

Disk|Crypt|Net: rethinking the stack for high-performance video streaming

Ilias Marinos
University of Cambridge

Robert N.M. Watson
University of Cambridge

Mark Handley
University College London

Randall R. Stewart
Netflix Inc.

ABSTRACT

Conventional operating systems used for video streaming employ an in-memory disk buffer cache to mask the high latency and low throughput of disks. However, data from Netflix servers show that this cache has a low hit rate, so does little to improve throughput. Latency is not the problem it once was either, due to PCIe-attached flash storage. With memory bandwidth increasingly becoming a bottleneck for video servers, especially when end-to-end encryption is considered, we revisit the interaction between storage and networking for video streaming servers in pursuit of higher performance.

We show how to build high-performance userspace network services that saturate existing hardware while serving data directly from disks, with no need for a traditional disk buffer cache. Employing netmap, and developing a new *diskmap* service, which provides safe high-performance userspace direct I/O access to NVMe devices, we amortize system overheads by utilizing efficient batching of outstanding I/O requests, process-to-completion, and zerocopy operation. We demonstrate how a buffer-cache-free design is not only practical, but required in order to achieve efficient use of memory bandwidth on contemporary microarchitectures. Minimizing latency between DMA and CPU access by integrating storage and TCP control loops allows many operations to access only the last-level cache rather than bottle-necking on memory bandwidth. We illustrate the power of this design by building Atlas, a video streaming web server that outperforms state-of-the-art configurations, and achieves ~72Gbps of plaintext or encrypted network traffic using a fraction of the available CPU cores on commodity hardware.

CCS CONCEPTS

• **Networks** → **Network services**; • **Software and its engineering** → **Operating systems**;

KEYWORDS

Network stacks; Storage stacks; Network Performance

ACM Reference format:

Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|Crypt|Net: rethinking the stack for high-performance video streaming. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017*, 14 pages. <https://doi.org/10.1145/3098822.3098844>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-4653-5/17/08...\$15.00

<https://doi.org/10.1145/3098822.3098844>

1 INTRODUCTION

More than 50% of Internet traffic is now video streamed from services such as Netflix. How well suited are conventional operating systems to serving such content? In principle, this is an application that might be well served by off-the shelf solutions. Video streaming involves long-lived TCP connections, with popular content served directly from the kernel disk buffer cache using the OS `sendfile` primitive, so few context switches are required. The TCP stack itself has been well tuned over the years, so this must be close to a best-case scenario for commodity operating systems.

Despite this, Netflix has recently committed a number of significant changes to FreeBSD aimed at improving streaming from their video servers. Perhaps current operating systems are not achieving close to the capabilities of the underlying hardware after all?

Previous work[18] has shown that a specialized stack can greatly outperform commodity operating systems for short web downloads of static content served entirely from memory. The main problem faced by the conventional stack for this workload was context switching between the kernel and OS to accept new connections; this solution achieves high performance by using a zero-copy architecture closely coupling the HTTP server and the TCP/IP stack in userspace, using netmap's[29] batching API to reduce the number of context switches to fewer than one per connection.

Such a workload is very different from video streaming; Netflix uses large servers with 12 or more cores and large amounts of RAM, but even so the buffer cache hit ratio is rather low - generally less than 10% of content can be served from memory without going to disk. At the same time, hardware trends point in the opposite direction: SSDs have moved storage much closer to the CPU, particularly in the form of NVMe PCIe-attached drives, and future non-volatile memory may move it closer still. In addition, on recent Intel CPUs, DMA to and from both storage and network devices is performed using DDIO[10] directly to the L3 cache rather than RAM. Storage latencies are now lower than typical network latencies. If we no longer need a disk buffer cache to mask storage latencies, can we rethink how we build these servers that stream the majority of Internet content?

We set out to answer this question. First we examine Netflix's changes to the FreeBSD operating system to understand the problems they faced building high-performance video servers. The story has become more complicated recently as the need for privacy has caused video providers to move towards using HTTPS for streaming. We examine the implications on the performance of the Netflix stack.

We then designed a special purpose stack for video streaming that takes advantage of low-latency storage. Our stack places the SSD directly in the TCP control loop, closely coupling storage, encryption, and the TCP protocol implementation. Ideally, a chunk of video content would be DMAed to the CPU's Last Level Cache (LLC), we could encrypt it in place to avoid thrashing the LLC and packetize it, then DMA it to the NIC, all without needing the data

to touch RAM. In practice, for the sort of high workloads Netflix targets, this ideal cannot quite be achieved. However we will show that it is possible to achieve approximately 70Gb/s of encrypted video streaming to anywhere between 6000 and 16000 simultaneous clients using just four CPU cores without using a disk buffer cache. This is 5% better than the Netflix stack can achieve using eight cores when all the content is already in the disk buffer cache, 50% better than the Netflix stack achieves when it has to fetch content from the SSD, and 130% more than stock FreeBSD/Nginx. Through a detailed analysis of PMC data from the CPU, we investigate the root causes of these performance improvements.

2 THE VIDEO STREAMING PROBLEM

Modern video streaming is rate-adaptive: clients on different networks can download different quality versions of the content. A number of standards exist for doing this, including Apple's HTTP Live Streaming[14], Adobe HTTP Dynamic Streaming[2] and MPEG-DASH[20]. Although they differ in details, all these solutions place the rate-adaptive intelligence at the client. Video servers are a dumb Content Distribution Network (CDN) delivering video files over HTTP or HTTPS, though they are often accessed through a DNS-based front-end that manages load across servers and attempts to choose servers close to the customer. Once the client connects, a steady stream of HTTP requests is sent to the server, requesting chunks of video content. HTTP persistent connections are used, so relatively few long-lived TCP connections are needed.

Video servers are, therefore, powerful and well-equipped general-purpose machines, at least in principle. All they do is repeatedly find the file or section of file corresponding to the chunk requested, and return the contents of that file over TCP. This should be a task for which conventional operating systems such as Linux and FreeBSD are well optimized. The main problem is simply the volume of data that needs to be served. How fast can a video server go?

In December 2015 the BBC iPlayer streaming service was achieving 20Gb/s[4] from a server using nginx on Linux, and featuring 24 cores on two Intel Xeon E5-2680v3 processors, 512 GB DDR4 RAM, and a 8.6TB RAID array of SSDs. This is expensive hardware, and 20Gb/s, while fast, is well below the memory bandwidth, disk bandwidth and network capacity. Is it possible to do better?

2.1 Case Study: The Netflix Video Streamer

Netflix is one of the largest video streaming providers in the world. During peak hours, Netflix along with YouTube video streaming traffic accounts for well over 50% of the US traffic [30]. To serve this traffic, Netflix maintains its own CDN infrastructure, located in PoPs and datacenters worldwide. Their server appliances use FreeBSD and the nginx web server, serving the video and audio components to their customers over HTTP [21] or, more recently, HTTPS. The servers run mostly a read-only workload while serving content; during scheduled content updates they serve fewer clients than normal.

Historically, to respond to an HTTP request for static content, a web server application would have to invoke `read` and `write` system calls consecutively to transfer data from a file stored on disk to a network socket. In the best case scenario, the file would already be present in the disk buffer cache, and then the read would complete

quickly; otherwise it would have to wait for the file to be fetched from disk and DMAed to RAM. This approach introduces significant overheads; the application spends a great deal of time blocking for I/O completion, and the contents of the file are redundantly copied to and from userspace, requiring high context switch rates, without the web server ever looking at the contents.

Modern commodity webservers offload most of this work to the kernel. Nginx uses the `sendfile` system call to allow a zero-copy transfer of data from the kernel disk-buffer cache to the relevant socket buffers without the need to involve the user process. Since the Virtual File System (VFS) subsystem and the disk buffer cache are already responsible for managing the in-memory representation of files, this is also the right place to act as an interface between the network and storage stacks. Upon `sendfile` invocation, the kernel maps a file page to a `sendfile` buffer (`sf_buf`), attaches an `mbuf` header and enqueues it to the socket buffer. With this approach, unnecessary domain transitions and data copies are completely avoided.

The BBC iPlayer servers used commodity software—nginx on Linux—using `sendfile` in precisely this way. Netflix, however, has devoted a great deal of resources to optimize further the performance of their CDN caches.

Among the numerous changes Netflix has made, the most important key bottlenecks that have been addressed include:

- The synchronous nature of `sendfile`.
- Problems with VM scaling when thrashing the disk buffer cache.
- Performance problems experienced at the presence of high ingress packet rates.
- Performance problems when streaming over HTTPS.

We will explore these changes in more detail, as they cast important light on how modern server systems scale.

2.1.1 Asynchronous `sendfile`

The `sendfile` system call optimizes data transfers, but requires blocking for I/O when a file page is not present in memory. This can greatly hinder performance with event-driven applications such as nginx. Netflix servers have large amounts of RAM—192GB is common—but the video catalog on each server is much larger. Buffer cache hit rates of less than 10% are common on most servers. This means that `sendfile` will often block, tying up nginx resources.

Netflix implemented a more sophisticated approach known as *asynchronous `sendfile`*. The system call never blocks for I/O, but instead returns immediately before the I/O operation has completed. The `sendfile` buffers with the attached `mbufs` are enqueued to the socket, but the socket is only marked ready for transmission when all of the inflight I/O operations have completed successfully. Upon encountering a failed I/O operation the error is irrecoverable: the socket is marked accordingly so that the application receives an error at a subsequent system call and closes it.

Netflix upstreamed their asynchronous `sendfile` implementation to the mainline FreeBSD tree in early 2016.

2.1.2 VM scaling

With a catalogue that greatly exceeds the DRAM size, and with asynchronous `sendfile` being more aggressive, the VM subsystem became a bottleneck in performance. In particular, upon VM page

exhaustion, all VM allocations were being blocked, waiting for pages to be reclaimed by the paging daemon, and stalling actual work.

Netflix uses several techniques to mitigate this problem. First, DRAM is divided into smaller partitions, each assigned to different *fake* NUMA domains, with a smaller number of CPU cores given affinity to each domain. This gives more efficient scaling with multiple cores by reducing lock contention. Second, in situations where free memory hits a low watermark, proactive scans reclaim pages in the VM page allocation context, avoiding the need to wake the paging daemon. Finally, reclaimed pages are re-enqueued to the inactive memory queues in batches to amortize the lock overhead.

2.1.3 RSS-assisted TCP LRO

Large Receive Offload (LRO) is a common technique used to amortize CPU usage when experiencing high rate inbound TCP traffic. The LRO engine aggregates consecutive packets that belong to the same TCP session before handing them to the network stack. This way, per-packet processing costs can be significantly reduced. To be CPU-efficient, the coalescing window for LRO aggregation is usually bounded by a predefined timeout or a certain number of packets. With thousands of TCP connections, packets belonging to the same session are likely to arrive interleaved with many other packets, making LRO less effective.

To tackle this problem, Netflix uses *RSS-assisted LRO*: it sorts incoming TCP packets into buckets based on their RSS (Receive Side Scaling) hash and the time at the end of the interrupt. This ordering brings packets from a flow that arrived widely separated in time together, so they appear to have arrived consecutively. As a result they can be successfully merged when they are fed to the LRO engine. This optimization helped reduce CPU utilization by ~5-30%, depending on the congestion control algorithm, and the interrupt coalescing tuning parameters.

2.1.4 In-kernel TLS

End-to-end encryption introduces a new challenge in building high-performance network services. Suddenly, optimized zerocopy interfaces such as `sendfile` are rendered useless, since they conflict with the fundamental nature of encrypted traffic. The kernel is unaware of the TLS protocol and it is no longer possible to use zerocopy operations from storage to the network subsystem. To serve data over a TLS connection, the conventional stack needs to fall back to userspace using traditional POSIX reads and writes when performing encryption. This reintroduces overheads that have been completely eliminated in the case of plaintext transfers. Netflix initially reported that enabling TLS decreased throughput on their servers from 40Gb/s to 8.5Gb/s[33].

To regain the advantages of `sendfile` for encrypted traffic, Netflix devised a hybrid approach to split work between kernel and userspace: the TLS session management, and negotiation remains in the userspace TLS library (openssl), but the kernel is now modified to include bulk data encryption in the `sendfile` pipeline. The TLS handshake is still handled by the userspace TLS library. Once the session keys are derived, nginx uses two new socket options to communicate the session key to the kernel, and to instruct the kernel to enable encryption on that socket. Once a *ChangeCipherSuite*

message is sent from the application, the in-kernel TLS state machine arms encryption on that particular socket. When `sendfile` is issued on a TLS socket, the kernel hands over the data to one of the dedicated TLS kernel threads for encryption, and only enqueues them to the socket buffer after encryption has been completed.

This approach brings most of the original `sendfile`'s benefits, but the semantics are no longer the same as in the plaintext case: the kernel cannot perform in-place encryption of data, as this would invalidate the buffer cache entries. Instead of being zero-copy, once the file is in the buffer cache, `sendfile` then needs to clone the data to another buffer; this can then be used to hold the ephemeral encrypted data, and mapped to the socket buffer.

2.2 Netflix Performance

We have used the Netflix software stack in order to demonstrate the effectiveness of the aforementioned performance improvements. Our video server is equipped with an Intel Xeon E5-2667-v3 8-core CPU, 128GB RAM, two dual-port Chelsio T580 40GbE NIC adaptors, and four Intel P3700 NVMe disks with 800GB capacity each. Our full test setup is described in more detail in §4. We stress-tested the system using HTTP persistent connections retrieving 300KB video chunks as we increase the number of simultaneous active clients.

Figure 1 shows throughput (left) and CPU utilization (right) as the number of concurrent HTTP persistent connections is varied. We show curves for the Netflix stack and for the stock nginx/FreeBSD stack for both the case where all content is served from the disk buffer cache (100% BC) and where all content needs to be fetched from the SSDs (0% BC). Normal Netflix workloads are nearer the latter than the former.

When all content is served from the buffer cache, there is no significant difference in performance, either in throughput or CPU utilization. This is expected, as the Netflix improvements do not tackle this easy case. When content has to be served from SSDs, the Netflix improvements show their effectiveness, almost doubling throughput from 39Gb/s to 72Gb/s. However, all eight cores are almost saturated at this workload.

Figure 2 shows the equivalent throughput and CPU utilization curves for encrypted transfers. The Netflix stack gives substantial performance improvements over stock nginx/FreeBSD, but now the cores are all saturated, leading to a drop in performance. When all data must be fetched from SSD, performance drops to 47Gb/s, a reduction of 35% compared to unencrypted streaming, despite the kernel TLS implementation.

Why does this performance reduction occur? Modern Intel CPUs use AESNI instructions and can reduce the encryption overhead to as low as 1 CPU cycle/byte, provided that the data are warm in the LLC. We used the CPU performance counters to understand further what is going on.

When serving plaintext content from the buffer cache, we see that Netflix *write* memory throughput is low, 20Gb/s, but *read* memory throughput is around 100Gb/s. This is not unreasonable, as the data does not need to be DMAed from the disk. When fetching data from the SSD, *write* memory throughput rises by about 70Gb/s, as data is DMAed to RAM from disk, and *read* memory throughput increases to 120Gb/s. We also see a fairly high rate, 90 million/sec, of reads due to LLC misses. In principle it should be possible to serve this

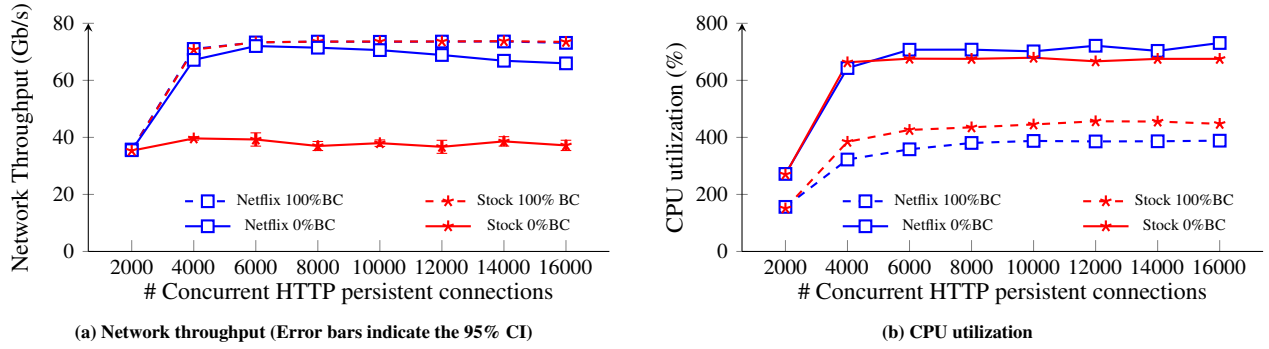


Figure 1: Plaintext performance, Netflix vs Stock FreeBSD, zero and 100% Buffer Cache (BC) ratios.

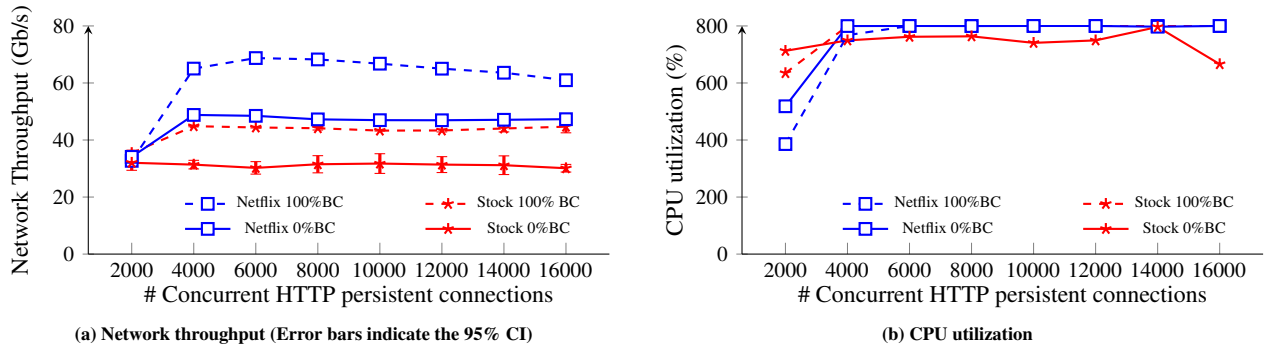


Figure 2: Encrypted performance, Netflix vs Stock FreeBSD, zero and 100% Buffer Cache (BC) ratios.

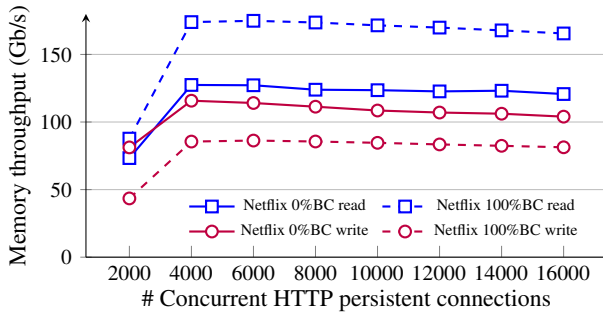


Figure 3: Encrypted Netflix memory performance

workload with ~ 72 Gb/s of read and write memory throughput, so the Netflix stack is working memory a little harder than is strictly necessary, even with plaintext workloads.

When serving encrypted content, the story becomes more complicated. The Netflix memory throughput is shown in Figure 3. Irrespective of whether data comes from the disks or from the buffer cache, memory read throughput is approximately 2.6 times the network throughput. Indeed, the 175 Gb/s read rate when serving from the disk buffer cache is getting closer to the memory speed of this hardware, indicating that memory accesses are likely to be a bottleneck. The system also shows a high rate of LLC miss events—200

million/sec—indicating that the cores are now waiting on memory much of the time, and explaining why CPU utilization is 100%.

2.3 Discussion

Netflix optimizations have clearly delivered significant improvements in the video streaming performance of FreeBSD, both for serving plaintext and encrypted content. However, it is also clear that memory is being worked very hard when serving these workloads. With a conventional stack it is extremely hard to pin down precisely why this is the case. We have profiled the stack, and with Netflix's VM improvements there are no obvious bottlenecks remaining.

Current Intel CPUs DMA to and from the LLC using DDIO, rather than direct to memory. In principle it ought to be possible to DMA data from the SSD and then DMA it to the NIC without it ever touching main memory. Would it also be possible to encrypt that data as it passes through the LLC? With a conventional stack though, it is clear that this is not happening. We speculate that this is because the stack is too asynchronous. Data is DMAed from the SSD to disk buffer cache, initially landing in the LLC. However, it is not immediately consumed, so gets flushed to memory. Subsequently the kernel copies the buffer, loading it into the LLC as a side effect. If it is not immediately encrypted it gets flushed again. The kernel goes to encrypt the data, causing it to be re-loaded into the LLC. The encrypted data is not immediately sent by TCP, so it gets flushed a third time. Finally it is DMAed to the NIC, requiring it to be loaded

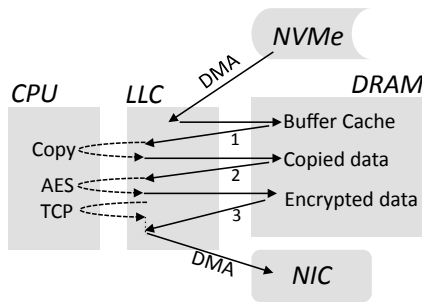


Figure 4: Possible Memory Accesses with the Netflix Stack

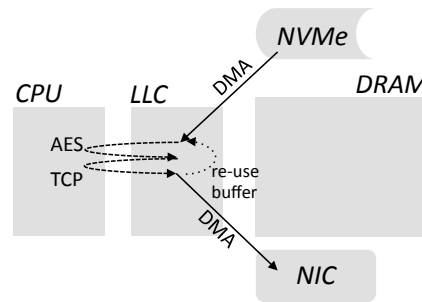


Figure 5: Desired Memory Accesses with Specialized Stack.

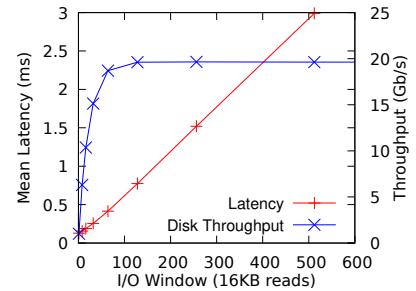


Figure 6: NVMe Controller Latency and Throughput

again. This process is shown in Figure 4, and requires three reads from memory, compared to the 2.6 times we observe, so presumably sometimes the data is not flushed, saving some memory reads.

Netflix’s newest production streamers are equipped with latest 16-core (32 hyperthreads) Intel CPUs and 100GbE NICs, but their maximum target service throughput is limited to ~90Gb/s. At this rate, they use 460Gbit/sec of read/write memory bandwidth—this is 96% of the measured hardware limits, and approximately five times the network throughput. This ratio is in general agreement with the results in Figure 3. At this utilization, CPU stalls waiting for memory become a major problem, and CPU utilization varies considerably, ranging from 75% to 90% with only small changes in demand.

3 TOWARDS A SPECIALIZED VIDEO STREAMING STACK

In a conventional stack, it is very hard to avoid all the memory traffic that we saw with the Netflix stack. Could a specialized, much more synchronous stack do better? How might we architect such a stack so that data remains in the LLC to the maximum extent possible?

As buffer cache hit ratios are so low with Netflix’s workload, clearly we need to design a system that works well when data has to be fetched from the SSD. The storage system needs to be very tightly coupled to the rest of the stack, so that as soon as data arrives from storage in the LLC, it is processed to completion by the rest of the application and network stacks without the need for any context switching or queuing. To accomplish this, data need to be fetched from storage *just-in-time*: in the typical case, the storage system must be clocked off the TCP protocol’s ACK clock, because arriving TCP ACKs cause TCP’s congestion window to inflate, allowing the transmission of more data. Only when this happens can data from the storage system be immediately consumed by the network without adding a queue to cope with rate mismatches.

The outline of a solution then looks like:

- (1) A TCP ACK arrives, freeing up congestion window.
- (2) This triggers the stack to request more data from the SSD to fill that congestion window.
- (3) The SSDs return data, ideally placing it in the LLC.
- (4) The read completion event causes the application to encrypt the data in-place, add TCP headers, and trigger the transmission of the packets.

- (5) The network completion event frees the buffer, allowing it to be reused for a subsequent disk read.

This design closely couples all the pipeline stages, so maximises use of the LLC, and never copies any data, though it does encrypt in place. The hope is that memory accesses resemble Figure 5.

For this to work, the SSD must be capable of very low latency, as it is placed directly in the TCP ACK-clock loop, and it must simultaneously be capable of high throughput. Today’s PCIe-attached NVMe SSDs have low latency, but before building a system, we profiled some drives to see how well they balance throughput and latency. Figure 6 shows the request latency and disk throughput as a function of the I/O window when making 16KB reads from an Intel P3700 800GByte NVMe drive. NVMe drives maintain a request queue, and the I/O window is the number of requests queued but not yet completed. It is clear that with an I/O window of around 128 requests, these drives can achieve maximum throughput while simultaneously maintaining latency of under 1ms. This is significantly smaller than the network RTT over typical home network links, even to a server in the same city, indicating that we should be able to place such an SSD directly into the TCP control loop.

3.1 Storage Stack

Traditional OS storage stacks pay a price in terms of efficiency to achieve generality and safety. These inefficiencies include copying memory between kernel and userspace, extra abstraction layers such as the Virtual File System, as well as POSIX API overheads needed for backwards compatibility and portability. However, all these overheads used to be negligible when compared to the latency and throughput of spinning disks.

Today PCIe-attached flash and the adoption of new host controller interfaces such as NVMe radically change the situation. Off-the-shelf hardware can achieve read throughput up to 28Gbps and access latencies as low as 20 μ s for short transfers [23]. However, if we wish to integrate storage into the network fast path, we cannot afford to pay the price of going through the conventional storage stack.

Our approach is to build a new high performance storage stack that is better suited to integration into our network pipeline. Before we explain its design and implementation though, we provide a brief overview of how NVMe drives interface with the operating system.

Function	Parameters	Description
<code>nvme_open()</code>	I/O qpair control block, character device, buffer memory size, flags	Configure, initialize, and attach to NVMe disk's queue pair.
<code>nvme_read()</code>	I/O description block (<code>struct iodesc</code>), metadata, error	Craft and enqueue a <i>READ</i> I/O request for a particular disk, namespace, starting offset, length, destination address etc.
<code>nvme_write()</code>	I/O description block (<code>struct iodesc</code>), metadata, error	Craft and enqueue a <i>WRITE</i> I/O request for a particular disk, namespace, starting offset, length, source buffer etc.
<code>nvme_sqsync()</code>	I/O qpair control block	Update the NVMe device's queue pair doorbell (via a dedicated <code>ioctl</code>) to initiate processing of pending I/O requests.
<code>nvme_consume_completions()</code>	I/O qpair control block, number of completions to consume	Consumes completed I/O requests (takes care of out-of-order completions). Invokes a per I/O request specific callback, set by the application layer (via <code>struct iodesc</code>).

Table 1: libnvme API functions (not exhaustive).

3.1.1 NVMe disk operation and data structures

PCIe NVMe disk controllers use circular queues of command descriptors residing in host memory to serve I/O requests for disk logical blocks (LBAs). The host places NVMe I/O commands in a *submission queue*; each command includes the operation type (e.g., READ, WRITE), the initial LBA address, the length of the request, the source or destination buffer address in host main memory, and various flags. Once commands have been enqueued, the device driver notifies the controller that there are requests waiting by updating the submission queue's tail doorbell—this is a device register, similar to NIC TX and RX doorbells for packet descriptors. Multiple I/O commands can be in progress at a time, and the disk firmware is allowed to perform out-of-order completions. For this to work, each submission queue is associated with a *completion queue*. This is used by the disk to communicate I/O completion events to the host. The OS is responsible for consuming command completions, and then notifies the controller via a completion queue doorbell so that completion slot entries can be reused.

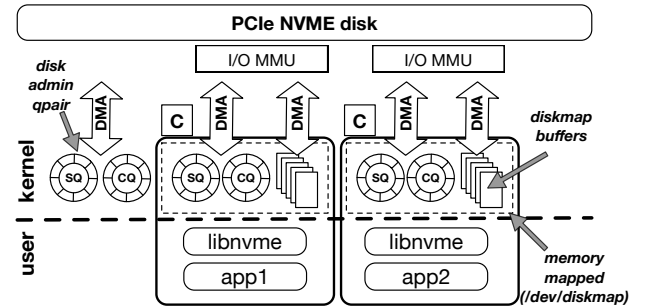
Unlike older SATA/AHCI Solid State Disks, which usually feature a single submission/completion queue pair with a limited number of slot entries, NVMe devices support a highly configurable number of queue pairs and depths, which greatly helps with scaling to multiple CPU cores and permits a share-free, lockless design.

3.1.2 Diskmap

Inspired by *netmap* [29], a high-performance packet processing framework which maps the NIC buffer rings to userspace, we designed and built *diskmap*, a conceptually similar system that uses kernel-bypass to allow userspace applications to directly map the memory used to DMA disk logical blocks to and from NVMe storage. From a high-level viewpoint, *diskmap* and *netmap* have many similarities, but the two DMA models and the nature of operations are fundamentally different, so a different approach is required.

With *diskmap*, userspace applications are directly responsible for crafting, enqueueing, and consuming I/O requests and completions, while the OS uses hardware capabilities to enforce protection and synchronization. The system is comprised of two parts:

- A kernel module used to initialize and configure devices that are to be used in *diskmap*-mode, as well as providing a thin syscall layer to abstract DMA operation initiation,

**Figure 7: High-level architecture of a diskmap application.**

- An accompanying userspace library which implements the NVMe driver and provides the API (see table 1) to abstract typical operations to the device such as read and write.

The architecture is shown in Figure 7. When the *diskmap* kernel module loads, the NVMe device is partially detached from the in-kernel storage stack: the actual datapath queue pairs are now disconnected and readily available for attaching to user-level applications. The device administration queue pairs, however, remain connected to the conventional in-kernel stack without being exposed to userspace. This allows all low-level configuration and privileged operations (e.g., device reset, `nvmeformat`) to continue working normally. It should be straightforward to allow some of the datapath queue pairs to remain connected to the in-kernel stack, possibly operating on different NVMe namespaces, but our implementation currently does not support this mode of operation.

During initialization, the kernel pre-allocates non-pageable memory for all the objects that are required for NVMe device operations, including submission and completion queues, PRP lists [22], and the *diskmap* buffers themselves. These will be shared by the NVMe hardware and the userspace applications and used for data transfers. Relative addressing is used to calculate pointers for any object in the shared memory region in a position-independent manner. Each *diskmap* buffer descriptor holds a set of metadata information: a unique index, the current buffer length, and the buffer address that the *libnvme* library uses when constructing NVMe I/O commands. The buffer size and queue depths are configurable via `sysctl` knobs. Similarly to *netmap*, the shared memory area is exposed by the kernel via a dedicated character device.

To connect to the *diskmap* data path, a userspace application maps the shared memory into its own virtual address space and issues a

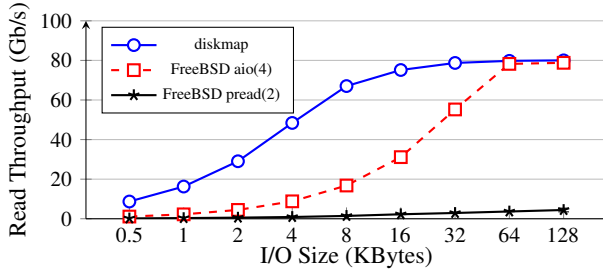


Figure 8: Read throughput, *diskmap* vs. *aio(4)* vs. *pread(2)*

dedicated *diskmap ioctl* to indicate the disks and queue pairs that should be attached, as well as the number of *diskmap* buffers required. This functionality is abstracted within a single *libnvmf* library call (see table 1). When an application calls *nvmf_read* or *nvmf_write*, the library is responsible for translating the I/O request for certain disk blocks into the corresponding NVMe commands and enqueueing them in the disk submission queue. The application layer then invokes a system call to update the relevant disk doorbell and initiate processing of the pending I/O commands.

Unlike their POSIX equivalents which block until the I/O is completed, *libnvmf* facilitates a non-blocking, event-driven model. With each I/O request, the application specifies a callback function which will be invoked upon I/O completion. A high-level I/O request might need to be split into several low-level NVMe commands, and this complicates the handling of out-of-order completion. *Libnvmf* hides this complexity, and only invokes the application-specified callback function when all the in-flight NVMe commands that comprise that particular transfer have completed.

Diskmap enforces memory safety by taking advantage of the IOMMU on newer systems. When an application requests the attachment of a datapath queue pair and a number of *diskmap* buffers, the kernel maps the relevant shared memory to the PCIe device-specific IOMMU page table. Since all the buffers are statically pre-allocated by the kernel upon initialization, there is no need to dynamically update the IOMMU page table with transient mappings and unmappings, which would otherwise significantly affect performance [3, 19]. *Diskmap* could also operate in an unsafe manner using direct physical memory addresses if the IOMMU is disabled. In both cases, userspace remains unaware of the change, and there is no special handling required either in the *libnvmf* library or the application itself.

3.1.3 *Diskmap* Performance

Before we integrate *diskmap* into our video server, we wish to understand how well it performs. Figure 8 shows the read throughput obtained using *diskmap* for a range of I/O request sizes. A single *diskmap* thread binds to four NVMe disks, fills their submission queues, and we measure throughput. We compare *diskmap* against FreeBSD *pread*, and *aio* (also single threads). *Aio* is a native FreeBSD interface which uses *kqueue* [17] and *kevent* to allow asynchronous I/O requests to be initiated by userspace. It is highly optimized, and allows I/O request batching with a single system call to increase performance. When an NVMe interrupt indicates completion, userspace is notified via *kqueue*.

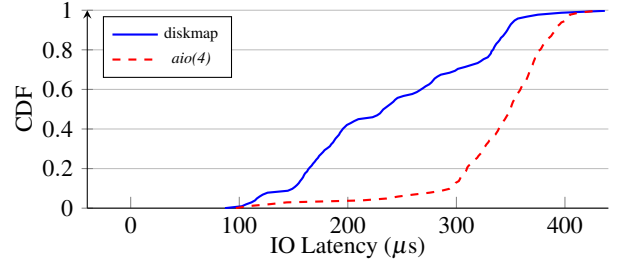


Figure 9: *diskmap* vs. *aio(4)* - I/O latency, read size: 512 bytes, I/O window: 128 requests

Although *aio* performs well for large reads, it is less stellar with smaller requests. *Diskmap* exhibits much higher throughput than *aio* unless request sizes are at least 64KB in size. With *diskmap* there is a sweet spot around 16KB—here, performance is close to the limits of the hardware, but the transfers are still small enough that they are comparable to today’s default TCP initial window size. This makes *diskmap* an excellent fit for our video server.

Figure 9 shows latency when using 512 byte read requests, and an I/O window of 128 requests. Such small requests stress the stack; despite this *diskmap* exhibits very low latency. Finally, Figure 6 was gathered using *diskmap*, and shows that both low latency and high throughput can be obtained simultaneously.

3.1.4 To batch or not to batch

One technique often used to improve throughput is batching. Modern NICs are able to achieve very high packet rates even with small transfer units; for example more than 40Mpps is possible with 64 byte packets [9]. For the CPU to keep up, batching is required to amortize the system call overhead (e.g., in the case of *netmap*) and the cost of accessing/updating the device doorbell registers. With today’s NVMe disks the situation is different: the minimum transfer unit of these devices typically ranges from 512 to 4096 bytes, and the IOPS rates that can be achieved are much lower than the NIC equivalent packet rates. We find that the CPU can fill the disk firmware pipeline and saturate the device for the whole range of supported I/O lengths per operation without needing to batch requests. In situations where the CPU is nearly saturated though, batching is still efficient in saving CPU cycles by amortizing the system call overhead.

3.2 Network Stack Integration

In a conventional configuration the work to send persistent files from disks to the network is distributed across many different subsystems that operate asynchronously and are loosely co-scheduled. In contrast, we seek tighter control of the execution pipeline. Scheduling work from a single thread, we can come closer to a process-to-completion ideal, minimizing the lifespan of data transfers across all the layers of the stack; from disk to application, from application to the network stack, and finally to the NICs. This should reduce pressure on the LLC and take advantage of contemporary microarchitectural properties such as Intel’s Data Direct IO (DDIO) [10].

To take advantage of *diskmap* in the network fast path, we need to embed the networking code in the same process thread. Several

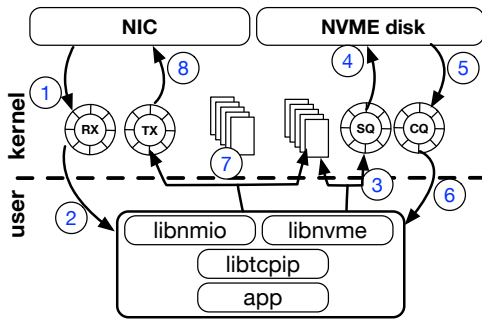


Figure 10: Atlas high-level control flow.

userspace network stacks have been presented recently, demonstrating dramatic improvements both in terms of latency and throughput [15, 16, 18]. We started from the Sandstorm [18] implementation, and modified it accordingly to build our network stack. Sandstorm was originally designed to serve small static objects, typical of web traffic, from the main memory. Although we leverage several of Sandstorm’s original design ideas, modifications were necessary because the requirements are fundamentally different:

- Unlike typical web traffic workloads, which mostly consist of short-lived connections, we want to optimize for long-lived HTTP-persistent connections used by video streaming.
- Content served by a video streamer does not fit in main memory, rendering some of Sandstorm’s optimizations such as multiple packet replicas and content pre-segmentation inapplicable.

Given the workload characteristics, doing network packet segmentation or calculating the checksums of large data in software would be a performance bottleneck. Instead we leverage NIC hardware support for TCP Segmentation Offload (TSO). This required modifications in netmap’s NIC driver, in particular for Chelsio T5 40GbE adapters, and in Sandstorm’s core TCP protocol implementation.

As the movie catalog size is significantly larger than RAM, a video streamer needs to serve ~90% of requests from disk in the typical case (see §2.1). This means that most of the time the OS buffer cache does not really act as a cache anymore; it merely serves as a temporary buffer pool to store the data that are enqueued to socket buffers. At the same time this comes with considerable overhead associated with nominally being a cache, including pressure in the VM subsystem, lock contention and so forth. In Atlas, we completely remove the buffer cache as an interface between the storage and network stacks.

To integrate the storage and network stacks, we implemented a mechanism similar to the conventional stack’s `sendfile`. Upon the reception of an HTTP GET request, the application layer web server issues a `tcpip_sendfile` library call. This call instructs the network stack to attach the in-memory object representation of a persistent file to the particular connection’s TCB. After this point, the network stack is responsible for fetching the data from the disk and transmitting them to the network (see Fig. 10). It invokes a callback to the application layer only when everything has been sent, or some other critical event has been encountered such as the connection being closed by the remote host.

Given the low latency of NVMe disks compared to WAN RTTs, rather than the read-ahead approach used by a conventional stack, we

use an on-demand mechanism to fetch data from disks and transmit them to the network. We have found that to get the peak throughput from NVMe disks, I/O requests larger than 8KB need to be issued. Seeking to optimize for the typical case and achieve the highest throughput from the NVMe disks, we have chosen to delay the I/O requests for a particular flow until the received ACKs inflate the space in the TCP congestion window to a reasonably high value— $10 \times \text{MSS}$ in our implementation. Of course, there are cases where this mechanism cannot be applied: for example, if a TCP connection experiences a retransmit timeout, or the effective window is smaller than this high-watermark value and all sent data is acknowledged, then we fall back issuing smaller I/O requests.

As our stack does not buffer in-flight data sent by TCP, retransmitted data must be re-fetched from disk. We use the TCP sequence number offset of the lost segment to decide which data to re-fetch and retransmit. When encrypted traffic is considered, it is worth noting that this socket-buffer-free approach fits well with ciphersuites like AES GCM [28] which do not require interpacket dependencies to work; instead the GCM counter can be easily derived from the TCP sequence numbers, including for retransmissions.

Our system prototype, Atlas, does not implement a sophisticated filesystem: disks are treated as flat namespaces, and files are laid out in consecutive disk blocks.

4 EVALUATION

We saw in Section 2.2 how the Netflix stack outperforms stock FreeBSD, both on encrypted workloads and with plaintext when the buffer cache hit ratio is low. However, profiling of the Netflix stack indicated potential inefficiencies that appeared to be inherent, and motivated our design of Atlas. We now wish to see to what extent our hypothesis was correct regarding the merits of a special purpose stack, and the need to integrate storage into the TCP control loop. We will compare the performance of Atlas against a Netflix-optimized setup, using contemporary hardware.

Our testbed consists of four machines; one server, two clients, and one middlebox to allow us to emulate realistic network round trip times. The test systems are connected via a 40GbE cut-through switch. Our video server is equipped with an Intel Xeon E5-2667-v3 8-core CPU, 128GB RAM, two dual-port Chelsio T580 40GbE NIC adapters, and four Intel P3700 NVMe disks with 800GB capacity each. Our two systems emulating large numbers of clients are equipped with Intel Xeon E5-2643-v2 6-core CPUs, and 64GB RAM. One uses a dual-port Chelsio T580 40GbE, while the other has an Intel XL710 40GbE controller. Finally, the middlebox has a 6-core Intel E5-2430L-v2 and 64GB RAM.

Atlas runs on FreeBSD 12-CURRENT; for Netflix we use the Netflix production release which is also based on FreeBSD 12, but also includes all the Netflix optimization patches, including those mentioned in §2.1, and tuning. The two client systems run Linux 4.4.8, and finally the middlebox runs FreeBSD 12.

We wish to model how a video streaming server would perform in the wild, but with all our machines connected to the same LAN using 40GbE links, the round-trip times are on the order of a few microseconds. This is not representative of the RTTs seen by production video servers, and would distort TCP behavior. To emulate more realistic latencies, we employ a middlebox which adds latency to

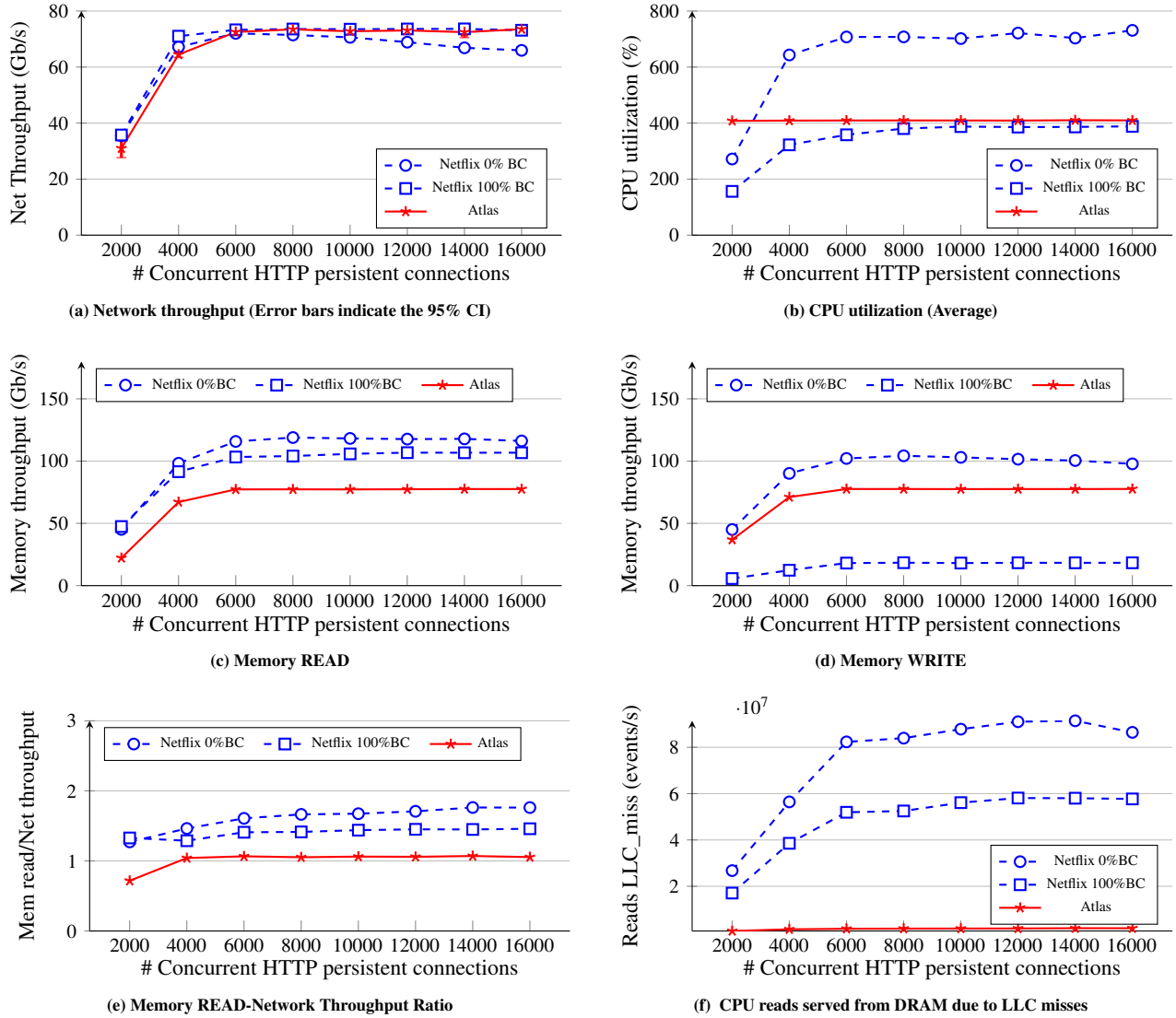


Figure 11: Plaintext performance, Netflix vs. Atlas, zero and 100% Buffer Cache (BC) ratios.

traffic going from the client to the server. The middlebox supports a configurable set of delay bands - we use this feature to add different delays to different flows, with latencies chosen from the range 10 to 40ms. In the middlebox, each newly received packet is hashed and buffered, and a per-flow constant delay¹ is introduced before the packet is forwarded on. To reduce stress on the middlebox and avoid it becoming a bottleneck, we only route client-to-server traffic through it, as the data rate in this direction is much lower.

We wish to test the systems under a range of loads and with varying numbers of clients. As we don't have Netflix's intelligent client, we rely on a load generator that models the sort of requests seen by a video server. We populate the disks with small files (~300KB), each corresponding to the equivalent of a video chunk. We use *weighthttp*

¹This avoids packet reordering within a flow.

to generate HTTP-persistent traffic with multiple concurrent clients². Each client establishes a long-lived TCP connection to the server, and generates a series of HTTP requests with a new request sent immediately after the previous one is served.

The Netflix configuration uses all eight available CPU cores in the video server. For Atlas we only use four CPU cores with one stack instance pinned to each core for the whole range of experiments. We expect each stack instance to bottleneck on different resources depending on the workload: plaintext HTTP traffic should not be a CPU-intensive task and thus we expect the performance of each stack instance to only be limited by the disk; in contrast encrypted network traffic puts heavy pressure both on the Last Level Cache

²This tool has been modified to support requesting multiple URLs.

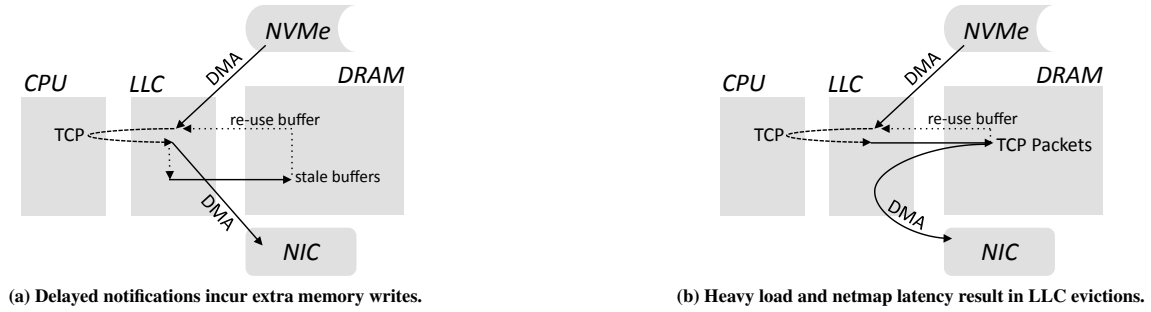


Figure 12: Principal sub-optimal Atlas memory access patterns for unencrypted traffic.

and the CPU cores which need to access and encrypt all the data before they can be transmitted.

4.1 Plaintext HTTP-persistent Traffic

We wish to evaluate the performance of Atlas and Netflix stacks with a plaintext HTTP workload as we vary the number of concurrent active HTTP connections. In the Netflix case we also need to include an extra dimension that impacts performance: the disk buffer cache hit ratio. In these benchmarks we are able to accurately control the disk buffer cache efficiency by adjusting the amount of distinct content that is requested by the clients. In the worst case, each video chunk is only requested once during the duration of the test, requiring the server to fetch all content from the disks; in the best case, the requested content is already cached in main memory and the server does not need to access the disks at all. Atlas does not utilize a buffer cache: all data requests, even repetitive ones, are served from the disk, so Atlas is not sensitive to the choice of workload.

Figures 11a and 11b show the network throughput and CPU utilization achieved by both systems. Atlas and the Netflix setup with a maximally effective buffer cache (100% BC) manage to saturate both the 40GbE NICs, achieving roughly ~73Gbps of HTTP throughput with higher numbers of concurrent connections. For less than 4,000 simultaneous connections Atlas achieves slightly less throughput (~13%) compared to the Netflix setup. We believe that this happens because Atlas often delays making an I/O request until the available TCP window of a flow grows to a larger value (10*MSS) so that it can improve disk throughput with slightly larger reads. Better tuning when the system has so much headroom would avoid this.

With the uncachable workload (0% BC), we observe that Netflix experiences a small performance drop as the number of connections increases. Although VM subsystem pressure is handled by the Netflix configuration much more gracefully than stock FreeBSD, with more connections requesting new data from the disks, the rate of proactive calls to reclaim memory increases, negatively affecting performance. This 0% BC workload comes much closer to the real-world situation: Netflix video streamers typically get low to no benefit from the in-memory buffer cache (<10% hit ratios), except perhaps on occasions when new and very popular content is added to the catalog and a spike in the disk buffer cache efficiency is observed. Atlas does not suffer such a performance drop-off, so is well suited to such uncachable workloads.

It is interesting to observe the CPU utilization of the Netflix setup for the two different workloads. The CPU utilization almost doubles when the buffer cache is thrashed and disk activity is required. It should be noted that the CPU utilization results reported for Atlas are hardly representative of the actual work performed. Atlas relies on polling for disk I/O completions and new packets on the NIC rings, so the CPU cores are constantly spinning even though the actual load might be light, and hence the CPU utilization measured remains constant at ~400% when using four cores.

From a microarchitectural viewpoint, where do Atlas's performance benefits actually come from? Atlas requires ~77Gb/s of memory read and write throughput respectively to saturate the NICs. This comes very close to a one-to-one ratio between network and memory read throughput (see Fig. 11e), indicating that Atlas does not suffer from multiple detours to RAM due to LLC evictions. In contrast, Netflix requires more memory read throughput—almost 1.5× the network throughput—to achieve similar network performance.

Although it is quite efficient, our expectation was that Atlas, due to DDIO, would demonstrate even better memory traffic efficiency. Why is this not the case? If we consider the ratio of memory reads to network throughput when Atlas serves 2,000 clients, we observe that the memory traffic required is about 65% of the network throughput achieved – clearly in this case DDIO is helping save memory traffic. Data is being loaded from storage into the LLC by DMA, and about 35% of it is still in the LLC when it is DMAed to the NIC. For this data, the data flow is like that shown in Figure 12a. Note that the memory write throughput is actually higher than memory read because netmap does not provide timely enough TX completion notifications to allow buffers to be immediately reused. We believe that detours to main memory could be reduced even further if netmap supported a fine-grained, low-latency mechanism to communicate TX DMA completion notifications to userspace applications: such a mechanism would allow us to utilize a strict LIFO approach while recycling DMA I/O buffers that could significantly reduce the stack's working set, increasing DDIO efficacy.

Why is the situation different for higher number of concurrent connections? We believe that the answer is related to the load that the stack experiences; for more than 4K concurrent connections the disks are close to saturation. Atlas then builds deeper queues on the I/O and NIC rings. This increases the working set of the stack since more diskmap buffers need to be associated with I/O commands and connections at each instant. At this point the storage and NIC

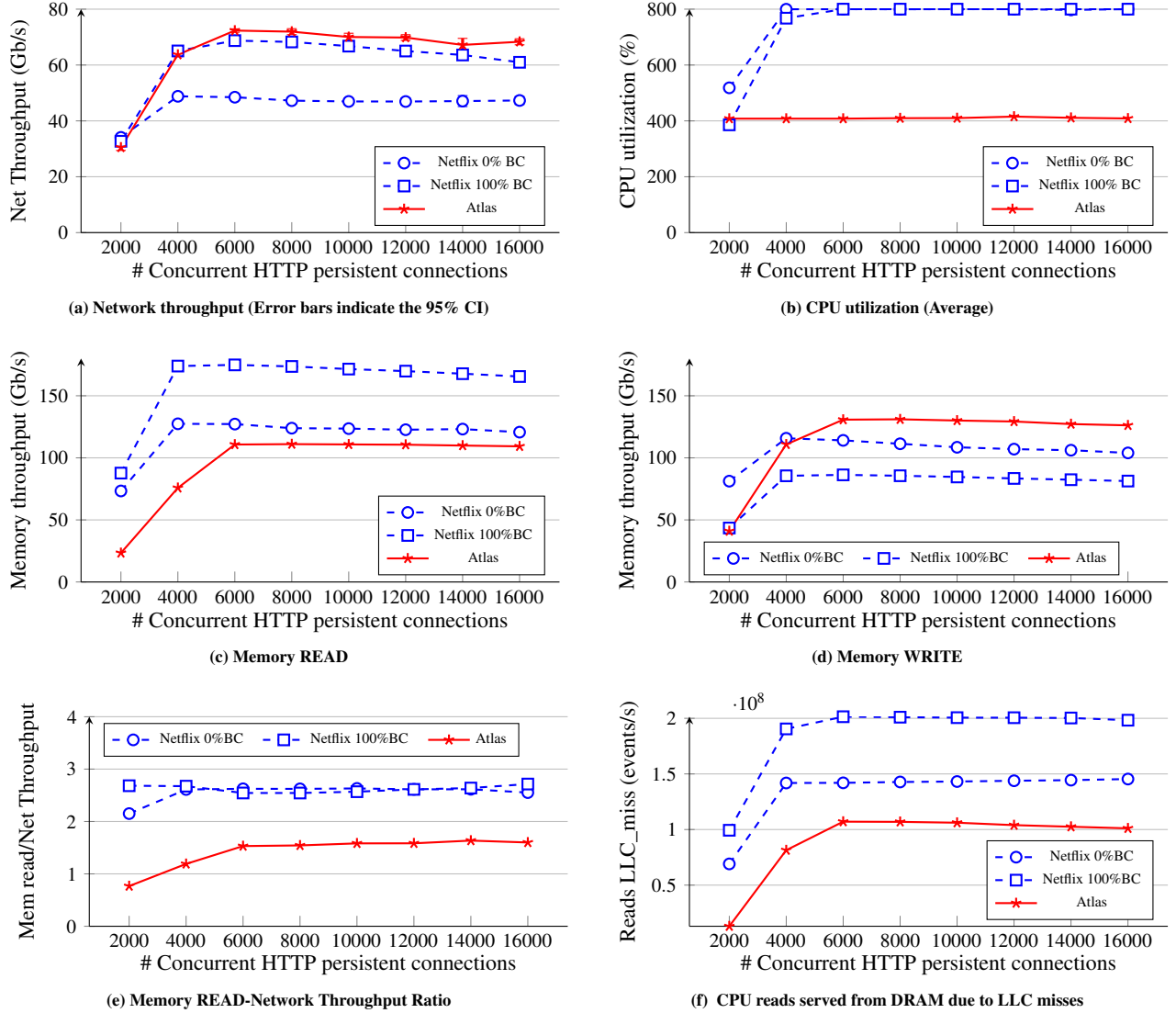


Figure 13: Encrypted performance, Netflix vs. Atlas, zero and 100% Buffer Cache (BC) ratios.

DMA are no longer closely coupled enough in Atlas's event loop so that all data remains in the LLC until transmission. Memory usage looks more like Figure 12b. In any case, when we look at LLC misses (Figure 11f), we see Atlas does not experience any CPU stalls whatsoever while waiting for reads to be served from memory, indicating that the memory read throughput observed is entirely due to DMA to the NIC.

4.2 Encrypted HTTP-persistent Traffic

The need to encrypt traffic significantly complicates the process of serving content. We expect high performance from Netflix's setup, since unlike stock FreeBSD, it can still take advantage of `sendfile` with the in-kernel TLS implementation (§ 2.1). The semantics, however, are different from the plaintext case. In-place

encryption is not an option as it would invalidate the buffer cache entries, so the stack needs to encrypt the data out-of-place, increasing the memory and LLC footprint.

To avoid our tests being impacted by CPU saturation on our client systems, rather than implementing a full TLS layer we have decided to emulate the TLS overhead by doing encryption and authentication of the data with dummy keys before it is actually transmitted. The HTTP headers are still transmitted in plaintext, so that the client software can parse the HTTP response and count the received data without needing to spend additional CPU cycles decrypting data, but the server encrypts everything else as normal. We believe that this setup closely approximates the actual TLS protocol's overheads, especially given that the initial TLS handshake's overhead will be negligible for flow durations encountered with video streaming.

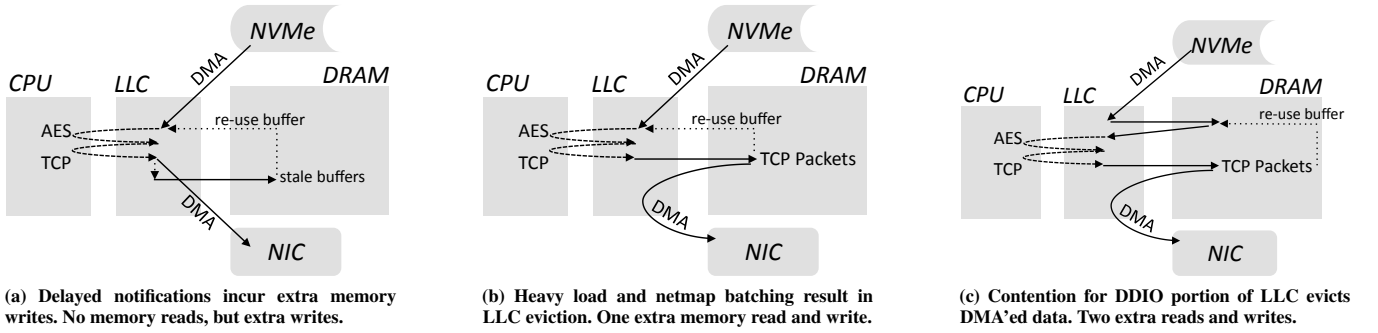


Figure 14: Principal sub-optimal Atlas memory access patterns for encrypted traffic.

For Atlas we used the internal OpenSSL GCM API that takes advantage of ISA extensions. This uses AESNI instructions for encryption and the PCLMUL instruction for ghash, so as to accelerate AES 128bit in Galois Counter Mode [28] (AES128-GCM). For a fair comparison, we modified the Netflix stack to implement a similar design: in particular, we have modified the Netflix implementation to allow plaintext transmission of HTTP headers, which are passed to the kernel as a parameter to the `sendfile` syscall, while the data are still encrypted. The Netflix implementation allows the use of different backends for encryption including support for offloading encryption to special PCIe hardware. Our experiments include results with, according to Netflix, the most optimized software-only implementation: Intel’s ISA-L library, which not only uses ISA extensions to accelerate crypto, but also utilizes non-temporal instructions to reduce pressure on the CPU’s Last Level Cache.

Figures 13a and 13b show network throughput and CPU utilization while serving encrypted traffic with a zero and 100% buffer cache hit ratios. When serving more than 4,000 connections, Atlas achieves higher throughput than Netflix when the buffer hit ratio is 100%, ~72Gb/s as opposed to ~68Gb/s peak throughput for Netflix.

When the workload is not cacheable Atlas, achieves 50% more network throughput than Netflix, while only using four cores. Netflix saturates all the CPU cores even when no disk activity is required, so uncacheable traffic caused storage stack overhead to be introduced, fewer CPU cycles are available for encryption and network protocol processing, greatly reducing throughput.

With under 2,000 active connections, we again see slightly sub-optimal Atlas throughput for the same reasons as with plaintext. As active connections increase, all three curves demonstrate a small performance degradation. This is to be expected when a resource—the CPU in this case—is saturated. Increasing the number of requests can only hinder performance by building deeper queues in stacks and by putting more pressure on memory. However, the reduction is small and both systems handle overload gracefully.

Measuring memory throughput while serving such workloads reveals a big difference between the two systems (Figures 13c and 13d). Clearly encryption affects memory throughput: Atlas memory read throughput reaches ~110Gb/s, roughly a ~43% increase compared to the plaintext case. Netflix, however, requires ~175Gb/s of read throughput when serving the cached workload. When serving the uncacheable workload it requires about ~127Gb/s for more than 4,000 concurrent connections. This might seem counter-intuitive

since the uncacheable workload should trigger more memory traffic due to LLC/memory pressure, but if we look at Fig. 13e, the ratio of memory read throughput to network throughput is actually unchanged at 2.6. For the whole range of connections benchmarked, Atlas remains more effective than Netflix in terms of memory traffic efficiency, requiring $1.5\times$ the network throughput as opposed to $2.6\times$ for Netflix.

The Atlas memory read results indicate that it was not possible to retain all the data in the LLC for the full duration of the TX pipeline, from disk to encryption to NIC for most workloads, though it is often possible for 2,000 concurrent connections when the memory access pattern in Fig. 14a dominates. We believe that the increased memory write throughput observed for Atlas in Figure 13d is related to dirty cache line evictions of encrypted data, which occur after the NIC has finished DMAing the encrypted data out; this does not affect performance. Under heavy load a fraction of the data was evicted to main memory and has to be re-read, either by the CPU while encrypting, or by the NIC during DMA for transmission, or both. The pattern in Fig. 14b is primarily due to a small amount of extra latency introduced by netmap batching, combined with heavy pressure on the LLC. The extra eviction in Fig. 14c prior to encryption is responsible for the LLC misses in Fig. 13f, and occurs because to avoid DMA thrashing the LLC, only a fraction of the LLC is available for DDIO. Once this is exhausted, new DMAs will evict older DMA buffers if the stack is even slightly slow getting round to encrypting them.

5 NEW DESIGN PRINCIPLES

We developed diskmap and the clean-slate Atlas stack to explore the boundaries of achievable performance through a blend of software specialization and microarchitectural awareness. The resulting prototype exhibits significant performance improvements over conventional designs. However, and perhaps counterintuitively, we believe that many of the resulting design principles are reusable, and could be applied within current network- and storage-stack designs.

Reduced latency and increased bandwidth for storage, arising out of new non-volatile storage technologies, fundamentally change the dynamic in storage-stack design. Previously, substantial investment in CPU to improve disk layout decisions and mask spindle latency was justified, and the use of DRAM to prefetch and cache on-disk contents offered significant improvements in both

latency and bandwidth [25]. Now, the argument for an in-DRAM buffer cache is dramatically reduced, as on-demand retrieval of data (e.g., on receiving a TCP ACK opening the congestion window), regardless of existing presence in DRAM, is not only feasible, but may also be more efficient than buffered designs.

Optimizing Last-Level Cache (LLC) use by DMA must be a key design goal to avoid being bottlenecked on memory bandwidth. A key insight here is that if the **aggregate bandwidth-delay product** across the I/O subsystem (e.g., from storage DMA receive through to NIC DMA send) can fit within the LLC, DRAM accesses can be largely eliminated. This requires careful bounding of latency across the I/O and compute path, proportionally decreasing the product, which discourages designs that defer processing – e.g., those that might place inbound DMA from disk, encryption, and outbound DMA to the NIC in different threads – an increasing design choice made to better utilize multiple hardware threads. In the Netflix stack, substantial effort is gone to mitigate cache misses, including use of prefetch instructions and non-temporal loads and stores around AES operations. Ironically, these mitigations may have the effect of further increasing the degree to which higher latency causes the bandwidth-delay product to exceed LLC size. This optimization goal also places pressure on copying designs: copies from a buffer cache to encrypted per-packet memory doubles the cache footprint, halving the bandwidth-delay product that can be processed on a package.

Integrating control loops to minimize latency therefore also becomes a key concern, as latency reduction requires a “process-to-completion” across control loops in I/O and encryption. Allowing unbounded latency due to handoffs between threads, or even in using larger queues between protocol-stack layers, is unacceptable, as it will increase effective latency, in turn increasing the bandwidth-delay product, limiting the work that can fit into the LLC.

Userspace I/O frameworks also suffer from latency problems, as they have typically been designed to maximize batching and asynchrony in order to mitigate system-call expense. Unlike netmap, diskmap facilitates latency minimization by allowing user code to have fine-grained notification of memory being returned for reuse, and by minimizing in-kernel work loops that otherwise increase LLC utilization. This is critical to ensuring that “free” memory in the LLC is reused, rather than unnecessarily spilling its unused contents from the LLC to DRAM by allocating further memory.

Zero-copy is not just about reducing software memory copies. While zero-copy operation has long been a goal in network-stack designs, attention has primarily been paid to data movement performed directly through the architecture – e.g., by avoiding unnecessary memory copies as data is passed between user and kernel space, or between kernel subsystems. It is clear from our research that, to achieve peak performance, system programmers must also eliminate or mitigate implied data movement in the hardware – with a special focus on memory-subsystem and I/O behavior where data copying in the microarchitecture or by DMA engines comes at extremely high cost that must be carefully managed. This is made especially challenging by the relative opacity of critical performance behaviors: as data copying and cache interactions move further from the processor pipeline, tools such as hardware performance counters become decreasingly able to attribute costs to the originating parties. For example, no hardware that we had access to was able to attribute cache-line allocation to specific DMA sources, which would

have allowed a more thorough analysis of NIC vs. NVMe cache interactions.

Larger than DRAM-size workloads are important for two reasons: a long tail of content used by large audiences (e.g., with respect to video and software updates), and also because DRAM is an uneconomical form of storage due to high cost and energy use as compared to flash memory. The Atlas design successfully deemphasizes DRAM use in favor of on-package cache and fast flash, avoiding loading content into volatile memory for longer than necessary.

Netflix has already begun to explore applying some of these design principles to their FreeBSD-based network stack. A key concern to reduce memory bandwidth utilization has been to improve the efficiency of cache use, which has to date been accomplished through careful use of prefetching and non-temporal operations. These in fact prove harmful compared to a more optimal design such as Atlas due to increasing the effective bandwidth-delay product. Reducing cache inefficiency by eliminating the buffer cache is challenging in the current software environment, especially when some key content sees high levels of reuse. However, reducing latency between storage DMA and encryption is plausible, by shifting data encryption close to storage I/O completions, avoiding redundant detours to DRAM.

6 RELATED WORK

We briefly discuss previous work related to Atlas.

Conventional Stack Optimizations: System call overheads and redundant data copies have been previously identified as a bottleneck of conventional OSes. Multiple past studies have focussed on optimizing OS primitives to achieve better system performance. IO-Lite [24] unifies data management between userspace applications and kernel subsystems, utilizing page-based mechanisms to safely share data. FlexSC [31] provides system call batching by allowing userspace and kernel to share the system call pages, avoiding that way CPU cache pollution. Megapipe [13] demonstrates significant performance improvements by employing system call batching, and introducing a bidirectional per-core pipe for data and event exchange between kernel and userspace. Past research has shown that maintaining flow affinity, and minimizing sharing is key to achieving network IO scalability on multicore systems [7, 8, 26].

Userspace I/O frameworks: Netmap [29] implements high-throughput network I/O, by exposing DMA memory to userspace and relying on batching to substantially reduce context switch overhead. Similarly, Intel’s DPDK [11] utilizes kernel-bypass to provide user-level access to the network interface using hardware virtualization. Intel’s Storage Performance Development Kit (SPDK) [32] is a contemporary effort to diskmap, that implements a high throughput and low latency NVMe storage I/O framework, by running the NVMe device drivers fully in userspace. Unlike SPDK, diskmap does not fully expose NVMe devices to userspace (e.g., device doorbells, administrative queue pairs): the OS kernel is still mediating for device administrative operations, and DMA operations are abstracted with system calls for protection.

Microkernels and User-level Stacks: Microkernel designs such as Mach [1], shift core services to userspace. The Exokernel [12] and SPIN [6] reduce shared subsystems to enable userspace-accessible low-level interfaces for hardware access. Recently, inspired by microkernel services, multiple user-level network stacks leveraged

OS-bypass to demonstrate dramatic throughput and latency improvements over the conventional kernel stacks [15, 16, 18].

Dataplane and Research OSeS: Arrakis [27] is a research operating system that leverages hardware virtualization to efficiently decouple the control and data planes. Diskmap was partially inspired by Arrakis, which first applied the idea of fast and safe user-level storage data plane. Similarly, IX [5] uses hardware virtualization to enforce safety, while utilizing zerocopy APIs and adaptive batching to achieve high performance network IO.

7 CONCLUSIONS

In this paper we presented Atlas, a high-performance video streaming stack which leverages OS-bypass and outperforms conventional and state-of-the-art implementations. Through measurement of the Netflix stack, we show how traditional server designs that feature a buffer cache to hide I/O latency suffer when serving typical video streaming workloads. Based on these insights, we show how to build a stack that directly includes storage in the network fast path.

Finally, we discuss the highly asynchronous nature of the conventional stack's components, and how it contributes to lengthening I/O datapaths, while wasting opportunities for exploiting microarchitectural properties. We show how, using a specialized design, it is possible to achieve tighter control over the complete I/O pipeline from the disk up to network hardware, achieving high throughput and making more efficient use of memory and CPU cycles on contemporary microarchitectures.

8 ACKNOWLEDGEMENTS

We thank Drew Gallatin from Netflix for his comments and invaluable assistance with the Netflix stack and workloads. Additionally, we gratefully acknowledge Navdeep Parhaar from Chelsio for arranging 40GbE NICs for us, and assisting us with TSO support for the netmap driver. Finally, we would also like to thank Jim Harris from Intel's Storage Division, Serafeim Mellos, our anonymous reviewers, and our shepherd Keith Winstein for their insightful comments.

This work was supported by a Google PhD Fellowship, and a NetApp Faculty Fellowship.

REFERENCES

- [1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevianian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Technical report, Computer Science Department, Carnegie Mellon University, August 1986.
- [2] Adobe HTTP Dynamic Streaming. <http://www.adobe.com/content/dam/Adobe/en/devnet/hds/pdfs/adobe-hds-specification.pdf>.
- [3] N. Amit, M. Ben-Yehuda, D. Tsafir, and A. Schuster. vIOMMU: Efficient IOMMU Emulation. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.
- [4] BBC Digital Media Distribution: How we improved throughput by 4x. <http://www.bbc.co.uk/blogs/internet/entries/>.
- [5] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, Dec. 2016.
- [6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.
- [7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.
- [8] 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 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.
- [9] Chelsio 40GbE Netmap Performance. <http://www.chelsio.com/wp-content/uploads/resources/T5-40Gb-FreeBSD-Netmap.pdf>.
- [10] Intel Data Direct IO. <http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html>.
- [11] Intel Data Plane Development Kit. <http://dpdk.org>.
- [12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.
- [13] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 135–148. USENIX Association, 2012.
- [14] HTTP Live Streaming. <https://tools.ietf.org/html/draft-pantos-http-live-streaming-23>.
- [15] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling Network Protocol Innovation with User-level Stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, Apr. 2014.
- [16] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association.
- [17] J. Lemon. Kqueue - A Generic and Scalable Event Notification Facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153, Berkeley, CA, USA, 2001. USENIX Association.
- [18] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.
- [19] A. Markuze, A. Morrison, and D. Tsafir. True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 249–262, New York, NY, USA, 2016. ACM.
- [20] Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats, April 2012. ISO/IEC 23009-1 (<http://standards.iso.org/ittf/PubliclyAvailableStandards>).
- [21] Netflix Appliance Software. <https://openconnect.netflix.com/en/software/>.
- [22] NVMe Express Specification 1.2.1. <http://www.nvmeexpress.org/specifications/>.
- [23] Intel P3608 NVMe drive. <http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-p3608-series.html>.
- [24] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *Operating systems review*, 33:15–28, 1998.
- [25] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 79–95, New York, NY, USA, 1995. ACM.
- [26] 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, pages 337–350, New York, NY, USA, 2012. ACM.
- [27] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.
- [28] RFC5288: AES Galois Counter Mode (GCM) Cipher Suites for TLS.
- [29] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.
- [30] Sandvine 2015 Global Internet Phenomena Report. <https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/global-internet-phenomena-report-latin-america-and-north-america.pdf>.
- [31] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.
- [32] Intel Storage Performance Development Kit. <http://www.spdk.io>.
- [33] R. Stewart, J.-M. Gurney, and S. Long. Optimizing TLS for high-bandwidth applications in FreeBSD. In *Proc. Asia BSD conference*, 2015.