Design and Implementation of the lwIP TCP/IP Stack". *Swedish Institute of Computer Science*, 2001.

[13] W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987. Aug. 2007.

[14] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen. *Architecture of Systemtap: A Linux Trace/Probe Tool*. 2005.

[15] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and C. Wagner. "Data ONTAP GX: A Scalable Storage Cluster". *Proc. USENIX FAST*. 2007.

[16] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. "Lazy Asynchronous I/O for Event-driven Servers". *Proc. USENIX ATC*. 2004.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat. "A Scalable, Commodity Data Center Network Architecture". *Proc. ACM SIGCOMM*. 2008.

[18] A. Ford, C. Raiciu, M. J. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. Oct. 14, 2015.

[19] GitHub. *Modern HTTP benchmarking tool*. https://github.com/wg/wrk. Jul. 2013.

[20] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-tsvwg-quic-protocol-02. Jan. 13, 2016.

[21] S. Han, K. Jang, K. Park, and S. Moon. "PacketShader: A GPU-accelerated Software Router". *Proc. ACM SIGCOMM*. 2010.

[22] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O". *Proc. USENIX OSDI*. 2012.

[23] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. "mSwitch: A Highly-scalable, Modular Software Switch". *Proc. ACM SOSR*. 2015.

[24] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. "Rekindling Network Protocol Innovation with User-level Stacks". *ACM SIGCOMM CCR*, Apr. 2014.

[25] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. "Is It Still Possible to Extend TCP?": *Proc. ACM IMC*. 2011.

[26] Intel. *Intel DPDK: Data Plane Development Kit*. http://dpdk.org/. Sep. 2013.

[27] Intel. *Introduction to the Storage Performance Development Kit (SPDK)*. https://software.intel.com/en-us/articles/introduction-to-the-storage-erformance-development-kit-spdk. Sep. 18, 2015.

[28] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC 1323. May 1992.

[29] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems". *Proc. USENIX NSDI*. 2014.

[30] Jesper Dangaard Brouer. *Unlocked 10Gbps TX wirespeed smallest packet single core*. http://netoptimizer.blogspot.de/2014/10/unlocked-10gbps-tx-wirespeed-smallest.html.

[31] A. Kantee. "Environmental Independence: BSD Kernel TCP/IP in Userspace". *AsiaBSDCon*, 2009.

[32] M. Larsen and F. Gont. *Recommendations for Transport-Protocol Port Randomization*. RFC 6056. Jan. 2011.

[33] J. Leverich and C. Kozyrakis. “Reconciling High Server Utilization and Sub-millisecond Quality-of-service”. *Proc. ACM EuroSys*. 2014.

[34] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. “The Emergence of Networking Abstractions and Techniques in TinyOS”. *Proc. USENIX NSDI*. 2004.

[35] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. “Scalable Kernel TCP Design and Implementation for Short-Lived Connections”. *Proc. ACM ASPLOS*. 2016.

[36] M. Zhuang and B. Aker. *memaslap: Load testing and benchmarking a server*. http://docs.libmemcached.org/bin/memaslap.html.

[37] I. Marinos, R. N. Watson, and M. Handley. “Network Stack Specialization for Performance”. *Proc. ACM SIGCOMM*. 2014.

[38] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996.

[39] M. Mathis and J. Mahdavi. “Forward Acknowledgement: Refining TCP Congestion Control”. *Proc. ACM SIGCOMM*. 1996.

[40] A. Medina, M. Allman, and S. Floyd. “Measuring the Evolution of Transport Protocols in the Internet”. *ACM SIGCOMM CCR*, Apr. 2005.

[41] *memcached - a distributed memory object caching system*. http://memcached.org/.

[42] Microsoft. *Windows I/O Completion Ports*. Microsoft White Paper. 2012.

[43] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. “The Click Modular Router”. *Proc. ACM SOSP*. 1999.

[44] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. *Known TCP Implementation Problems*. RFC 2525. Mar. 1999.

[45] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. “Fastpass: A Centralized "Zero-queue" Datacenter Network”. *Proc. ACM SIGCOMM*. 2014.

[46] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. “Improving Network Connection Locality on Multicore Systems”. *Proc. ACM EuroSys*. 2012.

[47] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. “Arrakis: The Operating System is the Control Plane”. *Proc. USENIX OSDI*. Oct. 2014.

[48] J. Postel. *Transmission Control Protocol*. RFC 793. Sep. 1981.

[49] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. “TCP Fast Open”. *Proc. ACM CoNEXT*. 2011.

[50] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. “How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP”. *Proc. USENIX NSDI*. 2012.

[51] A. Ramaiah, R. Stewart, and M. Dalal. *Improving TCP’s Robustness to Blind In-Window Attacks*. RFC 5961. Aug. 2010.

[52] L. Rizzo, M. Carbone, and G. Catalli. “Transparent Acceleration of Software Packet Forwarding using netmap”. *Proc. IEEE Infocom*. Mar. 2012.

[53] L. Rizzo, G. Lettieri, and M. Honda. “Netmap as a Core Networking Technology”. *AsiaBSDCon*, 2014.

[54] L. Rizzo. “netmap: A Novel Framework for Fast Packet I/O”. *Proc. USENIX ATC*. Jun. 2012.

[55] L. Rizzo. “Revisiting Network I/O APIs: The Netmap Framework”. *Queue*, Jan. 2012.

[56] L. Rizzo and G. Lettieri. “VALE, a Switched Ethernet for Virtual Machines”. *Proc. ACM CoNEXT*. 2012.

[57] P. Sarolahti and M. Kojo. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP)*. RFC 4138. Aug. 2005.

[58] P. Sarolahti, M. Kojo, and K. Raatikainen. “F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts”. *ACM SIGCOMM CCR*, Apr. 2003.

[59] L. Soares and M. Stumm. “FlexSC: Flexible System Call Scheduling with Exception-less System Calls”. *Proc. USENIX OSDI*. 2010.

[60] H. Tazaki, R. Nakamura, and Y. Sekiya. “Library Operating System with Mainline Linux Network Stack”. *Proc. netdev0.1*. Feb. 2015.

[61] The Open Group. *Networking Services, Issue 4*. Sep. 1994.

[62] M. Zec, L. Rizzo, and M. Mikuc. “DXR: Towards a Billion Routing Lookups Per Second in Software”. *ACM SIGCOMM CCR*, Sep. 2012.