The “other” FreeBSD optimizations used by Netflix to serve video at 800Gb/s from a single server
Or..

“How badly can I break Netflix’s performance when I disable optimizations?”
Motivation:

● Since 2021, Netflix has been able to serve almost 800Gb/s of TLS encrypted video traffic from a single server.
● How much are the various optimizations made to FreeBSD over the years helping?
Note: Most of the optimizations discussed in this slide deck were done outside of Netflix, by members of the FreeBSD community.
If I have seen further, it is by standing on the shoulders of giants.

- Isaac Newton
REMOVING ONE OPTIMIZATION CAN MAKE PERFORMANCE TUMBLE DOWN
Netflix Video Serving Workload

- FreeBSD-current
- NGINX web server
- Video served via sendfile(2) and encrypted using software kTLS
Netflix 400G Video Serving Hardware

- **AMD EPYC 7502P (“Rome”)**
  - 32 cores @ 2.5GHz
  - 256GB DDR4-3200
    - 8 channels
    - ~150GB/s mem bw
    - Or ~1.2Tb/s in networking units
  - 128 lanes PCIe Gen4
    - ~250GB/s of IO bandwidth
    - Or ~2Tb/s in networking units
Netflix 400G Video Serving Hardware

● 2x Mellanox ConnectX-6 Dx
  ○ Gen4 x16, 2 full speed 100GbE ports per NIC
    ■ 4 x 100GbE in total
  ○ Support for NIC kTLS offload
● 18x WD SN720 NVME
  ○ 2TB
  ○ PCIe Gen3 x4
Measurement Metrics

- Measure the maximum stable bandwidth of a configuration
- Use this bandwidth, and the CPU utilization, to arrive at a new “Gb/s per Percent CPU” metric.
Optimal Configuration

- Dataflow using NIC kTLS & sendfile
- All VM and NIC optimizations enabled
- Baseline Bandwidth: 375Gb/s @ 53% CPU
  - Or 7.1Gbs/pcpu
Section 1:

Sendfile & kTLS
Netflix 400Gb/s Video Serving Data Flow

When not using sendfile, data is copied to userspace & encrypted by the host CPU, then copied back to the kernel.

400Gb/s == 50GB/s

~400GB/sec of memory bandwidth and ~64 PCIe Gen 4 lanes are needed to serve 400Gb/s
What is sendfile?

- Specify a file and a socket to send it on
- Kernel sends directly from the page cache
  - No data is copied to userspace
  - Nginx never sees the data it is sending
Problem: Disk reads can block
sendfile

- When an nginx worker is blocked, it
cannot service other requests
- Solutions to prevent nginx from blocking
  like aio or thread pools scale poorly
Solution: Asynchronous sendfile

- sendfile() becomes “fire and forget”
- Empty buffers are appended to the TCP socket buffer. TCP stops when it sees an empty buffer.
- When disk read completes, disk interrupt handler informs TCP it is ready to send
Asynchronous sendfile

Disks

Socket Buffer

Internet
Asynchronous sendfile

Disks

Socket Buffer

Internet
Asynchronous sendfile

Socket Buffer

Disks

Internet
Asynchronous sendfile
Asynchronous sendfile

Disks → Socket Buffer → Internet
Asynchronous sendfile

Socket Buffer

Disks

Internet
Asynchronous sendfile
What is kTLS?

- Bulk crypto is moved into the kernel
  - Handshakes are still done in userspace
  - Required for async sendfile based dataflow with no copies or context switches.
- Doing crypto in the kernel almost quadruples CPU efficiency
- Originated at Netflix
Netflix 400Gb/s Video Serving Data Flow

When not using sendfile, data is copied to userspace & encrypted by the host CPU, then copied back to the kernel.

400Gb/s == 50GB/s

~400GB/sec of memory bandwidth and ~64 PCIe Gen 4 lanes are needed to serve 400Gb/s.
Netflix 400Gb/s Video Serving Data Flow

Using sendfile and software kTLS, data is encrypted by the host CPU.

400Gb/s == 50GB/s

~200GB/sec of memory bandwidth and ~64 PCIe Gen 4 lanes are needed to serve 400Gb/s
Netflix 400Gb/s Video Serving Data Flow

Using sendfile and software kTLS, data is encrypted by the host CPU.

400Gb/s == 50GB/s

~200GB/sec of memory bandwidth and ~64 PCIe Gen 4 lanes are needed to serve 400Gb/s
Netflix 400Gb/s Video Serving Data Flow

Using sendfile and NIC kTLS, data is encrypted by the NIC.

400Gb/s == 50GB/s

~100GB/sec of memory bandwidth and ~64 PCIe Gen 4 lanes are needed to serve 400Gb/s
Disable kTLS (and async sendfile)

- I was expecting just elevated CPU and memory bandwidth
  - Max BW is ~40Gb/s with 100% CPU
  - Bottlenecked on lock contention on aio queues
    - Nginx uses aio to avoid blocking when sending files without async sendfile.
Disable kTLS (and async sendfile)
PMC Flame Graph
Disable kTLS (and async sendfile)

- Attempt 2: Use nginx thread pools
  - 90Gb/s, 80% CPU
  - A lot of time spent accessing memory
    - Copy out file data from kernel to nginx
    - Crypto in userspace SSL
    - Extra memcpy in nginx for SSL
    - Copy in data to kernel from nginx
Disable kTLS (and async sendfile)
Disable sendfile (but use kTLS)

- 75Gb/s, 80% CPU
  - VM lock contention
  - A lot of time spent accessing memory
    - Copy out file data from kernel to nginx
    - Extra memcpy in nginx for SSL
    - Copy in data to kernel from nginx
    - Crypto in-place in kernel
Disable sendfile (but use kTLS)
disable sendfile (but use NIC kTLS)

- 95Gb/s, 80% CPU
  - VM lock contention
  - A lot of time spent accessing memory
    - Copy out file data from kernel to nginx
    - Extra memcpy in nginx for SSL
    - Even though it is not doing encryption, it still copies into a 16k buffer
    - Copy in data to kernel from nginx
Disable sendfile (but use NIC kTLS)
ISA-L

- Intel Intelligent Storage Acceleration Library
  - In ports as security/isal-kmod
  - Works well on AMD CPUs as well as Intel

- Highly optimized accelerated AES block ciphers
  - Has options to use non-temporal instructions, which avoids read-modify-write cache miss when storing crypto results
Enable Sendfile & kTLS, but disable ISA-L crypto

- 180Gb/s, 80% CPU
  - CPU / Memory bound in aesni crypto
  - Unlike ISA-L
    - We take cache misses when storing encrypted data
    - Data is copied
Enable Sendfile & kTLS, but disable ISA-L crypto
Enable Sendfile & kTLS

- 240Gb/s, 80% CPU
  - CPU / Memory bound in ISA-L crypto
Enable Sendfile & kTLS

![Bar chart showing Gbs/pcpu comparison between different configurations: ktls, no isal, ktls, isal, and nic ktls. The chart indicates that nic ktls has the highest Gbs/pcpu, followed by ktls, isal, and then ktls, no isal.]
Section 2: Virtual Memory Optimizations
UMA VM Page Cache

- A per-cpu pool of free pages that can be accessed locklessly
- Managed via UMA (Universal Memory Allocator)
- Only works for free pages, not pages that are recycled into the inactive or active page queues
Disable UMA VM Page Cache

- 60Gb/s 95% CPU
- Severe lock contention on VM free page queue
VM Batch Queues

- A way to free multiple pages to a page queue with a single lock
Disable VM Batch Queues

- 280Gb/s 95% CPU
- Severe lock contention on VM inactive page queue
VM Batch Queues

- no pg cache: low performance
- no batch queue: moderate performance
- optimal: high performance
SF_NOCACHE

- SF_NOCACHE causes data sent by sendfile() to be freed directly, and to not linger on the inactive page queues.
- Used when we don’t expect data to be re-used.
Disable SF_NOCACHE

- 120Gb/s at 55% CPU
  - Lock contention on the inactive page queue
  - Nginx pauses cause clients to run away
Disable SF_NOCACHE
16KB Pages (arm64)

- Arm64 recently added support for 16K pages.
- A lot of our kernel time is spent in page management.
- Large performance improvement:
  - 345Gb/s @ 80% CPU -> 368Gb/s @ 66% CPU
  - Ampere Q80-30, 128GB RAM, CX6-DX
Section 3: Network Stack Optimizations
TCP Large Receive Offload (LRO)

- LRO aggregates multiple received packets from the same TCP connection
- It reduces trips through the network stack
  - This reduces connection lookups, lock acquisitions and releases, decisions about when to send TCP acks, etc.
Disable TCP Large Receive Offload

- 330G 65% CPU
  - Health limited by NIC drops, clients go away
TCP Large Receive Offload

![Bar chart showing Gbs/pcpu comparison between no LRO and optimal case]

- No LRO: 4 Gbs/pcpu
- Optimal: 8 Gbs/pcpu
RSS accelerated LRO

Connection 0
Packet 0

Connection 1
Packet 0

Connection 2
Packet 0

Connection 3
Packet 0

Connection 4
Packet 0

Connection 5
Packet 0

Connection 254
Packet 3

Connection 255
Packet 3

Connection 0
Packet 0

Connection 0
Packet 1

Connection 0
Packet 2

Connection 0
Packet 3

Connection 1
Packet 0

Connection 1
Packet 1

Connection 255
Packet 2

Connection 255
Packet 3

SORT
Disable RSS accelerated LRO

- 365G 70% CPU
  - Health limited by NIC drops, clients go away
  - Basically the same efficiency as no LRO
RSS accelerated LRO

Gbs/pcpu

- no LRO
- no RSS assist
- optimal
TCP Large Send Offload (TSO)

- Like LRO, we reduce the number of trips through the network stack.
- Rather than sending 2 (or 8 or 43) packets to the NIC, we send one. NIC breaks (segments) it into 2 (or 8 or 43) packets on the wire.
- Avoids having to allocate headers for each, look up ethernet addresses, and interact with NIC hardware for each packet.
TSO Disabled

- 180G 85% CPU
  - Needed to disable IRQ coalescing to avoid transmit drops
  - A lot more time spent in network related functions.
TCP Large Send Offload (TSO)
Disable TSO and LRO

- 170G 85% CPU
  - Needed to disable IRQ coalescing to avoid transmit drops
TCP Large Send Offload (TSO) and LRO

![Bar chart showing performance comparison between no TSO, no LRO, no TSO & no LRO, and optimal modes.]

- **no TSO**: 2 Gbps/pcpu
- **no LRO**: 5 Gbps/pcpu
- **no TSO & no LRO**: 1 Gbps/pcpu
- **optimal**: 7 Gbps/pcpu
But wait, there’s … not … more..

- 800Gb prototype sitting on datacenter floor due to shipping exception 😞
- Something to talk about next year?
800G Prototype Details

- Dell R7525
- 2x AMD EPYC 7713 64c / 128t (128c / 256t total)
- 3x xGMI links between sockets
- 512 GB RAM
- 4x Mellanox ConnectX-6 Dx (8x 100GbE ports)
- 16x Intel Gen4 x4 14TB NVME
Initial Results: 420Gb/s

- Ran in 1NPS mode
- Network Siloing mode
- CPUs mostly idle
  - AMD guessed that xGMI was down-linking to x2
  - Set xGMI speed to 18GT/s and forced link width to x16, and disabled dynamic link width management
Results with DLWM forced: 500Gb/s

- Ran in 1NPS mode
- Network Siloing mode
  - NVME data DMA’ed to NIC’s NUMA Node
- xGMI link usage very uneven:
  - 15GB/s, 4GB/s and 2GB/s
  - Turns out that NVME is not evenly distributed by IO Quadrants
  - Even hashing of cross-socket to xGMI depends on evenly distributed IO
How to Improve xGMI Hashing

- Hashing based on device doing DMA
  - NVME is very uneven
  - NICs are much less uneven
  - “Network Siloing” normally does DMA from NVME to remote node, local to NIC
- Flip things, and do DMA from NVME to local buffers
- The NICs are doing DMA across xGMI
Results with local DMA to NVME node: 670Gb/s

- Much more even xGMI hashing:
  - 10/10/7 GB/s
- Problematic because:
  - Daemon that “locks” content into memory is not NUMA aware & can lead to page daemon thrashing.
  - Still pressure on xGMI links
Disk centric siloing

- Associate disk controllers with NUMA nodes
- Associate NUMA affinity with files
- Associate network connections with NUMA nodes
- Move connections to be “close” to the disk where the contents file is stored.
- After the connection is moved, there will be 0 NUMA crossings for **bulk** data.
Disk centric siloing problems

- No way to tell link partner that we want LACP to direct traffic to a different switch/router port
  - So TCP acks and http requests will come in on the “wrong” port
- Moving connections can lead to TCP re-ordering due to using multiple egress NICs
- Some clients issue http GET requests for different content on the same TCP connection
  - Content may be on different NUMA domains!
Disk centric siloing problems

● Moving NIC TLS sessions is expensive
  ○ Session will be established before content location is known
  ○ Once content location is known, crypto state needs to torn down on the original egress NIC and re-established on the NIC close to the media file.
Disk centric siloing problems

- Affinities are wrong for most things
  - Nginx worker accepted the connection on the NUMA node near the ingress NIC, so all sends on the socket will originate from the wrong node.
  - TCP/IP, ktls, etc, data structures allocated on node near ingress NIC
  - Incoming TCP acks will be handled on ingress NIC
  - TCP pacing done by pacer on “wrong” node
Disk centric siloing problems

- **Network Siloing:** Each connection hashed by LACP hash over IP/port.
  - Hundreds of thousands of unique IP/port combos
  - Sharding of conns to NUMA domains is nearly perfect
- **Disk Siloing:**
  - Each connection is hashed by content location
  - 8 to 32 drives considered
  - Sharding is almost always uneven
Disk centric siloing problems

● Uneven sharding can lead to hot NUMA nodes
  ○ Hot node constantly paging due to lack of RAM
  ○ Hot node NICs overloaded, leading to output drops while cold node’s NICs are underused
“Disk Centric Siloing” Results: 731Gb/s

- Much less xGMI traffic
- Limited by NIC output drops, not CPU.
- Cause of drops is now largely due to:
  - Page daemon interfering with nginx on popular node
  - Uneven loading on NICs due to content popularity differences. (NICs on popular node doing 94Gb/s, others doing 89Gb/s)