Network Stack Specialization for Performance

Ilias Marinos
University of Cambridge

Robert N.M. Watson
University of Cambridge

Mark Handley
University College London

ABSTRACT

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

This paper presents Sandstorm, a clean-slate userspace network stack that exploits knowledge of web server semantics, improving throughput over current off-the-shelf designs while retaining use of conventional operating-system and programming frameworks. Based on Netmap, our novel approach merges application and network-stack memory models, aggressively amortizes stack-internal TCP costs based on application-layer knowledge, tightly couples with the NIC event model, and exploits low-latency hardware access. We compare our approach to the FreeBSD and Linux network stacks with nginx as the web server, demonstrating ~3.5x throughput improvement, while experiencing low CPU utilization, linear scaling on multicore systems, and saturating current 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

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 big 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 90's. 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 will revisit the idea of specialized network stacks. In particular we develop Sandstorm, a specialized user-space stack for serving static web content. Importantly, however, our approach does not simply shift the network stack to userspace: we also promote tight integration and specialization of application and stack, achieving cross-layer optimizations antithetical to current design practices.

Servers such as Sandstorm could be used to serve images such as the Facebook logo, or as front end caches to popular dynamic content. This is a role that conventional stacks should be good at: nginx uses the `sendfile()` system call to hand over serving static content to the operating system. FreeBSD and Linux then implement zerocopy 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 should be hard to beat.

Our userspace web server implementation is built upon FreeBSD's Netmap [20] 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 it can achieve more than three times the throughput on datacenter-style networks when serving small image files typical of many web pages. Our implementation does not currently take advantage of TSO, so we believe that further performance improvements may also be possible.

Hotnets '13, November 21–22, 2013, College Park, MD, USA.

The demonstrated performance improvement comes from three places. First, we achieve a complete zerocopy stack, 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.

In this paper we discuss many of the issues that affect performance in conventional stacks, present a complete system design that avoids these pitfalls, and analyze where the performance improvements come from compared to a conventional stack that uses `sendfile()`. Further, we argue that the stack-as-library approach does actually support software reuse; for example, most of our TCP implementation could be reused, and would especially suit applications such as RPC that demand low latency.

2. SPECIAL PURPOSE ARCHITECTURE

What is the minimum 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, the implementations of these do not need to conform to the conventional sockets model, split between userspace and kernel, or even implement features such as dynamic TCP segmentation. For a web server that serves static content such as the Facebook logo to huge numbers of clients, 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:

- Conventional network stacks support zerocopy for OS-maintained data – e.g., file-system blocks in the buffer cache, but not for application-provided HTTP headers. Can we implement a complete zerocopy stack where packet buffers are passed from the NIC all the way to the application and vice versa for all data?

- Conventional stacks make extensive use of queueing and buffering in order to mitigate context switches and keep CPUs and NICs busy, at the cost of substantially increased cache footprint. 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 – e.g., 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 so that after the first client is served, no work whatsoever is 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 collocating 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 the above, is there a programming abstraction that allows components to be reused for other applications that may benefit from server specialization?

2.1 A zerocopy web server

Luigi Rizzo's Netmap 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 ring. Combined with batching to reduce system call overhead, 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 can 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.

The majority of the work is performed while processing an incoming TCP ACK. The IP header is checked; if it is acceptable, a hash table is used to locate the TCB. The offset of the ACK number from the start of the connection is used to directly locate the next pre-packaged packet to send and, if permitted by the congestion and receive windows, any subsequent packets that also can be sent. To transmit these packets, the destination address and port are rewritten, and the TCP and IP checksums incrementally updated. The packet is then sent directly to the NIC using Netmap. All reads of the ACK header and modifications of the transmitted packet are performed in a single pass, ensuring both headers and the TCB remain in the CPU's L1 cache.

Under high workloads it is possible that a second connection may need to send the same packet before it has finished being DMAed to the NIC. Thus more than one copy of each packet will need to be prepared to allow two packets containing the same data but different destinations to be in the DMA ring at a time. Our current implementation does this in advance, but it would be better to copy on demand whenever the high-water mark is raised, and then retain and reuse the new copies to avoid copying for subsequent connections.

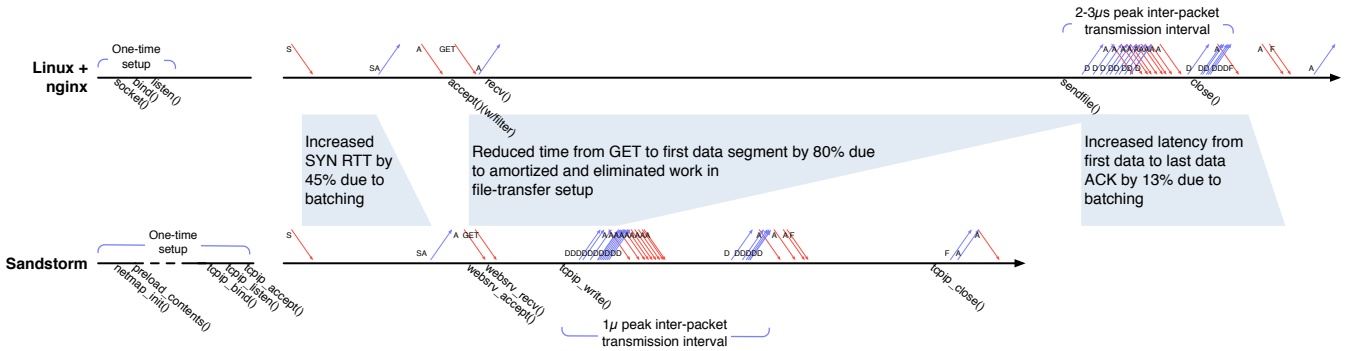


Figure 1: Several tradeoffs are visible in these packet traces taken from busy (but unsaturated) Linux and Sandstorm servers.

Figure 1 illustrates the tradeoff space through traces taken from busy (but unsaturated) Linux and Sandstorm servers. A batched design has a measurable effect on TCP round-trip time. On the other hand, Sandstorm amortizes or eliminates substantial parts of per-request processing through a more efficient architecture. Under this light load, these benefits are pronounced; at saturation the effect is even more significant.

2.2 Microkernel services on monolithic kernels

Although monolithic kernels are the *de facto* standard for networked system deployment, robustness and flexibility concerns continue to drive development towards microkernel-like approaches. Sandstorm offers several of their qualities:

Rapid deployment & reusability: Our prototype stack is highly modular, and synthesized from bottom-up using traditional dynamic libraries as building blocks (*components*) to construct a special-purpose system. Each library component corresponds to a stand-alone service that exposes a well-defined API. Sandstorm is built by combining three basic components (see Figure 2):

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

Splitting functionality into reusable stand-alone components does not require us to sacrifice the benefits of exploiting cross-layer knowledge to optimize performance. For example, the presented web server application 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 as easy, if not easier, than conventional systems, as application-specific performance-centric code shifts out of the kernel into 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.
- 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.

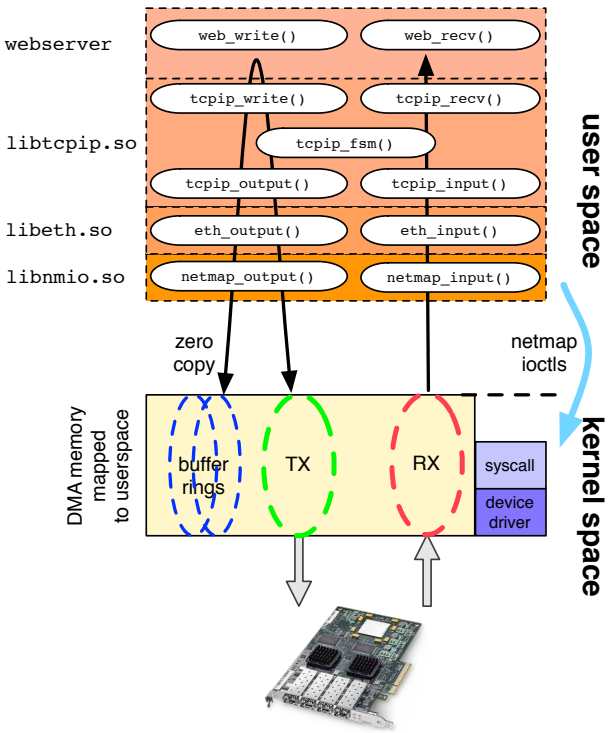
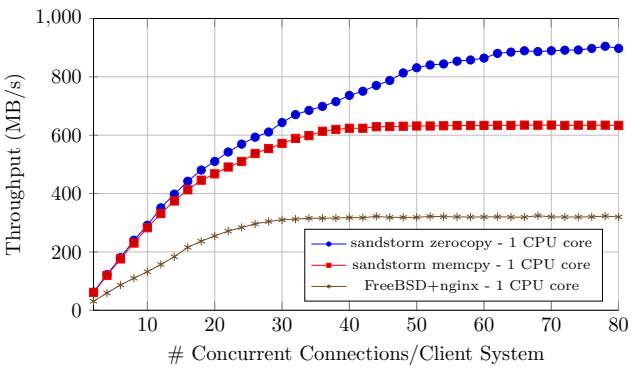


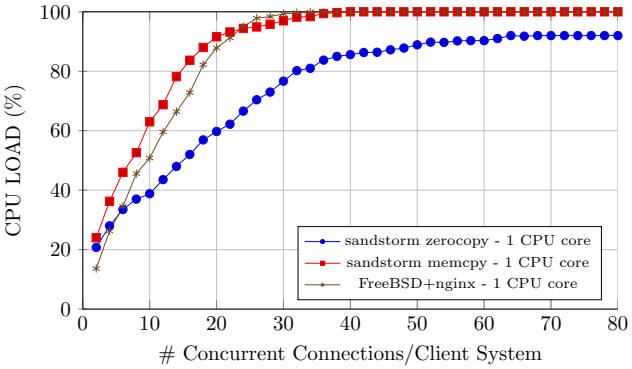
Figure 2: High-level architecture view

3. EVALUATION

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) Network throughput (old hw), 1 NIC



(b) CPU utilization (old hw), 1 NIC

Figure 3: Serving a ˜22KB image on a single-core server over 10GbE

a high-performance, low-footprint web server that adopts a non-blocking, event-driven model: it relies on OS primitives (e.g. `kqueue()`) for readiness event notifications as well as data movement (e.g. `sendfile()`) and asynchronously processes requests.

Contemporary web pages are immensely content-rich, but they are mainly comprised 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 [5]. The conventional approach taken by the network stack and web server application has been tuned to perform well when delivering large content by utilizing OS primitives and NIC hardware features. On the contrary, multiple simultaneous short-lived HTTP connections are considered a heavy workload that stresses the kernel-userspace interface and reveals performance bottlenecks: the size of the transmitted data is not enough to compensate for the system cost. Based on this insight, we have tailored the testbed setup to explore the behaviour of our system when faced with these challenges.

To explore possible Sandstorm performance gains, 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 3.10.10 and FreeBSD 10-CURRENT with NetFlix-originated patches optimizing the kernel flowtable and `sendfile()`. For all the benchmarks presented, we have configured nginx to serve content from a ramdisk in order to eliminate any disk bottlenecks. We used weighttp [3] for load generation.

For the old hardware, we use three systems, two clients and one server, all of which are connected to a 10G crossbar switch. All test systems are equipped with Intel 82598EB dual port 10GbE NIC, 8GB RAM, and two 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 2013, these are well-equipped contemporary servers.

3.1 Old hardware

In Figure 3, we present network throughput as a function of concurrent requests. Sandstorm achieves substantially higher throughput than the conventional model, even for the minimum number of concurrent connections tested. For example, it exhibits a 35% performance gain with just two simultaneous connections. The limiting factor for such a workload is not link capacity but rather latency, which is significantly lower with our stack even though a little transmit delay is intentionally introduced so as to achieve efficient batching.

One of the key design decisions in our system architecture is the use of zerocopy transmissions. To evaluate the effectiveness of this choice, we explore the performance of a Sandstorm variant which relies on a single memory copy to create the packets for transmission. Figure 3 shows a 30% network throughput decrease as result of the associated memory copy overhead.

In order to confirm this observation, we experimented with a different setup: we invoked two memcopy-Sandstorm instances on two different 10GbE NICs and pinned them to different CPUs. This time the requests from each client system were served by a single stack instance. The rationale be-

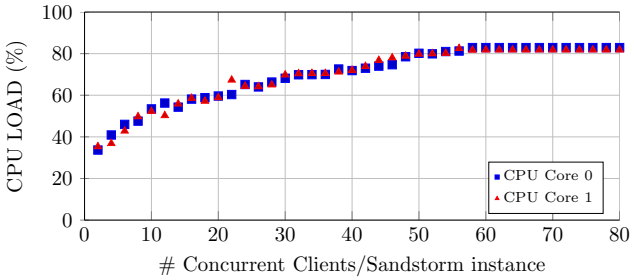
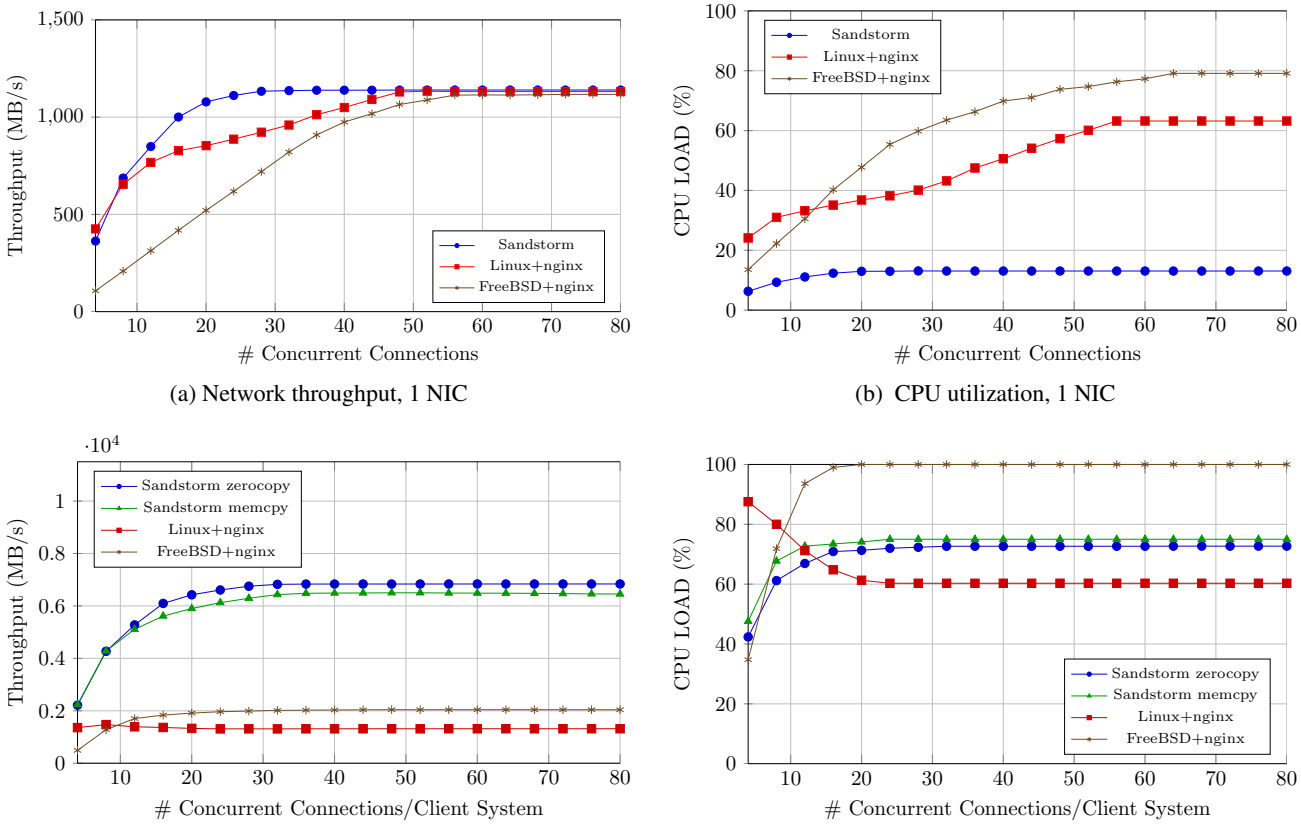


Figure 4: CPU Load per Sandstorm instance



(a) Network throughput, 1 NIC

(b) CPU utilization, 1 NIC

(c) Network throughput, 6 NICs

(d) CPU utilization, 6 NICs

Figure 5: Serving a ˜22KB file over HTTP

hind this scenario was that memory-associated bottlenecks would manifest as increased CPU load for both stack instances, although each of them was actually serving only half the workload compared to the single core case (one client system per Sandstorm instance instead of two). Figure 4 confirms our intuition; we observe only 10% CPU load improvement compared to the single-core instance while only serving half the previous load on each CPU core.

It was not feasible to fully saturate the single NIC with the available clients and such small files, although the zerocopy Sandstorm comes close. As the file size increases, the gap closes somewhat, but at 100KB file size, our single-core Netmap stack is still 54% faster than nginx, and has saturated the 10Gbps link.

3.2 New hardware

One of the key differences of the newer hardware over the old one is the support for Data Direct I/O, a feature that allows Direct Memory Access (DMA) originating over PCIe from the NIC to access the processor's Last-Level Cache (LLC). With this hardware the performance improvements over the conventional stacks are even greater: Sandstorm saturates a single 10GbE NIC at only 13% CPU load and scales linearly up to six NICs achieving ˜55Gbps at ˜73% CPU load (see Figure 5). Both Linux and FreeBSD manage to saturate a single NIC, although the actual throughput at medium loads is significantly lower than the one achieved by Sandstorm due to latency (see Figure 6). Moreover, the conventional stacks prove unable to scale with multiple NICs: they both top at ˜16Gbps with four NICs. Interestingly, we have experienced a collapse on Linux with six NICs.

Memory copies on newer hardware prove to be less expensive than on older hardware. Our microbenchmarks reveal that the reason the memcopy variant achieves slightly lower throughput than the zerocopy variant is that it uses 5-10% more CPU cycles per request. It has not saturated the memory, but memory latency does show up as increased CPU cycles per request. The configuration only uses three cores (one per two NICs) and so the cores are saturated at approximately 75% CPU load. Slightly higher throughput might be possible with asymmetric allocation of NICs to cores, but it would make a direct comparison harder - in any event the zero copy variant is already saturating all six NICs.

We could not evaluate Sandstorm's scalability for more than six NICs since we did not have available PCIe slots; however, due to the low CPU utilization and no evidence of memory bandwidth bottlenecks, we believe there are still available resources to scale to seven and perhaps eight NICs.

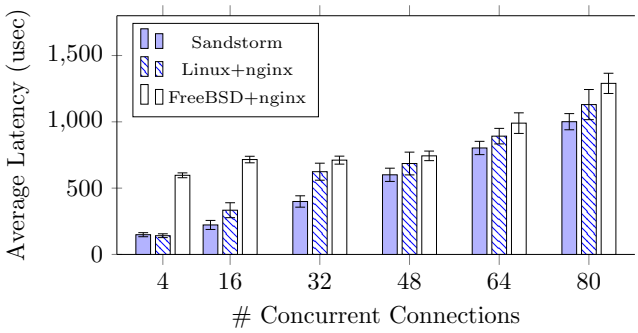


Figure 6: Average Latency (new hardware), 1 NIC

4. RELATED WORK

Web server and network stack performance optimization is not a new research area. Past studies have come up with several optimization techniques as well as completely different design choices. Existing approaches in HTTP server and network stack designs range from user-space and kernel-based implementations up to specialized operating systems.

With conventional approaches, userspace applications [1, 2] utilize the native general-purpose network stacks by heavily relying on operating-system primitives to achieve data movement and event notification [15]. Several proposals [6, 12, 19] focused on reducing the overhead of such primitives (e.g. KQueue, epoll, sendfile). IO-Lite [16] unifies the data management between OS subsystems and userspace applications by providing mechanisms to safely and concurrently share data. Pesterev and Wickizer [8, 17] proposed efficient techniques to improve performance of commodity stacks by controlling network connection locality and taking advantage of modern multicore systems. Similarly, MegaPipe [11] shows significant gains by introducing a bidirectional, per-core pipe to facilitate data exchange and event notification between kernel and user space 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 [7], a kernel-based web server which uses the socket interface. Similar to kHTTPd, TUX [13] is another noteworthy example of in-kernel network applications. TUX achieves great performance by eliminating the socket layer and by pinning the static content it serves in memory. We have adopted several of these ideas in our prototype stack, although it is not a kernel-based approach.

Microkernel designs such as Mach [4] have long appealed to OS designers, pushing core OS services (such as network stacks) into user processes so that they can be more easily developed, customized, and multiply-instantiated. The Cheetah webserver was built on top of the Exokernel [10] 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 [14] likewise blend operating-system and application components at compile-time, trimming unneeded software elements to accomplish extremely small memory footprints – although not necessarily implying application-specific specialization of OS services. OS-bypass with userspace network processing is another technique explored by the academic community with several studies such as Arsenic [18], U-Net [21], Linux PF_RING [9] and Netmap [20].

5. CONCLUSIONS

We have described Sandstorm, a high-performance web server based on a specialized network stack exploiting a total-system zerocopy design, aggressive cross-layer amortization and information flow, and a synchronous structure simultaneously promoting simple design and extremely high performance. The performance impact of these specializations is particularly noticeable in data center environments due to a dramatic drop in the effective round-trip time (across interconnect and server-side software) allowing short TCP connections to fill the available pipe much more quickly than conventional designs.

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. It is our belief that a blend of specialized network stacks with performance-critical applications such as web services, can dramatically improve the scalability of current hardware, reducing costs and energy demands, offering a practical alternative to general-purpose designs.

6. ACKNOWLEDGMENTS

We would like to thank Peter Tsonchev, Arvind Jadoo and Cristian Petrescu for their help in the genesis of this work, Luigi Rizzo for his insightful feedback on the ideas, Scott Long, Adrian Chadd, and Lawrence Stewart at Netflix, and Jim Harris at Intel, for their assistance in performance measurement and optimization.

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

References

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

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

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

[4] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, *Mach: A New Kernel Foundation for UNIX Development*, Computer Science Department, Carnegie Mellon University, 1986.

[5] Mohammad Al-Fares, Khaled Elmeleegy, Benjamin Reed, and Igor Gashinsky, *Overclocking the Yahoo!: CDN for faster web page loads*, Proceedings of the 2011 acm sigcomm conference on internet measurement conference, 2011, pp. 569–584.

[6] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel, *A scalable and explicit event delivery mechanism for UNIX*, Proceedings of the annual conference on USENIX Annual Technical Conference, 1999, pp. 19–19.

[7] Moshe Bar, *Kernel Korner: kHTTPd, a Kernel-Based Web Server*, Linux Journal **2000** (August 2000), no. 76es.

[8] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich, *An analysis of Linux scalability to many cores* (2010).

[9] Luca Deri et al., *Improving passive packet capture: Beyond device polling*, Proc. of SANE, 2004.

[10] Dawson R Engler, M Frans Kaashoek, et al., *Exokernel: An operating system architecture for application-level resource management*, ACM SIGOPS Operating Systems Review, 1995, pp. 251–266.

[11] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy, *MegaPipe: a new programming interface for scalable network I/O*, Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, 2012, pp. 135–148.

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

[13] Chuck Lever, Sun-Netscape Alliance, and Stephen P Molloy, *An analysis of the TUX web server*, Ann Arbor **1001** (2000), 48103–4943.

[14] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft, *Unikernels: library operating systems for the cloud*, Proceedings of the eighteenth international conference on architectural support for programming languages and operating systems, 2013, pp. 461–472.

[15] Marshall Kirk McKusick and George V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, Pearson Education, 2004.

[16] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel, *IO-Lite: A unified I/O buffering and caching system*, Operating systems review **33** (1998), 15–28.

[17] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris, *Improving network connection locality on multicore systems*, Proceedings of the 7th ACM european conference on Computer Systems, 2012, pp. 337–350.

[18] Ian Pratt and Keir Fraser, *Arsenic: A user-accessible gigabit ethernet interface*, INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, 2001, pp. 67–76.

[19] Niels Provos and Chuck Lever, *Scalable network I/O in Linux*, Proceedings of the annual conference on USENIX Annual Technical Conference, 2000, pp. 38–38.

[20] Luigi Rizzo, *netmap: a novel framework for fast packet I/O*, Proceedings of the 2012 USENIX conference on Annual Technical Conference, 2012, pp. 9–9.

[21] T. von Eicken, A. Basu, V. Buch, and W. Vogels, *U-Net: a user-level network interface for parallel and distributed computing*, Proceedings of the fifteenth ACM symposium on Operating systems principles, 1995, pp. 40–53.