Network Stack Specialization for Performance

Ilias Marinos
University of Cambridge
Ilias.Marinos@cl.cam.ac.uk

Robert N.M. Watson
University of Cambridge
Robert.Watson@cl.cam.ac.uk

Mark Handley
University College London
M.Handley@cs.ucl.ac.uk

Abstract

Contemporary network stacks are masterpieces of generality, supporting many edge-node and middle-node functions. Generality comes at a high performance cost: current APIs, memory models, and implementations drastically limit the effectiveness of increasingly powerful hardware. Generality has historically been required so that individual systems could perform many functions. However, as providers have scaled services to support millions of users, they have transitioned toward thousands (or millions) of dedicated servers, each performing a few functions. We argue that the overhead of generality is now a key obstacle to effective scaling, making specialization not only viable, but necessary.

We present Sandstorm and Namestorm, web and DNS servers that utilize a clean-slate userspace network stack that exploits knowledge of application-specific workloads. Based on the netmap framework, our novel approach merges application and network-stack memory models, aggressively amortizes protocol-layer costs based on application-layer knowledge, couples tightly with the NIC event model, and exploits microarchitectural features. Simultaneously, the servers retain use of conventional programming frameworks. We compare our approach with the FreeBSD and Linux stacks using the nginx web server and NSD name server, demonstrating 2–10× and 9× improvements in web-server and DNS throughput, lower CPU usage, linear multicore scaling, and saturated NIC hardware.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design — *Network communications*

General Terms: Design, performance

Keywords: Network stacks; network performance; network-stack specialization; clean-slate design

1. INTRODUCTION

Conventional network stacks were designed in an era where individual systems had to perform multiple diverse functions. In the last decade, the advent of cloud computing and the ubiquity of networking has changed this model; today, large content providers serve hundreds of millions of customers. To scale their systems, they are forced to employ many thousands of servers, with each providing only a single network service. Yet most content is still served with conventional general-purpose network stacks.

These general-purpose stacks have not stood still, but today's stacks are the result of numerous incremental updates on top of codebases that were originally developed in the early 1990s. Arguably, these network stacks have proved to be quite efficient, flexible, and reliable, and this is the reason that they still form the core of contemporary networked systems. They also provide a stable programming API, simplifying software development. But this generality comes with significant costs, and we argue that the overhead of generality is now a key obstacle to effective scaling, making specialization not only viable, but necessary.

In this paper we revisit the idea of specialized network stacks. In particular, we develop Sandstorm, a specialized userspace stack for serving static web content, and Namestorm, a specialized stack implementing a high performance DNS server. More importantly, however, our approach does not simply shift the network stack to userspace: we also promote tight integration and specialization of application and stack functionality, achieving cross-layer optimizations antithetical to current design practices.

Servers such as Sandstorm could be used for serving images such as the Facebook logo, as OCSP [20] responders for certificate revocations, or as front end caches to popular dynamic content. This is a role that conventional stacks should be good at: nginx [6] uses the `sendfile()` system call to hand over serving static content to the operating system. FreeBSD and Linux then implement zero-copy stacks, at least for the payload data itself, using scatter-gather to directly DMA the payload from the disk buffer cache to the NIC. They also utilize the features of smart network hardware, such as TCP Segmentation Offload (TSO) and Large Receive Offload (LRO) to further improve performance. With such optimizations, nginx does perform well, but as we will demonstrate, a specialized stack can outperform it by a large margin.

Namestorm is aimed at handling extreme DNS loads, such as might be seen at the root nameservers, or when a server is under a high-rate DDoS attack. The open-source state of the art here is NSD [5], which combined with a modern OS that minimizes data copies when sending and receiving UDP packets, performs well. Namestorm, however, can outperform it by a factor of nine.

Our userspace web server and DNS server are built upon FreeBSD's netmap [31] framework, which directly maps the NIC buffer rings to userspace. We will show that not only is it possible for a specialized stack to beat nginx, but on data-center-style networks when serving small files typical of many web pages, it can achieve

SIGCOMM'14, August 17–22, 2014, Chicago, Illinois, USA.

http://dx.doi.org/10.1145/2619239.2626311.

three times the throughput on older hardware, and more than six times the throughput on modern hardware supporting DDIO[1].

The demonstrated performance improvements come from four places. First, we implement a complete zero-copy stack, not only for payload but also for all packet headers, so sending data is very efficient. Second, we allow aggressive amortization that spans traditionally stiff boundaries – e.g., application-layer code can request pre-segmentation of data intended to be sent multiple times, and extensive batching is used to mitigate system-call overhead from userspace. Third, our implementation is synchronous, clocked from received packets; this improves cache locality and minimizes the latency of sending the first packet of the response. Finally, on recent systems, Intel's DDIO provides substantial benefits, but only if packets to be sent are already in the L3 cache and received packets are processed to completion immediately. It is hard to ensure this on conventional stacks, but a special-purpose stack can get much closer to this ideal.

Of course, userspace stacks are not a novel concept. Indeed, the Cheetah web server for MIT's XOK Exokernel [19] operating system took a similar approach, and demonstrated significant performance gains over the NCSA web server in 1994. Despite this, the concept has never really taken off, and in the intervening years conventional stacks have improved immensely. Unlike XOK, our specialized userspace stacks are built on top of a conventional FreeBSD operating system. We will show that it is possible to get all the performance gains of a specialized stack without needing to rewrite all the ancillary support functions provided by a mature operating system (e.g., the filesystem). Combined with the need to scale server clusters, we believe that the time has come to re-evaluate special-purpose stacks on today's hardware.

The key contributions of our work are:

- We discuss many of the issues that affect performance in conventional stacks, even though they use APIs aimed at high performance such as `sendfile()` and `recvmmsg()`.
- We describe the design and implementation of multiple modular, highly specialized, application-specific stacks built over a commodity operating system while avoiding these pitfalls. In contrast to prior work, we demonstrate that it is possible to utilize both conventional and specialized stacks in a single system. This allows us to deploy specialization selectively, optimizing networking while continuing to utilize generic OS components such as filesystems without disruption.
- We demonstrate that specialized network stacks designed for aggressive cross-layer optimizations create opportunities for new and at times counter-intuitive hardware-sensitive optimizations. For example, we find that violating the long-held tenet of data-copy minimization can increase DMA performance for certain workloads on recent CPUs.
- We present hardware-grounded performance analyses of our specialized network stacks side-by-side with highly optimized conventional network stacks. We evaluate our optimizations over multiple generations of hardware, suggesting portability despite rapid hardware evolution.
- We explore the potential of a synchronous network stack blended with asynchronous application structures, in stark contrast to conventional asynchronous network stacks supporting synchronous applications. This approach optimizes cache utilization by both the CPU and DMA engines, yielding as much as 2-10× conventional stack performance.

[1]Direct Data I/O. For more details, see Section 2.4

2. SPECIAL-PURPOSE ARCHITECTURE

What is the minimum amount of work that a web server can perform to serve static content at high speed? It must implement a MAC protocol, IP, TCP (including congestion control), and HTTP. However, their implementations do not need to conform to the conventional socket model, split between userspace and kernel, or even implement features such as dynamic TCP segmentation. For a web server that serves the same static content to huge numbers of clients (e.g., the Facebook logo or GMail JavaScript), essentially the same functions are repeated again and again. We wish to explore just how far it is possible to go to improve performance. In particular, we seek to answer the following questions:

- Conventional network stacks support zero copy for OS-maintained data – e.g., filesystem blocks in the buffer cache, but not for application-provided HTTP headers or TCP packet headers. Can we take the zero-copy concept to its logical extreme, in which received packet buffers are passed from the NIC all the way to the application, and application packets to be sent are DMAed to the NIC for transmission without even the headers being copied?
- Conventional stacks make extensive use of queuing and buffering to mitigate context switches and keep CPUs and NICs busy, at the cost of substantially increased cache footprint and latency. Can we adopt a bufferless event model that reimposes synchrony and avoids large queues that exceed cache sizes? Can we expose link-layer buffer information, such as available space in the transmit descriptor ring, to prevent buffer bloat and reduce wasted work constructing packets that will only be dropped?
- Conventional stacks amortize expenses internally, but cannot amortize repetitive costs spanning application and network layers. For example, they amortize TCP connection lookup using Large Receive Offload (LRO) but they cannot amortize the cost of repeated TCP segmentation of the same data transmitted multiple times. Can we design a network-stack API that allows cross-layer amortizations to be accomplished such that after the first client is served, no work is ever repeated when serving subsequent clients?
- Conventional stacks embed the majority of network code in the kernel to avoid the cost of domain transitions, limiting two-way communication flow through the stack. Can we make heavy use of batching to allow device drivers to remain in the kernel while colocating stack code with the application and avoiding significant latency overhead?
- Can we avoid any data-structure locking, and even cache-line contention, when dealing with multi-core applications that do not require it?

Finally, while performing all the above, is there a suitable programming abstraction that allows these components to be reused for other applications that may benefit from server specialization?

2.1 Network-stack Modularization

Although monolithic kernels are the *de facto* standard for networked systems, concerns with robustness and flexibility continue to drive exploration of microkernel-like approaches. Both Sandstorm and Namestorm take on several microkernel-like qualities:

Rapid deployment & reusability: Our prototype stack is highly modular, and synthesized from the bottom up using traditional dynamic libraries as building blocks (*components*) to construct a

special-purpose system. Each component corresponds to a standalone service that exposes a well-defined API. Our specialized network stacks are built by combining four basic components:

- The netmap I/O (`libnmio`) library that abstracts traditional data-movement and event-notification primitives needed by higher levels of the stack.
- `libeth` component, a lightweight Ethernet-layer implementation.
- `libtcpip` that implements our optimized TCP/IP layer.
- `libudpip` that implements a UDP/IP layer.

Figure 1 depicts how some of these components are used with a simple application layer to form Sandstorm, the optimized web server.

Splitting functionality into reusable components does not require us to sacrifice the benefits of exploiting cross-layer knowledge to optimize performance, as memory and control flow move easily across API boundaries. For example, Sandstorm interacts directly with `libnmio` to preload and push segments into the appropriate packet-buffer pools. This preserves a service-centric approach.

Developer-friendly: Despite seeking inspiration from microkernel design, our approach maintains most of the benefits of conventional monolithic systems:

- Debugging is at least as easy (if not easier) compared to conventional systems, as application-specific, performance-centric code shifts from the kernel to more accessible userspace.
- Our approach integrates well with the general-purpose operating systems: rewriting basic components such as device drivers or filesystems is not required. We also have direct access to conventional debugging, tracing, and profiling tools, and can also use the conventional network stack for remote access (e.g., via SSH).
- Instrumentation in Sandstorm is a simple and straightforward task that allows us to explore potential bottlenecks as well as necessary and sufficient costs in network processing across application and stack. In addition, off-the-shelf performance monitoring and profiling tools "just work", and a synchronous design makes them easier to use.

2.2 Sandstorm web server design

Rizzo's netmap framework provides a general-purpose API that allows received packets to be mapped directly to userspace, and packets to be transmitted to be sent directly from userspace to the NIC's DMA rings. Combined with batching to reduce system calls, this provides a high-performance framework on which to build packet-processing applications. A web server, however, is not normally thought of as a packet-processing application, but one that handles TCP streams.

To serve a static file, we load it into memory, and *a priori* generate all the packets that will be sent, including TCP, IP, and link-layer headers. When an HTTP request for that file arrives, the server must allocate a TCP-protocol control block (TCB) to keep track of the connection's state, but the packets to be sent have already been created for each file on the server.[2]

The majority of the work is performed during inbound TCP ACK processing. The IP header is checked, and if it is acceptable, a hash

[2]Our implementation packetizes on startup, but it could equally well do it on demand, the first time a file is requested, and the resulting packets retained to serve future requests, managed as an LRU cache.

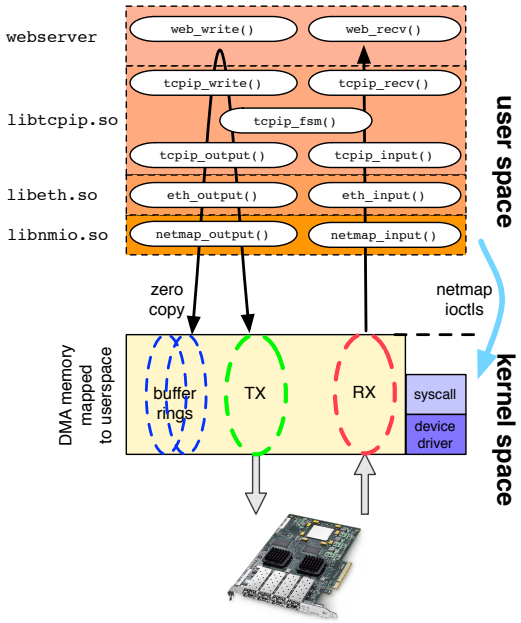


Figure 1: Sandstorm high-level architecture view.

table is used to locate the TCB. The offset of the ACK number from the start of the connection is used to locate the next prepackaged packet to send, and if permitted by the congestion and receive windows, subsequent packets. To send these packets, the destination address and port must be rewritten, and the TCP and IP checksums incrementally updated. The packet can then be directly fetched by the NIC using netmap. All reads of the ACK header and modifications to the transmitted packets are performed in a single pass, ensuring that both the headers and the TCB remain in the CPU's L1 cache.

Once a packet has been DMAed to the NIC, the packet buffer is returned to Sandstorm, ready to be incrementally modified again and sent to a different client. However, under high load, the same packet may need to be queued in the TX ring for a second client *before* it has finished being sent to the first client. The same packet buffer cannot be in the TX ring twice, with different destination address and port. This presents us with two design options:

- We can maintain more than one copy of each packet in memory to cope with this eventuality. The extra copy could be created at startup, but a more efficient solution would create extra copies on demand whenever a high-water mark is reached, and then retained for future use.
- We can maintain only one long-term copy of each packet, creating ephemeral copies each time it needs to be sent.

We call the former a *pre-copy* stack (it is an extreme form of zero-copy stack because in the steady state it never copies, but differs from the common use of the term "zero copy"), and the latter a *memcpy* stack. A pre-copy stack performs less per-packet work than a memcpy stack, but requires more memory; because of this, it has the potential to thrash the CPU's L3 cache. With the memcpy stack, it is more likely for the original version of a packet to be in the L3 cache, but more work is done. We will evaluate both approaches, because it is far from obvious how CPU cycles trade off against cache misses in modern processors.

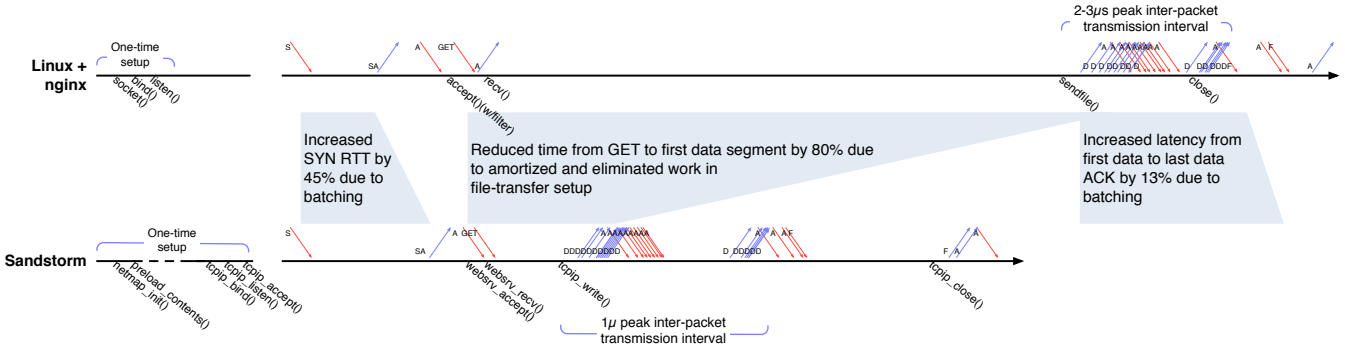


Figure 2: Several tradeoffs are visible in these packet traces taken on nginx/Linux and Sandstorm servers that are busy (but unsaturated).

Figure 2 illustrates tradeoffs through traces taken on nginx/Linux and pre-copy Sandstorm servers that are busy (but unsaturated). On the one hand, a batched design measurably increases TCP round-trip time with a relatively idle CPU. On the other hand, Sandstorm amortizes or eliminates substantial parts of per-request processing through a more efficient architecture. Under light load, the benefits are pronounced; at saturation, the effect is even more significant.

Although most work is synchronous within the ACK processing code path, TCP still needs timers for certain operations. Sandstorm's timers are scheduled by polling the Time Stamp Counter (TSC): although not as accurate as other clock sources, it is accessible from userspace at the cost of a single CPU instruction (on modern hardware). The TCP slow timer routine is invoked periodically (every ~500ms) and traverses the list of active TCBs: on RTO expiration, the congestion window and slow-start threshold are adjusted accordingly, and any unacknowledged segments are retransmitted. The same routine also releases TCBs that have been in *TIME_WAIT* state for longer than 2**MSL*. There is no buffering whatsoever required for retransmissions: we identify the segment that needs to be retransmitted using the oldest unacknowledged number as an offset, retrieve the next available prepackaged packet and adjust its headers accordingly, as with regular transmissions. Sandstorm currently implements TCP Reno for congestion control.

2.3 The Namestorm DNS server

The same principles applied in the Sandstorm web server, also apply to a wide range of servers returning the same content to multiple users. Authoritative DNS servers are often targets of DDoS attacks – they represent a potential single point of failure, and because DNS traditionally uses UDP, lacks TCP's three way handshake to protect against attackers using spoofed IP addresses. Thus, high performance DNS servers are of significant interest.

Unlike TCP, the conventional UDP stack is actually quite lightweight, and DNS servers already preprocess zone files and store response data in memory. Is there still an advantage running a specialized stack?

Most DNS-request processing is simple. When a request arrives, the server performs sanity checks, hashes the concatenation of the name and record type being requested to find the response, and sends that data. We can preprocess the responses so that they are already stored as a prepackaged UDP packet. As with HTTP, the destination address and port must be rewritten, the identifier must be updated, and the UDP and IP checksums must be incrementally updated. After the initial hash, all remaining processing is performed in one pass, allowing processing of DNS response headers to be performed from the L1 cache. As with Sandstorm, we can use pre-copy or memcpy approaches so that more than one response for the same name can be placed in the DMA ring at a time.

Our specialized userspace DNS server stack is composed of three reusable components, *libnmio*, *libeth*, *libudpip*, and a DNS-specific application layer. As with Sandstorm, Namestorm uses FreeBSD's netmap API, implementing the entire stack in userspace, and uses netmap's batching to amortize system call overhead. *libnmio* and *libeth* are the same as used by Sandstorm, whereas *libudpip* contains UDP-specific code closely integrated with an IP layer. Namestorm is an authoritative nameserver, so it does not need to handle recursive lookups.

Namestorm preprocesses the zone file upon startup, creating DNS response packets for all the entries in the zone, including the answer section and any glue records needed. In addition to type-specific queries for A, NS, MX and similar records, DNS also allows queries for ANY. A full implementation would need to create additional response packets to satisfy these queries; our implementation does not yet do so, but the only effect this would have is to increase the overall memory footprint. In practice, ANY requests prove comparatively rare.

Namestorm indexes the prepackaged DNS response packets using a hash table. There are two ways to do this:

- Index by concatenation of request type (e.g., A, NS, etc) and fully-qualified domain name (FQDN); for example, "`www.example.com`".
- Index by concatenation of request type and the wire-format FQDN as this appears in an actual query; for example, "*[3]*`www`*[7]*`example`*[3]*`com`*[0]*" where *[3]* is a single byte containing the numeric value 3.

Using the wire request format is obviously faster, but DNS permits compression of names. Compression is common in DNS answers, where the same domain name occurs more than once, but proves rare in requests. If we implement wire-format hash keys, we must first perform a check for compression; these requests are decompressed and then reencoded to uncompressed wire-format for hashing. The choice is therefore between optimizing for the common case, using wire-format hash keys, or optimizing for the worst case, assuming compression will be common, and using FQDN hash keys. The former is faster, but the latter is more robust to a DDoS attack by an attacker taking advantage of compression. We evaluate both approaches, as they illustrate different performance tradeoffs.

Our implementation does not currently handle referrals, so it can handle only zones for which it is authoritative for all the sub-zones. It could not, for example, handle the `.com` zone, because it would receive queries for `www.example.com`, but only have hash table entries for `example.com`. Truncating the hash key is trivial to do

Function	Parameters	Description
`tcpip_init()`	*none*	Initialize TCP layer (timers, callbacks etc).
`tcp_bind()`	tcb, ip, port	Bind a specific TCB to an IP and port.
`tcp_listen()`	tcb	Set a TCB to LISTEN state.
`tcp_accept()`	tcb, accept_callback	Set an application-specific accept callback to allow the application to control which connections to accept.
`tcp_recv()`	tcb, receive_callback	Set an application-specific receive callback to be called when data from a connection is available for the application.
`tcp_sent()`	tcb, sent_callback	Set an application-specific sent callback to be called when data previously written to a connection has been successfully delivered.
`tcp_write()`	tcb, content, number of segments	Push data to a TCP connection.
`tcp_close()`	tcb	Close a TCP connection, equivalent to BSD socket `shutdown()`.
`netmap_init()`	interfaces	Initialize netmap I/O library for specific interfaces.
`netmap_input_set_cb()`	interface, callback function	Set a callback to push raw RX data to higher layers (e.g., Ethernet).
`netmap_input()`	interface	Start the input engine on a specific interface.
`netmap_output()`	nring, pktbuf pool, slot index, flags	Depending on flags, memory or zero copy a packet to the TX ring – this function also implements TX batching.

Table 1: A selection of `libtcpip` and `libnmio` APIs.

1. Call RX poll to receive a batch of received packets that have been stored in the NIC's RX ring; block if none are available.
2. For each ACK packet in the batch:
    3. Perform Ethernet and IP input sanity checks.
    4. Locate the TCB for the connection.
    5. Update the acknowledged sequence numbers in TCB; update receive window and congestion window.
    6. For each new TCP data packet that can now be sent, or each lost packet that needs retransmitting:
        7. Find a free copy of the TCP data packet (or clone one if needed).
        8. Correct the destination IP address, destination port, sequence numbers, and incrementally update the TCP checksum.
        9. Add the packet to the NIC's TX ring.
        10. Check if δt has passed since last TX poll. If it has, call TX poll to send all queued packets.
11. Loop back to step 1

Figure 3: Outline of the main Sandstorm event loop.

as part of the translation to an FQDN, so if Namestorm were to be used for a domain such as `.com`, the FQDN version of hashing would be a reasonable approach.

2.4 Main event loop

To understand how the pieces fit together and the nature of interaction between Sandstorm, Namestorm, and netmap, we consider the timeline for processing ACK packets in more detail. Figure 3 summarizes Sandstorm's main loop. SYN/FIN handling, HTTP, and timers are omitted from this outline, but also take place. However, most work is performed in the ACK processing code.

One important consequence of this architecture is that the NIC's TX ring serves as the sole output queue, taking the place of conventional socket buffers and software network-interface queues. This is possible because retransmitted TCP packets are generated in the same way as normal data packets. As Sandstorm is fast enough to saturate two 10Gb/s NICs with a single thread on one core, data structures are also lock free.

When the workload is heavy enough to saturate the CPU, the system-call rate decreases. The receive batch size increases as calls to RX poll become less frequent, improving efficiency at the expense of increased latency. Under extreme load, the RX ring will fill, dropping packets. At this point the system is saturated and, as with any web server, it must bound the number of open connections by dropping some incoming SYNs.

Under heavier load, the TX-poll system call happens in the RX handler. In our current design, δt, the interval between calls to TX poll in the steady state, is a constant set to 80μs. The system-call rate under extreme load could likely be decreased by further increasing δt, but as the pre-copy version of Sandstorm can easily saturate all six 10Gb/s NICs in our systems for all file sizes, we have thus far not needed to examine this. Under lighter load, incoming packets might arrive too rarely to provide acceptable latency for transmitted packets; a 5ms timer will trigger transmission of straggling packets in the NIC's TX ring.

The difference between the pre-copy version and the memcpy version of Sandstorm is purely in step 7, where the memcpy version will simply clone the single original packet rather than search for an unused existing copy.

Contemporary Intel server processors support Direct Data I/O (DDIO). DDIO allows NIC-originated Direct Memory Access (DMA) over PCIe to access DRAM through the processor's Last-Level Cache (LLC). For network transmit, DDIO is able to pull data from the cache without a detour through system memory; likewise, for receive, DMA can place data in the processor cache. DDIO implements administrative limits on LLC utilization intended to prevent DMA from thrashing the cache. This design has the potential to significantly reduce latency and increase I/O bandwidth.

Memcpy Sandstorm forces the payload of the copy to be in the CPU cache from which DDIO can DMA it to the NIC without needing to load it from memory again. With pre-copy, the CPU only touches the packet headers, so if the payload is not in the CPU cache, DDIO must load it, potentially impacting performance. These interactions are subtle, and we will look at them in detail.

Namestorm follows the same basic outline, but is simpler as DNS is stateless: it does not need a TCB, and sends a single response packet to each request.

2.5 API

As discussed, all of our stack components provide well-defined APIs to promote reusability. Table 1 presents a selection of API functions exposed by `libnmio` and `libtcpip`. In this section we describe some of the most interesting properties of the APIs.

`libnmio` is the lowest-level component: it handles all interaction with netmap and abstracts the main event loop. Higher layers

(e.g., `libeth`) register callback functions to receive raw incoming data as well as set timers for periodic events (e.g., TCP slow timer). The function `netmap_ouput()` is the main transmission routine: it enqueues a packet to the transmission ring either by memory or zero copying and also implements an adaptive batching algorithm.

Since there is no socket layer, the application must directly interface with the network stack. With TCP as the transport layer, it acquires a TCB (TCP Control Block), binds it to a specific IPv4 address and port, and sets it to *LISTEN* state using API functions. The application must also register callback functions to accept connections, receive and process data from active connections, as well as act on successful delivery of sent data (e.g., to close the connection or send more data).

3. EVALUATION

To explore Sandstorm and Namestorm's performance and behavior, we evaluated using both older and more recent hardware. On older hardware, we employed Linux 3.6.7 and FreeBSD 9-STABLE. On newer hardware, we used Linux 3.12.5 and FreeBSD 10-STABLE. We ran Sandstorm and Namestorm on FreeBSD.

For the old hardware, we use three systems: two clients and one server, connected via a 10GbE crossbar switch. All test systems are equipped with an Intel 82598EB dual port 10GbE NIC, 8GB RAM, and two quad-core 2.66 GHz Intel Xeon X5355 CPUs. In 2006, these were high-end servers. For the new hardware, we use seven systems; six clients and one server, all directly connected via dedicated 10GbE links. The server has three dual-port Intel 82599EB 10GbE NICs, 128GB RAM and a quad-core Intel Xeon E5-2643 CPU. In 2014, these are well-equipped contemporary servers.

The most interesting improvements between these hardware generations are in the memory subsystem. The older Xeons have a conventional architecture with a single 1,333MHz memory bus serving both CPUs. The newer machines, as with all recent Intel server processors, support Data Direct I/O (DDIO), so whether data to be sent is in the cache can have a significant impact on performance.

Our hypothesis is that Sandstorm will be significantly faster than nginx on both platforms; however, the reasons for this may differ. Experience [18] has shown that the older systems often bottleneck on memory latency, and as Sandstorm is not CPU-intensive, we would expect this to be the case. A zero-copy stack should thus be a big win. In addition, as cores contend for memory, we would expect that adding more cores does not help greatly.

On the other hand, with DDIO, the new systems are less likely to bottleneck on memory. The concern, however, would be that DDIO could thrash at least part of the CPU cache. On these systems, we expect that adding more cores would help performance, but that in doing so, we may experience scalability bottlenecks such as lock contention in conventional stacks. Sandstorm's lock-free stack can simply be replicated onto multiple 10GbE NICs, with one core per two NICs to scale performance. In addition, as load increases, the number of packets to be sent or received per system call will increase due to application-level batching. Thus, under heavy load, we would hope that the number of system calls per second to still be acceptable despite shifting almost all network-stack processing to userspace.

The question, of course, is how well do these design choices play out in practice?

3.1 Experiment Design: Sandstorm

We evaluated the performance of Sandstorm through a set of experiments and compare our results against the nginx web server running on both FreeBSD and Linux. Nginx is a high-performance, low-footprint web server that follows the non-blocking, event-driven model: it relies on OS primitives such as `kqueue()` for readiness event notifications, it uses `sendfile()` to send HTTP payload directly from the kernel, and it asynchronously processes requests.

Contemporary web pages are immensely content-rich, but they mainly consist of smaller web objects such as images and scripts. The distribution of requested object sizes for Yahoo! CDN, reveals that 90% of the content is smaller than 25KB [11]. The conventional network stack and web-server application perform well when delivering large files by utilizing OS primitives and NIC hardware features. Conversely, multiple simultaneous short-lived HTTP connections are considered a heavy workload that stresses the kernel-userspace interface and reveals performance bottlenecks: even with `sendfile()` to send the payload, the size of the transmitted data is not quite enough to compensate for the system cost.

For all the benchmarks, we configured nginx to serve content from a RAM disk to eliminate disk-related I/O bottlenecks. Similarly, Sandstorm preloads the data to be sent and performs its pre-segmentation phase before the experiments begin. We use *weighttp* [9] to generate load with multiple concurrent clients. Each client generates a series of HTTP requests, with a new connection being initiated immediately after the previous one terminates. For each experiment we measure throughput, and we vary the size of the file served, exploring possible tradeoffs between throughput and system load. Finally, we run experiments with a realistic workload by using a trace of files with sizes that follow the distribution of requested HTTP objects of the Yahoo! CDN.

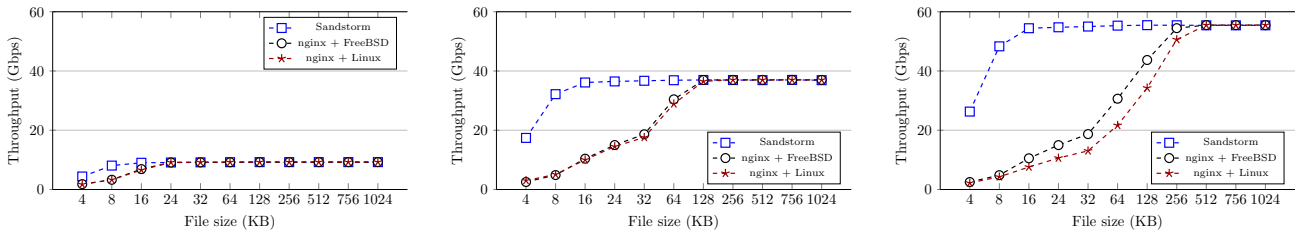
3.2 Sandstorm Results

First, we wish to explore how file size affects performance. Sandstorm is designed with small files in mind, and batching to reduce overheads, whereas the conventional `sendfile()` ought to be better for larger files.

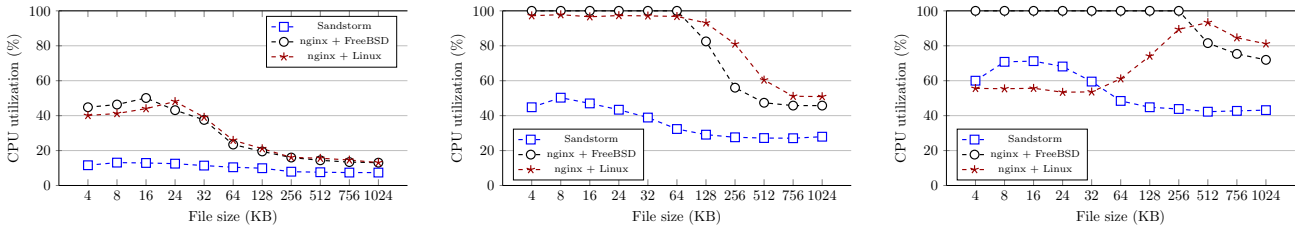
Figure 4 shows performance as a function of content size, comparing pre-copy Sandstorm and nginx running on both FreeBSD and Linux. With a single 10GbE NIC (Fig. 4a and 4d), Sandstorm outperforms nginx for smaller files by ~23–240%. For larger files, all three configurations saturate the link. Both conventional stacks are more CPU-hungry for the whole range of file sizes tested, despite potential advantages such as TSO on bulk transfers.

To scale to higher bandwidths, we added more 10GbE NICs and client machines. Figure 4b shows aggregate throughput with four 10GbE NICs. Sandstorm saturates all four NICs using just two CPU cores, but neither Linux nor FreeBSD can saturate the NICs with files smaller than 128KB, even though they use four CPU cores.

As we add yet more NICs, shown in Figure 4c, the difference in performance gets larger for a wider range of file sizes. With 6×10GbE NICs Sandstorm gives between 10% and 10× more throughput than FreeBSD for file sizes in the range of 4–256KB. Linux fares worse, experiencing a performance drop (see Figure 4c) compared to FreeBSD with smaller file sizes and 5–6 NICs. Low CPU utilization is normally good, but here (Figures 4f, 5b), idle time is undesirable since the NICs are not yet saturated. We have not identified any single obvious cause for this degradation. Packet traces show the delay to occur between the connection being accepted and the response being sent. There is no single kernel lock being held for especially long, and although locking is not negligible, it does not dominate, either. The system suffers one soft page fault for every two connections on average, but no hard faults, so data is already in the disk buffer cache, and TCB recycling is enabled. This is an example of how hard it can be to find performance problems with conventional stacks. Interestingly, this was an application-specific behavior triggered only on Linux: in benchmarks we conducted with other web servers (e.g., lighttpd [3], OpenLiteSpeed [7]) we did not experience a similar performance collapse on Linux with

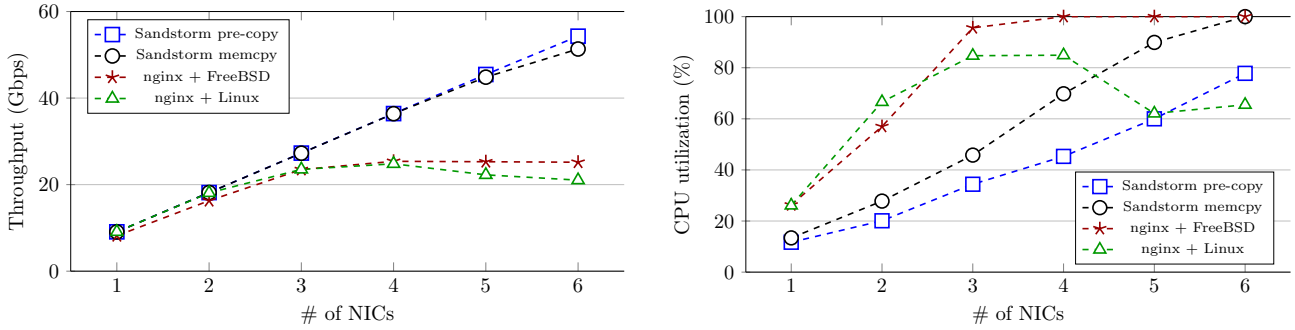


(a) Network throughput, 1 NIC **(b)** Network throughput, 4 NICs **(c)** Network throughput, 6 NICs



(d) CPU utilization, 1 NIC **(e)** CPU utilization, 4 NICs **(f)** CPU utilization, 6 NICs

Figure 4: Sandstorm throughput vs. file sizes and number of NICs.



(a) Network throughput **(b)** CPU utilization

Figure 5: Network throughput and CPU utilization vs. number of NICs while serving a Yahoo! CDN-like workload.

more than four NICs. We have chosen, however, to present the nginx datasets as it offered the greatest overall scalability in both operating systems.

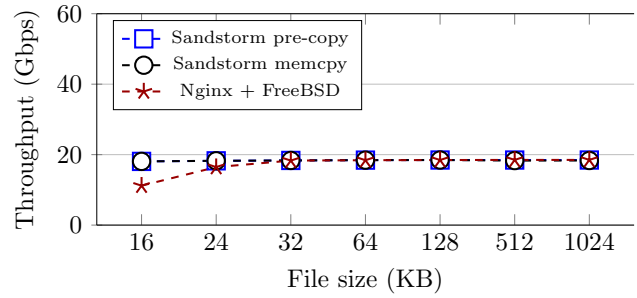
It is clear that Sandstorm dramatically improves network performance when it serves small web objects, but somewhat surprisingly, it performs better for larger files too. For completeness, we evaluate Sandstorm using a realistic workload: following the distribution of requested HTTP object sizes of the Yahoo! CDN [11], we generated a trace of 1000 files ranging from a few KB up to ~20MB which were served from both Sandstorm and nginx. On the clients, we modified *weighttp* to benchmark the server by concurrently requesting files in a random order. Figures 5a and 5b highlight the achieved network throughput and the CPU utilization of the server as a function of the number of the network adapters. The network performance improvement is more than 2× while CPU utilization is reduced.

Finally, we evaluated whether Sandstorm handles high packet loss correctly. With 80 simultaneous clients and 1% packet loss, as expected, throughput plummets. FreeBSD achieves approximately 640Mb/s and Sandstorm roughly 25% less. This is not fundamental, but due to FreeBSD's more fine-grained retransmit timer and its use of NewReno congestion control rather than Reno, which could also be implemented in Sandstorm. Neither network nor server is stressed in this experiment – if there had been a real congested link causing the loss, both stacks would have filled it.

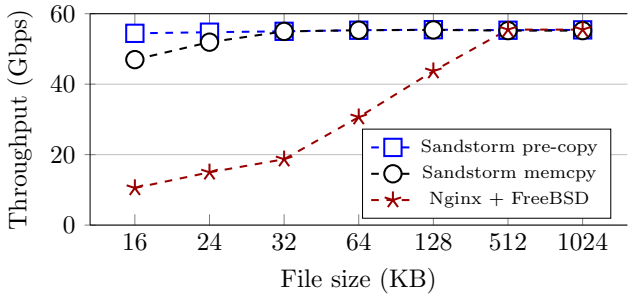
Throughout, we have invested considerable effort in profiling and optimizing conventional network stacks, both to understand their design choices and bottlenecks, and to provide the fairest possible comparison. We applied conventional performance tuning to Linux and FreeBSD, such as increasing hash-table sizes, manually tuning CPU work placement for multiqueue NICs, and adjusting NIC parameters such as interrupt mitigation. In collaboration with Netflix, we also developed a number of TCP and virtual-memory subsystem performance optimizations for FreeBSD, reducing lock contention under high packet loads. One important optimization is related to `sendfile()`, in which contention within the VM subsystem occurred while TCP-layer socket-buffer locks were held, triggering a cascade to the system as a whole. These changes have been upstreamed to FreeBSD for inclusion in a future release.

To copy or not to copy

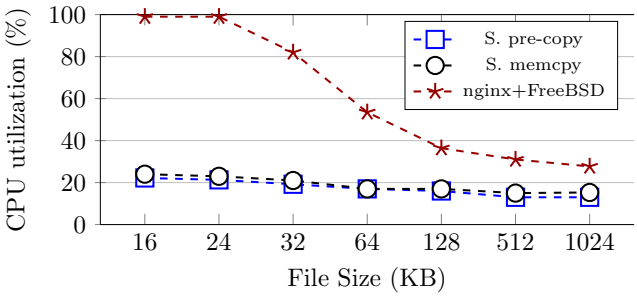
The pre-copy variant of Sandstorm maintains more than one copy of each segment in memory so that it can send the same segment to multiple clients simultaneously. This requires more memory than nginx serving files from RAM. The memcpy variant only enqueues copies, requiring a single long-lived version of each packet, and



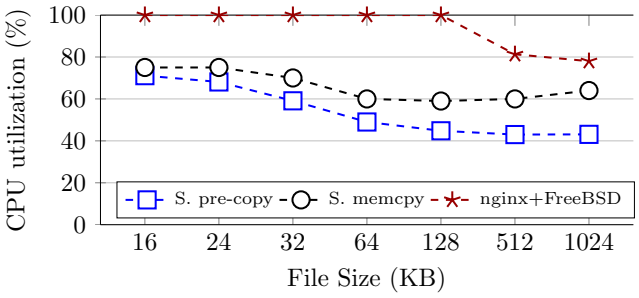
(a) Network throughput, 2 NICs



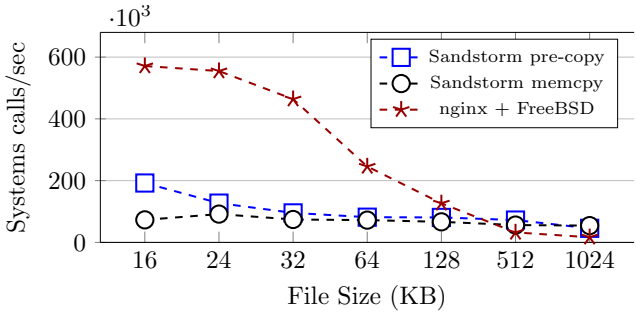
(b) Network throughput, 6 NICs



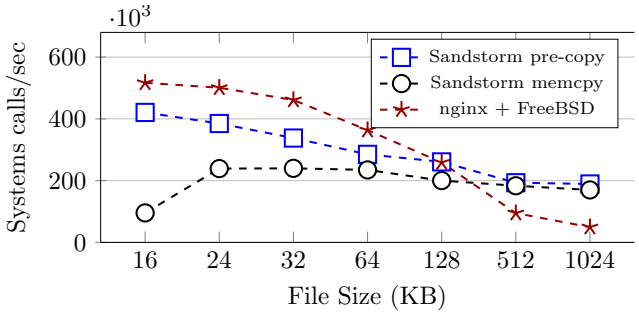
(c) CPU utilization, 2 NICs



(d) CPU utilization, 6 NICs



(e) System call rate, 2 NICs



(f) System call rate, 6 NICs

Figure 6: Sandstorm throughput and CPU utilization vs. variable number of NICs and file sizes.

uses a similar amount of memory to nginx. How does this `memcpy` affect performance? Figure 6 explores network throughput, CPU utilization, and system-call rate for two- and six-NIC configurations.

With six NICs, the additional `memcpy()` marginally reduces performance (Figure 6b) while exhibiting slightly higher CPU load (Figure 6d). In this experiment, Sandstorm only uses three cores to simplify the comparison, so around 75% utilization saturates those cores. The memcpy variant saturates the CPU for files smaller than 32KB, whereas the pre-copy variant does not. Nginx, using `sendfile()` and all four cores, only catches up for file sizes of 512KB and above, and even then exhibits higher CPU load.

As file size decreases, the expense of SYN/FIN and HTTP-request processing becomes measurable for both variants, but the pre-copy version has more headroom so is affected less. It is interesting to observe the effects of batching under overload with the memcpy stack in Figure 6f. With large file sizes, pre-copy and memcpy make the same number of system calls per second. With small files, however, the memcpy stack makes substantially fewer system calls per second. This illustrates the efficacy of batching: memcpy has saturated the CPU, and consequently no longer polls the RX queue as often. As the batch size increases, the system-call cost decreases, helping the server weather the storm. The pre-copy variant is not stressed here and continues to poll frequently, but would behave the same way under overload. In the end, the cost of the additional `memcpy` is measurable, but still performs quite well.

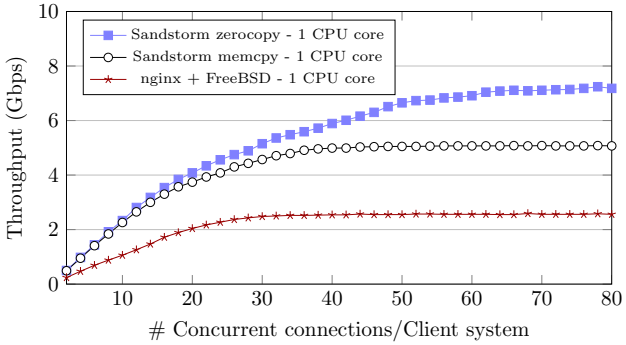
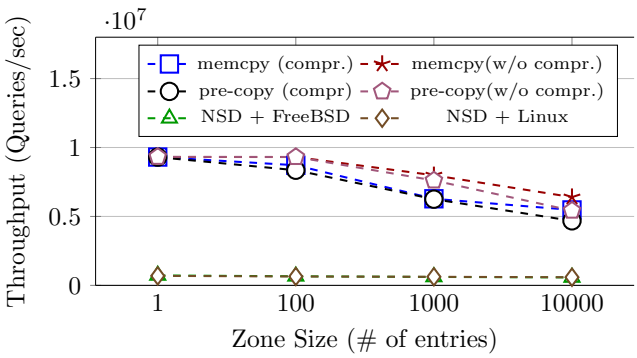
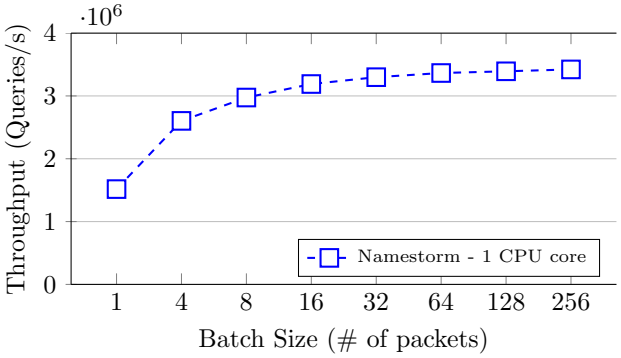


Figure 7: Network throughput, 1 NIC, ~23KB file, old hardware.



(a) Throughput, 1x10GbE NIC



(b) Batching efficiency

Figure 8: Namestorm performance measurements.

Results on contemporary hardware are significantly different from those run on older pre-DDIO hardware. Figure 7 shows the results obtained on our 2006-era servers. On the older machines, Sandstorm outperforms nginx by a factor of three, but the memcpy variant suffers a 30% decrease in throughput compared to pre-copy Sandstorm as a result of adding a single `memcpy` to the code. It is clear that on these older systems, memory bandwidth is the main performance bottleneck.

With DDIO, memory bandwidth is not such a limiting factor. Figure 9 in Section 3.5 shows the corresponding memory read throughput, as measured using CPU performance counters, for the network-throughput graphs in Figure 6b. With small file sizes, the pre-copy variant of Sandstorm appears to do more work: the L3 cache cannot hold all of the data, so there are many more L3 misses than with memcpy. Memory-read throughput for both pre-copy and nginx are closely correlated with their network throughput, indicating that DDIO is not helping on transmit: DMA comes from memory rather than the cache. The memcpy variant, however, has higher network throughput than memory throughput, indicating that DDIO is transmitting from the cache. Unfortunately, this is offset by much higher memory write throughput. Still, this only causes a small reduction in service throughput. Larger files no longer fit in the L3 cache, even with memcpy. Memory-read throughput starts to rise with files above 64KB. Despite this, performance remains high and CPU load decreases, indicating these systems are not limited by memory bandwidth for this workload.

3.3 Experiment Design: Namestorm

We use the same clients and server systems to evaluate Namestorm as we used for Sandstorm. Namestorm is expected to be significantly more CPU-intensive than Sandstorm, mostly due to fundamental DNS protocol properties: high packet rate and small packets. Based on this observation, we have changed the network topology of our experiment: we use only one NIC on the server connected to the client systems via a 10GbE cut-through switch. In order to balance the load on the server to all available CPU cores we use four dedicated NIC queues and four Namestorm instances.

We ran Nominum's *dnsperf* [2] DNS profiling software on the clients. We created zone files of varying sizes, loaded them onto the DNS servers, and configured dnsperf to query the zone repeatedly.

3.4 Namestorm Results

Figure 8a shows the performance of Namestorm and NSD running on Linux and FreeBSD when using a single 10GbE NIC. Performance results of NSD are similar with both FreeBSD and Linux. Neither operating system can saturate the 10GbE NIC, however, and both show some performance drop as the zone file grows. On Linux, NSD's performance drops by ~14% (from ~689,000 to ~590,000 Queries/sec) as the zone file grows from 1 to 10,000 entries, and on FreeBSD, it drops by ~20% (from ~720,000 to ~574,000 Qps). For these benchmarks, NSD saturates all CPU cores on both systems.

For Namestorm, we utilized two datasets, one where the hash keys are in wire-format (*w/o compr.*), and one where they are in FQDN format (*compr.*). The latter requires copying the search term before hashing it to handle possible compressed requests.

With wire-format hashing, Namestorm memcpy performance is ~11–13× better, depending on the zone size, when compared to the best results from NSD with either Linux or FreeBSD. Namestorm's throughput drops by ~30% as the zone file grows from 1 to 10,000 entries (from ~9,310,000 to ~6,410,000 Qps). The reason for this decrease is mainly the LLC miss rate, which more than doubles. *Dnsperf* does not report throughput in Gbps, but given the typical DNS response size for our zones we can calculate ~8.4Gbps and ~5.9Gbps for the smallest and largest zone respectively.

With FQDN-format hashing, Namestorm memcpy performance is worse than with wire-format hashing, but is still ~9–13× better compared to NSD. The extra processing with FQDN-format hashing costs ~10–20% in throughput, depending on the zone size.

Finally, in Figure 8a we observe a noticeable performance overhead with the pre-copy stack, which we explore in Section 3.5.

3.4.1 *Effectiveness of batching*

One of the biggest performance benefits for Namestorm is that netmap provides an API that facilitates batching across the system-call interface. To explore the effects of batching, we configured a single Namestorm instance and one hardware queue, and reran our benchmark with varying batch sizes. Figure 8b illustrates the results: a more than 2× performance gain when growing the batch size from 1 packet (no batching) to 32 packets. Interestingly, the performance of a single-core Namestorm without any batching remains more than 2× better than NSD.

At a minimum, NSD has to make one system call to receive each request and one to send a response. Recently Linux added the new `recvmmsg()` and `sendmmsg()` system calls to receive and send multiple UDP messages with a single call. These may go some way to improving NSD's performance compared to Namestorm. They are, however, UDP-specific, and `sendmmsg()` requires the application to manage its own transmit-queue batching. When we implemented Namestorm, we already had *libnmio*, which abstracts

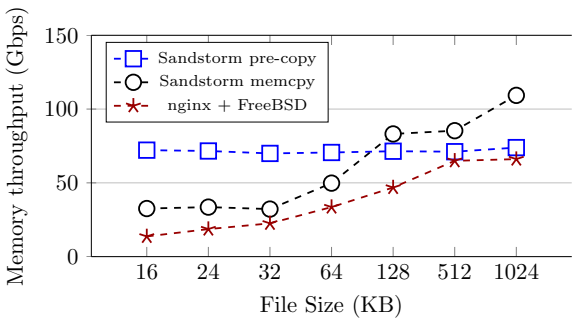


Figure 9: Sandstorm memory read throughput, 6 NICs.

and handles all the batching interactions with netmap, so there is no application-specific batching code in Namestorm.

3.5 DDIO

With DDIO, incoming packets are DMAed directly to the CPU's L3 cache, and outgoing packets are DMAed directly from the L3 cache, avoiding round trips from the CPU to the memory subsystem. For lightly loaded servers in which the working set is smaller than the L3 cache, or in which data is accessed with temporal locality by the processor and DMA engine (e.g., touched and immediately sent, or received and immediately accessed), DDIO can dramatically reduce latency by avoiding memory traffic. Thus DDIO is ideal for RPC-like mechanisms in which processing latency is low and data will be used immediately before or after DMA. On heavily loaded systems, it is far from clear whether DDIO will be a win or not. For applications with a larger cache footprint, or in which communication occurs at some delay from CPU generation or use of packet data, DDIO could unnecessarily pollute the cache and trigger additional memory traffic, damaging performance.

Intuitively, one might reasonably assume that Sandstorm's pre-copy mode might interact best with DDIO: as with `sendfile()` based designs, only packet headers enter the L1/L2 caches, with payload content rarely touched by the CPU. Figure 9 illustrates a therefore surprising effect when operating on small file sizes: overall memory throughput from the CPU package, as measured using performance counters situated on the DRAM-facing interface of the LLC, sees significantly less traffic for the memcpy implementation relative to the pre-copy one, which shows a constant rate roughly equal to network throughput.

We believe this occurs because DDIO is, by policy, limited from occupying most of the LLC: in the pre-copy cases, DDIO is responsible for pulling untouched data into the cache – as the file data cannot fit in this subset of the cache, DMA access thrashes the cache and all network transmit is done from DRAM. In the memcpy case, the CPU loads data into the cache, allowing more complete utilization of the cache for network data. However, as the DRAM memory interface is not a bottleneck in the system as configured, the net result of the additional memcpy, despite better cache utilization, is reduced performance. As file sizes increase, the overall footprint of memory copying rapidly exceeds the LLC size, exceeding network throughput, at which point pre-copy becomes more efficient. Likewise, one might mistakenly believe simply from inspection of CPU memory counters that nginx is somehow benefiting from this same effect: in fact, nginx is experiencing CPU saturation, and it is not until file size reaches 512K that sufficient CPU is available to converge with pre-copy's saturation of the network link.

By contrast, Namestorm sees improved performance using the memcpy implementation, as the cache lines holding packet data must be dirtied due to protocol requirements, in which case performing the memcpy has little CPU overhead yet allows much more efficient use of the cache by DDIO.

It is much more difficult to reason about the interaction between a conventional operating system, applications like nginx, and DDIO's effect on L3 cache behavior. Ideally, we would experiment by disabling DDIO and monitoring the L3 cache miss rate, but there is no way to disable it on the Xeon E5 CPUs we have used, nor modify the policy that controls the fraction of the cache used by DDIO.

4. DISCUSSION

We developed Sandstorm and Namestorm to explore the hypothesis that fundamental architectural change might be required to properly exploit rapidly growing CPU core counts and NIC capacity. Comparisons with Linux and FreeBSD appear to confirm this conclusion far more dramatically than we expected: while there are small-factor differences between Linux and FreeBSD performance curves, we observe that their shapes are fundamentally the same. We believe that this reflects near-identical underlying architectural decisions stemming from common intellectual ancestry (the BSD network stack and sockets API) and largely incremental changes from that original design.

Sandstorm and Namestorm adopt fundamentally different architectural approaches, emphasizing transparent memory flow within applications (and not across expensive protection-domain boundaries), process-to-completion, heavy amortization, batching, and application-specific customizations that seem antithetical to general-purpose stack design. The results are dramatic, accomplishing near-linear speedup with increases in core and NIC capacity – completely different curves possible only with a completely different design.

4.1 Current network-stack specialization

Over the years there have been many attempts to add specialized features to general-purpose stacks such as FreeBSD and Linux. Examples include `sendfile()`, primarily for web servers, `recvmmsg()`, mostly aimed at DNS servers, and assorted socket options for telnet. In some cases, entire applications have been moved to the kernel [13, 24] because it was too difficult to achieve performance through the existing APIs. The problem with these enhancements is that each serves a narrow role, yet still must fit within a general OS architecture, and thus are constrained in what they can do. Special-purpose userspace stacks do not suffer from these constraints, and free the programmer to solve a narrow problem in an application-specific manner while still having the other advantages of a general-purpose OS stack.

4.2 The generality of specialization

Our approach tightly integrates the network stack and application within a single process. This model, together with optimizations aimed at cache locality or pre-packetization, naturally fit a reasonably wide range of performance-critical, event-driven applications such as web servers, key-value stores, RPC-based services and name servers. Even rate-adaptive video streaming may benefit, as developments such as MPEG-DASH and Apple's HLS have moved intelligence to the client leaving servers as dumb static-content farms.

Not all network services are a natural fit. For example, CGI-based web services and general-purpose databases have inherently different properties and are generally CPU- or filesystem-intensive, deemphasizing networking bottlenecks. In our design, the control loop and transport-protocol correctness depend on the timely execution of application-layer functions; blocking in the application cannot be tolerated. A thread-based approach might be more suitable for such cases. Isolating the network stack and application into different threads still yields benefits: OS-bypass networking costs less, and

saved CPU cycles are available for the application. However, such an approach requires synchronization, and so increases complexity and offers less room for cross-layer optimization.

We are neither arguing for the exclusive use of specialized stacks over generalized ones, nor deployment of general-purpose network stacks in userspace. Instead, we propose selectively identifying key scale-out applications where informed but aggressive exploitation of domain-specific knowledge and micro-architectural properties will allow cross-layer optimizations. In such cases, the benefits outweigh the costs of developing and maintaining a specialized stack.

4.3 Tracing, profiling, and measurement

One of our greatest challenges in this work was the root-cause analysis of performance issues in contemporary hardware-software implementations. The amount of time spent analyzing network-stack behavior (often unsuccessfully) dwarfed the amount of time required to implement Sandstorm and Namestorm.

An enormous variety of tools exist – OS-specific PMC tools, lock contention measurement tools, tcpdump, Intel vTune, DTrace, and a plethora of application-specific tracing features – but they suffer many significant limitations. Perhaps most problematic is that the tools are not *holistic*: each captures only a fragment of the analysis space – different configuration models, file formats, and feature sets.

Worse, as we attempted inter-OS analysis (e.g., comparing Linux and FreeBSD lock profiling), we discovered that tools often measure and report results differently, preventing sensible comparison. For example, we found that Linux took packet timestamps at different points than FreeBSD, FreeBSD uses different clocks for DTrace and BPF, and that while FreeBSD exports both per-process and per-core PMC stats, Linux supports only the former. Where supported, DTrace attempts to bridge these gaps by unifying configuration, trace formats, and event namespaces [15]. However, DTrace also experiences high overhead causing bespoke tools to persist, and is unintegrated with packet-level tools preventing side-by-side comparison of packet and execution traces. We feel certain that improvement in the state-of-the-art would benefit not only research, but also the practice of network-stack implementation.

Our special-purpose stacks are synchronous; after netmap hands off packets to userspace, the control flow is generally linear, and we process packets to completion. This, combined with lock-free design, means that it is very simple to reason about where time goes when handling a request flow. General-purpose stacks cannot, by their nature, be synchronous. They must be asynchronous to balance all the conflicting demands of hardware and applications, managing queues without application knowledge, allocating processing to threads in order to handle those queues, and ensuring safety via locking. To reason about performance in such systems, we often resort to statistical sampling because it is not possible to directly follow the control flow. Of course, not all network applications are well suited to synchronous models; we argue, however, that imposing the asynchrony of a general-purpose stack on all applications can unnecessarily complicate debugging, performance analysis, and performance optimization.

5. RELATED WORK

Web server and network-stack performance optimization is not a new research area. Past studies have come up with many optimization techniques as well as completely different design choices. These designs range from userspace and kernel-based implementations to specialized operating systems.

With the conventional approaches, userspace applications [1, 6] utilize general-purpose network stacks, relying heavily on operating-system primitives to achieve data movement and event notification [26]. Several proposals [23, 12, 30] focus on reducing the overhead of such primitives (e.g., KQueue, `epoll`, `sendfile()`). IO-Lite [27] unifies the data management between OS subsystems and userspace applications by providing page-based mechanisms to safely and concurrently share data. Fbufs [17] utilize techniques such as page remapping and shared memory to provide high-performance cross-domain transfers and buffer management. Pesterev and Wickizer [28, 14] have proposed efficient techniques to improve commodity-stack performance by controlling connection locality and taking advantage of modern multicore systems. Similarly, MegaPipe [21] shows significant performance gain by introducing a bidirectional, per-core pipe to facilitate data exchange and event notification between kernel and userspace applications.

A significant number of research proposals follow a substantially different approach: they propose partial or full implementation of network applications in kernel, aiming to eliminate the cost of communication between kernel and userspace. Although this design decision improves performance significantly, it comes at the cost of limited security and reliability. A representative example of this category is kHTTPd [13], a kernel-based web server which uses the socket interface. Similar to kHTTPd, TUX [24] is another noteworthy example of in-kernel network applications. TUX achieves greater performance by eliminating the socket layer and pinning the static content it serves in memory. We have adopted several of these ideas in our prototype, although our approach is not kernel based.

Microkernel designs such as Mach [10] have long appealed to OS designers, pushing core services (such as network stacks) into user processes so that they can be more easily developed, customized, and multiply-instantiated. In this direction, Thekkath et al [32], have prototyped capability-enabled, library-synthesized userspace network stacks implemented on Mach. The Cheetah web server is built on top of an Exokernel [19] library operating system that provides a filesystem and an optimized TCP/IP implementation. Lightweight libOSes enable application developers to exploit domain-specific knowledge and improve performance. Unikernel designs such as MirageOS [25] likewise blend operating-system and application components at compile-time, trimming unneeded software elements to accomplish extremely small memory footprints – although by static code analysis rather than application-specific specialization.

OS-bypass and userspace network processing is another technique explored by the academic community with several studies such as Arsenic [29], U-Net [33], Linux PF_RING [16], as well as netmap [31]. lwIP [4] is a lightweight general-purpose network stack which was designed for low-end, embedded devices; although performance and specialization are not their primary goals, we have borrowed several ideas for our TCP API. mTCP [22] is another recent effort in userspace networking that demonstrates significant performance improvements: likewise, this work is related, but differs from ours in that it offers a general-purpose stack aiming to minimize integration effort for existing applications, while our proposal sacrifices backward compatibility in favor of specialization and performance. Finally, Solarflare OpenOnload [8] is one of the most sophisticated commercial proposals: it is a hybrid stack, capable of operating both in usermode and kernel, minimizing interrupts, data copies, and context switches. It requires, however, vendor-specific hardware (NIC) and likewise retains the BSD sockets API rather than exploiting domain-specific knowledge and cross-layer optimizations.

6. CONCLUSION

In this paper, we have demonstrated that specialized userspace stacks, built on top of netmap framework, can vastly improve the performance of scale-out applications. These performance gains

sacrifice generality by adopting design principles at odds with contemporary stack design: application-specific cross-layer cost amortizations, synchronous and buffering-free protocol implementations, and an extreme focus on interactions between processors, caches, and NICs. This approach reflects a widespread adoption of scale-out computing in data centers, which deemphasizes multifunction hosts in favor of increased large-scale specialization. Our performance results are compelling: a 2–10× improvement for web service, and a roughly 9× improvement for DNS service. Further, these stacks have proven easier to develop and tune than conventional stacks, and their performance improvements are portable over multiple generations of hardware.

General-purpose operating system stacks have been around a long time, and have demonstrated the ability to transcend multiple generations of hardware. We believe the same should be true of special-purpose stacks, but that tuning for particular hardware should be easier. We examined performance on servers manufactured seven years apart, and demonstrated that although the performance bottlenecks were now in different places, the same design delivered significant benefits on both platforms.

7. ACKNOWLEDGMENTS

We thank Peter Tsonchev, Arvind Jadoo and Cristian Petrescu for their help in the genesis of this work, and Luigi Rizzo for his feedback on the ideas in this paper. We also gratefully acknowledge Scott Long, Adrian Chadd, and Lawrence Stewart at Netflix, and Jim Harris from Intel's Storage Division, for their assistance in performance measurement, analysis, and optimization. Finally, we thank David Chisnall, Nikola Gvozdiev, Charalampos Rotsos, Malte Schwarzkopf, Bjoern Zeeb, our anonymous reviewers, and our shepherd Paul Barford for their insightful comments.

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 and FA8750-11-C-0249, by EMC/Isilon, by Google, Inc, and by the EU CHANGE FP7 project. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

8. REFERENCES

[1] Apache http server. `http://httpd.apache.org/`.

[2] DNSPerf. `http://nominum.com/measurement-tools/`.

[3] Lighttpd. `http://www.lighttpd.net/`.

[4] lwIP. `http://savannah.nongnu.org/projects/lwip/`.

[5] Name Server Daemon. `http://www.nlnetlabs.nl/nsd/`.

[6] Nginx web server. `http://nginx.org/`.

[7] OpenLiteSpeed. `http://open.litespeedtech.com/`.

[8] Solarflare OpenOnload. `http://www.openonload.org/`.

[9] weighttp: a lightweight benchmarking tool for webservers. `http://redmine.lighttpd.net/projects/weighttp/wiki`.

[10] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tavanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the 1986 USENIX Annual Technical Conference*, ATC '86. USENIX, June 1986.

[11] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky. Overclocking the Yahoo!: CDN for faster web page loads. In *Proceedings of the 2011 Internet Measurement Conference*. ACM, 2011.

[12] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, ATC '99. USENIX, 1999.

[13] M. Bar. Kernel Korner: kHTTPd, a Kernel-Based Web Server. *Linux Journal*, 2000(76), Aug. 2000.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 2010 USENIX Conference on Operating Systems Design and Implementation*, OSDI '10. USENIX, 2010.

[15] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, ATC '04. USENIX, 2004.

[16] L. Deri et al. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE 2004*. NLUUG, 2004.

[17] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93. ACM, 1993.

[18] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*. ACM, 2008.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 251–266. ACM, 1995.

[20] S. Galperin, S. Santesson, M. Myers, A. Malpani, and C. Adams. X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol-OCSP. 2013.

[21] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation*, OSDI '12. USENIX, 2012.

[22] E. 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. In *Proceedings of the 2014 USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14. USENIX, 2014.

[23] J. Lemon. KQueue: A Generic and Scalable Event Notification Facility. In *Proceedings of the 2001 USENIX Annual Technical Conference, FREENIX track*. USENIX, June 2001.

[24] C. Lever, M. A. Eriksen, and S. P. Molloy. An Analysis of the TUX Web Server. Technical Report 00-8, Center for Information Technology Integration (CITI), University of Michigan, Ann Arbor, MI, Nov. 2000.

[25] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the 2013 International Conference on Architectural support for Programming Languages and Operating Systems*, ASPLOS '13. ACM, 2013.

[26] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, Reading, MA, USA, April 2004.

[27] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1), 2000.

[28] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12. ACM, 2012.

[29] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit Ethernet interface. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, 2001.

[30] N. Provos and C. Lever. Scalable Network I/O in Linux. In *Proceedings 2000 USENIX Annual Technical Conference*, ATC '00. USENIX, 2000.

[31] L. Rizzo. netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12. USENIX, 2012.

[32] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking (TON)*, 1(5):554–565, 1993.

[33] 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 1995 ACM Symposium on Operating Systems Principles*, SOSP '95. ACM, 1995.