Network and storage stack specialisation for performance

Ilias Marinos

University of Cambridge
Computer Laboratory
Wolfson College

December 2018

This dissertation is submitted for
the degree of Doctor of Philosophy

To my parents, Roula and Andreas.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation is not substantially the same as any that I have submitted or that is being concurrently submitted for a degree, diploma, or other qualification at the University of Cambridge, or any other University or similar institution.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.

Network and storage stack specialisation for performance

Ilias Marinos

Summary

In order to serve hundreds of millions of users, contemporary content providers employ tens of thousands of servers to scale their systems. The system software in these environments, however, is struggling to keep up with the increase in demand: contemporary network and storage stacks, as well as related APIs (e.g., BSD socket API) follow a 'one-size-fits-all' design, heavily emphasising generality and feature richness at the cost of performance, leaving crucial hardware resources unexploited. Despite considerable prior research in improving I/O performance for conventional stacks, substantial hardware potential still remains unexploited because most of these proposals are fundamentally limited in their scope and effectiveness, as they still have to fit in a general-purpose design.

In this dissertation, I argue that *specialisation* and *microarchitectural awareness* are necessary in system software design to effectively exploit hardware capabilities, and scale I/O performance. In particular, I argue that trading off generality and compatibility, allows us to radically re-architect the stack emphasising application-specific optimisations and efficient data movement throughout the hardware to improve performance.

I first demonstrate that conventional general-purpose stacks fail to effectively utilise contemporary hardware while serving critical Internet workloads, and show why modern microarchitectural properties play a critical role in scaling I/O performance. I then identify core decisions in Operating Systems design that, although they were originally introduced to optimise performance, are now proven redundant or even detrimental. I propose clean-slate, specialised architectures for network and storage stacks designed to exploit modern hardware properties, and application domain-specific knowledge in order to sidestep historical bottlenecks in systems I/O performance, and achieve great scalability. With thorough evaluation of my systems, I illustrate how specialisation and greater microarchitectural awareness could lead to dramatic performance improvements, which could ultimately translate to improved scalability and reduced capital expenditure simultaneously.

Acknowledgements

I suppose this whole journey actually begun many years ago, when Robert N.M. Watson trusted me to do a Google Summer of Code project with him, and later when he invited me to Cambridge for an internship. His high standards, passion for research, and seemingly infinite knowledge around computer systems is what truly motivated me to pursue an MSc and a PhD. Robert has generously given me freedom and unwavering support to do research on things I liked, no matter how risky they seemed; I am truly grateful for this. I am also deeply indebted to Robert, for his patience, guidance, and help at critical junctures throughout my PhD. Later, I met Mark Handley a man who effectively served as a co-supervisor during my PhD and has greatly impacted my journey into systems research. Mark taught me how to think, how to do quality research, and has been my inspiration to this day. None of this work would have been possible without him; my deepest gratitude extends to Mark for his invaluable guidance, endless patience during our collaboration, and more importantly for never losing his trust in me. All in all, I have been immensely fortunate to meet, and having worked with Mark and Robert; after all these years, I consider them as true friends.

My good friend Serafeim Mellos deserves credit and gratitude, for challenging, motivating and supporting me over the years; from the very beginning when we were undergraduate students, to the final proofreading of this document. I also owe special thanks to my friend Dimos Pediaditakis, for his invaluable help and continuous support even when things were looking grim. Of course, I am extremely thankful to my Sofia; your support, patience and encouragement has seen me through tumultuous times.

Dozens of brilliant people have helped me immensely and taught me many things at the Computer Laboratory. More specifically, I thank Kumar Sharad, Laurent Simon, Bjoern Zeeb, Charalampos Rotsos, George Parisis, Sheharbano Khattak, Salvator Galea, Gianni Antichi, and Yury Audzevich. My thanks also go to Costin Raiciu and Jon Crowcroft for generously agreeing to examine my dissertation.

Throughout my PhD, I had the honor and privilege of working with distinguished researchers in industry. In particular, I thank Vytautas Valancius and Daniel Eisenbud for our collaboration and for giving me the opportunity to work at the USPS team at Google.

This work was supported by a Google PhD fellowship in systems and networking, and a NetApp Advanced Technology Group research grant.

Last, but foremost I am forever grateful to my parents, who have devoted their life to raise me — with dedication, unselfishness and loving support. I dedicate this dissertation to them. And I need to thank them in Greek now...

Το μεγαλύτερο «ευχαριστώ» όλων φυσικά το οφείλω στους γονείς μου, που αφιέρωσαν τη ζωή τους για να με μεγαλώσουν· με αγάπη, αφοσίωση και αυτοθυσία. Θα σας είμαι ευγνώμων για πάντα! Η διατριβή αυτή ειναι αφιερωμένη σ΄ εσάς!

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Conventional network and storage 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; instead of a single system performing multiple functions, today, large content providers are scaling out by employing thousands of servers, each providing a single network service. Yet most content is still managed and served with general-purpose stacks.

Historically, an operating system (OS) is responsible for resource sharing, management and isolation, as well as providing a set of common services to facilitate application development. Contemporary widely-deployed operating systems (typically variants of Linux, BSD or Windows) are extremely flexible, and support a wide range of middle- and edge-node functions, as well as hardware: from interactive desktop environments and low-power smartphones to high-loaded, performance-critical middleboxes, file servers, and web servers. The ability to support many different applications using a stable general-purpose interface is arguably very useful, as it dramatically simplifies software development, promotes portability on different hardware, and reduces maintenance costs. This design decision, however, does not come without compromises: the application requirements vary substantially, and designing a general-purpose operating system interface to accommodate every possible need comes at a high performance cost, wasting substantial hardware potential.

Improving Input/Output (I/O) performance has been a long-standing goal of systems research. Over the years, it led to numerous optimisations of commodity network and storage stacks, and yet it has been a losing battle: the problem with these enhancements is that each serves a narrow role, yet still must fit within a general OS architecture, and thus are constrained in their scope and effectiveness. Hence, it would be very hard, if not impossible, to explore disruptive techniques in systems design to improve I/O performance while retaining the current OS structure. For example, the conventional operating systems rely on the kernel-user level split between application and network/storage stack to achieve protection and sharing — could we eliminate the kernel-user split to amortise the system overhead by reducing the domain crossings? Microarchitectural design has been advancing rapidly, bringing drastic changes to I/O data flow

throughout the hardware, and yet contemporary OSes have not been altered structurally to take advantage of these new features. For example, Intel's Data Direct I/O (DDIO) allows Direct Memory Access (DMA) to occur via the processor's last-level cache (LLC) when the data are available there, avoiding that way extra roundtrips to the main memory. Conventional operating systems, however, leverage a loosely coupled, highly asynchronous model of operation among different components (applications, device drivers, network, and storage stacks) to optimise for CPU multiplexing, making it hard to achieve tight control of the working set and exploit such microarchitectural features — could we introduce a more synchronous design to exploit contemporary microarchitectures and optimise cache locality and memory efficiency? Conventional operating systems suffer from redundant memory copies of data among kernel and user space — could we realise a design that achieves full zero-copy memory flow from the hardware buffers to the application layer?

As the hardware capabilities evolve rapidly and we transition to microsecond-scale I/O, the conventional stacks have become a serious bottleneck for performance-critical services, with host-side processing accounting for the major share of network and storage performance overhead. Indeed, it would be reasonable to expect that trading off generality would yield some performance improvement. What is far from clear though is the extent of the benefit, and whether that would be sufficient to justify the costs associated with specialised approaches in system design (flexibility, maintenance, financial costs). For example, it would likely be unwise to invest in specialisation if there were hidden fundamental hardware bottlenecks that are preventing I/O scalability, or if plainly the attainable performance gains coming from a novel design are insignificant. In this dissertation, we argue that it is imperative to trade-off generality, in favour of specialisation to significantly push network and storage I/O performance. In particular, we discuss the pitfalls and limitations of general-purpose network and storage stacks and we present the design and implementation of specialised stacks, which demonstrate dramatic performance improvements over the conventional stacks while still retaining many of the benefits of general-purpose operating systems. To achieve this we employ design principles, such as transparent zero-copy memory flow from the hardware to the application layer, aggressive cross-layer optimisations where possible, optimisations based on microarchitectural awareness, and techniques to amortise system-related overheads.

Part of this work has been the result of collaboration with Netflix, a major video streaming provider. Netflix's Content Delivery Networks (CDN) are built from highly specialised nodes with a strong emphasis on performance across the I/O stacks (network and storage), as they need to serve many thousands of clients simultaneously. Netflix was inspired by the outcome of this work, and has already begun to explore applying some of the design principles presented in this dissertation to their FreeBSD-based video streaming stack.

1.0.1 Contributions

In this dissertation, I describe the following principal contributions:

1. I investigate I/O performance limitations of contemporary off-the-shelf hardware. I show why modern microarchitectural properties play a critical role in scaling I/O performance, and I argue why contemporary systems software architecture should be optimised for efficient hardware use.

2. I question long-held tenets in operating systems design, and I show how these previously beneficial optimisations could prove redundant or even harmful under the presence of certain critical workloads. For example, I show how the traditional OS buffer cache could actually negatively impact performance while serving a typical video streaming workload, or how software zero-copy could hinder DMA performance for certain workloads on modern CPUs.

3. I present the design and architecture of clean-slate, specialised network and storage stacks, that avoid conventional OS pitfalls to greatly scale I/O performance. I show how such architectures manage to minimise data movement throughout modern microarchitectures, to avoid historical bottlenecks (e.g., memory bandwidth saturation) in systems performance that stem from software design, and are often incorrectly attributed to hardware. Finally, I conduct a thorough evaluation of the specialised stacks, as well as contemporary state-of-the-art systems.

4. I quantify the performance benefits of specialisation with critical Internet workloads, and demonstrate this way the existence of a substantial potential in reducing capital expenditure while scaling out performance-critical applications.

1.0.2 Collaborations

The designs, architectures and algorithms presented in this dissertation are the result of collaboration with my supervisors Robert N.M. Watson (University of Cambridge), and Mark Handley (University College London).

Randall R. Stewart helped us understand better the challenges introduced by Netflix workloads, and gave us access to the Netflix video streaming stack implementation.

1.1 Dissertation outline

This thesis is structured as follows:

Chapter 2 gives an overview of the state-of-the-art systems, frameworks and techniques in high-performance I/O, and highlights the limitations in scaling I/O performance with conventional OSes and hardware. We also trace the evolution of hardware microarchitecture, and motivate our work by identifying and demonstrating shortcomings in contemporary software systems.

Chapter 3 describes the design and implementation of the Sandstorm and Namestorm clean-slate userspace network stacks, that utilise domain-specific knowledge and system techniques to reduce system costs, and saturate contemporary networking hardware (NIC) for applications such as web and DNS servers. We show why this architecture is more suitable to handle critical workloads (e.g., web traffic), and we demonstrate the power of this design by comparing against contemporary conventional stacks and applications.

Chapter 4 describes Atlas, a specialised library OS system that leverages kernel-bypass to accelerate storage and network I/O, and ultimately improve system performance while serving video streaming workloads. Furthermore, we discuss fundamental shortcomings stemming from conventional stacks' architecture, and we show novel techniques that allow us to take advantage of modern hardware properties, to eliminate historical bottlenecks and scale I/O performance. We evaluate our solution against a production state-of-the-art stack that currently powers Netflix's CDN fleet.

Chapter 5 highlights directions for future work and concludes this dissertation.

1.2 Related publications

Most of the work described in this dissertation is part of peer-reviewed publications:

[MWH13] Ilias Marinos, Robert N. M. Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. HotNets-XII. College Park, Maryland: ACM, 2013, 9:1–9:7.

[MWH14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 175–186.

[MWH+17] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. "Disk, Crypt, Net: Rethinking the Stack for High-performance Video Streaming". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 211–224.

I have also co-authored the following publications, which are not being presented in the context of this dissertation:

[AWC+14] Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. "TESLA: Temporally Enhanced System Logic Assertions". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 19:1–19:14.

[GWA+15] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, et al. "Clean Application Compartmentalization with SOAAP". in: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 1016–1031.

Chapter 2

Background

Conventional general-purpose operating systems are ubiquitously deployed and form the core of today's networked and storage systems. Recently though, the rapid increase in user demand revealed that, despite many years of evolution, commodity stacks fail to efficiently utilise contemporary hardware, forcing providers to wastefully rely on system scale-out to cope with the demand.

In this chapter, we discuss the mismatch in recent software and hardware capabilities, which is the root cause for reduced system I/O performance while serving certain workloads. We also consider significant advancements in microarchitecture design that present critical software optimisation opportunities, which have not been previously explored with conventional operating systems. Finally, we describe the properties, limitations and previously proposed enhancements of current conventional I/O stacks, as well as alternative approaches in systems design.

2.1 Mind the gap: the hardware - software evolution

Nowadays, the rapid evolution of hardware capabilities should allow us, in principle at least, to achieve low latency and high throughput with performance-critical applications. Unlike older processors, modern CPUs feature numerous cores (e.g., 24-cores is not uncommon for latest generations of Intel CPUs) and typical datacenter servers could have more than one socket per server (NUMA). Over the last 5-10 years, physically integrating components such as the memory controller hub (MCH or Northbridge), and the I/O controller (including the PCIe root complex) in the CPU package has been made possible, and modern x86 microarchitecture has evolved to jettison external shared buses (e.g., the Front-Side Bus) allowing for dramatic improvements in memory and PCIe latency and bandwidth. The CPU caches have gotten significantly larger, and with technologies such as DDIO [DDIO] they currently play a critical role serving as the hub between CPU cores, DRAM, and PCIe peripherals. Latest Intel Xeon CPUs, for example, feature three levels of cache with up to 60MB of on-die L3 cache, while 10-15 years ago state-of-the-art CPUs would typically have only two levels of on-die cache (with

~1MB L2 cache). To achieve connectivity with the main memory (DRAM), CPUs now feature multiple, high-speed channels allowing for very fast memory transfers in the order of hundreds of Gb/s. With regards to peripheral connectivity, the third generation of PCIe serial interconnect allows for a usable bandwidth of up to ~7.87Gb/s per lane with 40 available lanes per CPU package on latest configurations (summing up to an impressive total of ~315Gb/s). Compared to the first PCIe revisions (2003-2005), this is a $4\times$ increase in throughput per PCIe lane, with plenty more lanes available in modern CPUs. For storage, flash memory is becoming faster (with off-the-shelf devices capable of 28Gb/s read throughput for short operations, and latency as low as 20μs), more reliable, and affordable while jettisoning conventional storage buses/interfaces (e.g., AHCI/SATA) and attaching storage directly to the PCIe interconnect. For networking, 10 and 40GbE Network Interface Controllers (NICs) could be safely considered commodity for datacenter servers, while high-end CDN providers are rapidly adopting 100GbE NICs in production. To set the context and better understand the rate of evolution, over the last 10 years, we have witnessed a ~10x or more increase in NIC and storage bandwidth, while at the same time such hardware became more accessible and affordable.

What is the *status quo* in operating system (OS) design? Conventional network and storage stacks were originally designed in an era where individual systems had to perform multiple diverse functions. As a result, all the critical subsystems including the network and storage APIs and schedulers have been optimised to allow multiple users and applications to share the scarce processing resources, trading-off throughput and latency for efficient CPU multiplexing. In the subsequent years, however, CPU cycles were made more readily available thanks to rapid advances in semiconductor manufacturing.

On the networking side, the extra processing capacity has been invested in general-purpose features. Most conventional network stacks have been extended to feature a selection of built-in stateful firewalls, multiple flexible routing tables and mechanisms, traffic classification and queueing disciplines for quality of service, several pluggable TCP congestion control mechanisms, debugging features, load balancing features etc: the same stacks were now flexible enough to serve the needs of user desktops, back-end servers (web servers, databases etc), embedded devices, and middleboxes (load balancers, shapers etc).

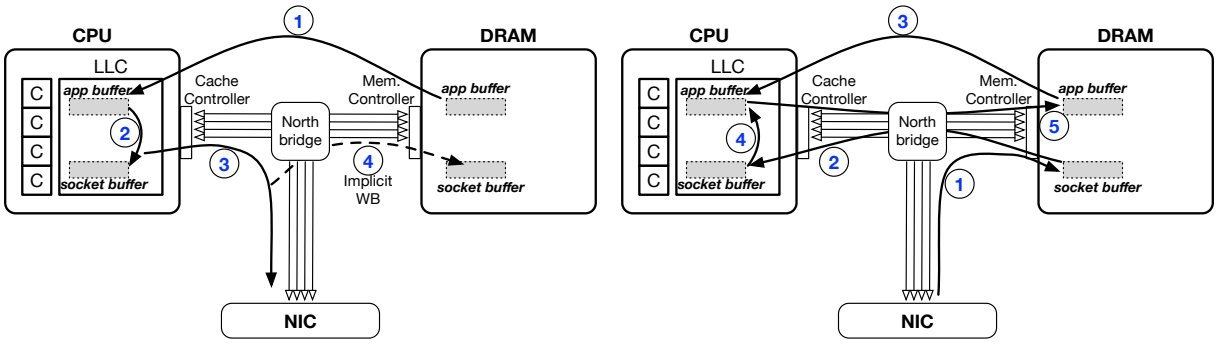
On the storage side, the systems community has extensively tried to trade off CPU cycles for improvements in I/O latency and throughput, aiming to work around the major performance bottlenecks of spinning disks. Complex filesystem data structures and sophisticated techniques have been employed in an attempt to mask hardware inefficiencies: larger filesystem block sizes are being used to increase I/O throughput at the expense of storage capacity, file blocks are laid sequentially to avoid the poor performance of random disk accesses, and asynchronous writes of metadata are being enforced (soft updates) to reduce the amount of disk writes. To further reduce the high storage I/O latency and improve responsiveness for interactive applications and networking, more CPU and RAM resources have been devoted: the *buffer cache* transparently caches pages originating from persistent storage in DRAM, to accelerate performance of subsequent requests by completely eliminating new I/O operations for the same data.

Over the years, these stacks have evolved to large and complex codebases, that in an attempt to be general-purpose have picked extra significant costs associated with more flexible designs.

Nowadays, however, due to the dramatic increase in user and application demands we actively seek to push the limits in I/O performance in order to fully exploit the hardware capabilities, and reduce capital expenditure for large cloud and content providers. Indeed the networking and storage hardware is steadily getting faster, straining the server memory and computation resources that are necessary to process I/O traffic: a 100GbE adapter, for example, is capable of delivering a 64byte packet to the CPU last-level cache every ~5.1ns, while the same LLC access time from a CPU core is roughly 10-15ns. Furthermore, extra CPU cycles are needed to accommodate the needs of sophisticated applications or protocols (e.g., with end-to-end encryption the CPU needs to access and encrypt all the data to be transmitted). To keep up with the increasing demand, it is imperative to achieve more efficient use of hardware resources. Modern storage hardware is now affordable and fast enough, making several of the aforementioned storage stack optimisations redundant or even harmful. Similarly, contemporary feature-rich network stacks struggle to keep up with NIC hardware capabilities, consuming precious processing power that could otherwise made available for application processing. We argue that by reducing the software complexity in terms of abstraction layers, general-purpose features/structures, and by making the path between application and I/O hardware shorter, we can eliminate traditional major bottlenecks in systems performance and come a step closer to utilising the full modern hardware potential.

2.2 Microarchitectural evolution

Memory bandwidth has historically been one of the most significant bottlenecks in scaling I/O performance [ABC+08; BBB11; DEA+09; WM95]. Very recently, Netflix has reported that their video-streaming CDN nodes have hit *the memory wall* while using contemporary hardware (see §4.2.1). Efficient data movement throughout the hardware is key to achieving performance: when a system experiences high memory latency, or even saturation of the memory channels, the overall system performance is erratic. CPU utilisation is a software-facing metric that traditionally shows the proportion of wall clock time spent actually running instructions vs. waiting for I/O or entering the idle (low-power) state. The problem with such metrics though is that different CPU execution units serve different functions (e.g., arithmetic logic unit (ALU), load-store unit (LSU), cryptographic accelerator etc), and this is not comprehensively captured in current software-facing metrics. For example, in cases where the memory bandwidth is saturated, the CPU frequently stalls waiting for data to be moved from and to the main memory: in such cases CPU utilisation is no longer an accurate metric of load or efficiency, since the same instructions are suddenly significantly more expensive (in terms of CPU cycles). Similarly, DMA operations are also affected and experience the same contention for memory bandwidth, resulting in vastly reduced performance.



(a) TX side. **(b)** RX side.

Figure 2.1: Memory traffic patterns while sending and receiving packets on pre-2011 x86 microarchitectures. Roughly 2 bytes of read/write memory traffic for each byte transmitted, and 4 bytes of memory traffic for each byte received.

Software architecture and algorithms can directly affect memory traffic efficiency by changing the way a workload is represented, achieving that way more or less efficient use of the microarchitecture.

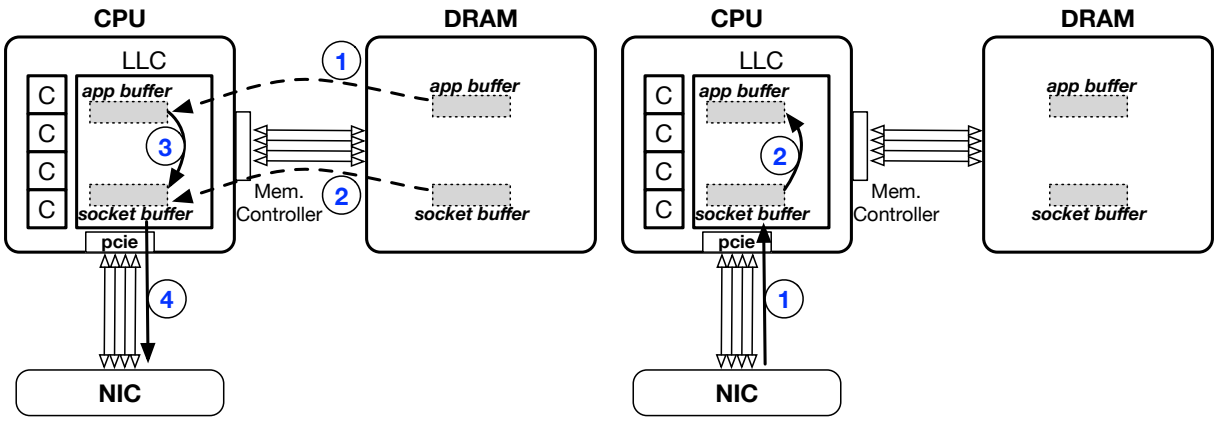
In this section, we explore how the evolution of microarchitectural design affects the memory traffic patterns due to I/O, and ultimately I/O performance.

2.2.1 Memory traffic: DMA traffic and CPU cache interaction

Direct memory access (DMA) is a feature of computer systems, that allows peripherals or other hardware subsystems to access the main memory without the mediation of the central processing unit (CPU). Without DMA, any input/output operation would typically fully occupy the CPU to copy data for the duration of the I/O operation. Instead, the CPU only needs to initiate the transfer and dedicated DMA engines are responsible for moving the relevant data from/to memory. That way I/O transfers should, in principle at least, introduce only a minimal CPU overhead.

In this subsection, we use network I/O as an example for the memory traffic analysis, but the same principles are applicable for all kinds of DMA-based I/O over the PCIe interconnect (e.g., PCIe-attached disks, crypto accelerators).

Figure 2.1 illustrates the memory traffic patterns on pre-2011 Intel x86 microarchitectures, while an application is sending and receiving network packets using the conventional BSD socket API. An application is maintaining an in-memory buffer to store the data that are ready to be transmitted; the data access operations on the application buffer trigger CPU cache misses, fetching data from the main memory every time it is needed; write back to DRAM is not required, since the buffer is not dirtied. The socket buffer needs to also be fetched from the main memory, but this only happens for the first time: typically the socket buffer should be sufficiently small to fit in the LLC, and given the subsequent operations, it should be in either of the `EXCLUSIVE`,`MODIFIED`,`SHARED` states. When the CPU attempts a write to the socket



(a) TX side. **(b)** RX side.

Figure 2.2: Memory traffic patterns while sending and receiving packets on post-2011 x86 microarchitectures. Received and transmitted network packets could ideally be served completely from the Last Level Cache without extra memory transactions.

buffer the cache state changes between one of the aforementioned states (e.g., `EXCLUSIVE` to `MODIFIED`), without ever getting to `INVALID`. It is worth noting that these state transitions ideally do not imply any extra memory traffic: with write-back policy only caches are updated on write, not memory. Once the NIC requests to DMA out part of the socket buffer there are two cases:

1. If the socket buffer is still present in the LLC (dirtied), most snooping hardware would either invalidate and evict the appropriate cache lines so that the DMA is served from DRAM, or 'redirect' the DMA transfer to be served from the LLC while also triggering an implicit write-back at the same time. Worse, the DMA read request could cause several earlier generations to issue a speculative read while the Cache Coherency protocol tries to locate the data in the cache, resulting in multiple roundtrips to the main memory.

2. Otherwise, the DMA will be served from DRAM.

With contemporary microarchitectures (see Figure 2.2) the situation is radically different: the *Northbridge* is eliminated, and its functions (e.g., snoop agent, memory controller) are now co-located into the CPU package[1], without the need of a Front-Side Bus which has traditionally been a performance bottleneck. Similarly, the PCI express interface is also integrated onto the CPU package, instead of an I/O hub or *Southbridge*. With these optimisations, the new x86 generations have the potential to increase the effective I/O bandwidth and reduce latency, by eliminating a significant part of the memory activity. Figure 2.2a illustrates a high-level view of the potential memory patterns on transmit, with the newer generations of hardware: the application and socket buffers might either need to be loaded once from the main memory, or ideally the data accesses on these buffers could be fully satisfied from the larger caches without

[1]Commonly known as *uncore* functions or *system agent*.

any misses, and then can remain in the LLC for the whole lifespan of the application, depending of course on the software re-use frequency, as well as the overall CPU cache pressure. The DMA read operation from the NIC is directly served from the LLC, without causing any data evictions [DDIO].

There are also several differences on the network receive side, between older and newer generations of microarchitectures. Figure 2.1b shows the potential memory traffic patterns on older generations. DMA write operations have the NIC transferring data to the host main memory. If these data (packet payloads or control structures) happen to be in the CPU cache hierarchies, they are invalidated. This means that subsequent data access operations from the network stack on these memory addresses will trigger CPU cache misses, and the data will be read back from the main memory, in order to eventually be copied to the application buffers. On newer microarchitectures with the Data Direct I/O technology the memory flow is different (Fig. 2.2b): a DMA write operation from a PCIe peripheral can be handled with two modes of operation: *(i)* a `WRITE UPDATE` which results to an in-place update of the appropriate LLC cache lines, without further detours to the main memory, or *(ii)* a `WRITE ALLOCATE` which allocates lines in the LLC[2] to store the DMA data (potentially without any extra memory traffic if the data delivered do not exist in the cache hierarchies). With this approach, the data are immediately available for processing by the CPU cores, without extra cache misses. Eliminating the redundant memory traffic, alleviates pressure on the memory channels: this way, memory bandwidth is not saturated, and memory latency can remain low avoiding CPU pipeline stalls.

2.2.2 Memory - I/O interactions, and performance

Optimising software for memory traffic efficiency is hardly a new research area. For conventional OSes most efforts focussed, to a larger extent, in achieving transparent memory flow by reducing the absolute number of memory copies in software where possible (e.g., zero-copy operation between storage and network kernel subsystems). Zero copy optimisations achieve more efficient data movement through the architecture, by eliminating multipliers on cache footprint for the data being processed. This approach alone, however, is not sufficient by itself to guarantee more efficient data movement throughout the hardware: contemporary system software optimisations mostly focus on processor-driven interactions with the microarchitecture, but not DMA interactions with the LLC. Data copying (both from the CPU and DMA operations) and cache interactions are complex, and come at substantial cost. How do the aforementioned modern microarchitectural properties affect memory traffic, and ultimately I/O performance in practice? We attempt to explore these implications through a set of microbenchmarks.

Our testing software utilises kernel-bypass, to fetch raw data blocks from a fast PCIe NVMe disk, and immediately transmit them to the network, using a 40GbE NIC. This requires initiating

[2] `WRITE ALLOCATE` is restricted, by policy, to use only a fraction of the cache (typically 10%) to avoid thrashing.

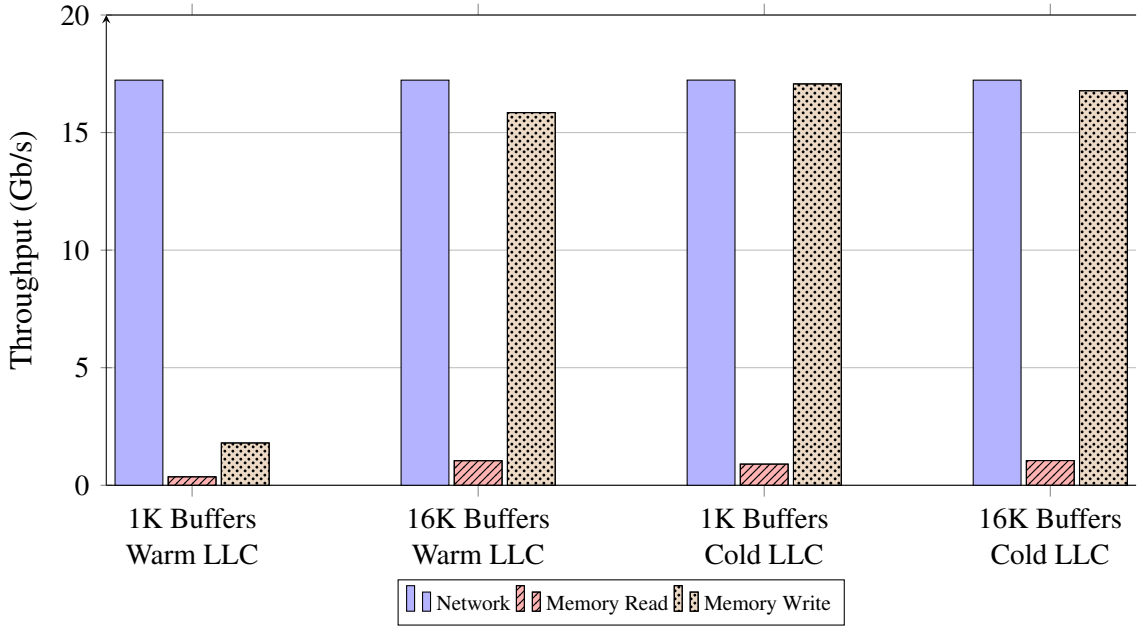


Figure 2.3: Microbenchmark: fetching 8KB data blocks from the NVMe disk, and *zero-copy* transmitting them to the network. Exploring the potential impact of various working-set sizes, and cache state to I/O and memory activity.

DMA from the PCIe-attached storage to load the data into the main memory, and then initiating a second DMA from the main memory to the PCIe-attached NIC to transmit the packet data. We utilise hardware offload features to perform network packet segmentation and checksumming operations, so there are no significant processing requirements for this task. We instead want to explore the interactions of competing raw I/O traffic in terms of CPU cache utilisation/efficacy, and memory traffic activity. Figure 2.3 illustrates the network, memory read, and memory write throughput while sending data from the disk to the network. Memory throughput (read or write) refers to the amount of data transferred between the CPU package and the main memory (DRAM) at the unit of time, and it is measured using hardware uncore performance counters. For this particular microbenchmark, we present four workload variants: the core functionality remains the same throughout all experiments, however we vary the size of memory used to accommodate DMA operations, as well as implicitly control DDIO efficacy by controlling the state of the cache. We achieve a *warm* LLC, by explicitly accessing the DMA buffers[3] immediately before the experiment commencement. The four workloads, in detail, are:

- **1K I/O Buffers, Warm LLC:** For this variant we use enough DMA buffers so that they conveniently fit in the CPU last level cache, but are still sufficient to achieve peak performance with our hardware (NVMe disk and NIC). Furthermore, we ensure that the DMA memory is warm in the CPU cache when the experiment starts. We observe that the system is able to achieve best performance (NVMe disk is saturated at ~17.2Gb/s), while operating almost fully from the LLC without putting any pressure to the memory channels (negligible increase in memory bandwidth).

[3]By DMA buffers we refer to the memory that the DMA operations will load from or store into.

- **16K I/O Buffers, Warm LLC:** The amount of DMA memory used for this experiment is significantly larger compared to the previous workload. Since we do not really need more buffers to saturate the hardware (by maintaining more in-flight I/O operations to fill the disk pipeline), using more buffers practically results to an inefficient way of reclaiming/reusing DMA memory. This is obvious in the graph, as the memory write throughput almost matches the network throughput: DMA memory cannot fit in the LLC, and data are being evicted resulting in higher memory bandwidth. In any case, disk I/O completions are consumed immediately by our tool, and hence it manages to DMA them out to the network before they get flushed out to DRAM (so no significant memory read traffic is required).
- **1K I/O Buffers, Cold LLC:** This configuration is similar to the first workload presented previously with the only difference that the DMA buffers are not warm in the LLC when the experiment commences. Perhaps surprisingly this variation has a huge impact: DDIO is limited by policy to utilise only a fraction of the LLC for allocations (unlike the first workload which could potentially utilise all of the LLC space), and thus newer DMA operations need to evict older ones generating memory write traffic equal to the achieved disk read throughput. Again though, freshly DMAed data from the disk are consumed before they get flushed, so no extra memory read activity is required.
- **16K I/O Buffers, Cold LLC:** Memory activity for this workload is almost identical as with a warm cache. Since the DMA memory size used largely exceeds the LLC size, we cannot avoid evictions, and thus memory write throughput remains constantly equal to the disk read throughput for the duration of the experiment.

All of the above workloads achieve peak performance given the hardware capabilities, but at various costs in terms of memory bandwidth: it is obvious, for example, that larger working sets result to higher memory write traffic. In all the cases above however, memory read activity remains low because our software ensures the immediate consumption of completed disk DMA operations.

What happens if the CPU or the software is not efficient enough to keep up with the rate of completed DMA operations from the hardware? We simulate this behaviour by introducing extra delay between the disk I/O completion and the NIC transmission events. To achieve this we explicitly maintain a larger queue depth for completed I/O operations; the deeper the queue, the higher the delay between disk read completion and network transmission. Figure 2.4 highlights the differences in memory activity: deeper queues result in higher memory activity in both directions. Why is this the case? Although data are being transferred from the disk directly to the LLC, by the time the software initiates the NIC transmission, they have already been flushed to DRAM and need to be read back again. This behaviour closely resembles the situation experienced in modern I/O stacks: *bufferbloat* due to deep driver and intermediate software queues, as well as *asynchrony* stemming from context switching leads to highly inefficient use of the available memory bandwidth.

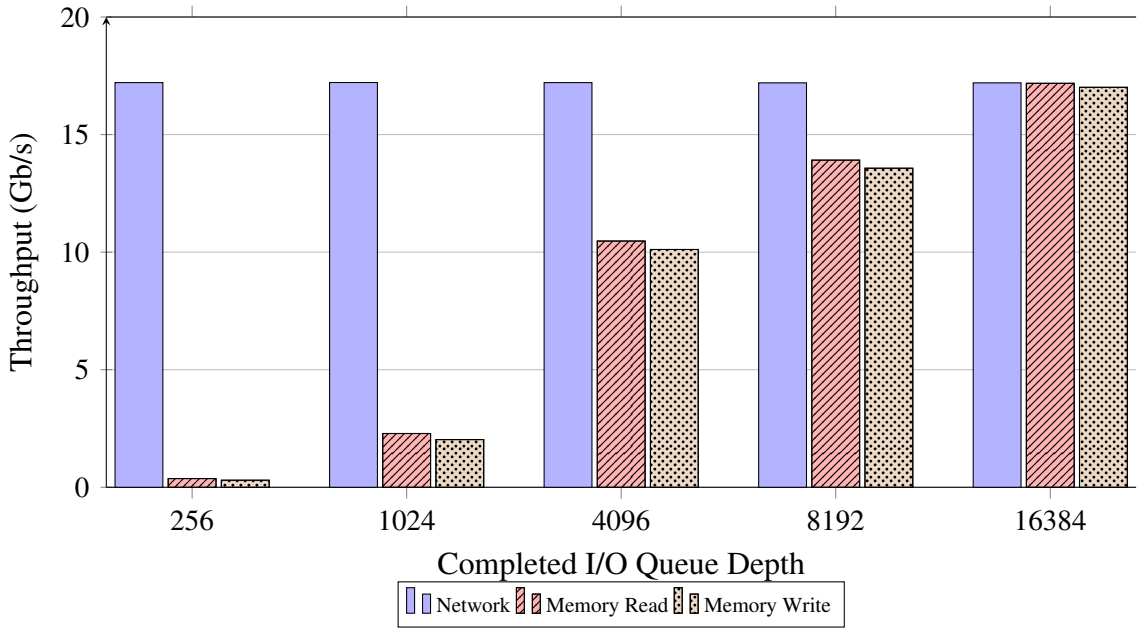
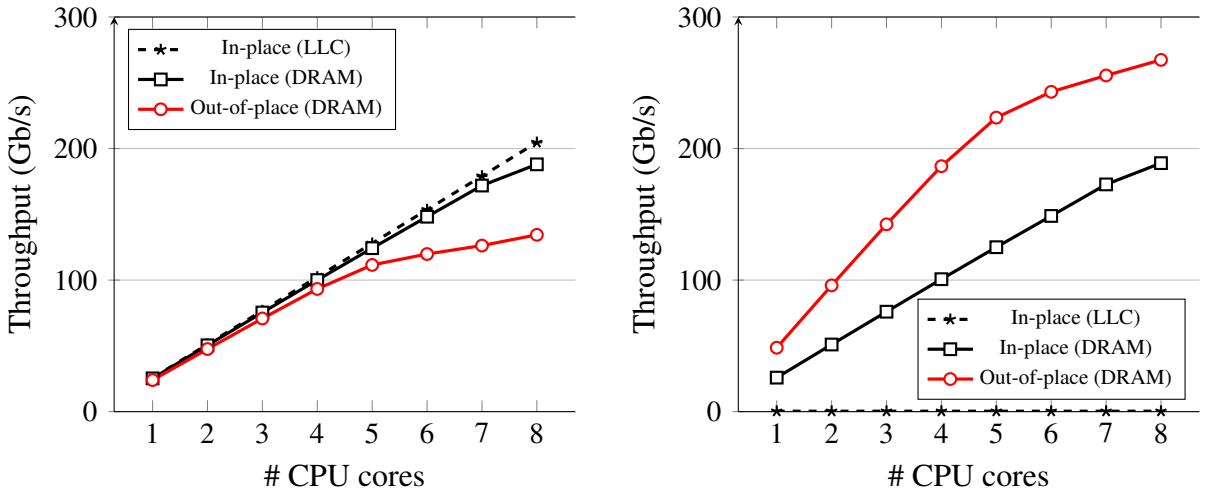


Figure 2.4: Microbenchmark: fetching 8KB data blocks from the NVMe disk, and *zero-copy* transmitting them to the network. We deliberately delay the consumption of completed disk I/O operations by building queues, in order to explore how this affects memory activity.

2.2.3 Memory - I/O - encryption interactions, and performance

In the recent years, privacy and confidentiality concerns drive the industry to rapidly employ ubiquitous end-to-end encryption (TLS). TLS is not only used to protect common web traffic that transfers smaller amounts of data such as web browsing, but also for services such as video streaming that typically uses long flows and accounts for more than 50% of Internet traffic [GIPR+]. As of January 2017, more than 99% of YouTube traffic is encrypted [GGL17]. In February 2017, the movement to encrypt the Internet traffic has reached an impressive milestone: approximately half of the web traffic is now served over HTTPS [Geb17].

What is the cost of end-to-end encryption with contemporary off-the-shelf hardware? In a pursuit of addressing the high-energy requirements, and the relatively poor performance of software-only implementations of AES, modern CPUs feature hardware-acceleration for common cryptographic functions (AES-NI) [ADF+10; Gue10]. This in combination with the Advanced Vector Extensions (AVX) [AVX18] and Carry-less Multiplication (CLMUL) [GK14] extensions of the x86 instruction set allow for highly parallel, hardware accelerated implementations that achieve fast and secure data encryption and decryption, using the Advanced Encryption Standard (AES). Figure 2.5 illustrates the encryption performance of a contemporary CPU, and how it is affected by LLC utilisation and memory activity. We explore two approaches when encrypting data: *in-place* encryption where the original buffer that held the plaintext data is overwritten with the ciphertext, and an *out-of-place* or *copying* encryption approach where we store the encrypted data to a second new buffer so that the plaintext data are being preserved in the original buffer. The dataset represented by the dashed lines highlights the maximum per-



(a) Encryption Throughput (Error bars indicate the standard deviation)

(b) Memory Read Throughput

Figure 2.5: Bulk AES-GCM128 encryption performance using AES-NI on 8-core Haswell CPU. *In-place* encryption uses the same buffer for source and destination, while *out-of-place* encryption uses different buffers. The *LLC* dataset uses small buffers that fit in the LLC (2MB), while the *DRAM* identifier indicates that the benchmark uses large buffers that do not fit in the LLC (6GB).

formance achieved on this CPU: it uses small buffers that fit in the LLC so that detours to the main memory are avoided, while also utilising in-place encryption to efficiently use the LLC space. The peak throughput achieved is a total of ~204Gbps, or roughly 25.5Gbps per core (1 Cycle/Byte). When the buffers to be encrypted cannot fit in the cache, memory bandwidth becomes the major bottleneck in performance. In this case, in-place encryption requires 1 byte of data to be read from main memory and 1 byte written back, for every byte encrypted: still, it achieves near linear scaling on multiple cores with a total encryption throughput of ~188Gb/s, at the cost of cumulative read/write memory throughput of ~378Gb/s. On the contrary, out-of-place encryption requires 2 bytes to be read from the main memory for each byte encrypted (because the source and destination addresses are different), and therefore it saturates the memory channels' capacity much sooner: using all of the available cores, the system achieves a total throughput of ~134Gb/s, at the cost of ~400Gb/s of cumulative memory traffic (~267Gb/s READ).

It is clear from the previous experiment that modern CPUs are able to achieve great bulk encryption performance. Encryption is a CPU-intensive task: how does it interact with I/O data transfers throughout the microarchitecture? Figure 2.6 explores the system behaviour while executing a simultaneously I/O- and CPU-intensive microbenchmark: data are being fetched from disks, get encrypted, and finally transmitted to the network. We want to investigate how data accessing and processing from the CPU affects memory traffic: if the CPU cannot keep up with the I/O traffic, data will accumulate in queues waiting for processing, and redundant memory traffic will spike. The benchmarking software, however, carefully controls the working-set size,

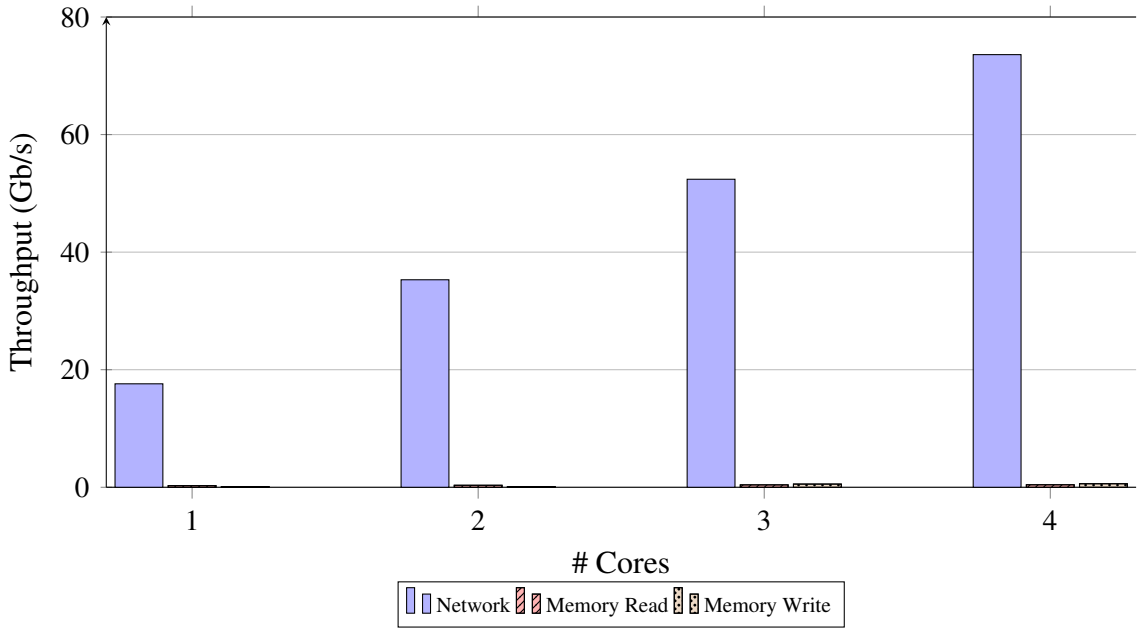


Figure 2.6: Network, Memory Read, and Write Throughput while fetching 8KB blocks from 4 NVMe disks, encrypting them (AES-GCM128), and transmitting them to the network using 2x40GbE NICs.

and achieves a synchronous pipeline: it manages to saturate the hardware, and scale linearly on multiple cores, while serving the workload fully from the CPU cache, without any detours to the main memory.

2.3 Performance limitations with conventional stacks

In this section we discuss known sources of overhead in contemporary network and storage stacks.

2.3.1 Motivation

We present a set of experiments in an attempt to highlight the shortcomings of conventional stacks, while serving critical workloads. In particular, we investigate the performance of contemporary, state-of-the-art web servers while serving workloads typical of web or video-streaming traffic.

2.3.1.1 Workload: short-lived TCP flows

Achieving good performance with short-lived connections is critical, as this kind of workload is typical of web traffic: recent studies reveal that the vast majority of HTTP connections transfer only a small amount of data (less than 25KiB) [AER+11; All12]. Despite the fact that modern

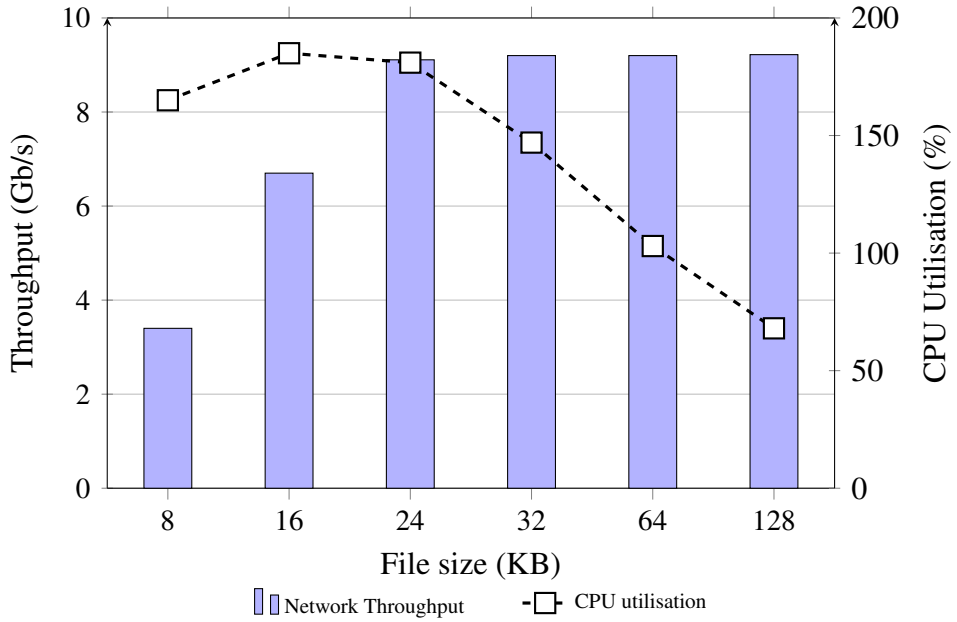


Figure 2.7: Network throughput and CPU utilisation vs. File Size, 2 nginx threads on FreeBSD, 'SandyBridge' CPU, 1x10GbE NIC.

web pages are content-rich, most contemporary browsers open several connections to each domain to fetch all the relevant web objects in parallel, and reduce latency. Unfortunately, HTTP pipelining (batching) is not a viable option, since it requires HTTP request serialisation which eventually increases the page loading times.

Figure 2.7 illustrates how the TCP flow lifespan (controlled by varying the content size served) affects the throughput and system behaviour. An effective network throughput of ~9.2Gb/s is achieved while serving file sizes of 24KB or larger. On the contrary, we observe that workloads that consist of short-lived connections (i.e., smaller file sizes) actually stress the conventional stack. This behaviour is counter-intuitive: it would be reasonable to expect that shorter flows are more likely to achieve peak performance with modern hardware, as they should put less pressure to the CPU cache. What is actually happening? With short-lived flows the system experiences high CPU utilisation as the amount of data that is being transferred with each HTTP connection is not sufficient to compensate for the cost of setting up and tearing down TCP connections; these require a series of expensive operations such as socket and buffer allocations/deallocations, and multiple domain transitions to achieve interaction between kernel and the userspace application. Additionally, shorter transfers result in greater relative network I/O overhead in comparison to bulk transfers, first because NIC hardware smart features such as TCP Segmentation Offload (TSO) cannot be as effective with shorter flows, and second because the system-related overhead remains roughly constant per file/message when using OS primitives such as `sendfile`.

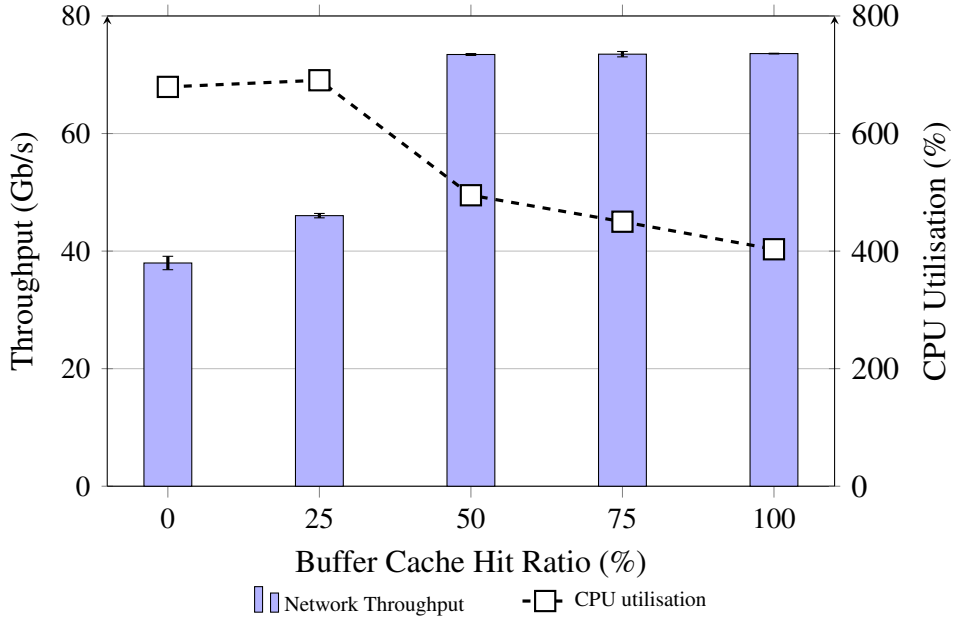


Figure 2.8: Network throughput and CPU utilisation vs. Buffer Cache Hit Ratio, Vanilla nginx + FreeBSD, 'Haswell' CPU, 2x40GbE NICs. Fetching 300KB video chunks over HTTP-persistent TCP connections.

2.3.1.2 Workload: long-lived TCP flows

The inefficiencies of commodity stacks while serving short-lived TCP flows have recently been highlighted by several studies [BPP+16; HMC+12; JWJ+14; MWH14]. On the contrary, conventional wisdom suggests that commodity stacks should perform great while serving workloads that consist of elephant flows (bulk transfers) [HMC+12]. Is this the case? As mentioned previously, bulk transfers are truly inexpensive, mostly due to two reasons: first, because NIC hardware acceleration (e.g., TSO, LRO) has the greatest efficiency with long flows, and second because the system cost for connections' bookkeeping is greatly amortised by the amount of data transferred. The previous hypothesis, however, requires two strong assumptions: *(i)* the commodity stacks serve static content, so that they can take advantage of zero-copy operations achieved with `sendfile`, an *(ii)* the working set (including the content served) can fully fit in the server's main memory. This could be a reasonable assumption for web caches that only serve very popular content such as the Google frontpage, but it is not really practical for generic CDN caches that are responsible for serving larger amounts of data (sometimes less popular) that greatly exceed modern DRAM sizes (e.g., in the range of petabytes for Akamai and Netflix CDN servers). Procurement of enough hardware to hold such workloads fully in DRAM (using sharding and/or other distributed computing technologies) would be at least wasteful and uneconomical from a financial standpoint, if not also impractical from a technical point of view. Therefore, most CDN providers need to include persistent storage in the network fast path. That should be fine given that `sendfile` was actually designed to optimise data transfers from persistent storage to the network. But what happens in practice?

Figure 2.8 shows the achieved network throughput and the CPU utilisation of a conventional server, while varying the fraction of data that is being served from persistent storage or the main memory. To manage this, we control the workload to achieve different buffer cache hit ratios on the server system. Additionally, we utilise only HTTP-persistent connections for our tests, to minimise the book-keeping costs on the server machine. The results are enlightening: the conventional OS evidently struggles with workloads that are not fully served from the main memory, experiencing reduced throughput and higher CPU utilisation. It is worth noting that observing CPU utilisation alone would be very misleading in this case: although it seems that there is plenty of CPU idle time, further analysis reveals that the system is frequently blocked reclaiming VM pages (for an extensive analysis see Chapter 4).

2.3.2 Scalability bottlenecks, and optimisations

The microbenchmarks presented previously suggest that despite decades of evolution and optimisation, contemporary I/O stacks still suffer substantial undiagnosed processor-time inefficiency while operating in important parts of the workload space such as common web traffic, or workloads larger than DRAM size. We attempt to explore the major scalability bottlenecks in conventional systems, and survey related work in the area of OS, network, and storage stack optimisations for performance.

Scaling Non-Blocking Execution: Modern web server applications need to handle thousands of file descriptors that abstract network connections and/or data files, in order to cope with CDN workloads. How do we address this problem? One possible systems-software approach would be to dedicate one thread per connection, and simply rely on blocking I/O calls such as the traditional UNIX `read` and `write` operations for data movement. In practice, this would be a bad design choice, since experience shows very poor system scalability as the number of threads/processes increases, due to heavy scheduling overhead, increased context switching rates, and substantial thread storage space. Non-blocking mode of operation for `read` and `write` is not a solution by itself either: with a naive approach, application threads could waste a lot of resources to continuously poll each and every file descriptor for readiness. Ideally, a more sophisticated solution would incorporate a pool of threads to allow scaling processing to multiple cores, and finally use an efficient mechanism for readiness event notifications.

In order to amortise the large context switching overhead of web servers that relied to one process per connection design approach, many solutions have transitioned to event-driven [PL00; THTTP; ZEUS], or hybrid architectures [PDZ99; WCB01]. In 1999, Banga et al [BMD99] have proposed a scalable implementation of the `select` system call that allowed applications to register interest to several sources of events, and greatly improved the performance of applications that handle large numbers of file descriptors (e.g., connections, files). This proposal motivated later work by Lemon [Lem01] on the `kqueue` interface for BSD: this work not only addresses the scalability issues of event-based notifications, but more importantly it also provides a more general-purpose notification interface by supporting multiple different types of

events (e.g., signals, asynchronous I/O, timers) in addition to the usual file descriptor events (e.g., socket readiness) [MNW14]. Linux also has its own implementation: `epoll` also supports stateful interest sets for events, albeit is less flexible than `kqueue` as it only supports file descriptor based events. These interfaces are now widely used by high-performance web-server implementations such as nginx to reduce the overhead of handling many concurrent connections.

System-Call Overhead: Past research has shown that system calls are expensive and could potentially hinder performance [DP93]. It would be inaccurate, however, to associate the system-call overhead solely with the domain crossing cost, especially on newer hardware: unlike older CPUs which had to raise a slow `INT $80` interrupt, the new `SYSCALL` instruction allows for low-latency roundtrip domain crossings in the range of 100-200 cycles or less, which is actually comparable to a bad cache miss. To our experience, the explicit overhead of domain crossing while experimenting with network and storage stacks was not significant; the implicit cost (instruction and data cache pollution, TLB shootdown etc) is harder to quantify though. Instead, what contributes the most to the system-call overhead is the amount of work being performed in the kernel context (e.g., allocations, sanity checks, memory copies from user to kernel space, synchronisation), as well as the frequency that this happens (e.g., one system call per UDP packet for transmission is really stressing the kernel-user interface). Finally, conventional operating systems fail to achieve linear scaling on multicore systems due to substantial synchronisation costs that are required to achieve compatibility with legacy system-call interfaces [CKZ+13].

To amortise the overheads associated with high context switching rates, the systems community has experimented extensively with the idea of system call *batching* or *clustering*. Cassyopia [RDH+03] is a compiler that attempts to perform optimisations across different address spaces while relying on profiling to identify optimisation opportunities. In particular, Cassyopia could greatly reduce the number of domain crossings by replacing groups of system calls with a single *multi-call*, while still maintaining the same functionality. Furthermore, it employs advanced techniques such as *looped multi-calls*, that allow chaining several system calls by feeding the output of a certain system call to the input of a subsequent one. The concept of *multi-calls* is also widely used in paravirtualised hypervisors [XNML9] to batch hypercalls and reduce the amount of transitions between the guest and the hypervisor.

FlexSC [SS10; SS11] identified that cache pollution and pipeline flushing are the primary factors of system-call overhead, and proposed an exception-less implementation of the system call interface by having user and kernel share the syscall pages: leveraging this asynchronous mechanism, FlexSC allows for parallel execution (batching) of system call requests. This solution, however, does not come without its own drawbacks: the asynchrony introduced between system call invocation and execution, could harm cache locality especially in multi-core systems. Unlike FlexSC, Megapipe [HMC+12] and VOS [VAK11] allow efficient system call batching of asynchronous I/O requests and completion notifications over synchronous channels: still, most of these approaches are not POSIX compliant, and it is difficult to apply them in practice

because they require significant kernel- or even user-level code modifications due to changes to the system call interface.

Linux has recently added the `recvmmsg` and `sendmmsg` calls to allow handling multiple UDP messages with a single system call. The new interface is implemented as a thin wrapper over the existing `sendmsg` and `recvmsg` implementations, instead of a bottom-up re-implementation: this optimisation assumes that the explicit kernel-user domain crossing cost is the major bottleneck, and hence it solely targets on reducing the absolute number of system calls. Still, this solution preserves the same number of per-packet locks/allocations/deallocations, and therefore the overall performance improvement is almost negligible according to our experience. For storage, system call batching is possible when using the POSIX `aio`[4] facility [The04] in FreeBSD and Linux: `lio_listio` allows to issue multiple I/O requests in a single system call. To a certain extent, the `sendfile` [SNDF1] call also amortises the domain-crossing overhead by shifting longer consecutive sequences of read/write I/O and synchronisation system calls into the kernel. Although generalised batched system calls see relatively little use in practice, the `sendfile` system call is widely used by state-of-the-art web servers.

Lack of Connection Affinity: With the higher packet rates (44Mpps or more is typical for a 40GbE NIC [CHLNT]) made possible by modern network adapters, it is imperative to distribute the processing load to multiple CPU cores in order to scale performance. In Linux and FreeBSD, the received network packets are spread across (either by software or hardware [SLNS]) different queues, and eventually different CPU cores on a per-flow basis using a 5-tuple hash. Although this solution has several shortcomings (mostly dependent on the workload characteristics), it usually achieves a reasonably fair distribution of load among the different CPU cores. However, this is not a comprehensive end-to-end solution, since it can only achieve partial connection affinity: due to the kernel-user level split and the shared listening socket (with the associated single `accept` backlog queue that enforces access serialisation), a user-level application thread could still accept and process connections that actually come from remote cores; this practically means that the RX/TX takes place on different hardware threads resulting in cache and lock contention issues on multicore systems [PSZ+12]. Advanced NIC hardware features such as accelerated Receive Flow Steering [SLNS] could help improve the situation, but this solution incurs significant setup complexity and is not very effective for workloads that consist of short-lived connections.

Recent Linux kernels, partly address the issue with the `SO_REUSEPORT` [KH13] option, which allows multiple threads/processes to bind to the same port number (avoiding the use of a single listening socket). Affinity-Accept [PSZ+12] and Megapipe [HMC+12], provide per-core accept queues and also ensure that all processing (including kernel and user threads) of a network flow is affined to the same core.

Redundant Memory Copies: With conventional operating systems, a user-level application relies on the BSD socket API for networking and uses calls such as `read, write, recvmsg,`

[4]The `aio` subsystem uses kernel processes to service asynchronous I/O requests.

`sendmsg` to communicate data from user buffers to the kernel and vice versa. These calls, however, introduce significant cost (in terms of memory traffic and CPU cache pressure) as the data need to be copied to different buffers while switching to different domains (kernel, user) for protection and synchronisation reasons. Explicit data copying is not a requirement to achieve the aforementioned goals though; it is, rather, a design choice to provide a synchronous copy behaviour: a user application is allowed to reuse a buffer, only after a system call that consumes these data (e.g., `send`) has returned. The systems community has observed that the bottleneck in systems processing has shifted from the CPU to the memory system long time ago [WM95]; still, conventional operating systems have not radically changed to reflect this reality, wasting CPU cycles and memory bandwidth to copy data between protection domains and kernel subsystems.

Several research proposals have concluded that there could be substantial performance gains by leveraging *sharing* semantics for I/O, as opposed to relying on data cloning (memory copying). Fast buffers ("fbufs") relies on page remapping and shared memory, to avoid copying and achieve high data throughput across protection domains [DP93]. IO-lite [PDZ00] extends "fbufs" to unify I/O buffer and cache management, and allows sharing immutable buffers among all the different subsystems of general-purpose OSes, including userspace applications, the network, and file systems. In the same context, the Streamline buffer management system [BBB11] achieves zero-copy communication between applications and/or I/O devices, and avoids context switches by leveraging multiple shared ring buffers and specialisation (indirection, compression etc). Finally, Stackmap [YHS+16] addresses latency problems with transactional workloads over TCP at the cost of compatibility, by replacing the socket API on the datapath with an extended zero-copy netmap API [Riz12], while still retaining the use of the in-kernel Linux TCP/IP stack.

Linux has added the `splice` [SPL2], `vmsplice` [VMSP2] and `tee` [TEE2] system calls to allow data movement between certain types of file descriptors (i.e., UNIX pipes) without any extra copying between kernel and user space. Similar functionality is also supported in the network transmit side: the contents of a particular file can be sent without needing to copy them from userspace, when using the `sendfile` [SNDF1] system call. The `sendfile` call basically instructs the kernel to construct the appropriate packet headers, and then use the scatter/-gather features of modern NICs to zero-copy VM pages (data) to a particular network connection. This works well when transmitting static data that are present in the page cache, however it cannot work for data that are dynamically generated in application buffers. To tackle this problem, Linux has recently upstreamed the `MSG_ZEROCOPY` patch set, in an attempt to allow zero-copy operations for dynamically created data in userspace buffers [CB17]; this solution changes the semantics of the traditional `sendmsg` call and requires extra synchronisation for the communication of completion notifications to userspace. Similarly, asynchronous I/O requests to raw disk devices in FreeBSD [LIO] also achieve zero-copy operations, by temporarily wiring user pages, and queueing requests directly to the disk hardware queues; like `MSG_ZEROCOPY`, this method carries significant overhead associated with user memory (un)pinning, and requires

extra bookkeeping to manage memory reclaiming in userspace.

Virtual File System overhead: In POSIX-compliant operating systems, almost every system resource is abstracted by a UNIX file (disk files, BSD sockets [Kle86] etc). This offers great programming convenience, but comes at the cost of performance [BCM+10]: to achieve interoperability between subsystems such as storage and network, common generic data structures (reference-counted file descriptors, in-memory objects such as inodes etc) need to be used to reference resources. For filesystems, this file abstraction, although convenient, is actually quite expensive: on average, 87% of the VFS time was spent to support semantics such as concurrency, hierarchical names and in-memory objects [VNP+14]. Similarly for networking: within a single UNIX process, the file-descriptor space is flat and shared. FreeBSD, for example, scans the file-descriptor bitmap to find the first unset bit (minimum unused integer), each time it needs to allocate a new socket. Naturally, this process becomes more and more expensive as the number of concurrent connections increases and, even worse, incurs extra overhead for synchronisation at the presence of multiple threads as it requires acquiring a per-process-global exclusive lock of the file-descriptor table.

Heavy and/or inefficient data structures: The `sk_buff` and `mbuf` systems are the workhorses of the Linux and FreeBSD network stacks respectively. These data structures have been designed and evolved to support a wide range of different use cases, and in order to achieve flexibility, performance is sacrificed under certain critical workloads. When considering message-oriented workloads (high packet rates with small payloads), for example, Linux experiences significant overhead due to excessive stress on the memory subsystem [HJP+10a]: it has to (de)allocate two `skb`-buffers per packet, one for the data and one for the metadata. To the other end of the spectrum, when considering workloads that consist of bulk transfers, FreeBSD requires numerous `mbuf` objects to be chained together to send larger chunks of data (since one mbuf can only wrap a single VM page): with high-bandwidth links, this practically means consuming `mbuf`s at rates in the order of millions per second, which puts significant pressure to the CPU caches, and the memory allocator.

Concurrency: Significant performance potential is sacrificed when scaling to multiple cores, due to lock contention, and CPU cache management/sharing issues. For example, Linux and FreeBSD utilise shared counters for reference-counted garbage collection, global lists of buffer descriptors (e.g., FreeBSD *pbuf*s to manage I/O operations), and protocol control blocks that enforce serialised access, limiting scalability to greater numbers of cores [BCM+10]. This situation is made worse due to inefficient cache management: for example, false cache line sharing could be the reason for substantially degraded application performance [BB08; BCM+10]. Some of these scalability bottlenecks could be easily fixed as the implementation can be modified so that the cores do not have to block and wait for each other (e.g., coarse-grained locks); sometimes though these bottlenecks are very difficult to eliminate due to either the semantics of the shared resource, or the overall system architecture, and POSIX interface properties [CKZ+13].

Kernel Scheduler overhead: Commodity OSes separate network protocol processing from the relevant consumer applications, in order to achieve scheduling flexibility and optimise for CPU multiplexing. As a result, they need to sacrifice latency, and heavily utilise queuing and intermediate buffering to facilitate context switching. To make things worse, packet and task scheduling decisions are loosely coupled, as they mostly rely on soft and device interrupts to distribute CPU cycles among protocol and application processing [MR97]. Previous research [PLZ+14] has showed that the scheduler's overhead could be quite significant; it could sum up to ~40% of the total packet processing overhead, depending on the process scheduling status at the time of a network event (e.g., a receiving process that sleeps).

Google is already extensively customising the Linux kernel scheduler to achieve more efficient load balancing for heterogeneous workloads that are executed simultaneously on the same hardware and require thousands of application threads [CW09].

2.4 Kernel bypass, and hardware offload

OS-bypass and direct userspace I/O is another approach explored by the systems community in an attempt to optimise performance, and work around conventional operating systems pitfalls. Most common arguments for leveraging kernel-bypass and implementing network and storage stacks in userspace include flexibility (in terms of deployment, debugging etc), as well as greater potential for application-specific optimisations (*specialisation*).

In 1993, Thekkath et al. [TNM+93], have prototyped modular, library-synthesised userspace network stacks implemented on Mach to facilitate code maintenance, customisation and ease of debugging. In the same context, Edwards et al. [EM95] showed how to build high-performance user-level TCP stacks, and demonstrated significant performance gains over the conventional in-kernel HP-UX network stack back in 1995. Seeking to address problems related to network latency and throughput with small messages (RPC-like), von Eicken et al. have presented U-Net [EBB+95], a novel architecture that completely eliminates the need for kernel mediation in the send and receive paths. Similarly, Arsenic [PF01] is a Gigabit Ethernet NIC with hardware support for rich virtualisation and packet filters, that allowed direct userspace I/O access without compromises in resource management or security. Several projects [MCZ06; MN03; RG05] explored mechanisms to efficiently virtualise high-performance NICs, raid controllers or graphic processor units (GPUs). Some of these hardware devices greatly simplified virtualisation complexity and reduced overhead, by supporting a number of virtual I/O channels that could be dedicated to a particular virtual machine or application. Nowadays, the latest PCIe generations also include similar I/O virtualisation support: this allows a single physical PCIe device to appear as multiple independent virtual devices (*virtual functions*), and relies to the IOMMU to enforce access protection.

Over the last few years, userspace I/O has drawn a lot more attention as a prominent method to successfully exploit the wasted hardware capabilities. Many recent frameworks provide the

necessary abstractions to streamline userspace application development with direct access to I/O devices. In 2005, PF_RING-DNA exposed full NIC DMA access to userspace in order to achieve lossless capturing of small packets at line speeds [Der05]. More recently, Intel has released the DPDK [DPDK] software suite that leverages hardware virtualisation and OS-bypass to provide full userspace I/O to the network interfaces. DPDK has gained significant traction in the enterprise, and it is utilised extensively in hypervisors, and in services based on network functions virtualisation. Similar to DPDK, Netmap [Riz12] implements high-throughput and safe network I/O, by exposing DMA memory to userspace. Unlike DPDK though, netmap abstracts DMA operations initiation with system calls, and relies on batching to effectively reduce the context-switch overhead. The PacketShader [HJP+10b] I/O engine utilises a custom network device driver that drops the use of `sk_buff` in favour of a more simple API, and relies on shared memory between kernel and userspace to pass packets in both directions. Finally, aiming to improve performance of BPF programs, and avoid the cost and complexity associated with `sk_buff` (de)allocations, the XDP kernel facility [HS16] processes RX packet-pages directly out of the driver.

Several network stacks have been developed in an attempt to replace the in-kernel stacks in cases that they proved to be unsuitable (e.g., due to excessively large memory footprint for low-end devices), or performed poorly. lwIP [LWIP], for example, is a lightweight general-purpose network stack which was designed for low-end, embedded devices; although performance and specialisation are not their primary goals, we have borrowed several ideas for our TCP API (see Chapter 3). mTCP [JWJ+14] is another recent effort in userspace networking that demonstrates significant performance improvements: likewise, this work is related, but differs from ours in that it offers a general-purpose stack aiming to minimise integration effort for existing applications, while our proposals sacrifice backward compatibility in favour of specialisation and performance. Finally, the Solarflare OpenOnload [SLRFO] is one of the most sophisticated commercial proposals: it is a hybrid stack, capable of operating both in user and kernel mode, minimising interrupts, data copies, and context switches. It requires, however, a vendor-specific hardware (NIC) to operate, and likewise retains the BSD sockets API rather than exploiting domain-specific knowledge for cross-layer optimisations.

A lot of research has focussed in the area of the hardware/software interface for I/O devices (NICs, disks etc). FlexNIC [KPS+16] rethinks and extends NIC abstractions, and provides a rich and flexible network DMA interface to reduce packet processing overheads; this work is orthogonal to ours, as our specialised stacks could benefit from such a DMA interface. Some hardware vendors provide rich packet filtering capabilities to offload security checks, and sophisticated load balancing decisions to the hardware (e.g., Solarflare 10GbE NICs). Others have gone further and designed advanced NIC ASICs that are able to run a fully-fledged TCP/IP stack [Cht6; ToE05]. Mostly known as TCP Offload Engine (TOE), this technology completely bypasses the in-kernel network stack, while still relying on the BSD socket layer abstraction for transparent application compatibility: the ASIC TCP/IP stack is DMAing data directly to and from the kernel socket buffers. This approach can dramatically reduce latency and CPU utili-

sation, however, this comes at the cost of flexibility, and scalability (due to limited resources). TCP Offload Engines are typically supplied as a black box for consumers, so deploying alternative congestion control mechanisms, advanced tuning, customisation, or even bug fixes could be difficult or even impossible.

Kernel bypass, and several other techniques that have been used to accelerate network packet processing have been recently employed to optimise storage I/O performance. Historically, magnetic disks have been the *de facto* standard in enterprise storage: spindle latency in this case was the limiting factor in scaling storage performance; with modern flash disks, hardware throughput and latency is no longer the bottleneck. Intel's SPDK [SPDK] is a contemporary effort to *diskmap* (see Chapter 4.3.1), that implements a high throughput and low latency NVMe storage I/O framework, by running the NVMe device drivers fully in userspace. FusionIO's Direct File System [JBL+10] also uses direct userspace I/O access that bypasses the buffer cache, to achieve up to 20% better performance over the ext3 filesystem while using vendor-specific (ioDrive) disks. Moneta-D [Cau10] is a software and hardware platform that reduces access latency of flash-based SSDs by offloading permissions checks to the hardware, and allowing DMA operation handling from userspace without any kernel mediation.

2.5 Alternative transport, and application-layer protocols

In the scope of this dissertation, we seek to improve system performance with transport- and application- layer protocols that are widely deployed in the real world (e.g., TCP, UDP, HTTP etc). Over the years, however, a number of researchers have argued about fundamental limitations in these extensively used protocol stacks that affect both network characteristics (e.g., latency), and system efficiency.

In 1989, David D. Clark et. al conducted an analysis of the TCP processing overhead, arguing that the observed performance limitations stem from poor implementation or hardware bottlenecks, and are not inherent to the protocol itself [CJR+89]. Although this study concludes that a solid implementation of TCP can keep up with high packet rates and throughput, it does not actually claim that network processing is cheap, but rather that most of the overhead is associated with functions such as buffer and timer management, checksums etc. Moreover it only investigates the TCP fast path (not connection establishment, retransmission etc), and is actually based on relatively older hardware, and slower network adapters. The emergence of the first 10GbE adapters revealed that faster networking hardware could actually strain server resources (CPU and memory bandwidth), and the associated cost could be attributed to both TCP and the required OS support (e.g., socket API) [Bal04; FHH+03]. Several systems have been designed to completely avoid the use of TCP, to work around all of the performance limitations related to both the protocol specifics (e.g., connection establishment delay due to 3-way handshake), as well as the conventional stack implementations' shortcomings (e.g., poor performance with short-lived connections). Facebook's memcached deployment relies on UDP in order to reduce

latency and overhead for `get` requests [NFG+13]. Similarly, MICA [LHA+14] is a scalable in-memory key-value store that relies on direct userspace I/O (via DPDK) and UDP to scale performance of read requests in large-scale deployments. In 2013, Google has presented QUIC, an experimental new transport designed from scratch to improve performance for HTTPS traffic, and replace the traditional HTTP (1.1 or 2), TLS, and TCP stacks. QUIC was built as a user-level transport on top of UDP [LRW+17; QUIC], and is currently widely deployed at Google servers. Although these systems manage to minimise system overheads, this does not come without compromises: reliable transmission and congestion control is still required for efficient operation, and now this has to be part of the application itself.

Following U-Net [EBB+95], the Infiniband hardware standard [Inf4] focussed on maximising throughput and reducing latency for parallel applications, by minimising OS involvement in the network fast path. Remote Direct Memory Access (RDMA) is another increasingly adopted model for userspace networking [RDMA], especially in datacenter environments. RDMA-based systems typically leverage specialised NIC adapters (Infiniband) to offload network processing, and give applications the ability to perform read and write operations from/to virtual memory regions of remote machines, without any need for OS intervention [DNH+14; MGL13; ORS+11]. RDMA can greatly reduce latency, however, it requires specialised hardware, and more importantly it assumes a mutual trust model between client and server for efficient usage[5], an assumption that is not valid for Wide Area Network (WAN) services.

Several workloads are considered particularly stressful for commodity stacks due to specific properties (e.g., short-lived flows as we have shown previously). To work around several of these issues the networking community has also experimented with several application-layer protocol modifications/enhancements. Modern web pages, for example, are highly content-rich, but the corresponding user web traffic is, perhaps counter-intuitively, composed of many shorter flows [AER+11; All12]. This happens because modern user browsers open multiple flows per server in order to parallelise fetching the different web objects of a page. Although HTTP/1.1 supports request pipelining, so that a client can use a single TCP connection to fetch multiple objects, this feature is rarely used because it requires HTTP request serialisation, and hence increased latency and head-of-line blocking occurs, leading to significantly higher page load times [AER+11]. To address these limitations Google's SPDY [SPDY] has been used as an enhancement to HTTP/1.1, and formed the basis of the technical specification of HTTP/2 [BPM15]. These protocols manage to significantly reduce the web page load times by multiplexing HTTP request and responses, and by supporting header compression, and prioritisation of requests. These application layer protocol optimisations assist to partially work around several performance limitations of conventional stacks, but are definitely not a comprehensive solution by themselves: the benefits are almost negligible with smaller pages, or with RPC/REST interfaces, and can be easily overwhelmed by dependencies and computation, while the presence of network packet loss can have detrimental effects in performance [WBK+14]. Nevertheless, these application-layer protocol enhancements are orthogonal to, and could ben-

[5] Otherwise, memory isolation guarantees are needed at a per-connection granularity.

efit from many of the system performance improvements presented in this dissertation.

2.6 In-kernel 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 [Bar00], a kernel-based web server which still relies on the BSD socket interface: being a kernel daemon itself, kHTTPd interfaces directly internal kernel structures and it avoids the extra overhead associated with the user-kernel interface (explicit domain crossing overheads, TLB flushing, memory copies etc). TUX [LEM00] is another noteworthy example of in-kernel network applications. Unlike kHTTPd, TUX emphasises on specialisation, and achieves greater performance by eliminating the socket layer and pinning the static content it serves in memory. We have adopted several of these ideas in our prototypes, although our approach is not kernel-based.

Several other optimisations violate the end-to-end principle [SRC84] in favour of performance, by blurring the barriers between the user-level application layers, and the kernel-level protocol processing. For example, HTTP accept filters in FreeBSD [ACF+H] allow an application to request the kernel to pre-process incoming connections, and ideally defer handing them over to the userspace application up until a complete HTTP request has been buffered by the kernel, so as to eliminate unnecessary context switching. Solaris' NCA accelerates web server performance by maintaining an in-kernel cache of accessed web pages, and either fully handles the request in kernel, or passes it to a user-level HTTP server via a `door` [NCA]. Similar to NCA, the Windows HTTP.sys response cache also accelerates HTTP performance, by handling several HTTP requests entirely in the kernel.

2.7 Microkernels, unikernels, and research operating systems

Microkernel designs such as Mach [ABG+86] have long appealed to OS designers, pushing core services (such as network stacks) into user processes so that they can be more easily developed, customised, and multiply-instantiated. Hydra's [WCC+74] capability-based, object-oriented design focusses on extensibility, allowing user-level programs to manage resources through multi-level policies. Exokernels also expose hardware features to userspace via a lightweight, low-level kernel interface, and implement system abstractions via library operating systems [EKO95]. The Cheetah web server [GEK+02] was built on top of an Exokernel library OS, allowing developers to greatly improve performance by directly accessing hardware and using a specialised network stack and file system; our Sandstorm stack presented in detail

OS	Language	Target Environment
OSv [KLC+14]	C/Java	KVM, Xen
RUMP [Kan12]	C	Linux kernel, POSIX
Drawbridge [PBH+11]	C	Windows "picoprocess"
MirageOS [MMR+13]	OCaml	Xen, KVM, kFreeBSD, POSIX
Exokernel [EKO95]	C	Aegis, XOK
Arrakis [PLZ+14]	C	Barrelfish, POSIX
IX [BPP+16]	C	"Dune" [BBM+12] process

Figure 2.9: A selection of library OSes, and their target environments.

in Chapter 3 was greatly inspired by this work. Scout [MP96] and Nemesis [LMB+06] also follow a similar direction, by restructuring the OS into libraries that are linked to user-level applications. The SPIN [BSP+95] operating system uses language and link-time mechanisms that allow loading of extension system services in kernel, aiming to achieve on-demand specialisation of the operating system according to the needs of each individual application. More recently, Drawbridge [PBH+11] has demonstrated a scalable redesign of a real commercial OS (Windows 7), to a library OS; it defines a deliberately narrow and lightweight ABI of 36 calls to virtualise a host kernel towards the library OS.

Unikernel designs such as MirageOS [MMR+13] likewise blend operating-system and application components at compile-time, trimming unneeded software elements to accomplish extremely small memory footprints— although by static code analysis rather than application-specific specialisation. Unikernels are single address space systems, and usually bundle an application with a selection of system components to produce a lightweight image. Most commonly, unikernel implementations use a single type-safe language (e.g., OCaml for MirageOS, Haskell for HalVM [GLIS+H]) and all the necessary basic components are developed from scratch. Unlike MirageOS and HalVM, OSv [KLC+14] and RUMP kernels [Kan12] focus on providing compatibility with a wide range of existing applications, avoiding the need to re-implement all the necessary building blocks (e.g., drivers, filesystems). ClickOS [MAR+14] modifies the ABI, to achieve high-performance, dense deployments of Xen VMs that specialise in network processing applications (e.g., routing, forwarding).

Arrakis [PLZ+14] is a research operating system that leverages hardware virtualisation to efficiently decouple the control and data planes for I/O-intensive applications. Diskmap (see Chapter 4.3.1) was partially inspired by Arrakis, which first applied the idea of a fast and safe user-level storage data plane. Similarly, IX [BPP+16] uses hardware virtualisation to enforce safety, while utilising zero-copy APIs and adaptive batching to achieve high performance network I/O.

2.8 Summary

Modern hardware capabilities are improving at a tremendous pace, and this presents a major challenge for contemporary systems software, which plainly fails to effectively utilise the full hardware potential. This chapter traces the most influential accomplishments in network, and storage hardware, as well as CPU microarchitecture. It also surveys many existing software systems that are widely used for Internet or data center servers. Through a set of microbenchmarks we illustrate the mismatch between contemporary OS and hardware capabilities in terms of network and storage I/O performance. We start by examining major changes in CPU microarchitecture along with their practical implications (§2.2), then discuss challenges, limitations, and existing optimisations of conventional operating systems when serving common critical workloads (§2.3), and finally survey alternative approaches in systems design that attempt to overcome known limitations of commodity stacks, and push I/O performance (§2.4, §2.7).

Conventional operating systems suffer from generality, which in contemporary implementations translates to high asynchrony, bufferbloat, and legacy suboptimal data structures, mechanisms and APIs: these flaws have a detrimental impact in I/O performance, especially in modern hardware generations. Chapters 3 and 4 discuss the design of clean-slate stacks that manage to overcome historical bottlenecks in scaling I/O performance, and fully utilise hardware resources.

Chapter 3

Network stack specialisation

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

In this chapter, we present Sandstorm and Namestorm, web and DNS servers that utilise a clean-slate userspace network stack that exploits knowledge of application-specific workloads. In particular, these systems are focussed on workloads that fit in the main memory (DRAM), so persistent storage is intentionally left out of the network fast path; we relax this constraint in chapter 4. Based on the netmap framework, our novel approach merges application and network-stack memory models, aggressively amortises protocol-layer costs based on application-layer knowledge, couples tightly with the NIC event model, and exploits microarchitectural features. Simultaneously, the servers retain use of conventional programming frameworks. We compare our approach with the FreeBSD and Linux stacks using the nginx web server and NSD name server, demonstrating 2–10× and 9× improvements in web-server and DNS throughput, lower CPU usage, linear multicore scaling, and saturated NIC hardware.

3.1 Introduction

Conventional network stacks were designed in an era where individual systems had to perform multiple diverse functions. In the last decade, the advent of cloud computing and the ubiquity of networking has changed this model; today, large content providers serve hundreds of millions of customers. To scale their systems, they are forced to employ many thousands of servers, with

each providing only a single network service. Yet most content is still served with conventional general-purpose network stacks.

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

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

Servers such as Sandstorm could be used for serving images such as the Facebook logo, as OCSP [MAM+99] responders for certificate revocations, or as front end caches to popular dynamic content. This is a role that conventional stacks should be good at: nginx [NGINX] uses the `sendfile` system call to hand over serving static content to the operating system. FreeBSD and Linux then implement zero-copy stacks, at least for the payload data itself, using scatter-gather to directly DMA the payload from the disk buffer cache to the NIC. They also utilise the features of smart network hardware, such as TCP Segmentation Offload (TSO) and Large Receive Offload (LRO) to further improve performance. With such optimisations, nginx does perform well, but as we will demonstrate, a specialised stack can outperform it by a large margin.

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

Our userspace web server and DNS server are built upon FreeBSD's netmap [Riz12] framework, which directly maps the NIC buffer rings to userspace. We will show that not only is it possible for a specialised stack to beat nginx, but on data-center-style networks when serving small files which is typical of many web pages, it can achieve three times the throughput on older hardware, and more than six times the throughput on modern hardware supporting DDIO[1].

The demonstrated performance improvements come from four places. First, we implement a complete zero-copy stack, not only for payload but also for all packet headers, so sending data is

[1]Direct Data I/O. For more details, see Section 3.2.4

very efficient. Second, we allow aggressive amortisation 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 minimises the latency of sending the first packet of the response. Finally, on recent systems, Intel's DDIO provides substantial benefits, but only if packets to be sent are already in the L3 cache and received packets are processed to completion immediately. It is hard to ensure this on conventional stacks, but a special-purpose stack can get much closer to this ideal.

Of course, userspace stacks are not a novel concept. Indeed, the Cheetah web server for MIT's XOK Exokernel [EKO95] operating system took a similar approach, and demonstrated significant performance gains over the NCSA web server in 1994. Despite this, the concept has never really taken off, and in the intervening years conventional stacks have improved immensely. Unlike XOK, our specialised userspace stacks are built on top of a conventional FreeBSD operating system. We will show that it is possible to get all the performance gains of a specialised stack without needing to rewrite all the ancillary support functions provided by a mature operating system (e.g., the file system). Combined with the need to scale server clusters, we believe that the time has come to re-evaluate special-purpose stacks on today's hardware.

The key contributions of our work are:

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

porting synchronous applications. This approach optimises cache utilisation by both the CPU and DMA engines, yielding as much as 2-10× conventional stack performance.

3.2 Special-purpose architecture

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

- Conventional network stacks support zero copy for OS-maintained data – e.g., filesystem blocks in the buffer cache, but not for application-provided HTTP headers or TCP packet headers. Can we take the zero-copy concept to its logical extreme, in which received packet buffers are passed from the NIC all the way to the application, and application packets to be sent are DMAed to the NIC for transmission without even the headers being copied?

- Conventional stacks make extensive use of queuing and buffering to mitigate context switches and keep CPUs and NICs busy, at the cost of substantially increased cache footprint and latency. Can we adopt a bufferless event model that reimposes synchrony and avoids large queues that exceed cache sizes? Can we expose link-layer buffer information, such as available space in the transmit descriptor ring, to prevent buffer bloat and reduce wasted work constructing packets that will only be dropped?

- Conventional stacks amortise expenses internally, but cannot amortise repetitive costs spanning application and network layers. For example, they amortise TCP connection lookup using Large Receive Offload (LRO) but they cannot amortise the cost of repeated TCP segmentation of the same data transmitted multiple times. Can we design a network-stack API that allows cross-layer amortisations to be accomplished such that after the first client is served, no work is ever repeated when serving subsequent clients?

- Conventional stacks embed the majority of network code in the kernel to avoid the cost of domain transitions, limiting two-way communication flow through the stack. Can we utilise efficient batching to allow device drivers to remain in the kernel while co-locating stack code with the application and avoiding significant latency overhead?

- Can we avoid any data-structure locking, and even cache-line contention, when dealing with multi-core applications that do not require it?

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

3.2.1 Network-stack modularisation

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

Rapid deployment & reusability: Our prototype stack is highly modular, and synthesised from the bottom up using traditional dynamic libraries as building blocks (*components*) to construct a special-purpose system. Each component corresponds to a stand-alone service that exposes a well-defined API. Our specialised network stacks are built by combining four basic components:

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

Figure 3.1 depicts how some of these components are used with a simple application layer to form Sandstorm, the optimised web server.

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

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

- Debugging is at least as easy (if not easier) compared to conventional systems, as application-specific, performance-centric code shifts from the kernel to more accessible userspace.
- Our approach integrates well with the general-purpose operating systems: rewriting basic components such as device drivers or filesystems is not required. We also have direct access to conventional debugging, tracing, and profiling tools, and can also use the conventional network stack for remote access (e.g., via SSH).

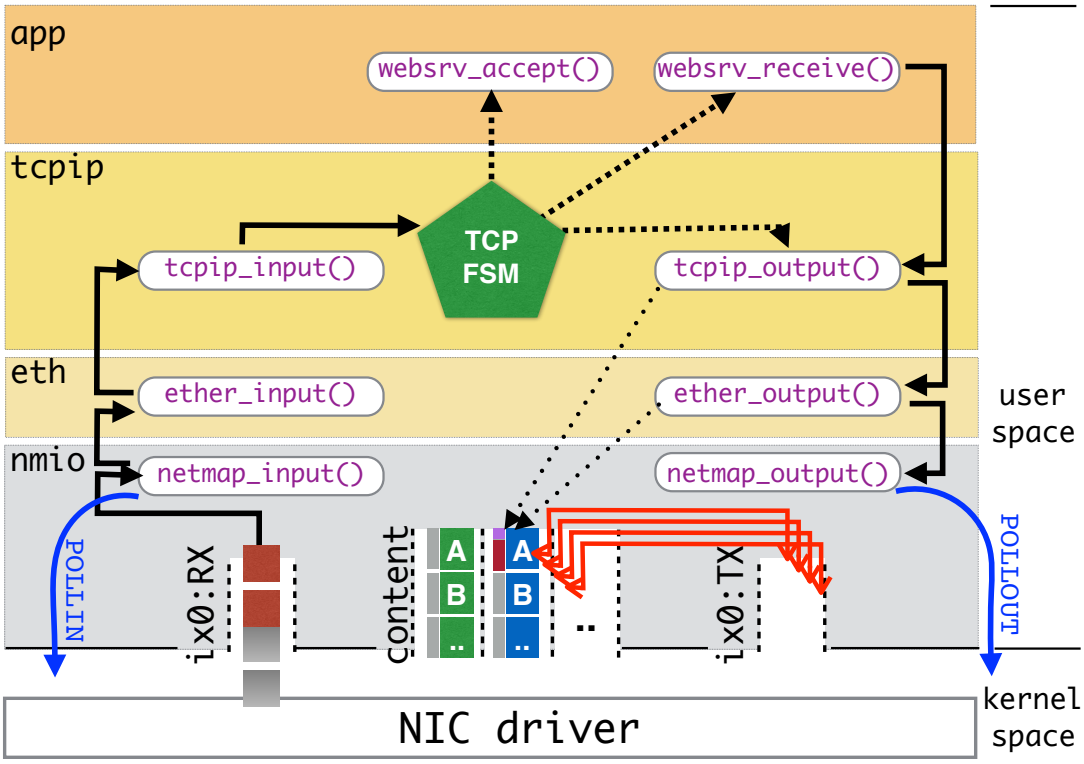


Figure 3.1: Sandstorm high-level architecture view.

- 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.

3.2.2 Sandstorm web server design

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

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

file is requested, and the resulting packets retained to serve future requests, managed as a Least Recently Used (LRU) cache.

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

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

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

- We can maintain only one long-term copy of each packet, creating ephemeral copies each time it needs to be sent.

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

Figure 3.2 illustrates trade-offs through traces taken on nginx/Linux and pre-copy Sandstorm servers that are busy (but unsaturated). On the one hand, a batched design measurably increases TCP round-trip time with a relatively idle CPU. On the other hand, Sandstorm amortises or eliminates substantial parts of per-request processing through a more efficient architecture. Under light load, the benefits are pronounced; at saturation, the effect is even more significant.

Although most work is synchronous within the ACK processing code path, TCP still needs timers for certain operations. Sandstorm's timers are scheduled by polling the Time Stamp Counter (TSC): although not as accurate as other clock sources, it is accessible from userspace

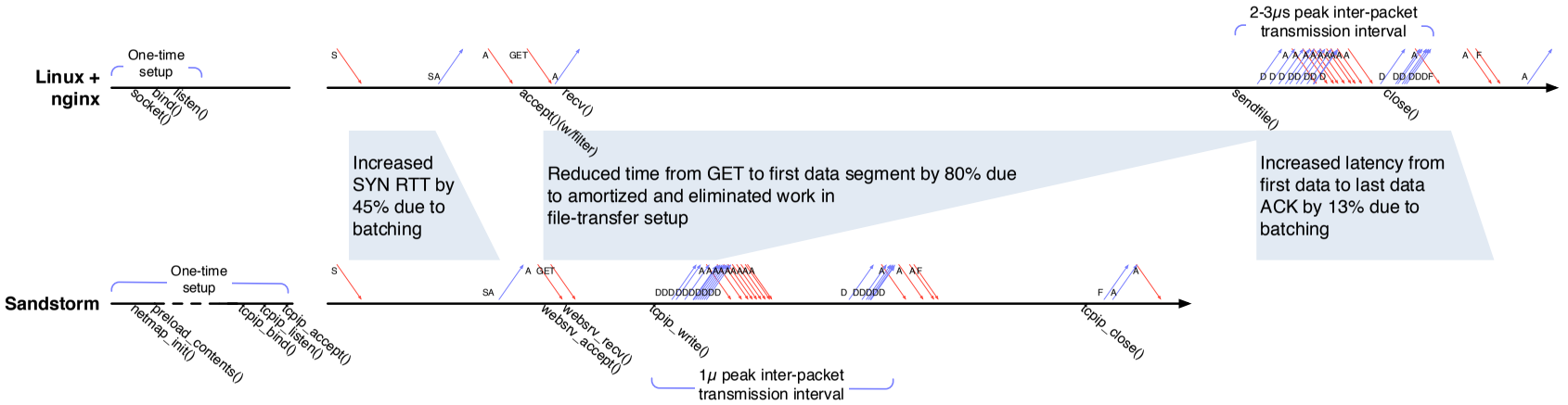


Figure 3.2: Several trade-offs are visible in these packet traces taken on nginx/Linux and Sandstorm servers that are busy (but unsaturated).

at the cost of a single CPU instruction (on modern hardware). The TCP slow timer routine is invoked periodically (every ~500ms) and traverses the list of active TCBs: on RTO expiration, the congestion window and slow-start threshold are adjusted accordingly, and any unacknowledged segments are retransmitted. The same routine also releases TCBs that have been in *TIME_WAIT* state for longer than 2**MSL*. There is no buffering whatsoever required for retransmissions: we identify the segment that needs to be retransmitted using the oldest unacknowledged number as an offset, retrieve the next available prepackaged packet and adjust its headers accordingly, as with regular transmissions. Sandstorm currently implements TCP Reno for congestion control.

3.2.3 The Namestorm DNS server

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

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

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

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

Namestorm preprocesses the zone file upon startup, creating DNS response packets for all the entries in the zone, including the answer section and any glue records needed. In addition to type-specific queries for A, NS, MX and similar records, DNS also allows queries for ANY. A full implementation would need to create additional response packets to satisfy these queries;

our implementation does not yet do so, but the only effect this would have is to increase the overall memory footprint. In practice, ANY requests appear to be comparatively rare.

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

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

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

Our implementation does not currently handle referrals, so it can handle only zones for which it is authoritative for all the sub-zones. It could not, for example, handle the `.com` zone, because it would receive queries for `www.example.com`, but only have hash table entries for `example.com`. Truncating the hash key is trivial to do as part of the translation to an FQDN, so if Namestorm were to be used for a domain such as `.com`, the FQDN version of hashing would be a reasonable approach.

3.2.4 Main event loop

To understand how the pieces fit together and the nature of interaction between Sandstorm, Namestorm, and netmap, we consider the timeline for processing ACK packets in more detail. Figure 3.3 summarises Sandstorm's main loop. SYN/FIN handling, HTTP, and timers are omitted from this outline, but also take place. However, most work is performed in the ACK processing code.

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

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

Figure 3.3: Outline of the main Sandstorm event loop.

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

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

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

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

and increase I/O bandwidth.

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

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

3.2.5 API

As discussed, all of our stack components provide well-defined APIs to promote reusability. Table 3.1 presents a selection of API functions exposed by `libnmio` and `libtcpip`. In this section we describe some of the most interesting properties of the APIs.

`libnmio` is the lowest-level component: it handles all interaction with netmap and abstracts the main event loop. Higher layers (e.g., `libeth`) register callback functions to receive raw incoming data as well as set timers for periodic events (e.g., TCP slow timer). The function `netmap_ouput` is the main transmission routine: it enqueues a packet to the transmission ring either by memory or zero copying and also implements an adaptive batching algorithm.

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

3.3 Evaluation

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

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

NICs, 128GB RAM and a quad-core Intel Xeon E5-2643 CPU. In 2014, these are well-equipped contemporary servers.

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

Our hypothesis is that Sandstorm will be significantly faster than nginx on both platforms; however, the reasons for this may differ. Experience [EGH+08] has shown that the older systems often bottleneck on memory latency, and as Sandstorm is not CPU-intensive, we would expect this to be the case. A zero-copy stack should thus be a big win. In addition, as cores contend for memory, we would expect that adding more cores does not help greatly.

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

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

3.3.1 Experiment design: Sandstorm

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

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

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

Table 3.1: Non-exhaustive list of `libtcpip` and `libnmio` APIs.

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

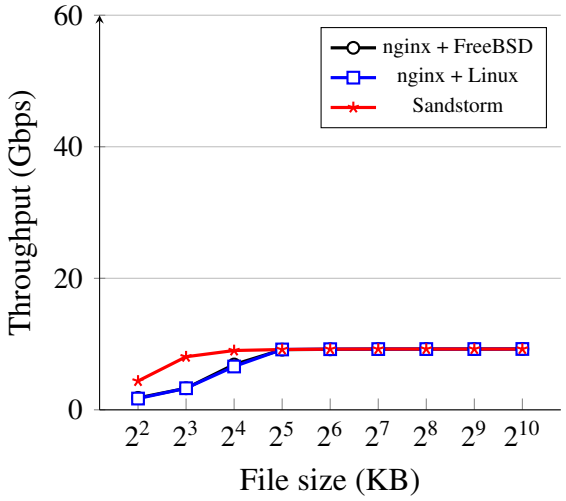
3.3.2 Sandstorm results

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

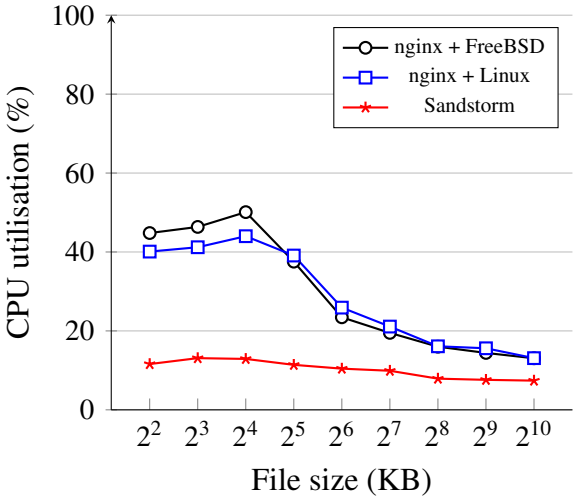
Figure 3.4 shows performance as a function of content size, comparing pre-copy Sandstorm and nginx running on both FreeBSD and Linux. With a single 10GbE NIC (Fig. 3.4a and 3.4b), Sandstorm outperforms nginx for smaller files by ~23–240%. For larger files, all three configurations saturate the link. Both conventional stacks are more CPU-hungry for the whole range of file sizes tested, despite potential advantages such as TSO on bulk transfers.

To scale to higher bandwidths, we added more 10GbE NICs and client machines. Figure 3.4c shows aggregate throughput with four 10GbE NICs. Sandstorm saturates all four NICs using just two CPU cores, but neither Linux nor FreeBSD can saturate the NICs with files smaller than 128KB, even though they use four CPU cores.

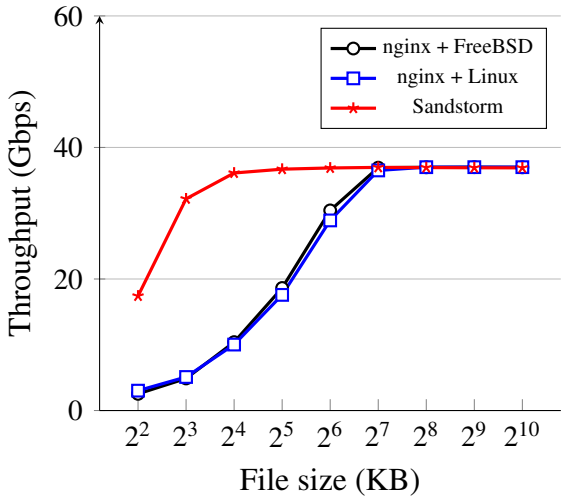
As we add yet more NICs, shown in Figure 3.4e, the difference in performance gets larger for a wider range of file sizes. With 6×10GbE NICs Sandstorm gives between 10% and 10× more throughput than FreeBSD for file sizes in the range of 4–256KB. Linux fares worse, experiencing a performance drop (see Figure 3.4e) compared to FreeBSD with smaller file sizes and 5–6 NICs. Low CPU utilisation is normally good, but here (Figures 3.4f, 3.5b), idle time is undesirable since the NICs are not yet saturated. We have not identified any single obvious cause for this degradation. Packet traces show the delay to occur between the connection being accepted and the response being sent. There is no single kernel lock being held for especially long, and although locking is not negligible, it does not dominate, either. The system suffers one soft page fault for every two connections on average, but no hard faults, so data is already in the disk buffer cache, and TCB recycling is enabled. This is an example of how hard it can be to find performance problems with conventional stacks. Interestingly, this was an application-specific behaviour triggered only on Linux: in benchmarks we conducted with other web servers (e.g., lighttpd [LHTPD], OpenLiteSpeed [OLSP]) we did not experience a similar performance collapse on Linux with more than four NICs. We have chosen, however, to present the nginx



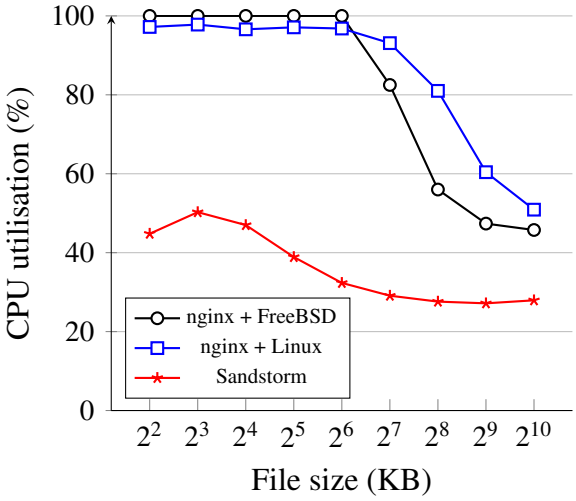
(a) Network throughput, 1 NIC



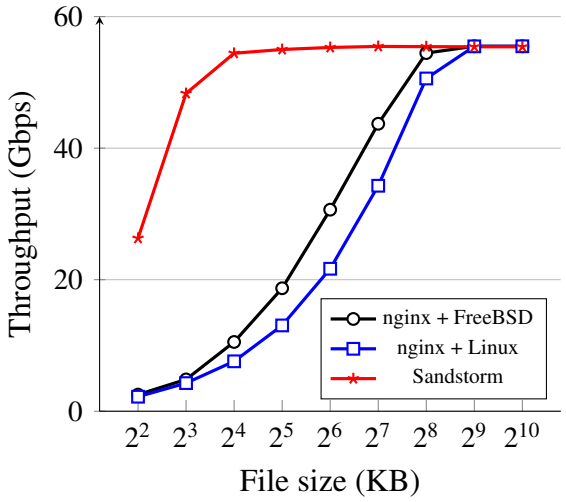
(b) CPU utilisation, 1 NIC



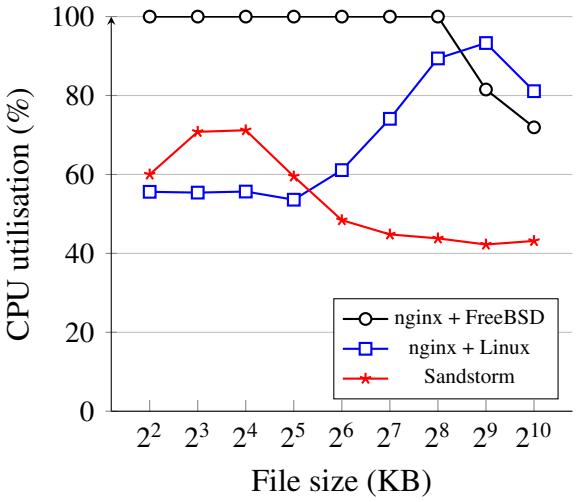
(c) Network throughput, 4 NICs



(d) CPU utilisation, 4 NICs

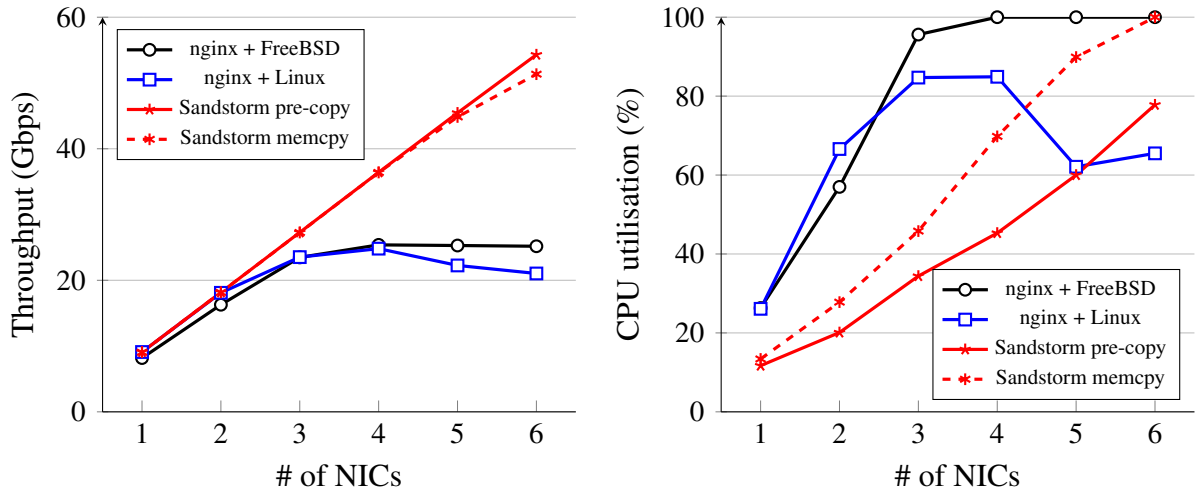


(e) Network throughput, 6 NICs



(f) CPU utilisation, 6 NICs

Figure 3.4: Sandstorm throughput vs. file sizes and number of NICs.



(a) Network throughput

(b) CPU utilisation

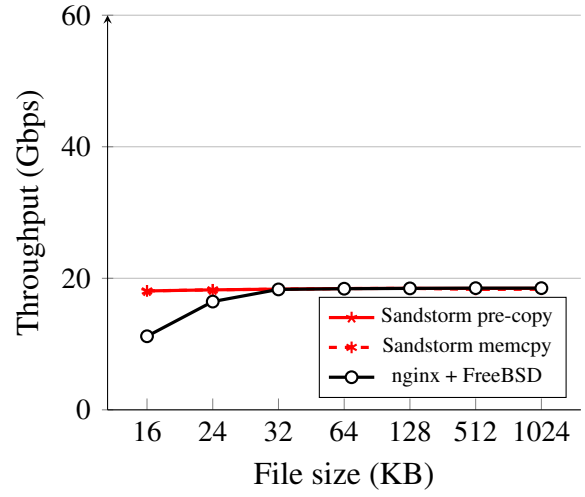
Figure 3.5: Network throughput and CPU utilisation vs. number of NICs while serving a Yahoo! CDN-like workload.

datasets as it offered the greatest overall scalability in both operating systems.

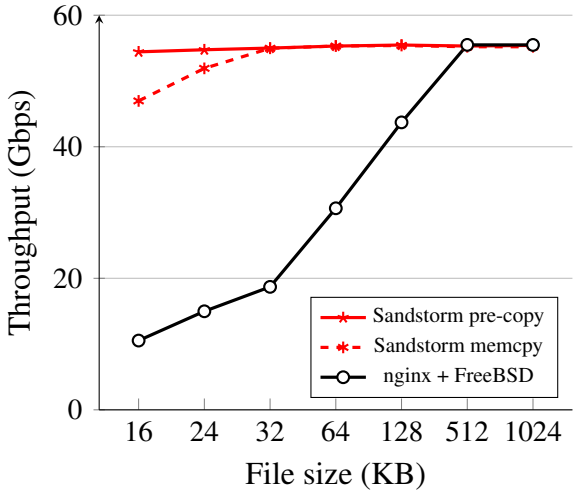
It is clear that Sandstorm dramatically improves network performance when it serves small web objects, but somewhat surprisingly, it performs better for larger files too. For completeness, we evaluate Sandstorm using a realistic workload: following the distribution of requested HTTP object sizes of the Yahoo! CDN [AER+11], we generated a trace of 1,000 files ranging from a few KB up to ~20MB which were served from both Sandstorm and nginx. On the clients, we modified *weighttp* to benchmark the server by concurrently requesting files in a random order. Figures 3.5a and 3.5b highlight the achieved network throughput and the CPU utilisation of the server as a function of the number of the network adapters. The network performance improvement is more than 2× while CPU utilisation is reduced.

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

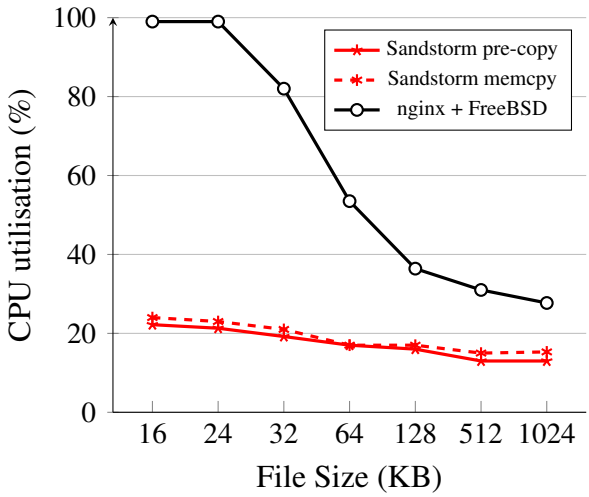
Throughout, we have invested considerable effort in profiling and optimising conventional network stacks, both to understand their design choices and bottlenecks, and to provide the fairest possible comparison. We applied conventional performance tuning to Linux and FreeBSD, such as increasing hash-table sizes, manually tuning CPU work placement for multiqueue NICs, and adjusting NIC parameters such as interrupt mitigation. In collaboration with Netflix, we also developed a number of TCP and virtual-memory subsystem performance optimisations for FreeBSD, reducing lock contention under high packet loads. One important optimisation is re-



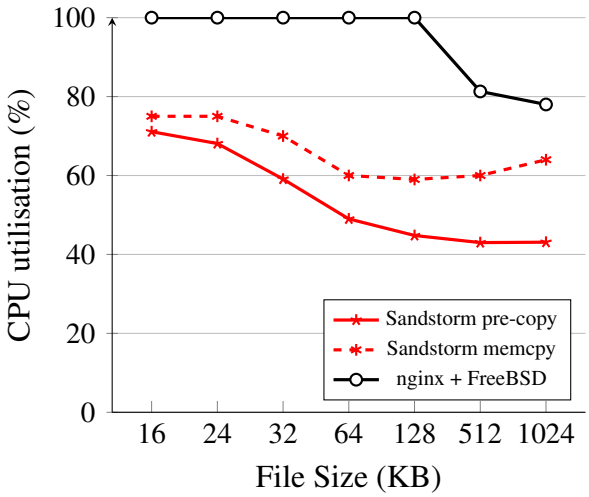
(a) Network throughput, 2 NICs



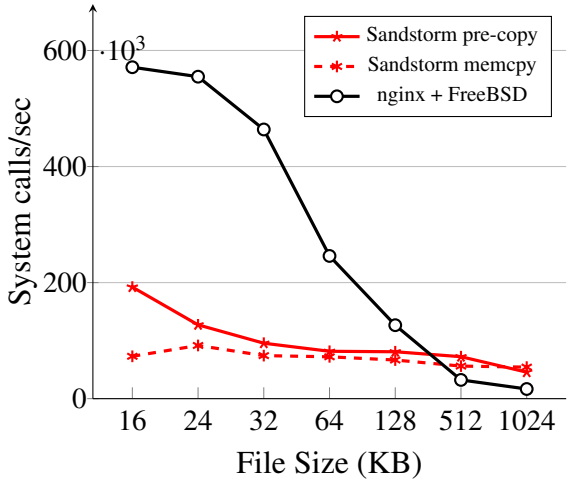
(b) Network throughput, 6 NICs



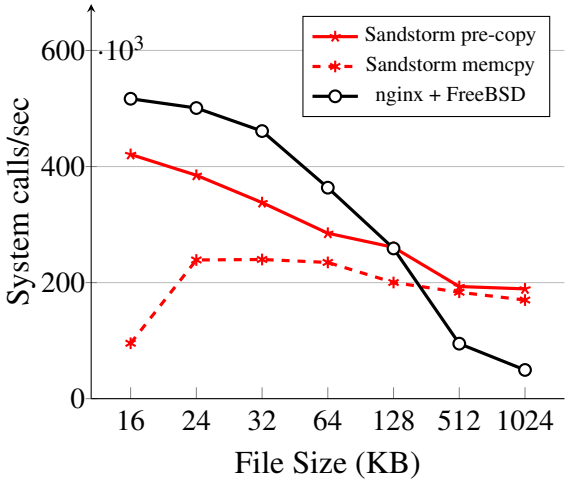
(c) CPU utilisation, 2 NICs



(d) CPU utilisation, 6 NICs



(e) System call rate, 2 NICs



(f) System call rate, 6 NICs

Figure 3.6: Sandstorm throughput, system call rate, and CPU utilisation vs. variable number of NICs and file sizes.

lated to `sendfile`, in which contention within the VM subsystem occurred while TCP-layer socket-buffer locks were held, triggering a cascade to the system as a whole. These changes have been upstreamed to FreeBSD for inclusion in a future release.

To copy or not to copy

The pre-copy variant of Sandstorm maintains more than one copy of each segment in memory so that it can send the same segment to multiple clients simultaneously. This requires more memory than nginx serving files from RAM. The memcpy variant only enqueues copies, requiring a single long-lived version of each packet, and uses a similar amount of memory to nginx. How does this `memcpy` affect performance? Figure 3.6 explores network throughput, CPU utilisation, and system-call rate for two- and six-NIC configurations.

With six NICs, the additional `memcpy` marginally reduces performance (Figure 3.6b) while exhibiting slightly higher CPU load (Figure 3.6d). In this experiment, Sandstorm only uses three cores to simplify the comparison, so around 75% utilisation saturates those cores. The memcpy variant saturates the CPU for files smaller than 32KB, whereas the pre-copy variant does not. Nginx, using `sendfile` and all four cores, only catches up for file sizes of 512KB and above, and even then exhibits higher CPU load.

As file size decreases, the expense of SYN/FIN and HTTP-request processing becomes measurable for both variants, but the pre-copy version has more headroom so is affected less. It is interesting to observe the effects of batching under overload with the memcpy stack in Figure 3.6f. With large file sizes, pre-copy and memcpy make the same number of system calls per second. With small files, however, the memcpy stack makes substantially fewer system calls per second. This illustrates the efficacy of batching: memcpy has saturated the CPU, and consequently it no longer polls the RX queue as often. As the batch size increases, the system-call cost decreases, helping the server weather the storm. The pre-copy variant is not stressed here and continues to poll frequently, but would behave the same way under overload. In the end, the cost of the additional `memcpy` is measurable, but still performs quite well.

Results on contemporary hardware are significantly different from those run on older pre-DDIO hardware. Figure 3.7 shows the results obtained on our 2006-era servers. On the older machines, Sandstorm outperforms nginx by a factor of three, but the memcpy variant suffers a 30% decrease in throughput compared to pre-copy Sandstorm as a result of adding a single `memcpy` to the code. It is clear that on these older systems, memory bandwidth is the main performance bottleneck.

With DDIO, memory bandwidth is not such a limiting factor. Figure 3.9 in Section 3.3.5 shows the corresponding memory read throughput, as measured using CPU performance counters, for the network-throughput graphs in Figure 3.6b. With small file sizes, the pre-copy variant of Sandstorm appears to do more work: the L3 cache cannot hold all of the data, so there are many more L3 misses than with memcpy. Memory-read throughput for both pre-copy and nginx

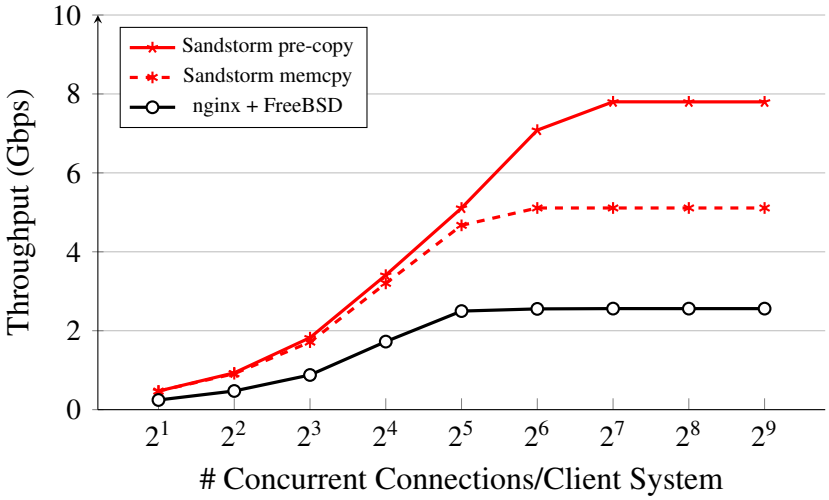


Figure 3.7: Network throughput, 1 NIC, ~23KB file, old hardware.

are closely correlated with their network throughput, indicating that DDIO is not helping on transmit: DMA comes from memory rather than the cache. The memcpy variant, however, has higher network throughput than memory throughput, indicating that DDIO is transmitting from the cache. Unfortunately, this is offset by much higher memory write throughput. Still, this only causes a small reduction in service throughput. Larger files no longer fit in the L3 cache, even with memcpy. Memory-read traffic starts to rise with files above 64KB. Despite this, performance remains high and CPU load decreases, indicating these systems are not limited by memory bandwidth for this workload.

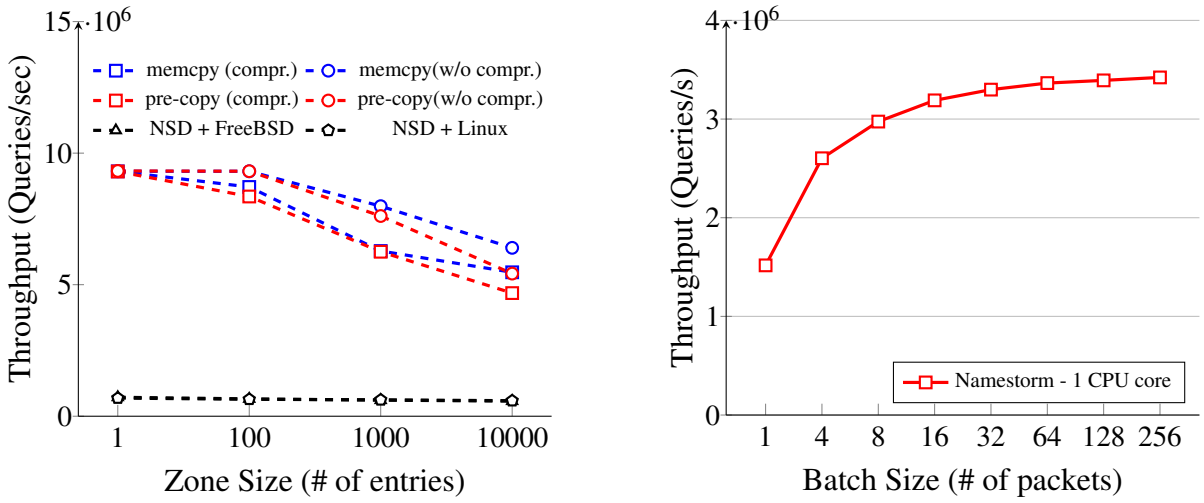
3.3.3 Experiment design: Namestorm

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

We ran Nominum's *dnsperf* [DNSP1+] DNS profiling software on the clients. We created zone files of varying sizes, loaded them onto the DNS servers, and configured dnsperf to query the zone repeatedly.

3.3.4 Namestorm results

Figure 3.8a shows the performance of Namestorm and NSD running on Linux and FreeBSD when using a single 10GbE NIC. Performance results of NSD are similar with both FreeBSD



(a) Throughput, 1x10GbE NIC

(b) Batching efficiency

Figure 3.8: Namestorm performance measurements.

and Linux. Neither operating system can saturate the 10GbE NIC, however, and both show some performance drop as the zone file grows. On Linux, NSD's performance drops by ~14% (from ~689,000 to ~590,000 Queries/sec) as the zone file grows from 1 to 10,000 entries, and on FreeBSD, it drops by ~20% (from ~720,000 to ~574,000 Qps). For these benchmarks, NSD saturates all CPU cores on both systems.

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

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

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

Finally, in Figure 3.8a we observe a noticeable performance overhead with the pre-copy stack, which we explore in Section 3.3.5.

3.3.4.1 Effectiveness of batching

One of the biggest performance benefits for Namestorm is that netmap provides an API that facilitates batching across the system-call interface. To explore the effects of batching, we configured a single Namestorm instance and one hardware queue, and reran our benchmark with varying batch sizes. Figure 3.8b illustrates the results: a more than 2× performance gain when growing the batch size from 1 packet (no batching) to 32 packets. Interestingly, the performance of a single-core Namestorm without any batching remains more than 2× better than NSD.

At a minimum, NSD has to make one system call to receive each request and one to send a response. Recently Linux added the new `recvmmsg` and `sendmmsg` system calls to receive and send multiple UDP messages with a single call. These may go some way to improving NSD's performance compared to Namestorm. They are, however, UDP-specific, and `sendmmsg` requires the application to manage its own transmit-queue batching. When we implemented Namestorm, we already had *libnmio*, which abstracts and handles all the batching interactions with netmap, so there is no application-specific batching code in Namestorm.

3.3.5 Data direct I/O

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

Intuitively, one might reasonably assume that Sandstorm's pre-copy mode might interact best with DDIO: as with `sendfile`-based designs, only packet headers enter the L1/L2 caches, with payload content rarely touched by the CPU. Figure 3.9 illustrates a therefore surprising effect when operating on small file sizes: overall memory throughput from the CPU package, as measured using performance counters situated on the DRAM-facing interface of the LLC, sees significantly less traffic for the memcpy implementation relative to the pre-copy one, which shows a constant rate roughly equal to network throughput.

This occurs because in the pre-copy cases the majority of the packet payloads (data) remain in the main memory: the CPU needs to fetch and modify only a single cache line per packet to

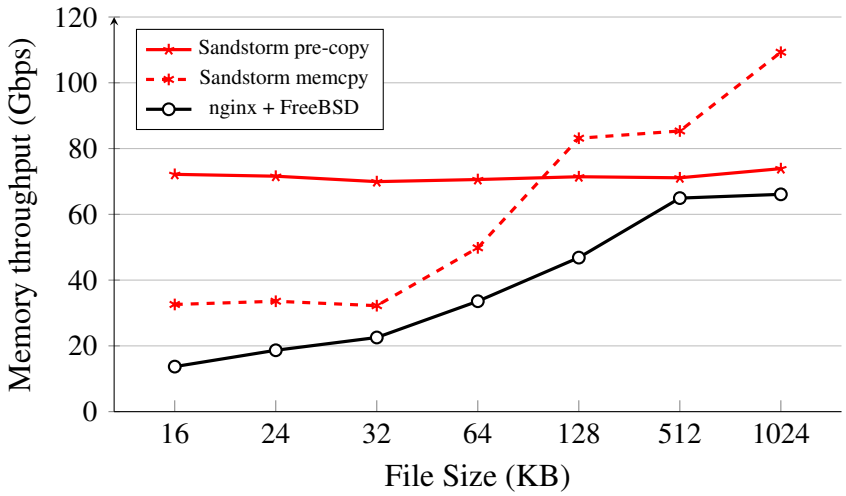


Figure 3.9: Sandstorm memory read throughput, 6NICs.

update the relevant packet headers, while the actual packet payload is fetched from the main memory with each DMA operation, without polluting the CPU cache[2]. This practically means that the per-packet processing cost remains constant, irrespective of the payload length. In the memcpy case, the CPU loads data into the cache, allowing more complete utilisation of the cache for network data. However, as the DRAM memory interface is not a bottleneck in the system as configured, the net result of the additional memcpy, despite better cache utilisation, is reduced performance. As file sizes increase, the overall footprint of memory copying rapidly exceeds the LLC size, exceeding network throughput, at which point pre-copy becomes more efficient. Likewise, one might mistakenly believe simply from inspection of CPU memory counters that nginx is somehow benefiting from this same effect: in fact, nginx is experiencing CPU saturation, and it is not until file size reaches 512K that sufficient CPU is available to converge with pre-copy's saturation of the network link.

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

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

[2]No cache allocation is happening, because the CPU did not load these data.

3.4 Discussion

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

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

3.4.1 Current network-stack specialisation

Over the years there have been many attempts to add specialised features to general-purpose stacks such as FreeBSD and Linux. Examples include `sendfile`, primarily for web servers, `recvmmsg`, mostly aimed at DNS servers, and assorted socket options for telnet. In some cases, entire applications have been moved to the kernel [Bar00; LEM00] because it was too difficult to achieve performance through the existing APIs. The problem with these enhancements is that each serves a narrow role, yet still must fit within a general OS architecture, and thus are constrained in what they can do. Special-purpose userspace stacks do not suffer from these constraints, and free the programmer to solve a narrow problem in an application-specific manner while still having the other advantages of a general-purpose OS stack.

3.4.2 The generality of specialisation

Our approach tightly integrates the network stack and application within a single process. This model, together with optimisations aimed at cache locality or pre-packetisation, naturally fit a reasonably wide range of performance-critical, event-driven applications such as web servers, key-value stores, RPC-based services and name servers. As we will show in Chapter 4, even rate-adaptive video streaming can benefit, as developments such as MPEG-DASH [DASH] and Apple's HLS [HLS] have moved intelligence to the client leaving servers as dumb static-content farms.

Not all network services, however, are a natural fit. For example, CGI-based web services and general-purpose databases have inherently different properties and are generally CPU- or

filesystem-intensive, de-emphasising networking bottlenecks. In our design, the control loop and transport-protocol correctness depend on the timely execution of application-layer functions; blocking in the application cannot be tolerated. A thread-based approach might be more suitable for such cases. Isolating the network stack and application into different threads still yields benefits: OS-bypass networking costs less, and saved CPU cycles are available for the application. However, such an approach requires synchronisation, and so increases complexity and offers less room for cross-layer optimisation.

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

3.4.3 Tracing, profiling, and measurement

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

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

Worse, as we attempted inter-OS analysis (e.g., comparing Linux and FreeBSD lock profiling), we discovered that tools often measure and report results differently, preventing sensible comparison. For example, we found that Linux took packet timestamps at different points than FreeBSD, FreeBSD uses different clocks for DTrace and BPF, and that while FreeBSD exports both per-process and per-core PMC stats, Linux supports only the former. Where supported, DTrace attempts to bridge these gaps by unifying configuration, trace formats, and event namespaces [CSL04]. However, DTrace also experiences high overhead causing bespoke tools to persist, and is unintegrated with packet-level tools preventing side-by-side comparison of packet and execution traces. We feel certain that improvement in the state-of-the-art would benefit not only research, but also the practice of network-stack implementation.

Our special-purpose stacks are synchronous; after netmap hands off packets to userspace, the control flow is generally linear, and we process packets to completion. This, combined with lock-free design, means that it is very simple to reason about where time goes when handling a request flow. General-purpose stacks cannot, by their nature, be synchronous. They must be asynchronous to balance all the conflicting demands of hardware and applications, managing

queues without application knowledge, allocating CPU cycles to threads in order to handle those queues, and ensuring safety via locking. To reason about performance in such systems, we often resort to statistical sampling because it is not possible to directly follow the control flow. Of course, not all network applications are well suited to synchronous models; we argue, however, that imposing the asynchrony of a general-purpose stack on all applications can unnecessarily complicate debugging, performance analysis, and performance optimisation.

3.5 Conclusions

In this chapter, we have demonstrated that specialised userspace stacks, built on top of netmap framework, can vastly improve the performance of scale-out applications. These performance gains sacrifice generality by adopting design principles at odds with contemporary stack design: application-specific cross-layer cost amortisations, synchronous and buffering-free protocol implementations, and an extreme focus on interactions between processors, caches, and NICs. This approach reflects a widespread adoption of scale-out computing in data centers, which de-emphasises multi-function hosts in favour of increased large-scale specialisation. Our performance results are compelling: a 2–10$\times$ improvement for web service, and a roughly 9$\times$ improvement for DNS service. Furthermore, these stacks have proven easier to develop and tune than conventional stacks, and their performance improvements are portable over multiple generations of hardware.

General-purpose operating system stacks have been around a long time, and have demonstrated the ability to transcend multiple generations of hardware. We believe the same should be true of special-purpose stacks, but 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.

Chapter 4

Network and storage stack specialisation

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 amortise system overheads by utilising efficient batching of outstanding I/O requests, process-to-completion, and zero-copy 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. Minimising 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 bottlenecking 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.

4.1 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?

In Chapter 3 we have shown that a specialised 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 user application and OS to accept new connections; our solution achieved high performance by using a zero-copy architecture closely coupling the HTTP server and the TCP/IP stack in userspace, using netmap's [Riz12] 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 [DDIO] 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 packetise 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 6,000 and 16,000 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.

4.2 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 [HLS], Adobe HTTP Dynamic Streaming [HDS], and MPEG-DASH [DASH]. 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 optimised. 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 [BBC] 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?

4.2.1 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 [GIPR+]. 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 [NFLX+] 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 optimise 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.

4.2.1.1 Asynchronous sendfile

The `sendfile` system call optimises 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 catalogue 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 in-flight 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.

4.2.1.2 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 (nginx processes would sleep in `vm_wait` instead of actually serving traffic).

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. This practically means that the pageout codepath is actually run in the context of a single nginx process, allowing the rest of the processes to continue serving traffic. Finally, reclaimed pages are re-enqueued to the inactive memory queues in batches to amortise the lock overhead.

4.2.1.3 RSS-assisted TCP LRO

Large Receive Offload (LRO) is a common technique used to amortise 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. By default, the FreeBSD LRO implementation could manage up to 8 packet aggregations at one time; this works well on a local network while serving a few fast TCP connections, but with thousands of TCP connections, packets belonging to the same session are likely to arrive interleaved with many other packets, making LRO substantially 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 (i.e. the original time of arrival). 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 optimisation helped reduce CPU utilisation by ~5-30%, depending on the congestion control algorithm, and the interrupt coalescing tuning parameters (LRO aggregation rate improved from ~1.1 to more than 2 packets per aggregation).

4.2.1.4 In-kernel TLS

End-to-end encryption introduces a new challenge in building high-performance network services. Suddenly, optimised zero-copy 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 zero-copy 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 [SGL15].

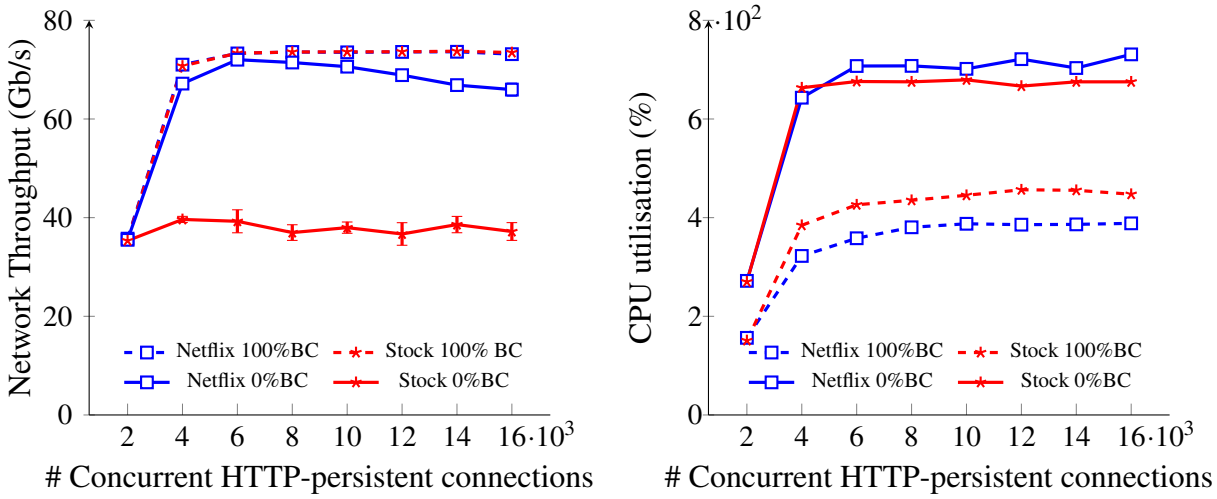
To regain the advantages of `sendfile` for encrypted traffic, Netflix devised a hybrid approach to split work between kernel and userspace: the TLS session management, and negotiation remains in the userspace TLS library (openssl), but the kernel is now modified to include bulk data encryption in the `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.

4.2.2 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 §4.4. We stress-tested the system using HTTP persistent connections retrieving 300KB video chunks as we increase the number of simultaneous active clients.

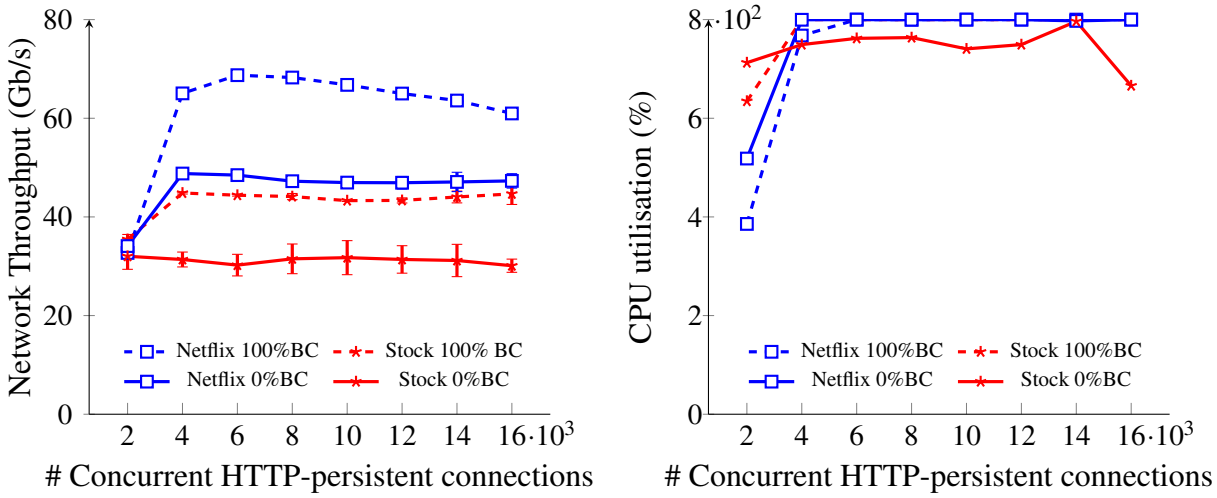
Figure 4.1 shows throughput (left) and CPU utilisation (right) as the number of concurrent HTTP persistent connections is varied. We show curves for the Netflix stack and for the stock nginx/FreeBSD stack for both the case where all content is served from the disk buffer cache (100% BC) and where all content needs to be fetched from the SSDs (0% BC). Normal Netflix workloads are nearer the latter than the former.



(a) Network throughput (Error bars indicate the 95% CI)

(b) CPU utilisation

Figure 4.1: Plaintext performance, Netflix vs Stock FreeBSD, zero and 100% Buffer Cache (BC) ratios.



(a) Network throughput (Error bars indicate the 95% CI)

(b) CPU utilisation

Figure 4.2: Encrypted performance, Netflix vs Stock FreeBSD, zero and 100% Buffer Cache (BC) ratios.

When all content is served from the buffer cache, there is no significant difference in performance, either in throughput or CPU utilisation. 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 4.2 shows the equivalent throughput and CPU utilisation 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

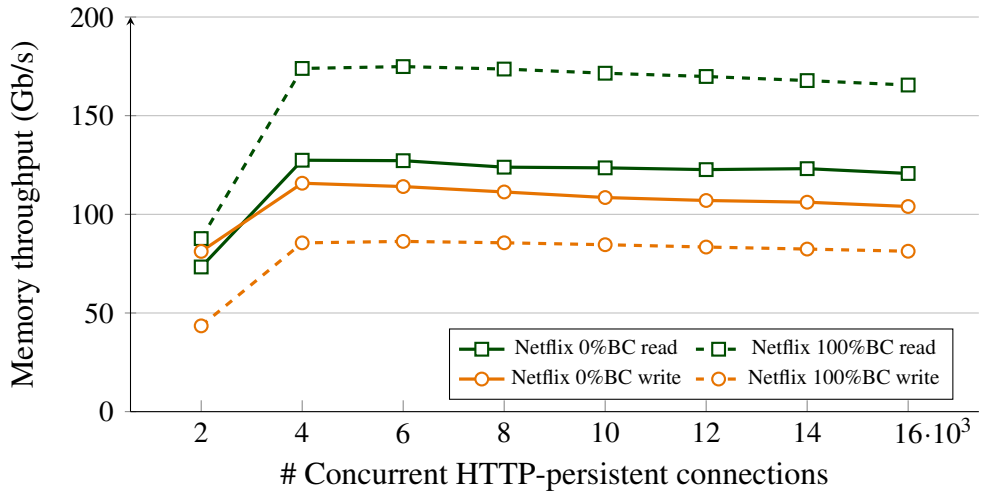


Figure 4.3: Encrypted Workload, Netflix memory performance.

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? The reader may intuitively think that because encryption is a CPU-intensive task, the CPU cores fail to keep up with the I/O data rates. This, however, is far from true in this case; 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 and we are not bottlenecked on memory bandwidth. Our microbenchmarks indicate that the attainable bulk encryption throughput while using AES-NI acceleration on our CPU (AES-GCM128) is a total maximum of up to ~205Gb/s, or ~25.5Gb/s per core when the buffer sizes are such that can fit in the LLC (no extra memory traffic). Clearly such throughput is more than sufficient to easily keep up with the achievable I/O rates of the storage and networking hardware, yet there is a huge discrepancy between this, and the previously achieved 47Gb/s. Why is this happening? 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[1] is low, 20Gb/s, but *read* memory throughput[2] 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 workload with ~72Gb/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

[1] *Write* memory throughput corresponds to the amount of data transferred from the CPU package to DRAM at the unit of time.

[2] *Read* memory throughput corresponds to the amount of data transferred from DRAM to the CPU package at the unit of time.

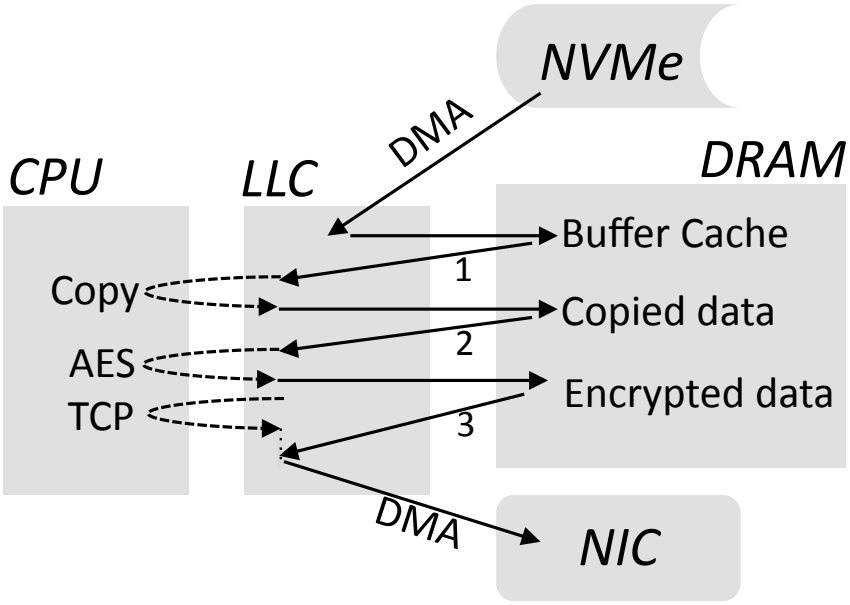


Figure 4.4: Possible memory accesses with the Netflix stack.

throughput is shown in Figure 4.3. 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 175Gb/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 utilisation is 800% (all eight cores are saturated).

4.2.3 Discussion

Netflix optimisations 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,

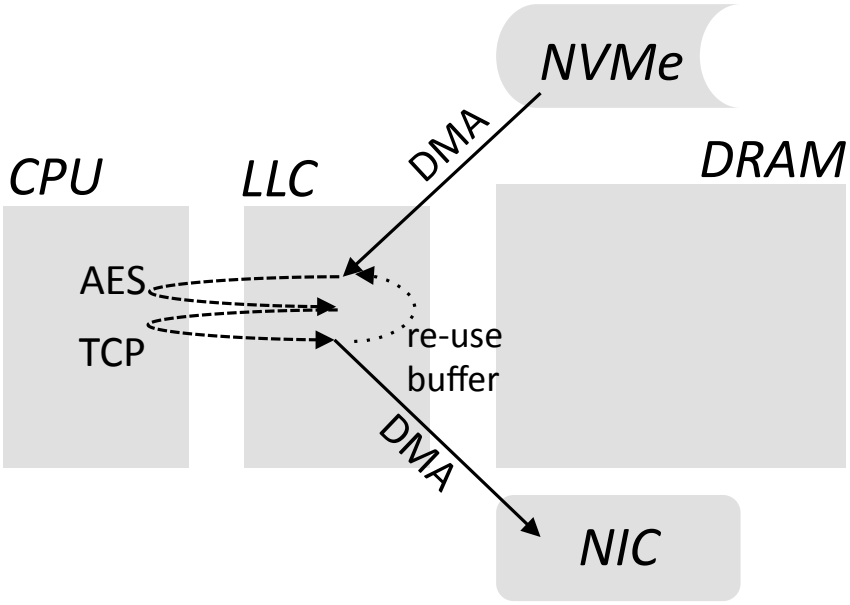


Figure 4.5: Desired memory accesses with a specialised stack.

so it gets flushed a third time. Finally it is DMAed to the NIC, requiring it to be loaded again. This process is shown in Figure 4.4, 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 4.3. At this utilisation, CPU stalls waiting for memory become a major problem, and CPU utilisation varies considerably, ranging from 75% to 90% with only small changes in demand. When operating close to the memory bandwidth saturation threshold, CPU performance is erratic; a linear increase in demand can lead to a non-linear increase in CPU utilisation.

4.3 Towards a specialised 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 specialised, 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

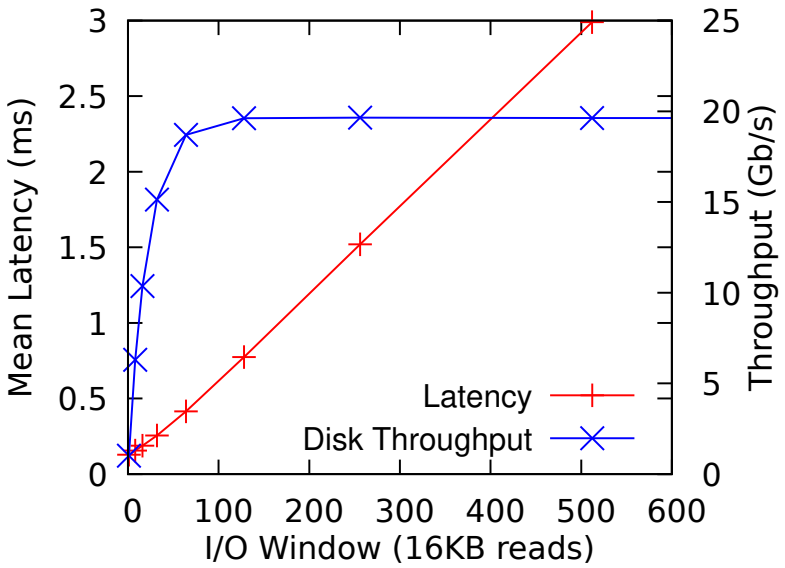


Figure 4.6: NVMe controller latency and throughput.

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 4.5.

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 4.6 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.

4.3.1 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 20μs for short transfers [p3608]. 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.

4.3.1.1 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.

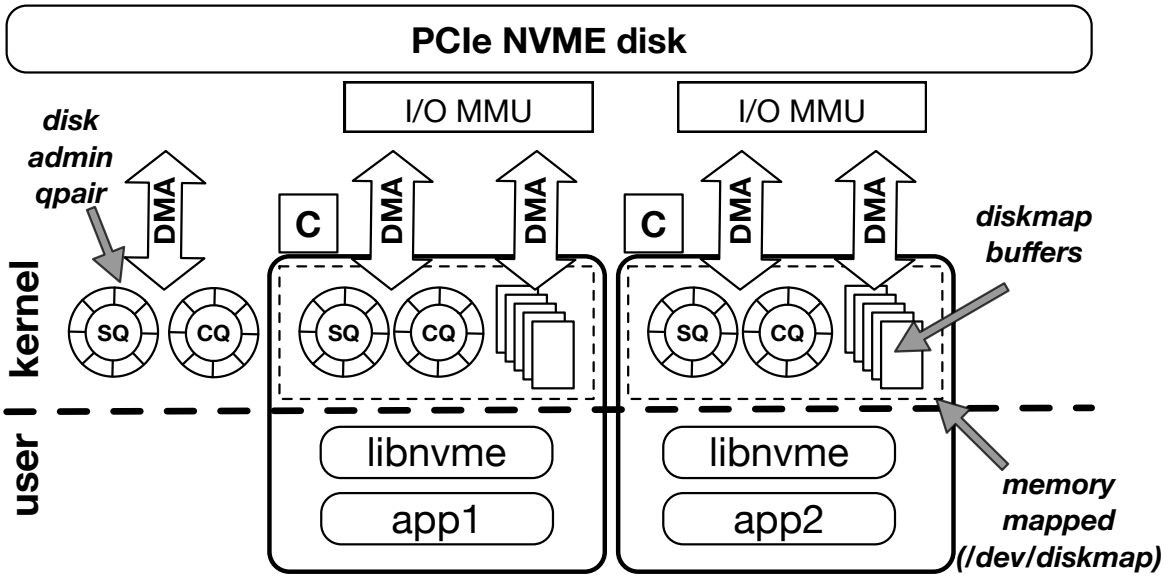


Figure 4.7: High-level architecture of diskmap applications.

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.

4.3.1.2 Diskmap

Inspired by *netmap* [Riz12], 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, enqueuing, and consuming I/O requests and completions, while the OS uses hardware capabilities to enforce protection and synchronisation. The system is comprised of two parts:

- A kernel module used to initialise and configure devices that are to be used in diskmap-mode, as well as providing a thin syscall layer to abstract DMA operation initiation,
- An accompanying userspace library which implements the NVMe driver and provides the API (see table 4.1) to abstract typical operations to the device such as read and write.

The architecture is shown in Figure 4.7. When the diskmap kernel module loads, the NVMe device is partially detached from the in-kernel storage stack: the actual datapath queue pairs are

Function	Parameters	Description
`nvme_open()`	I/O qpair control block, character device, buffer memory size, flags	Configure, initialize, and attach to NVMe disk's queue pair.
`nvme_read()`	I/O description block (`struct iodesc`), metadata, error	Craft and enqueue a *READ* I/O request for a particular disk, namespace, starting offset, length, destination address etc.
`nvme_write()`	I/O description block (`struct iodesc`), metadata, error	Craft and enqueue a *WRITE* I/O request for a particular disk, namespace, starting offset, length, source buffer etc.
`nvme_sqsync()`	I/O qpair control block	Update the NVMe device's Submission Queue doorbell (via a dedicated `ioctl`) to initiate processing of pending I/O requests.
`nvme_cqsync()`	I/O qpair control block	Update the NVMe device's Completion Queue doorbell (via a dedicated `ioctl`) to communicate to the disk firmware the consumption of completion notifications.
`nvme_consume_completions()`	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 `struct iodesc`).
`nvme_close()`	I/O qpair control block	Unregister the particular I/O queue pair, and release resources.

Table 4.1: `libnvme` API functions (not exhaustive).

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 initialisation, 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 [NVME], 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 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 libnvme library call (see table 4.1). When an application calls `nvme_read` or `nvme_write`, the library is responsible for translating the I/O request for certain disk blocks into the corresponding NVMe commands and enqueuing 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, libnvme 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 (see Listing 4.1). 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 completions. Libnvme 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 initialisation, there is no need to dynamically update the IOMMU page table with transient mappings and unmappings, which would otherwise significantly affect performance [ABT+11; MMT16]. 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

```
int read_done(char *buf, uint32_t lba, uint32_t lba_count,
     void *arg, int err);

int
read_done(char *buf, uint32_t lba, uint32_t lba_count,
     void *arg, int err) {
     if (err != E_NOERR) /* Do something. */
          return err;

     /* "buf" holds the newly read data. */
     return E_NOERR;
}

int
main() {
     /* Use /dev/nvme0, Submission/Completion Queue Pair 1. */
     nvme_open(..., &ioqp, "diskmap:nvme0-1", NBUFS_SZ, 0);
     iodesc.ioqp = &ioqp; /* "iodesc" is the "aiocb" equivalent. */
     iodesc.ns_id = NSID; /* Namespace ID. */
     iodesc.op_complete = read_done; /* Callback function. */
     iodesc.flags = 0;
     iodesc.lba_count = IO_SIZE_IN_SECTORS;

     for (;;) {
          /*
           * Generate random starting offset. Ideally this should be
           * pre-calculated outside this main event loop to avoid overhead.
           */
          iodesc.lba = random() %
               (media_size_in_sectors - IO_SIZE_IN_SECTORS);

          /* Enqueue READ operation. */
          nvme_read(&iodesc, NULL, &ret);
          if (ret != E_NOERR)
               break; /* Errors should be handled properly. */

          /* Update Submission Queue doorbell, if needed. */
          nvme_sqsync(&ioqp);

          /* Poll Completion Queue. */
          ncpls = nvme_peek_completions(&ioqp);

          /* The CB function will be invoked once per completion. */
          nvme_consume_completions(&ioqp, ncpls);

          /* Update Completion Queue doorbell, if needed. */
          nvme_cqsync(&ioqp);
     }

     [...]
}
```

Listing 4.1: Pseudocode for random **READ** operations using diskmap.

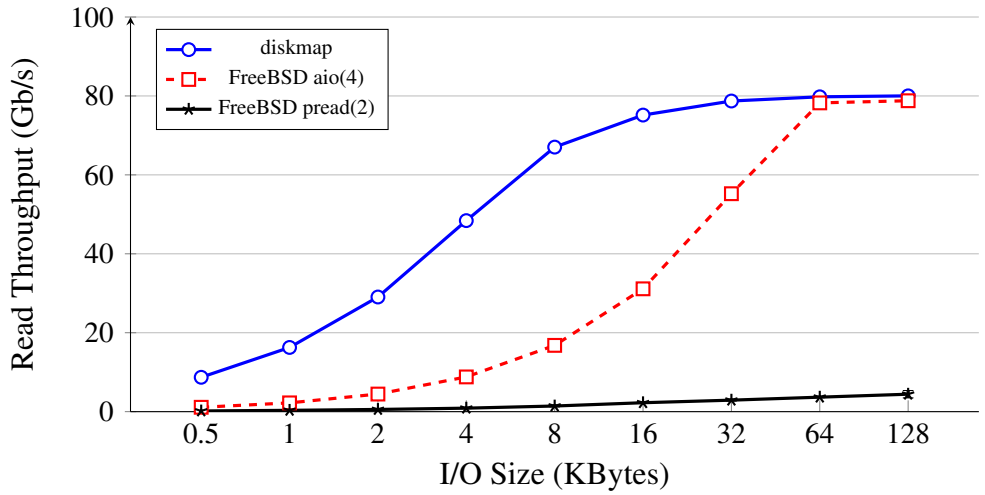


Figure 4.8: Read throughput, *diskmap* vs. *aio(4)* vs. *pread(2)*

either in the libnvme library or the application itself. Unlike SPDK [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.

4.3.1.3 Diskmap performance

Before we integrate diskmap into our video server, we wish to understand how well it performs. Figure 4.8 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 [Lem01] and kevent to allow asynchronous I/O requests to be initiated by userspace. It is highly optimised, 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.

Although aio performs well for large reads, it is less than 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 (10*MSS [CDC+13]). This makes diskmap an excellent fit for our video server.

Figure 4.9 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 4.6 was gathered using diskmap, and shows that both low latency and high throughput can be obtained simultaneously.

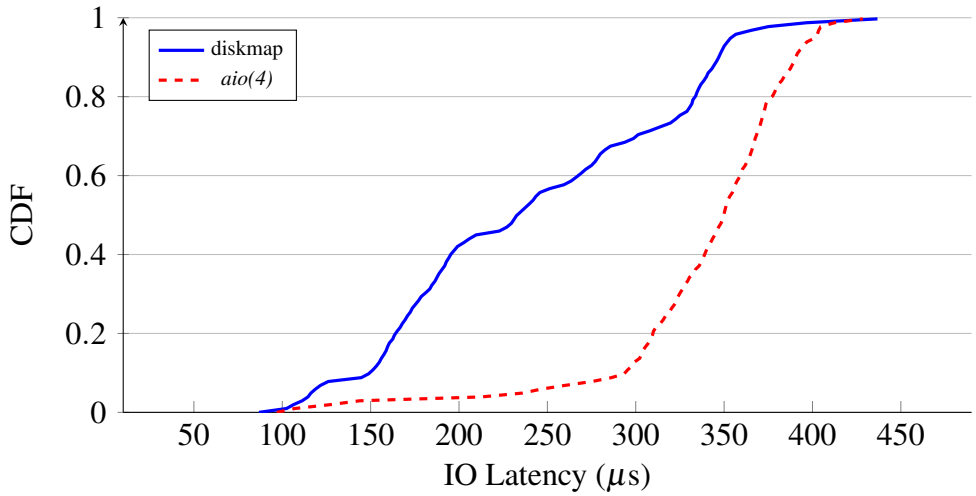


Figure 4.9: *diskmap* vs. *aio(4)* - I/O latency, read size: 512 bytes, I/O window: 128 requests.

4.3.1.4 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 [CHLNT]. For the CPU to keep up, batching is required to amortise 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 amortising the system-call overhead.

4.3.2 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, minimising 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) [DDIO].

To take advantage of diskmap in the network fast path, we need to embed the networking code in the same process thread. Several userspace network stacks have been presented recently,

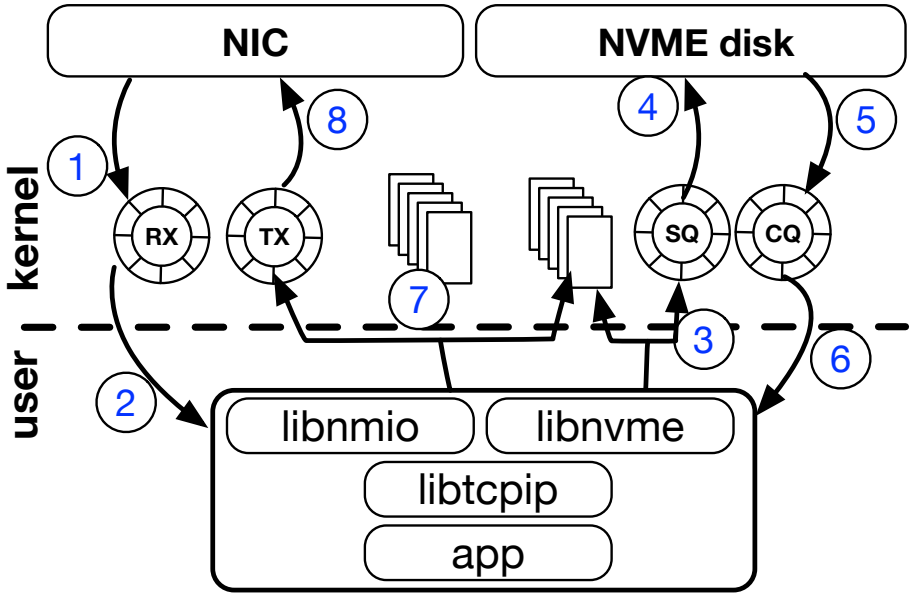


Figure 4.10: Atlas high-level control flow.

demonstrating dramatic improvements both in terms of latency and throughput [HHR+14; JWJ+14; MWH14]. We started from the Sandstorm implementation (presented in Chapter 3), 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 optimise 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 optimisations 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 catalogue size is significantly larger than RAM, a video streamer needs to serve ~90% of requests from disk in the typical case (see §4.2.1). 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. 4.10). 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 optimise 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*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 [SMC08] 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.

4.4 Evaluation

We saw in Section 4.2.2 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-optimised 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 optimisation patches, including those mentioned in §4.2.1, 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 in the order of a few microseconds. This is not representative of the RTTs seen by production video servers, and would distort TCP behaviour. To emulate more realistic latencies, we have implemented a middlebox which adds latency to traffic going from the clients 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. Figure 4.11 illustrates the high-level operation of the software middlebox. We rely on netmap and zero-copy I/O to achieve this solution in software with off-the-shelf hardware. The middlebox constantly polls the receive queues, and each newly received packet is hashed and buffered, and a per-flow constant delay is added before the packet is forwarded on. By enforcing constant delay to all packets belonging to the same TCP flow, we avoid introducing any TCP re-ordering effects which would hinder the experiments. The middlebox implementation adopts an event-driven, share-nothing approach, and scales linearly to multiple CPU cores (by pinning a process per core), provided NIC hardware support for multiple RX queues and RSS [SLNS].

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 *weighttp* to generate HTTP-persistent traffic with multiple concurrent clients[3]. 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

[3]This tool has been modified to support requesting multiple URLs.

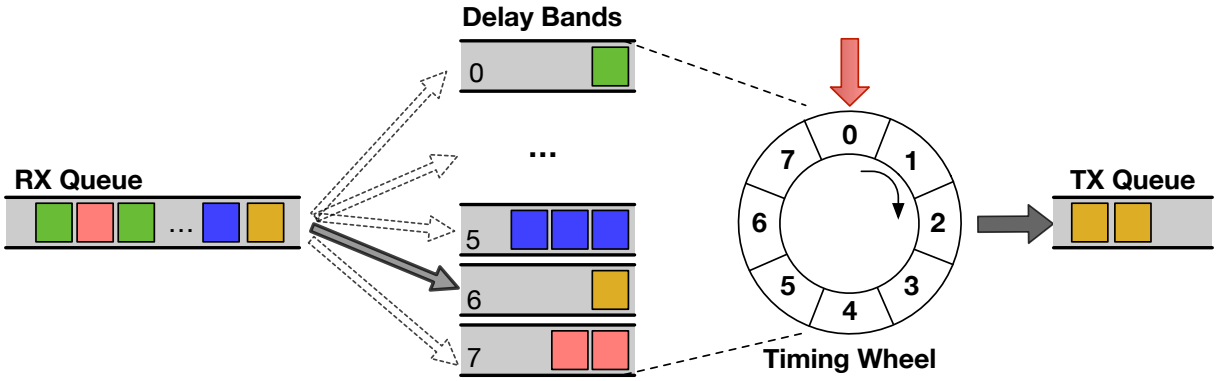


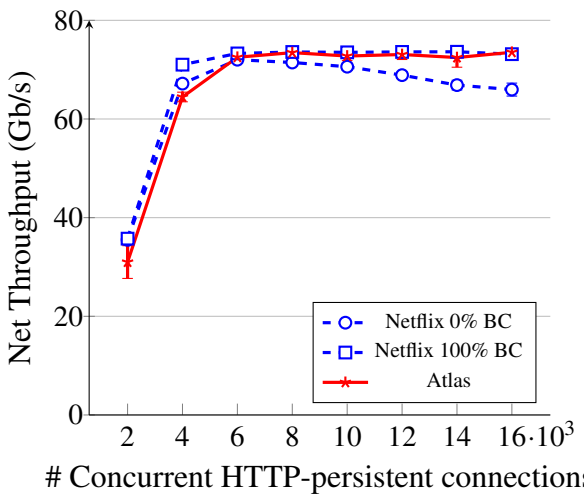
Figure 4.11: High-level view of the software middlebox architecture. Packets are distributed to buckets, each corresponding to a different target RTT. A Timing Wheel data structure [VL87] is used to facilitate multiple timers at O(1) time. On each clock tick, all packets buffered to the relevant bucket are being transmitted.

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

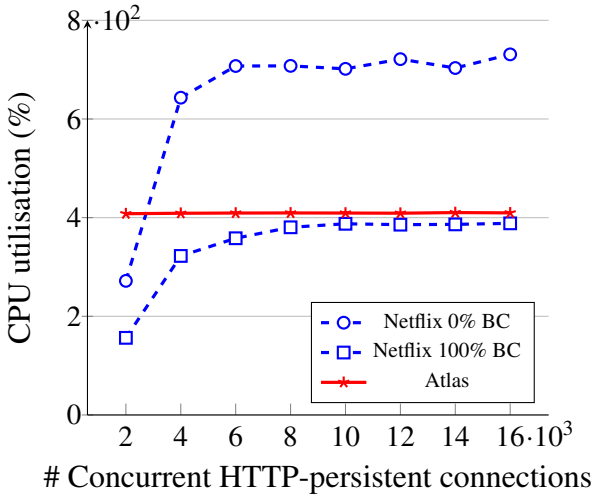
4.4.1 Plaintext HTTP-persistent traffic

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 utilise a buffer cache: all data requests, even repetitive ones, are served from the disk, so Atlas is not sensitive to the choice of workload.

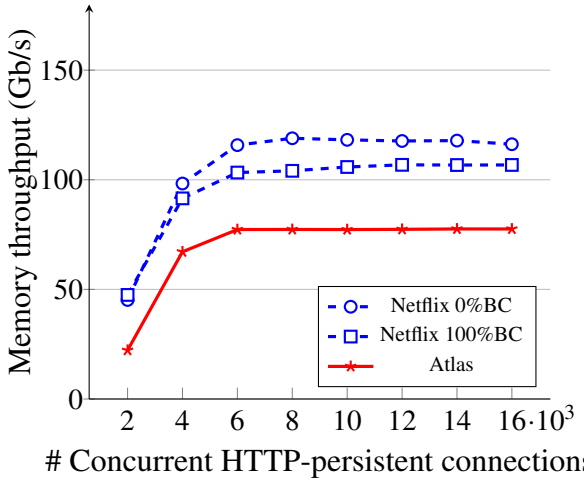
Figures 4.12a and 4.12b show the network throughput and CPU utilisation achieved by both systems. Atlas and the Netflix setup with a maximally effective buffer cache (100% BC) manage to saturate both the 40GbE NICs, achieving roughly ~73Gbps of HTTP throughput with higher numbers of concurrent connections. 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.



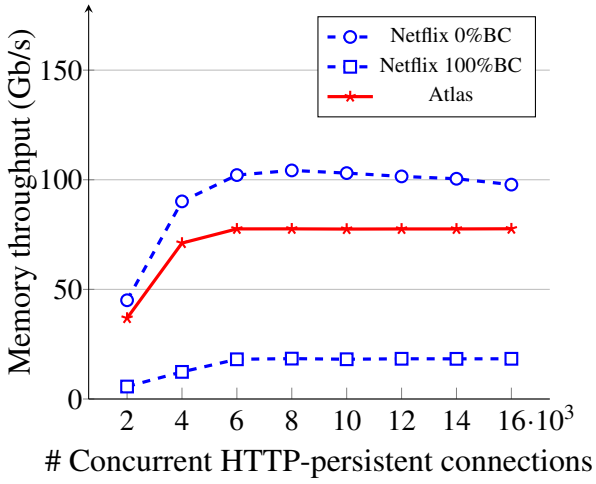
(a) Network throughput (Error bars indicate the 95% CI)



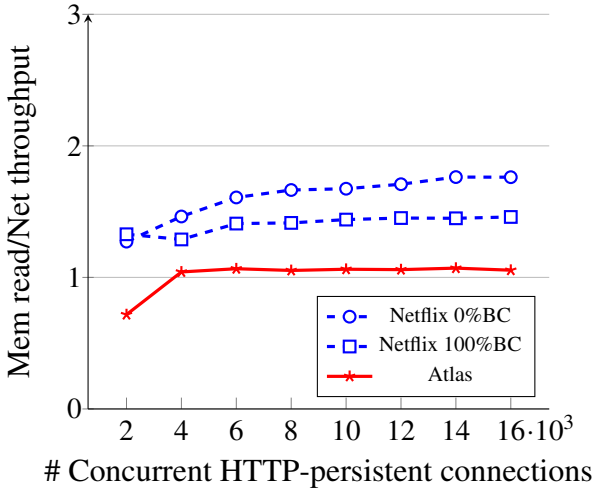
(b) CPU utilisation (Average)



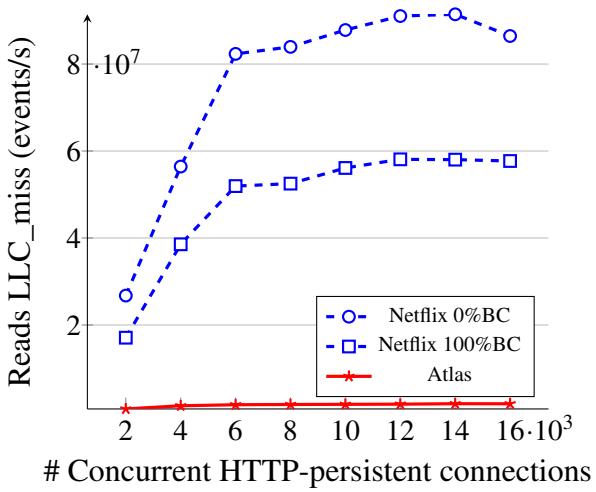
(c) Memory READ



(d) Memory WRITE



(e) Mem. READ-Network Throughput Ratio



(f) CPU reads served from DRAM due to LLC misses

Figure 4.12: Plaintext performance, Netflix vs. Atlas, zero and 100% Buffer Cache (BC) ratios.

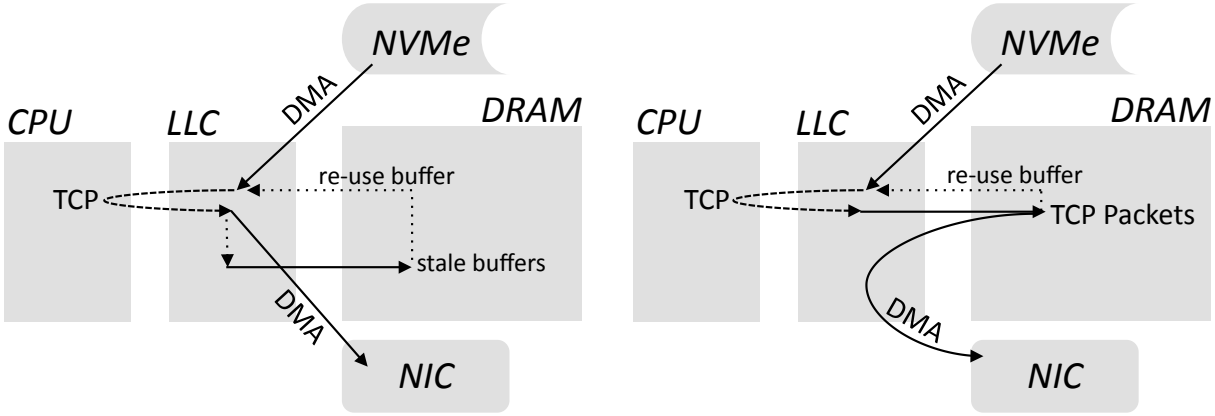
With the uncacheable workload (0% BC), we observe that Netflix experiences a small performance drop as the number of connections increases. Although VM subsystem pressure is handled by the Netflix configuration much more gracefully than stock FreeBSD, with more connections requesting new data from the 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 catalogue 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 uncacheable workloads.

It is interesting to observe the CPU utilisation of the Netflix setup for the two different workloads. The CPU utilisation almost doubles when the buffer cache is thrashed and disk activity is required. It should be noted that the CPU utilisation 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 utilisation 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. 4.12e), 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 4.13a. 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 utilise 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



(a) Delayed notifications incur extra memory writes.

(b) Heavy load and netmap latency result in LLC evictions.

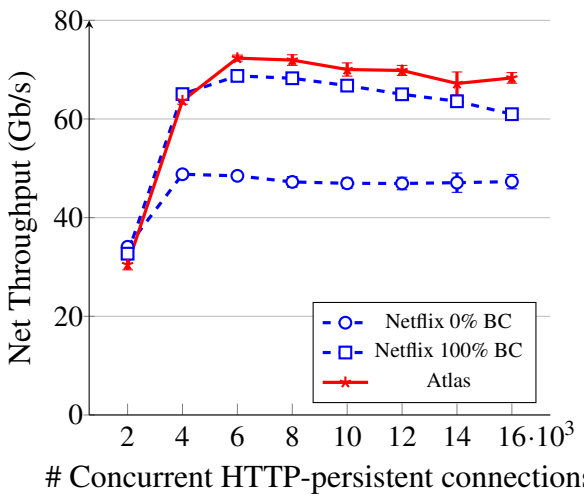
Figure 4.13: Principal sub-optimal Atlas memory access patterns for unencrypted traffic.

I/O commands and connections at each instant. At this point the storage and NIC 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 4.13b. In any case, when we look at LLC misses (Figure 4.12f), 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.

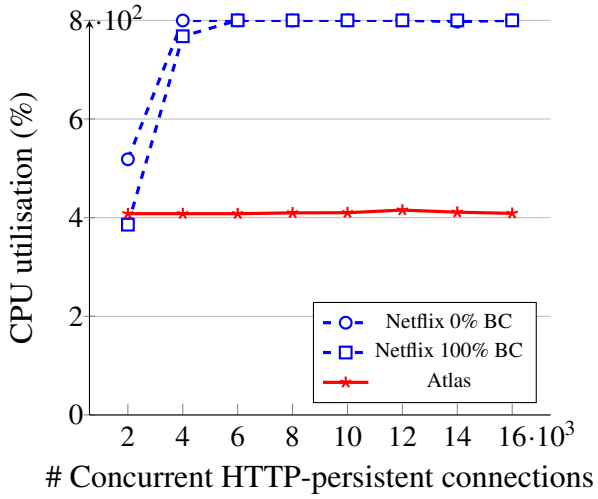
4.4.2 Encrypted HTTP-persistent traffic

The need to encrypt traffic significantly complicates the process of serving content. We expect high performance from Netflix' setup, since unlike stock FreeBSD, it can still take advantage of `sendfile` with the in-kernel TLS implementation (§4.2.1). 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.

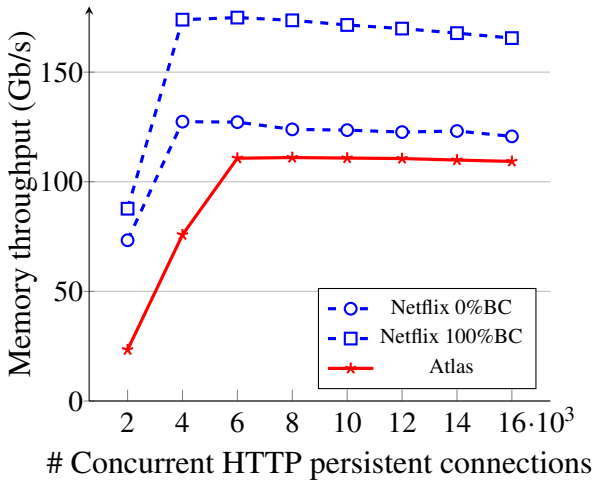
To avoid our tests being impacted by CPU saturation on our client systems, rather than implementing a full TLS layer we have decided to emulate the TLS overhead by doing encryption and authentication of the data with dummy keys before it is actually transmitted. The HTTP headers are still transmitted 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.



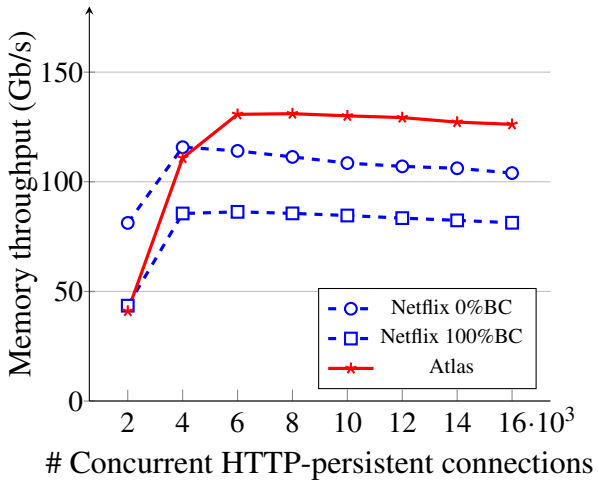
(a) Network throughput (Error bars indicate the 95% CI)



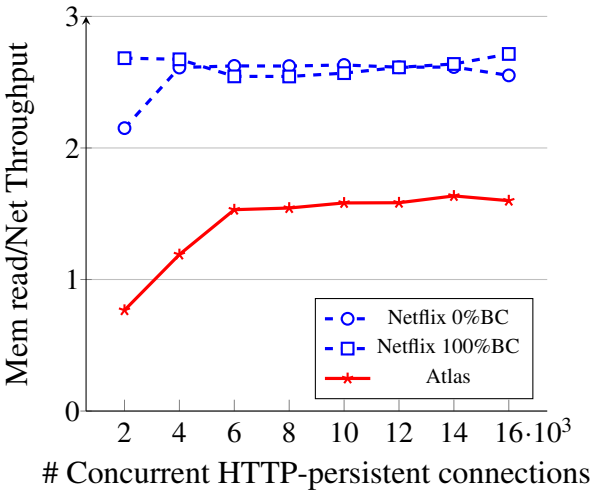
(b) CPU utilisation (Average)



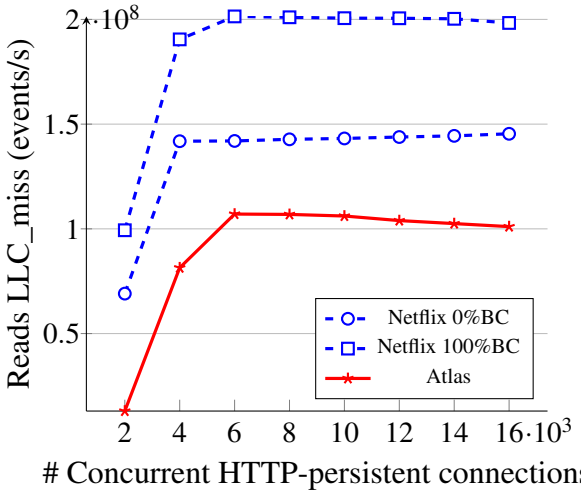
(c) Memory READ



(d) Memory WRITE



(e) Memory READ-Network Throughput Ratio



(f) CPU reads served from DRAM due to LLC misses

Figure 4.14: Encrypted performance, Netflix vs. Atlas, zero and 100% Buffer Cache (BC) ratios.

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 [SMC08] (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 optimised software-only implementation: Intel's ISA-L library, which not only uses ISA extensions to accelerate crypto, but also utilises non-temporal instructions to reduce pressure on the CPU's Last Level Cache.

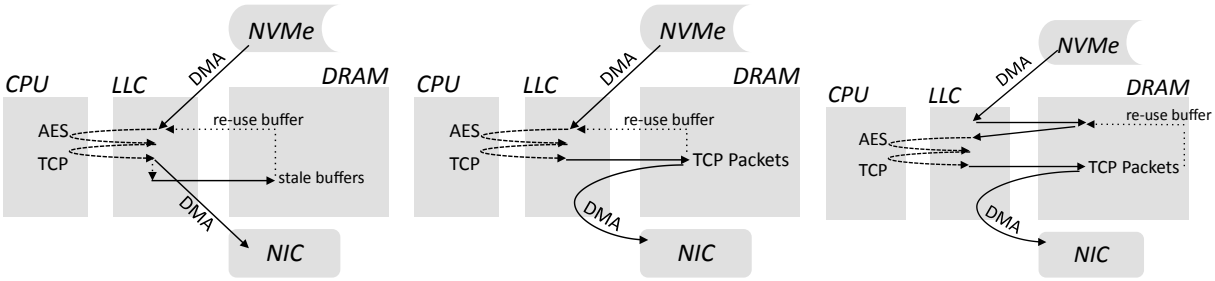
Figures 4.14a and 4.14b show network throughput and CPU utilisation while serving encrypted traffic with a zero and 100% buffer cache hit ratios. When serving more than 4,000 connections, Atlas achieves higher throughput than Netflix when the buffer hit ratio is 100%, ~72Gb/s as opposed to ~68Gb/s peak 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 suboptimal 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 4.14c and 4.14d). 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. 4.14e, 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× the network throughput as opposed to 2.6× 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



(a) Delayed notifications incur extra memory writes. No memory reads, but extra writes.

(b) Heavy load and netmap batching result in LLC eviction. One extra memory read and write.

(c) Contention for DDIO portion of LLC evicts DMA'ed data. Two extra reads and writes.

Figure 4.15: Principal sub-optimal Atlas memory access patterns for encrypted traffic.

in Fig. 4.15a dominates. We believe that the increased memory write throughput observed for Atlas in Figure 4.14d 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. 4.15b 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. 4.15c prior to encryption is responsible for the LLC misses in Fig. 4.14f, 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.

4.5 New design principles

We developed diskmap and the clean-slate Atlas stack to explore the boundaries of achievable performance through a blend of software specialisation and microarchitectural awareness. The resulting prototype exhibits significant performance improvements over conventional designs. However, and perhaps counter intuitively, 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 [PGG+95]. 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.

Optimising 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 increasingly popular design choice made to better utilise 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 optimisation 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 minimise 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 maximise batching and asynchrony in order to mitigate system-call expense. Unlike netmap, diskmap facilitates latency minimisation by allowing user code to have fine-grained notification of memory being returned for reuse, and by minimising in-kernel work loops that otherwise increase LLC utilisation. 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 behaviour 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 behaviours: 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 de-emphasises DRAM use in favour 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 utilisation 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.

4.6 Conclusions

In this Chapter 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 specialised 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.

Chapter 5

Future work and conclusions

This research is only a small step towards improving systems I/O performance. In the remainder of this chapter, I discuss possible future directions for this work, and finally I conclude this dissertation.

5.1 Improvements to netmap and Atlas

As I have shown in §4.4, although Atlas makes highly effective use of memory bandwidth compared to the Netflix stack, it still requires detours to the main memory to serve heavy workloads. I believe it would be feasible to achieve much higher throughputs without the need for extra memory traffic, if we address some issues with netmap that introduce latency. Chapter 2 suggests that it should also be possible to fully serve such workloads fully from the CPU's last level cache. Unfortunately, netmap mostly focusses on batching and asynchrony to amortise the system call overhead, and it does not explicitly optimise for LLC and I/O interactions. It is worth noting that one of the primary goals of netmap design was to achieve tight integration with the conventional OS, by supporting primitives such as `poll`, `select`, and `kqueue` for readiness notification: this is convenient and helps avoiding 'burning' cores (i.e., busy waiting) when polling the NIC RX rings, for example, but it introduces some asynchrony. A conceptually similar drawback to the netmap design is the lack of a mechanism to communicate DMA completion events to userspace in a timely manner: this practically means that Atlas (and other applications) cannot really reclaim DMA memory fast enough, resulting in unnecessary higher memory usage and evictions. I believe that applying some of the diskmap design principles to netmap would address these issues. Another source of asynchrony with netmap is the delay introduced to achieve batching. Batching in netmap is required mostly due to the following reasons: *(i)* to amortise the system call overhead, and *(ii)* to amortise the costs associated with the update of the NIC doorbell registers for DMA operations. Batching is probably required with very small packets at line rate, since we cannot afford to update the NIC registers for every packet, but with larger packets (including TSO segments) batching is mostly effective in reduc-

ing the absolute number of system calls. Exploring the DPDK approach [DPDK] which allows DMA initiation directly from userspace without a system call, would probably be beneficial by reducing the delay between I/O completion and initiation events (e.g., a DMA write from the disk completes, and a DMA read from the NIC starts).

5.2 Towards terabit/s host-side system processing

What is the right software architecture, and abstractions that would enable host-side network packet processing at rates of terabits/s? A 100GbE adapter can continuously place small packets to the CPU LLC almost at a ~5ns interval, while the CPU access time to the same cache is currently in the range of ~10-15ns. Indeed, the networking hardware is getting faster and faster; over the last decade we have experienced a 10× increase in bandwidth. The situation is no longer the same for CPU core performance though: instruction-level parallelism has hit a plateau, while clock speed cannot increase anymore, due to supply voltage, and heat dissipation issues. How can we bridge this gap? Clearly, the most straightforward solution is scaling processing to multiple cores, while at the same time minimising software complexity to achieve a tight processing pipeline, and minimise the CPU pipeline stalls due to memory. Possibly, we will also need to explore the potential of offloading certain operations to the hardware: for example, if CPU cycles is the new bottleneck in scaling performance, encryption could probably be offloaded to the NIC.

5.3 Conclusions

In this dissertation, I have described the architecture and design principles of specialised network and storage stacks, aiming to push I/O performance with conventional off-the-shelf hardware.

- In **Chapter 2**, I traced and discussed the evolution in hardware capabilities and conventional Operating System design. Through a set of microbenchmarks, I illustrated how much of the hardware potential is wasted due to shortcomings in conventional OSes. Finally, I surveyed background and related work for the areas covered in this dissertation.
- In **Chapter 3**, I presented Sandstorm and Namestorm, specialised userspace network stacks that optimise performance for web and DNS workloads. With this design, application and network stack memory models are merged into the same address space, creating ample opportunities for aggressive cross-layer optimisations, process-to-completion (reduced asynchrony) which is crucial for efficient hardware operation, and other techniques to amortise the system costs. Sandstorm achieves 2-10× throughput improvement in web traffic, and Namestorm 9× throughput improvement in DNS traffic respectively when compared to conventional configurations.

- In **Chapter 4**, I presented Atlas, a specialised video streaming stack. Atlas is designed to serve video streaming workloads, such as the ones experienced by Netflix. It leverages NVMe disks' capabilities to include the persistent storage directly in the network fast path, without the use of a traditional in-memory buffer cache. By tightly integrating the network and storage I/O control loop, higher performance was achieved; this is not only due to more efficient code expression with respect to processor microarchitecture, but rather due to achieving more efficient I/O data flow throughout the hardware. Atlas achieves ~50% throughput improvement with encrypted content, and simultaneously almost 50% reduction in memory read traffic when compared to the state-of-the-art Netflix video streaming stack.

Collectively, these chapters serve to validate the hypothesis I stated in Chapter 1. With my proof-of-concept stacks and thorough evaluation of conventional stacks, I have demonstrated that *(i)* general-purpose OSes fail to fully take advantage of modern hardware capabilities while serving common Internet workloads, and *(ii)* specialised stacks can achieve efficient data movement throughout the hardware, eliminating this way historical bottlenecks in scaling I/O performance.

Would it be possible to achieve similar specialisation and microarchitectural awareness with general-purpose operating systems? It is our belief that the design principles of the specialised stacks presented in this dissertation, are *generalised* enough to be applicable to a wide range of applications that are performance-sensitive. Historically, there has been a heated debate whether all this specialisation could actually be applied back to general-purpose operating systems in the form of extensions, or modifications of the existing components (network/storage stacks, BSD socket layer etc). We believe that in practice it would be very difficult, if not impossible, to achieve that degree of specialisation and microarchitectural awareness with commodity stacks: to achieve this would require changing the core design principles of contemporary operating systems, and that would definitely be a substantial engineering effort. Modern stacks have grown to millions of lines of code (e.g., more than 15M lines of code for the Linux kernel) with numerous network and storage controller drivers. Even if we could instantly change the core OS design principles and APIs, this cannot be transparent for userspace applications as they will need to adopt the new principles (e.g., a new POSIX interface): practically this means that modifications would be necessary in the user-level ecosystem, and backwards compatibility cannot be maintained. Hence, instead of trying to envisage a new future-proof, general-purpose operating system, we opted to come up with specialised and impactful solutions to critical problems. We believe that frameworks such as DPDK, netmap, and diskmap that multiplex and export physical resources safely to applications through a minimalistic set of low-level primitives allow for great specialisation of applications for performance, while at the same time maintaining many of the benefits of conventional operating systems.

The end-to-end principle in system design [SRC84] argues that careful selection of the boundaries of functions and abstraction layers is of utmost importance: OS developers have always

strived to invent the appropriate structures and abstractions that would be *generic* enough to promote reusability, and extended functionality. Over the years, however, the end-to-end principle in conventional OSes has been frequently violated in order to improve performance: transport protocol functionality has been implemented in the hardware (e.g., TSO, LRO), application layer functionality has been shifted in the kernel (`accept` filters, in-kernel TLS etc), caches etc. It is clear that *collapsing*, or *blurring* the layer boundaries has been used extensively to reduce overheads, and enhance system performance. We firmly believe that the end-to-end principle does *not* conflict with specialisation; Engler et al. [EKO95] advocate that applications know better than the OS of their resource management decisions, and should be given as much control as possible over them in order to have their own abstractions. Similarly, Lampson et al. [LS79] argue that the lower-level functionality provided by the Operating System should always be *replaceable* by an application-specific version of the function; this is clearly not the case with contemporary general-purpose Operating Systems, limiting that way the overall efficiency and scale of specialisation. If anything, it is the *lack* of end-to-end performance analysis within the microarchitecture of contemporary CPUs that has led to poor performance, as developers have omitted the key data source and sink, DMA, from their optimisations.

Is it future-proof to maintain a microarchitectural focus in I/O stacks' design or should we just focus on the programming model (ISA) and treat the microarchitecture as a black box? The truth is that Operating System developers have always explicitly taken into account the microarchitecture design to optimise conventional OSes for performance. For example, the emergence of SMP systems brought major redesign of core components: whole stacks have been re-engineered to reduce sharing, and promote parallel execution onto multiple cores. With Non-Unified Memory Access (NUMA) systems and PCIe affinity to particular CPU packages, the OS has been modified to support NUMA awareness when managing memory or scheduling tasks to particular cores. Furthermore, OS data structures have been modified accordingly to make effective use of modern CPU caches (e.g., with padding or alignment, trying to avoid false sharing or cache aliasing etc). In Chapter 3 we have presented benchmarks to demonstrate that our web stack is actually portable across different generations of hardware.

With this work we encourage deep microarchitectural awareness when designing performance-critical I/O stacks. Our principles and hypotheses, however, are not temporal or strictly limited in exploiting only contemporary hardware features; instead, we emphasise on minimising the data movement throughout the hardware, not only the architecture. Experience over the last two decades has shown that scalability walls are most often encountered when careless data movement leads to inefficient use of the microarchitecture. The importance of efficient data movement throughout the hardware seems to be further validated, by continuous trends in hardware design such as integration of controllers and peripherals in the CPU package, Cache Allocation Technology (that allows isolating and dedicating LLC fractions to particular applications), non-temporal SSE instruction set extensions to reduce cache pollution and minimise data movement, non-volatile RAM (NVRAM) etc. To further assist microarchitecture-based optimisations, it would really be beneficial if CPU designers were to provide better tools for in-

specting non-software interactions with the microarchitecture, such as by adding new counters that allow more direct analysis of DMA-originated memory interactions.

We have demonstrated how it is feasible to greatly improve system performance, with more efficient software architectures. Making efficient use of the microarchitecture and improving scalability is really important for many reasons (e.g., capital expenditure reduction). Among those, energy consumption is of special growing importance: U.S. datacenters alone use more than 90 billion kilowatt-hours of electricity annually, and this number is projected to increase to 140 billion kWh by 2020 [NB17]. Already, data centers' energy consumption around the globe accounts for ~3% of global electricity supply and almost 2% of the total carbon pollution per year. Improving system efficiency by half or more order of magnitude could allow us to drastically reduce the amount of servers, having a crucial impact in total energy consumption.

Bibliography

[ABC+08] Katerina Argyraki, Salman Baset, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Eddie Kohler, et al. “Can Software Routers Scale?” In: *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*. PRESTO ’08. Seattle, WA, USA: ACM, 2008, pp. 21–26 (cited on page 27).

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach: A New Kernel Foundation for UNIX Development*. Technical report. Computer Science Department, Carnegie Mellon University, Aug. 1986 (cited on page 47).

[ABT+11] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. “vIOMMU: Efficient IOMMU Emulation”. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’11. Portland, OR: USENIX Association, 2011, pp. 6–6 (cited on page 93).

[ACF+H] *accf_http(9) FreeBSD Kernel Developer’s Manual* (cited on page 47).

[ADF+10] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guildford, E. Ozturk, et al. *White paper: Breakthrough AES Performance with Intel AES New Instructions*. Technical report. Intel Corporation, 2010 (cited on page 33).

[AER+11] Mohammad Al-Fares, Khaled Elmeleegy, Benjamin Reed, and Igor Gashinsky. “Overclocking the Yahoo!: CDN for faster web page loads”. In: *Proceedings of the 2011 Internet Measurement Conference*. ACM. 2011 (cited on pages 35, 46, 63, 67).

[All12] Mark Allman. “Comments on Bufferbloat”. In: *SIGCOMM Comput. Commun. Rev.* 43.1 (Jan. 2012), pp. 30–37 (cited on pages 35, 46).

[AVX18] *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*. Technical report. Intel Corporation, 2018 (cited on page 33).

[AWC+14] Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. “TESLA: Temporally Enhanced System Logic Assertions”. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. Amsterdam, The Netherlands: ACM, 2014, 19:1–19:14 (cited on page 22).

[Bal04] Pavan Balaji. “Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck”. In: *In RAIT workshop ’04*. 2004, p. 2004 (cited on page 45).

[Bar00] Moshe Bar. “Kernel Korner: kHTTPd, a Kernel-Based Web Server”. In: *Linux Journal* 2000.76 (Aug. 2000) (cited on pages 47, 74).

[BB08] Raffaele Bolla and Roberto Bruschi. “PC-based Software Routers: High Performance and Application Service Support”. In: *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*. PRESTO ’08. Seattle, WA, USA: ACM, 2008, pp. 27–32 (cited on page 42).

[BBB11] Willem de Bruijn, Herbert Bos, and Henri Bal. “Application-Tailored I/O with Streamline”. In: *ACM Trans. Comput. Syst.* 29.2 (May 2011), 6:1–6:33 (cited on pages 27, 41).

[BBC] *BBC Digital Media Distribution: How we improved throughput by 4x*. Accessed on 20/03/2017. URL: `http://www.bbc.co.uk/blogs/internet/entries/` (cited on page 81).

[BBM+12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. “Dune: Safe User-level Access to Privileged CPU Features”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 335–348 (cited on page 48).

[BCM+10] 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”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–16 (cited on page 42).

[BMD99] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. “A Scalable and Explicit Event Delivery Mechanism for UNIX”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’99. Monterey, California: USENIX Association, 1999, pp. 19–19 (cited on page 38).

[BPM15] M. Belshe, R. Peon, and Ed. M. Thomson. *RFC 7540. Hypertext Transfer Protocol Version 2 (HTTP/2)*. 2015 (cited on page 46).

[BPP+16] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. “The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane”. In: *ACM Trans. Comput. Syst.* 34.4 (Dec. 2016), 11:1–11:39 (cited on pages 37, 48).

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, et al. “Extensibility Safety and Performance in the SPIN Operating System”. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP ’95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 267–283 (cited on page 48).

[Cau10] Caulfield, Adrian M. and De, Arup and Coburn, Joel and Mollow, Todor I. and Gupta, Rajesh K. and Swanson, Steven. “Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories”. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 385–395 (cited on page 45).

[CB17] John Corbet and Willem de Bruijn. *Zero-copy networking*. accessed on 12/11/2017. 2017. URL: https://lwn.net/Articles/726917/ (cited on page 41).

[CDC+13] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. *Increasing TCP’s Initial Window*. RFC 6928. Apr. 2013 (cited on page 95).

[CHLNT] *Chelsio 40GbE Netmap Performance*. http://www.chelsio.com/wp-content/uploads/resources/T5-40Gb-FreeBSD-Netmap.pdf (cited on pages 40, 96).

[Cht6] *Chelsio Terminator 6 ASIC*. accessed on 2/12/2017. URL: Terminator%206%20ASIC (cited on page 44).

[CJR+89] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. “An analysis of TCP processing overhead”. In: *IEEE Communications Magazine* 27.6 (June 1989), pp. 23–29 (cited on page 45).

[CKZ+13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 1–17 (cited on pages 39, 42).

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. “Dynamic Instrumentation of Production Systems”. In: *Proceedings of the 2004 USENIX Annual Technical Conference*. ATC ’04. Boston, MA: USENIX, 2004 (cited on page 75).

[CW09] Jonathan Corbet and Michael Waychison. *KS2009: How Google uses Linux*. accessed on 12/11/2017. 2009. URL: https://lwn.net/Articles/357658/ (cited on page 43).

[DASH] *Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*. ISO/IEC 23009-1 (`http://standards.iso.org/ittf/PubliclyAvailableStandards`). Apr. 2012 (cited on pages 74, 81).

[DDIO] *Intel Data Direct IO*. Accessed on 17/03/2017. URL: `http://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html` (cited on pages 25, 30, 80, 96).

[DEA+09] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, et al. "RouteBricks: Exploiting Parallelism to Scale Software Routers". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 15–28 (cited on page 27).

[Der05] L. Deri. "nCap: Wire-speed Packet Capture and Transmission". In: *Proceedings of the End-to-End Monitoring Techniques and Services on 2005. Workshop*. E2EMON '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 47–55 (cited on page 44).

[DNH+14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. "FaRM: Fast Remote Memory". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 401–414 (cited on page 46).

[DNSP1+] *DNSPerf*. `http://nominum.com/measurement-tools/` (cited on page 70).

[DP93] Peter Druschel and Larry L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. ACM, 1993 (cited on pages 39, 41).

[DPDK] *Intel Data Plane Development Kit*. `http://dpdk.org` (cited on pages 44, 112).

[EBB+95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. "U-Net: a user-level network interface for parallel and distributed computing". In: *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995 (cited on pages 43, 46).

[EGH+08] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. "Towards high performance virtual routers on commodity hardware". In: *Proceedings of the 2008 ACM CoNEXT Conference*. ACM. 2008, p. 20 (cited on page 63).

[EKO95] 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. Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266 (cited on pages 47–48, 53, 114).

[EM95] Aled Edwards and Steve Muir. "Experiences Implementing a High Performance TCP in User-space". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '95. Cambridge, Massachusetts, USA: ACM, 1995, pp. 196–205 (cited on page 43).

[FHH+03] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. "TCP Performance Re-visited". In: *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 70–79 (cited on page 45).

[Geb17] Gennie Gebhart. *Tipping the Scales on HTTPS: 2017 in Review*. accessed on 23/12/2017. 2017. URL: `https://www.eff.org/deeplinks/2017/12/tipping-scales-https` (cited on page 33).

[GEK+02] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. "Fast and Flexible Application-level Networking on Exokernel Systems". In: *ACM Trans. Comput. Syst.* 20.1 (Feb. 2002), pp. 49–83 (cited on page 47).

[GGL17] *Google Transparency Report: HTTPS encryption on the web*. accessed on 23/12/2017. 2017. URL: `https://transparencyreport.google.com/https/overview` (cited on page 33).

[GIPR+] *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` (cited on pages 33, 81).

[GK14] S. Gueron and M.E. Kounavis. *White Paper: Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. Technical report. Intel Corporation, 2014 (cited on page 33).

[GLIS+H] *Galois Inc, The Haskell Lightweight Virtual Machine (HaLVM)*. `https://github.com/galoisinc/halvm` (cited on page 48).

[Gue10] Shay Gueron. *White paper: Intel Advanced Encryption Standard (AES) New Instructions Set*. Technical report. Intel Corporation, 2010 (cited on page 33).

[GWA+15] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, et al. "Clean Application Compartmentalization with SOAAP". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 1016–1031 (cited on page 23).

[HDS] *Adobe HTTP Dynamic Streaming*. `http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/hds/pdfs/adobe-hds-specification.pdf` (cited on page 81).

[HHR+14] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. "Rekindling Network Protocol Innovation with User-level Stacks". In: *SIGCOMM Comput. Commun. Rev.* 44.2 (Apr. 2014), pp. 52–58 (cited on page 97).

[HJP+10a] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. "PacketShader: A GPU-accelerated Software Router". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 195–206 (cited on page 42).

[HJP+10b] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. "PacketShader: A GPU-accelerated Software Router". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 195–206 (cited on page 44).

[HLS] Roger Pantos and William May. *HTTP Live Streaming*. Internet-Draft draft-pantos-http-live-streaming-20. Work in Progress. Internet Engineering Task Force, Sept. 2016. 57 pp. (cited on pages 74, 81).

[HMC+12] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. "MegaPipe: a new programming interface for scalable network I/O". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. USENIX Association. 2012, pp. 135–148 (cited on pages 37, 39–40).

[HS16] Tom Herbert and Alexei Starovoitov. *eXpress Data Path*. accessed on 12/11/2017. 2016. URL: `https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf` (cited on page 44).

[Inf4] Infiniband Trade Organization. *Introduction to Infiniband for end users*. accessed on 24/11/2017. URL: `https://cw.infinibandta.org/document/dl/7268` (cited on page 46).

[JBL+10] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. "DFS: A File System for Virtualized Flash Storage". In: *Trans. Storage* 6.3 (Sept. 2010), 14:1–14:25 (cited on page 45).

[JWJ+14] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo 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. Seattle, WA: USENIX Association, 2014, pp. 489–502 (cited on pages 37, 44, 97).

[Kan12] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels; Joustavat käyttöjärjestelmät: Jokaytimen ja Tynkäytimien suunnittelu ja toteutus*. en. G4 Monografiaväitöskirja. 2012 (cited on page 48).

[KH13] Michael Kerrisk and Tom Herbert. *The SO_REUSEPORT socket option*. accessed on 12/11/2017. 2013. URL: `https://lwn.net/Articles/542629/` (cited on page 40).

[KLC+14] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. "OSv - Optimizing the Operating System for Virtual Machines". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 2014, pp. 61–72 (cited on page 48).

[Kle86] S. R. Kleiman. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX". In: 1986, pp. 238–247 (cited on page 42).

[KPS+16] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. "High Performance Packet Processing with FlexNIC". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 67–81 (cited on page 44).

[LEM00] Chuck Lever, Marius Aamodt Eriksen, and Stephen P. Molloy. *An Analysis of the TUX Web Server*. Technical report 00-8. Ann Arbor, MI: Center for Information Technology Integration (CITI), University of Michigan, Nov. 2000 (cited on pages 47, 74).

[Lem01] Jonathan Lemon. "Kqueue - A Generic and Scalable Event Notification Facility". In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 141–153 (cited on pages 38, 95).

[LHA+14] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-memory Key-value Storage". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 429–444 (cited on page 46).

[LHTPD] *Lighttpd*. Accessed 17/03/2017. URL: `http://www.lighttpd.com/` (cited on page 65).

[LIO] *lio_listio(2) FreeBSD System Calls Manual* (cited on page 41).

[LMB+06] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, et al. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications". In: *IEEE J.Sel. A. Commun.* 14.7 (Sept. 2006), pp. 1280–1297 (cited on page 48).

[LRW+17] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 183–196 (cited on page 46).

[LS79] Butler W. Lampson and Robert F. Sproull. "An Open Operating System for a Single-user Machine". In: *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*. SOSP '79. Pacific Grove, California, USA: ACM, 1979, pp. 98–105 (cited on page 114).

[LWIP] Adam Dunkels. *Design and Implementation of the lwIP*. 2001 (cited on page 44).

[MAM+99] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. Technical report. United States, 1999 (cited on page 52).

[MAR+14] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "ClickOS and the Art of Network Function Virtualization". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 459–473 (cited on page 48).

[MCZ06] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. "Optimizing Network Virtualization in Xen". In: *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*. ATEC '06. Boston, MA: USENIX Association, 2006, pp. 2–2 (cited on page 43).

[MGL13] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 103–114 (cited on page 46).

[MMR+13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, et al. "Unikernels: library operating systems for the cloud". In: *Proceedings of the 2013 International Conference on Architectural support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013 (cited on page 48).

[MMT16] Alex Markuze, Adam Morrison, and Dan Tsafrir. “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. Atlanta, Georgia, USA: ACM, 2016, pp. 249–262 (cited on page 93).

[MN03] Derek McAuley and Rolf Neugebauer. “A Case for Virtual Channel Processors”. In: *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*. NICELI ’03. Karlsruhe, Germany: ACM, 2003, pp. 237–242 (cited on page 43).

[MNW14] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd. Addison-Wesley Professional, 2014 (cited on page 39).

[MP96] David Mosberger and Larry L. Peterson. “Making Paths Explicit in the Scout Operating System”. In: *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’96. Seattle, Washington, USA: ACM, 1996, pp. 153–167 (cited on page 48).

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. “Eliminating Receive Livelock in an Interrupt-driven Kernel”. In: *ACM Trans. Comput. Syst.* 15.3 (Aug. 1997), pp. 217–252 (cited on page 43).

[MWH+17] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. “Disk, Crypt, Net: Rethinking the Stack for High-performance Video Streaming”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: ACM, 2017, pp. 211–224 (cited on page 22).

[MWH13] Ilias Marinos, Robert N. M. Watson, and Mark Handley. “Network Stack Specialization for Performance”. In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. HotNets-XII. College Park, Maryland: ACM, 2013, 9:1–9:7 (cited on page 22).

[MWH14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. “Network Stack Specialization for Performance”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM ’14. Chicago, Illinois, USA: ACM, 2014, pp. 175–186 (cited on pages 22, 37, 97).

[NB17] Jiacheng Ni and Xuelian Bai. “A review of air conditioning energy performance in data centers”. In: 67 (Jan. 2017) (cited on page 115).

[NCA] *Solaris Network Cache and Accelerator (NCA), System Administration Guide, Volume 3* (cited on page 47).

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, et al. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI'13. Lombard, IL: USENIX Association, 2013, pp. 385–398 (cited on page 46).

[NFLX+] *Netflix Appliance Software*. `https://openconnect.netflix.com/en/software/` (cited on page 81).

[NGINX] *Nginx web server*. Accessed 17/03/2017. URL: `http://nginx.org/` (cited on page 52).

[NSD] *Name Server Daemon (NSD)*. Accessed on 17/03/2017. URL: `http://www.nlnetlabs.nl/projects/nsd/` (cited on page 52).

[NVME] *NVM Express Specification 1.2.1*. `http://www.nvmexpress.org/specifications/` (cited on page 93).

[OLSP] *OpenLiteSpeed*. Accessed 17/03/2017. URL: `http://open.litespeedtech.com/` (cited on page 65).

[ORS+11] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 29–41 (cited on page 46).

[p3608] *Intel P3608 NVME drive*. `http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-p3608-series.html` (cited on page 90).

[PBH+11] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. "Rethinking the Library OS from the Top Down". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 291–304 (cited on page 48).

[PDZ00] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. "IO-Lite: a unified I/O buffering and caching system". In: *ACM Transactions on Computer Systems (TOCS)* 18.1 (2000) (cited on page 41).

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. "Flash: An Efficient and Portable Web Server". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '99. Monterey, California: USENIX Association, 1999, pp. 15–15 (cited on page 38).

[PF01] Ian Pratt and Keir Fraser. "Arsenic: A user-accessible gigabit Ethernet interface". In: *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE. 2001 (cited on page 43).

[PGG+95] 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. Copper Mountain, Colorado, USA: ACM, 1995, pp. 79–95 (cited on page 106).

[PL00] Niels Provos and Chuck Lever. "Scalable Network I/O in Linux". In: *Proceedings 2000 USENIX Annual Technical Conference*. ATC '00. San Diego, California: USENIX, 2000 (cited on page 38).

[PLZ+14] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, et al. "Arrakis: The Operating System is the Control Plane". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 1–16 (cited on pages 43, 48).

[PSZ+12] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. "Improving Network Connection Locality on Multicore Systems". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 337–350 (cited on page 40).

[QUIC] *QUIC, a multiplexed stream transport over UDP*. accessed on 23/12/2017. URL: `https://www.chromium.org/quic` (cited on page 46).

[RDH+03] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. "Cassyopia: Compiler Assisted System Optimization". In: *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*. HOTOS'03. Lihue, Hawaii: USENIX Association, 2003, pp. 18–18 (cited on page 39).

[RDMA] *Architectural Specifications for RDMA over TCP/IP*. accessed on 12/11/2017. URL: `http://www.rdmaconsortium.org` (cited on page 46).

[RG05] Mendel Rosenblum and Tal Garfinkel. "Virtual Machine Monitors: Current Technology and Future Trends". In: *Computer* 38.5 (May 2005), pp. 39–47 (cited on page 43).

[Riz12] Luigi Rizzo. "netmap: a novel framework for fast packet I/O". In: *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association. 2012, pp. 9–9 (cited on pages 41, 44, 52, 80, 91).

[SGL15] Randall Stewart, John-Mark Gurney, and Scott Long. "Optimizing TLS for High–Bandwidth Applications in FreeBSD". In: *Proc. Asia BSD conference*. 2015 (cited on page 84).

[SLNS] *Scaling in the Linux Networking Stack*. Accessed on 1/10/2017. URL: `https://www.kernel.org/doc/Documentation/networking/scaling.txt` (cited on pages 40, 99).

[SLRFO] *Solarflare OpenOnload*. Accessed on 1/10/2017. URL: http://www.openonload.org/ (cited on page 44).

[SMC08] Joseph A. Salowey, Dr. David A. McGrew, and Abhijit Choudhury. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. Technical report 5288. Aug. 2008. 8 pp. (cited on pages 98, 105).

[SNDF1] *sendfile(2) FreeBSD System Calls Manual* (cited on pages 40–41).

[SPDK] *Intel Storage Performance Development Kit*. http://www.spdk.io (cited on pages 45, 95).

[SPDY] *SPDY: An experimental protocol for a faster web*. accessed on 23/12/2017. URL: http://dev.chromium.org/spdy/spdy-whitepaper (cited on page 46).

[SPL2] *splice(2) Linux Programmer's Manual*. Sept. 2017 (cited on page 41).

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-end Arguments in System Design". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288 (cited on pages 47, 113).

[SS10] Livio Soares and Michael 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. Vancouver, BC, Canada: USENIX Association, 2010, pp. 33–46 (cited on page 39).

[SS11] Livio Soares and Michael Stumm. "Exception-less System Calls for Event-driven Servers". In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'11. Portland, OR: USENIX Association, 2011, pp. 10–10 (cited on page 39).

[TEE2] *tee(2) Linux Programmer's Manual*. Sept. 2017 (cited on page 41).

[The04] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. New York, NY, USA: IEEE, 2004 (cited on page 40).

[THTTP] Jef Poskanzer. *thttpd*. http://www.acme.com/software/thttpd/ (cited on page 38).

[TNM+93] Chandramohan A Thekkath, Thu D Nguyen, Evelyn Moy, and Edward D Lazowska. "Implementing network protocols at user level". In: *IEEE/ACM Transactions on Networking (TON)* 1.5 (1993), pp. 554–565 (cited on page 43).

[ToE05] *White paper: The Benefits of 10 Gbps TCP Offload*. Technical report. Chelsio Communications, 2005 (cited on page 44).

[VAK11] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. "The Case for VOS: The Vector Operating System". In: *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011*. 2011 (cited on page 39).

[VL87] G. Varghese and T. Lauck. "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility". In: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. SOSP '87. Austin, Texas, USA: ACM, 1987, pp. 25–38 (cited on page 100).

[VMSP2] *vmsplice(2) Linux Programmer's Manual*. Sept. 2017 (cited on page 41).

[VNP+14] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. "Aerie: Flexible File-system Interfaces to Storage-class Memory". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 14:1–14:14 (cited on page 42).

[WBK+14] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. "How Speedy is SPDY?" In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 387–399 (cited on page 46).

[WCB01] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-conditioned, Scalable Internet Services". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: ACM, 2001, pp. 230–243 (cited on page 38).

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System". In: *Commun. ACM* 17.6 (June 1974), pp. 337–345 (cited on page 47).

[WEIG] *weighttp: a lightweight benchmarking tool for webservers*. Accessed on 30/09/2017. URL: `http://redmine.lighttpd.net/projects/weighttp/wiki/` (cited on page 65).

[WM95] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24 (cited on pages 27, 41).

[XNML9] *Virtualization in Xen 3.0*. Accessed on 2/11/2017. URL: `http://www.linuxjournal.com/article/8909` (cited on page 39).

[YHS+16] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. "StackMap: Low-latency Networking with the OS Stack and Dedicated NICs". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 43–56 (cited on page 41).

[ZEUS] *Zeus Technology, Zeus Web server* (cited on page 38).