NETFLIX

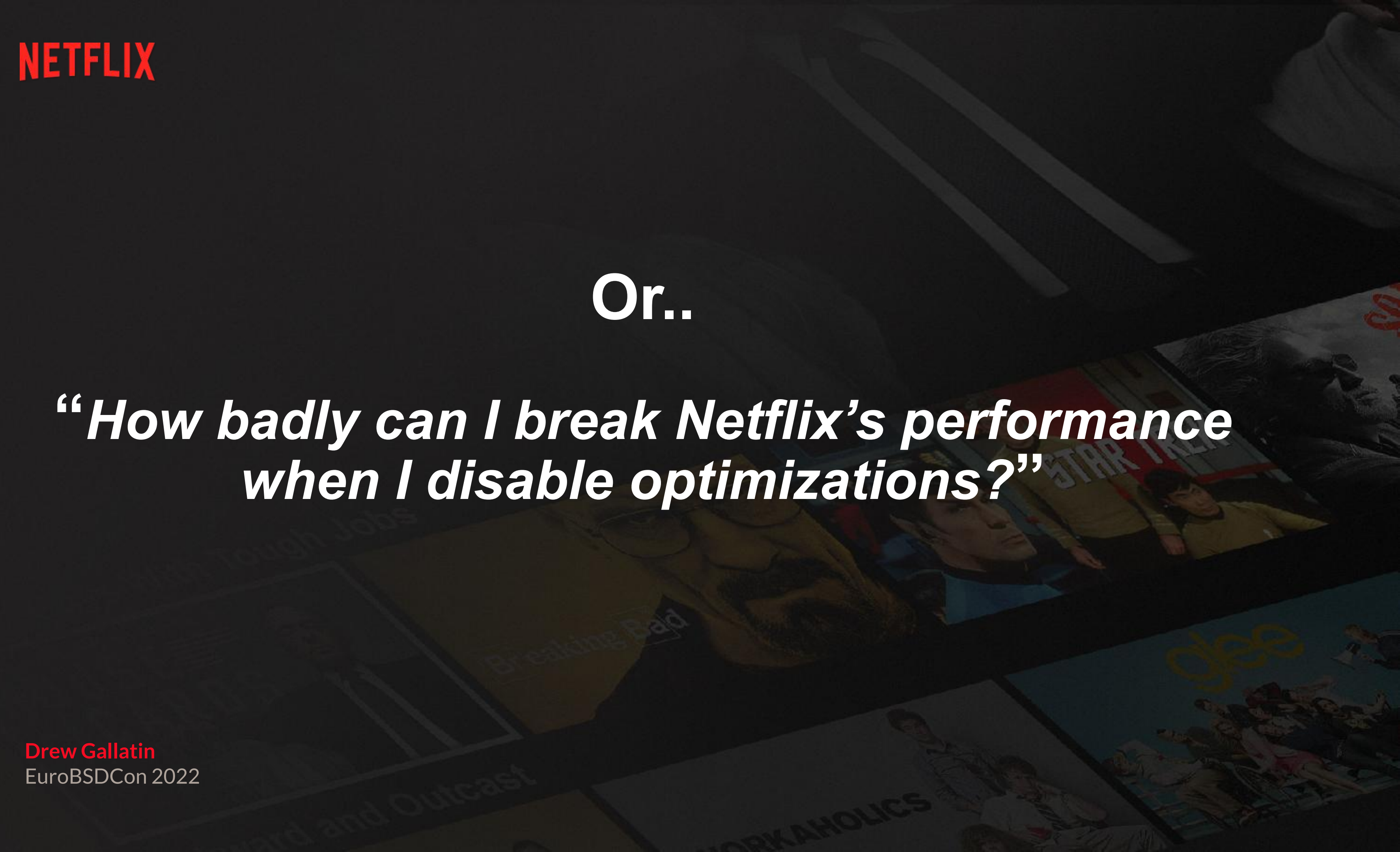
The “other” FreeBSD optimizations used by Netflix to serve video at 800Gb/s from a single server

Drew Gallatin
EuroBSDCon 2022

NETFLIX

Or..

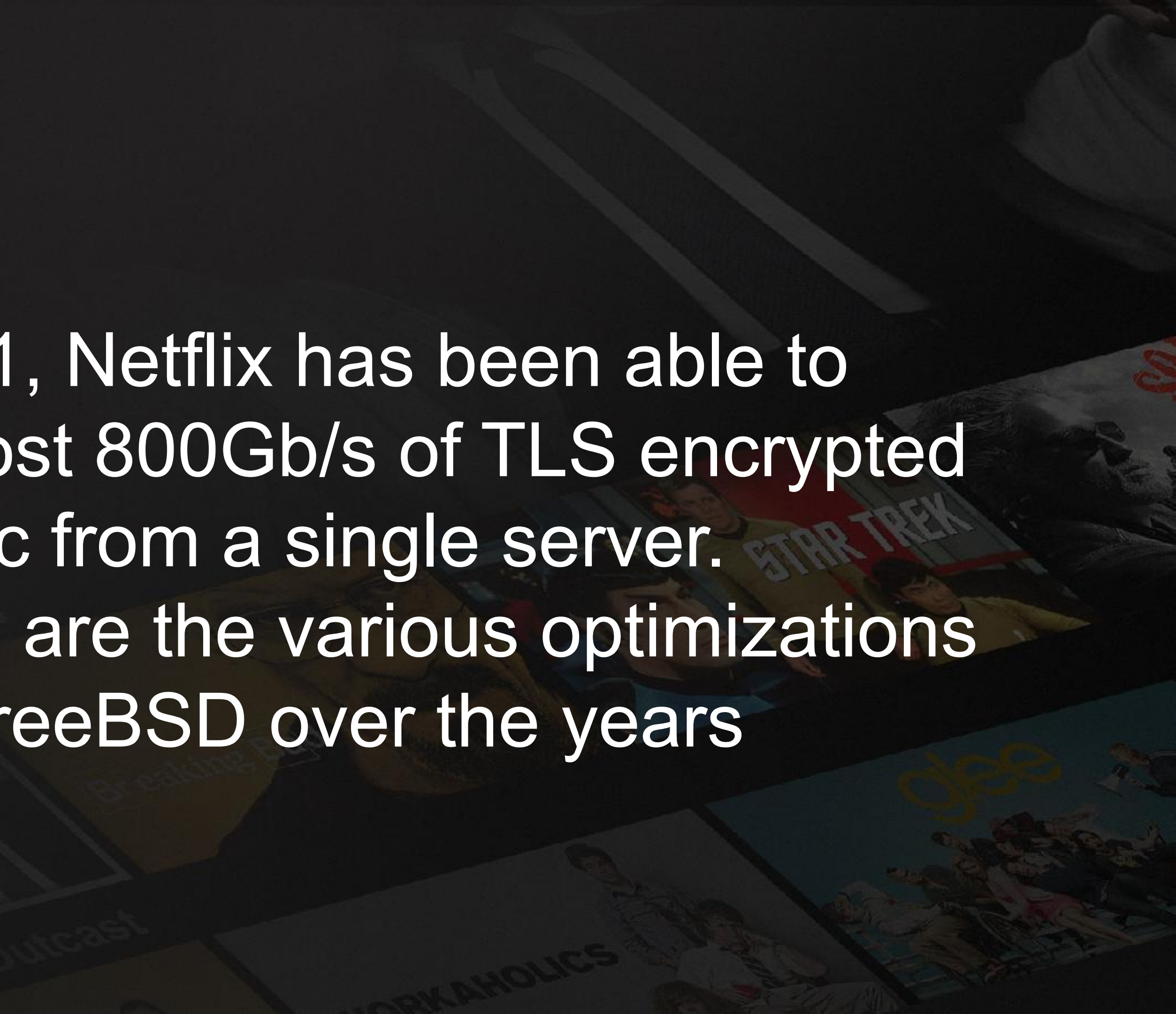
"How badly can I break Netflix's performance when I disable optimizations?"



Drew Gallatin
EuroBSDCon 2022

Motivation:

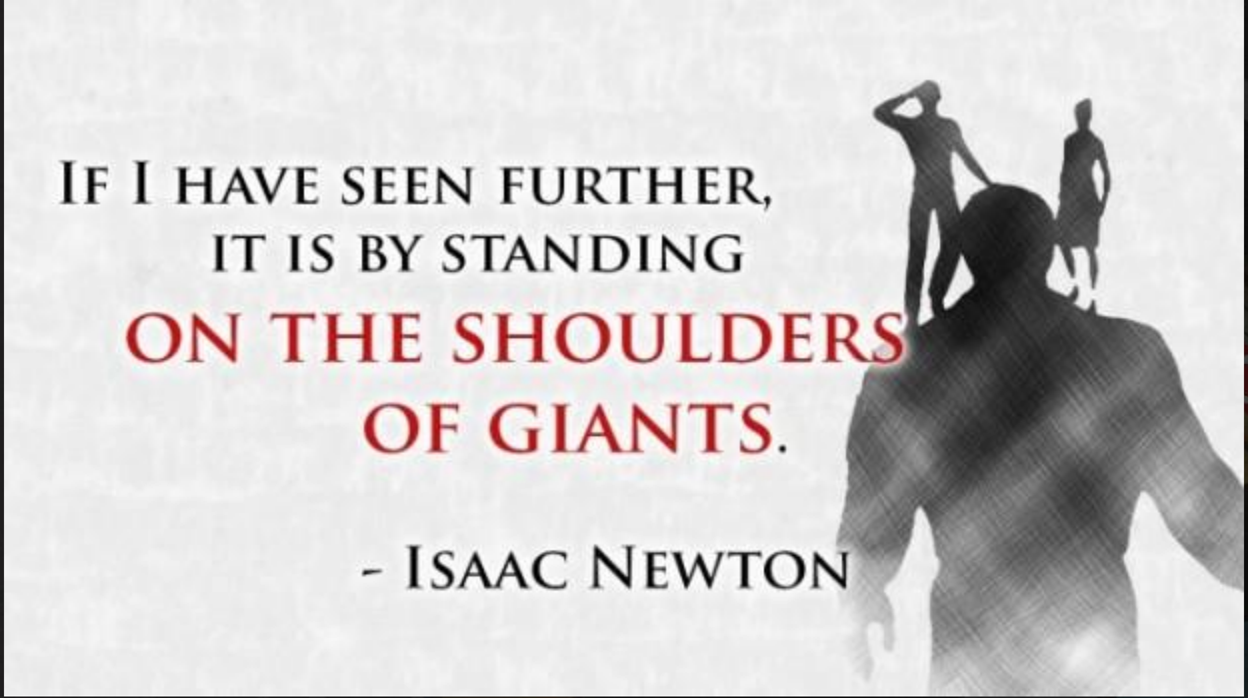
- Since 2021, Netflix has been able to serve almost 800Gb/s of TLS encrypted video traffic from a single server.
- How much are the various optimizations made to FreeBSD over the years helping?



Note: Most of the optimizations discussed in this slide deck were done outside of Netflix, by members of the FreeBSD community

Drew Gallatin

EuroBSDCon 2022



If I have seen further,
it is by standing
**on the shoulders
of giants.**

- Isaac Newton

REMOVING ONE OPTIMIZATION
CAN MAKE PERFORMANCE TUMBLE DOWN
imgflip.com

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

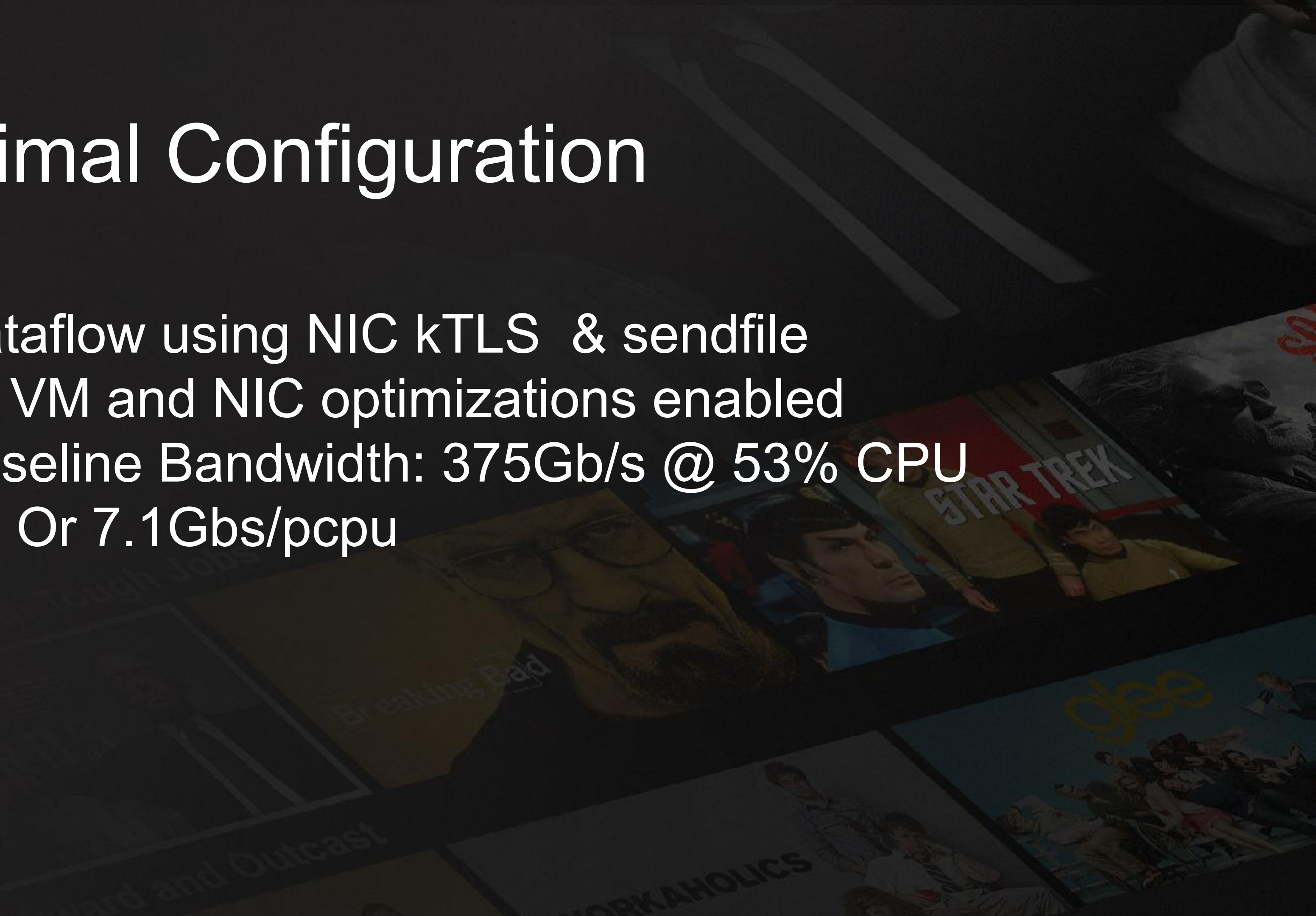
- 2x Mellanox ConnectX-6 Dx
  - Gen4 x16, 2 full speed 100GbE ports per NIC
    - 4 x 100GbE in total
  - Support for NIC kTLS offload
- 18x WD SN720 NVME
  - 2TB
  - PCIe Gen3 x4

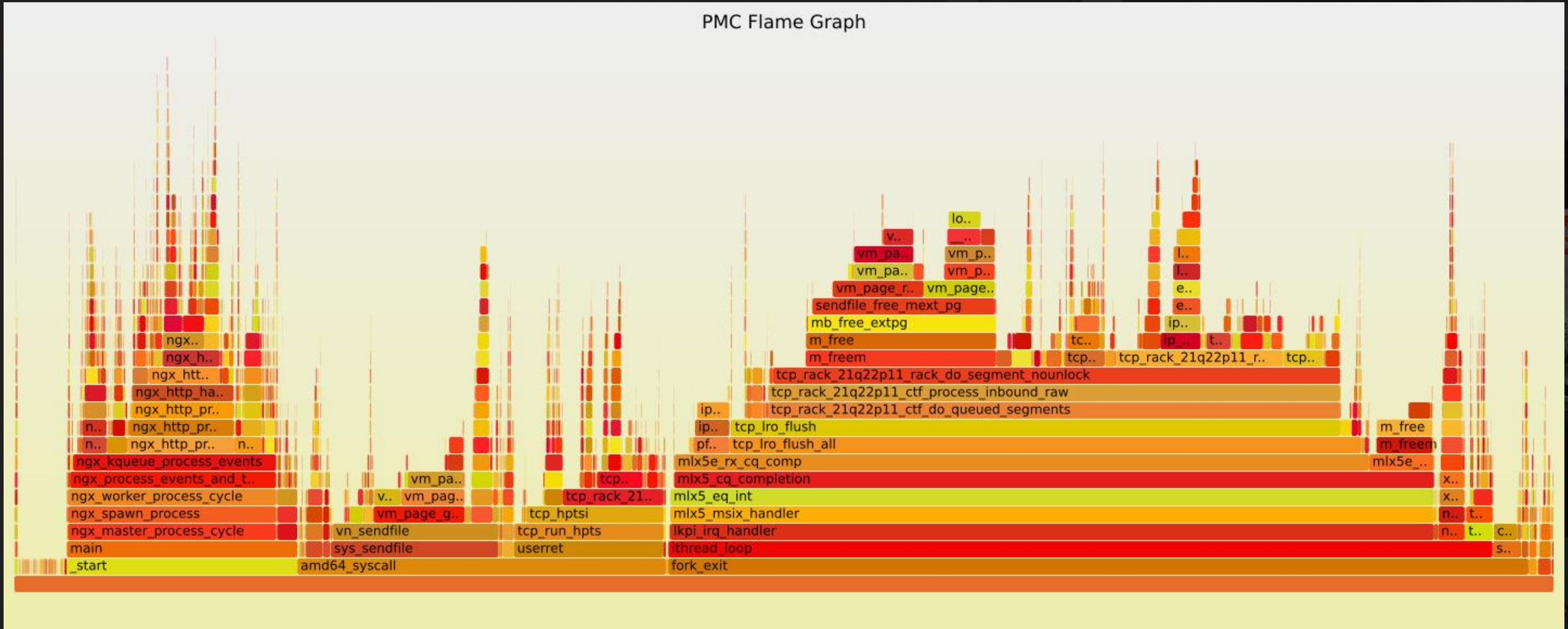
Measurement Metrics

- 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




PMC Flame Graph
lo..
v..
vm_pa..
vm_p..
vm_pa..
vm_p..
vm_page_r..
vm_page..
sendfile_free_mext_pg
mb_free_extpg
m_free
m_freem
l..
l..
e..
e..
ip..
ip..
t..
tc..
tcp..
tcp_rack_21q22p11_r..
tcp..
tcp_rack_21q22p11_rack_do_segment_nounlock
tcp_rack_21q22p11_ctf_process_inbound_raw
tcp_rack_21q22p11_ctf_do_queued_segments
ip..
ip..
tcp_lro_flush
pf..
tcp_lro_flush_all
m_free
m_freem
mlx5e_rx_cq_comp
mlx5e_..
mlx5_cq_completion
mlx5_eq_int
mlx5_msix_handler
lkpi_irq_handler
ithread_loop
fork_exit
x..
x..
n..
t..
n..
t..
c..
s..
ngx..
ngx_h..
ngx_htt..
ngx_http_ha..
ngx_http_pr..
n..
ngx_http_pr..
n..
ngx_http_pr..
n..
ngx_kqueue_process_events
ngx_process_events_and_t..
ngx_worker_process_cycle
ngx_spawn_process
ngx_master_process_cycle
main
_start
vm_pa..
v..
vm_pag..
vm_page_g..
vn_sendfile
sys_sendfile
amd64_syscall
tcp..
tcp_rack_21..
tcp_hptsi
tcp_run_hpts
userret

Section 1:

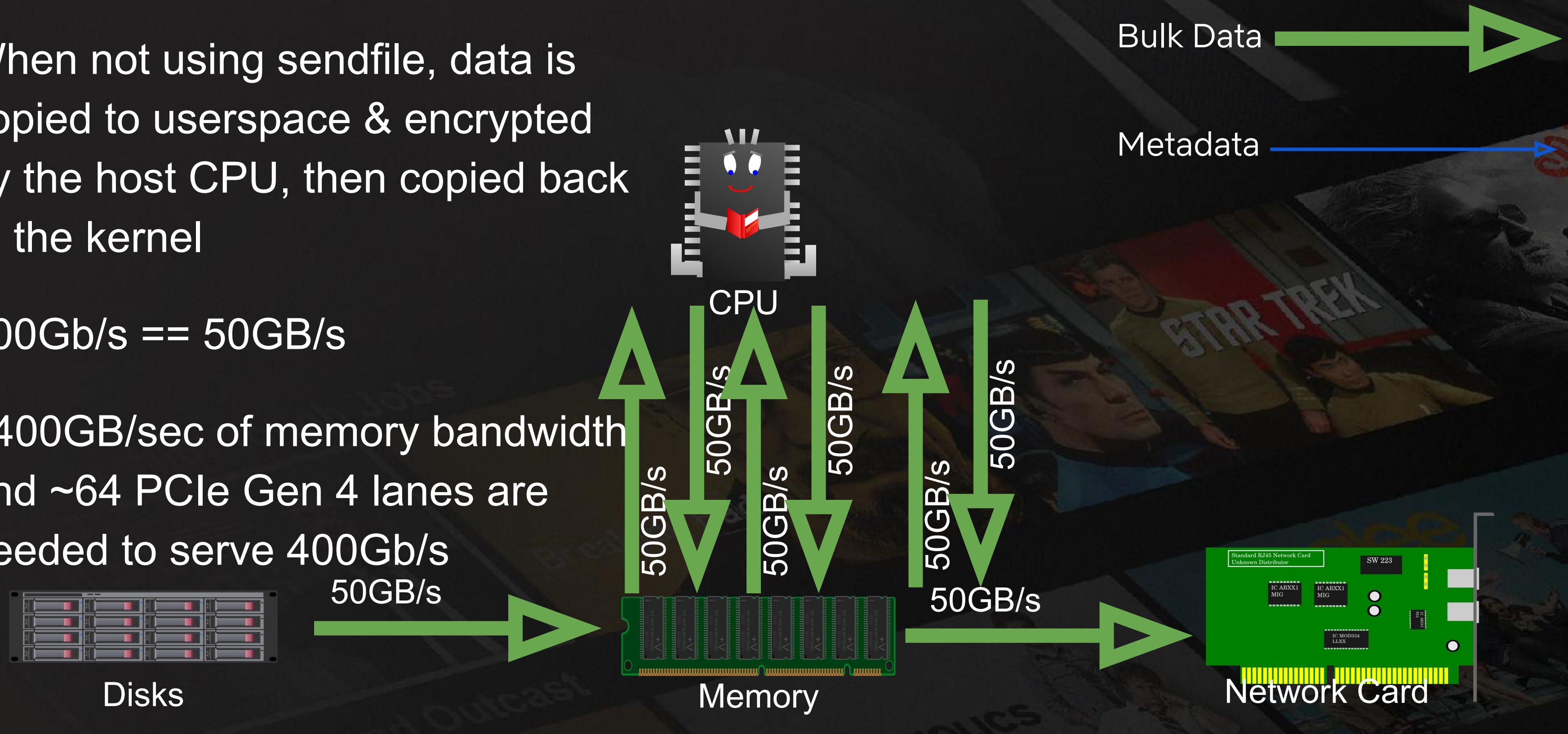
Sendfile & kTLS

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



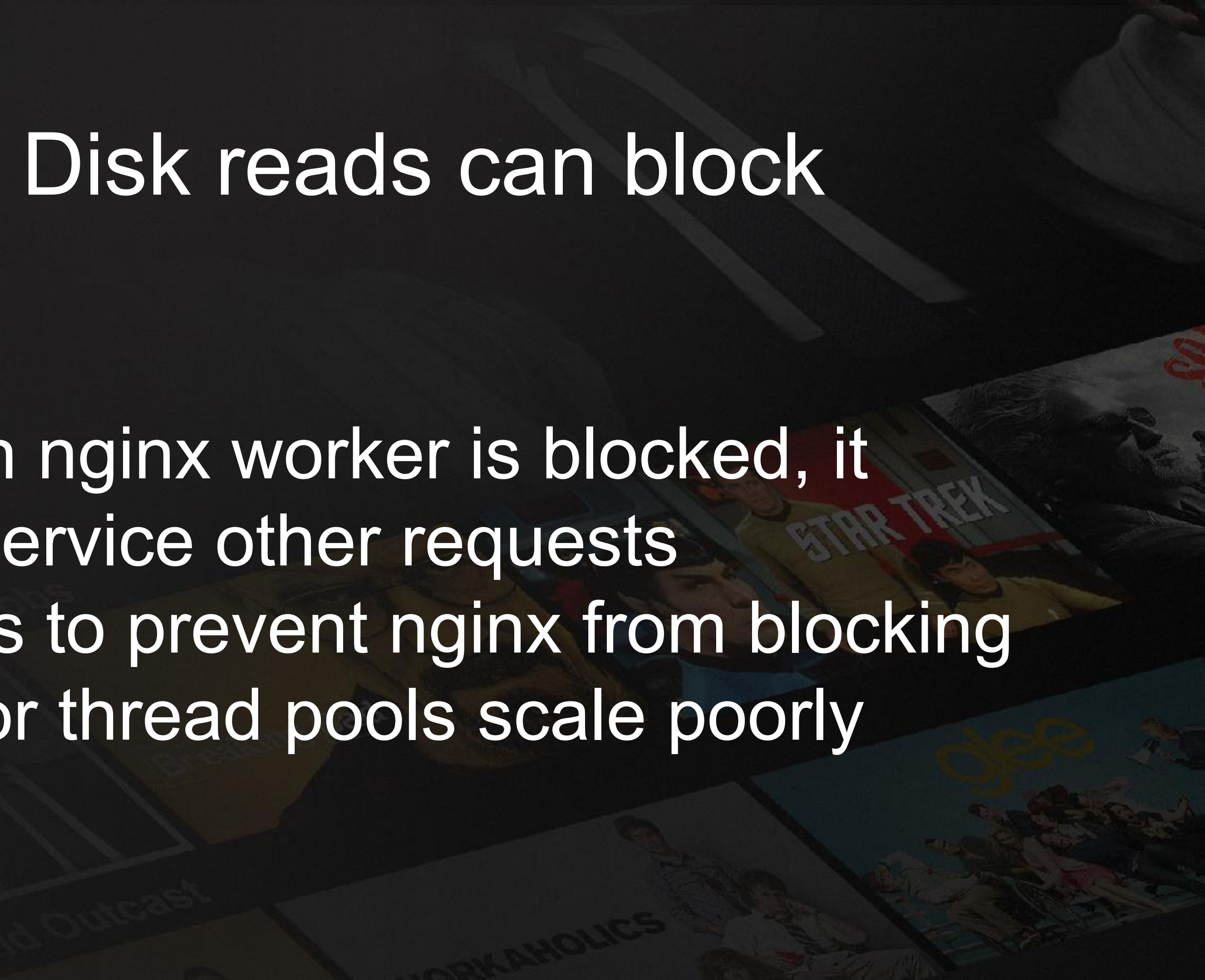
What is sendfile?

- 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

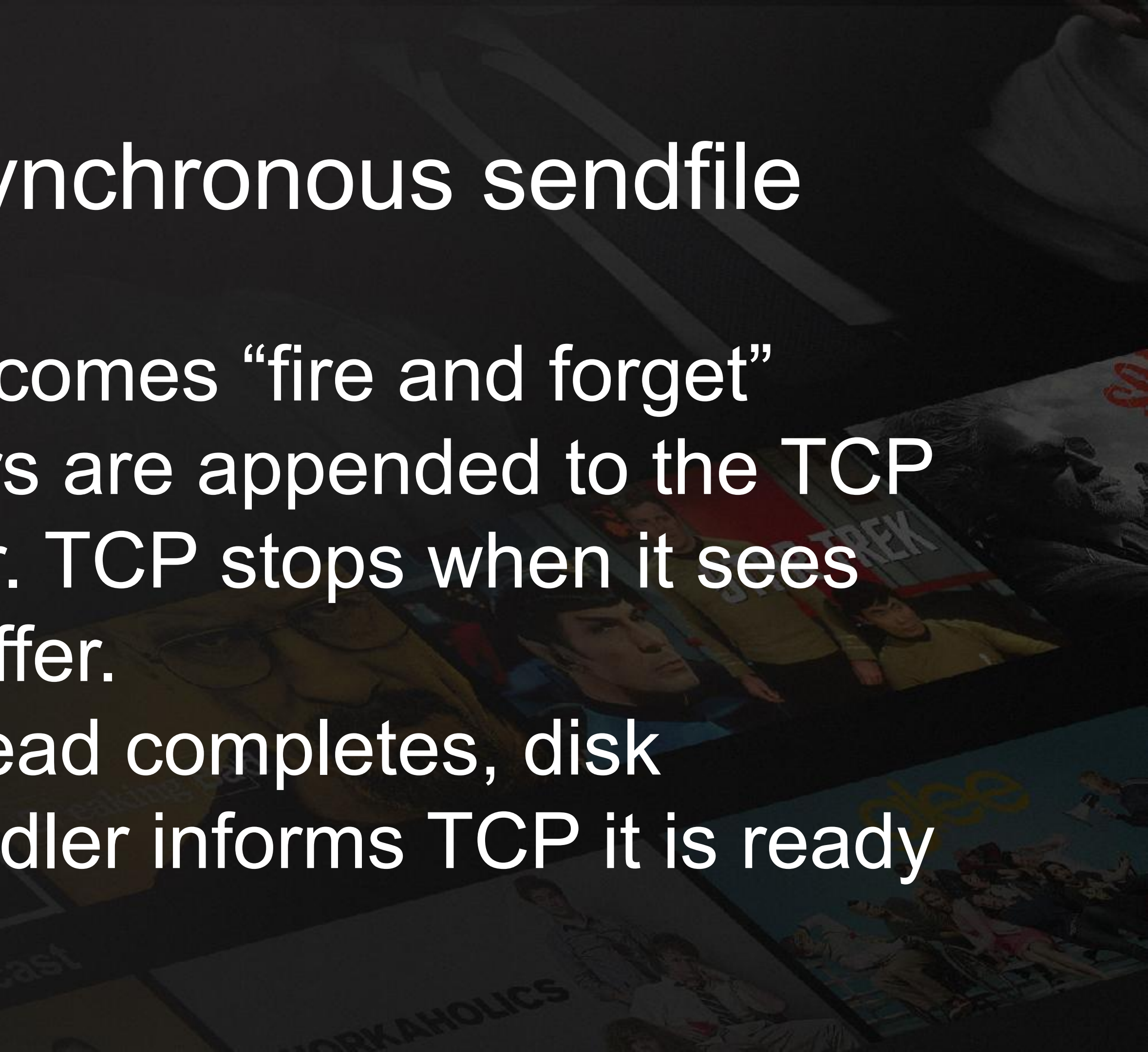
NETFLIX

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

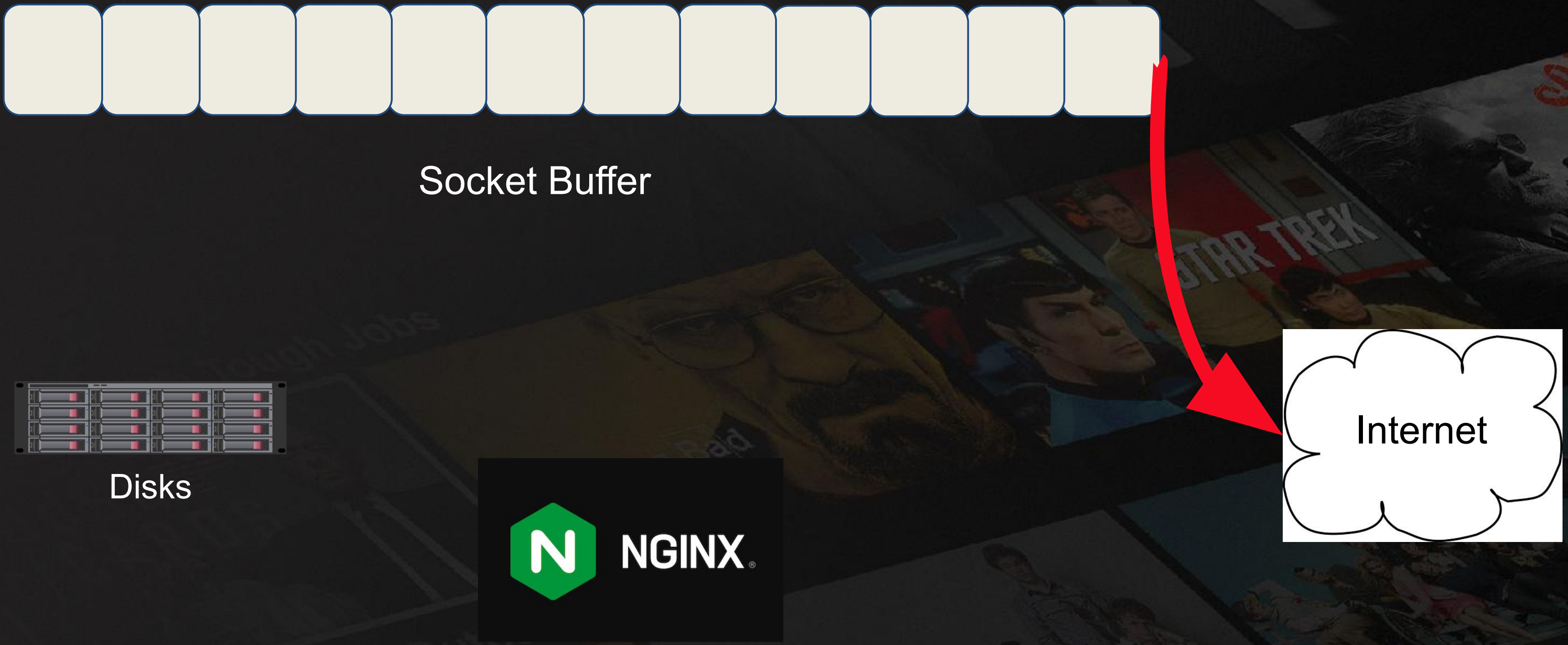


Solution: Asynchronous sendfile



- 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



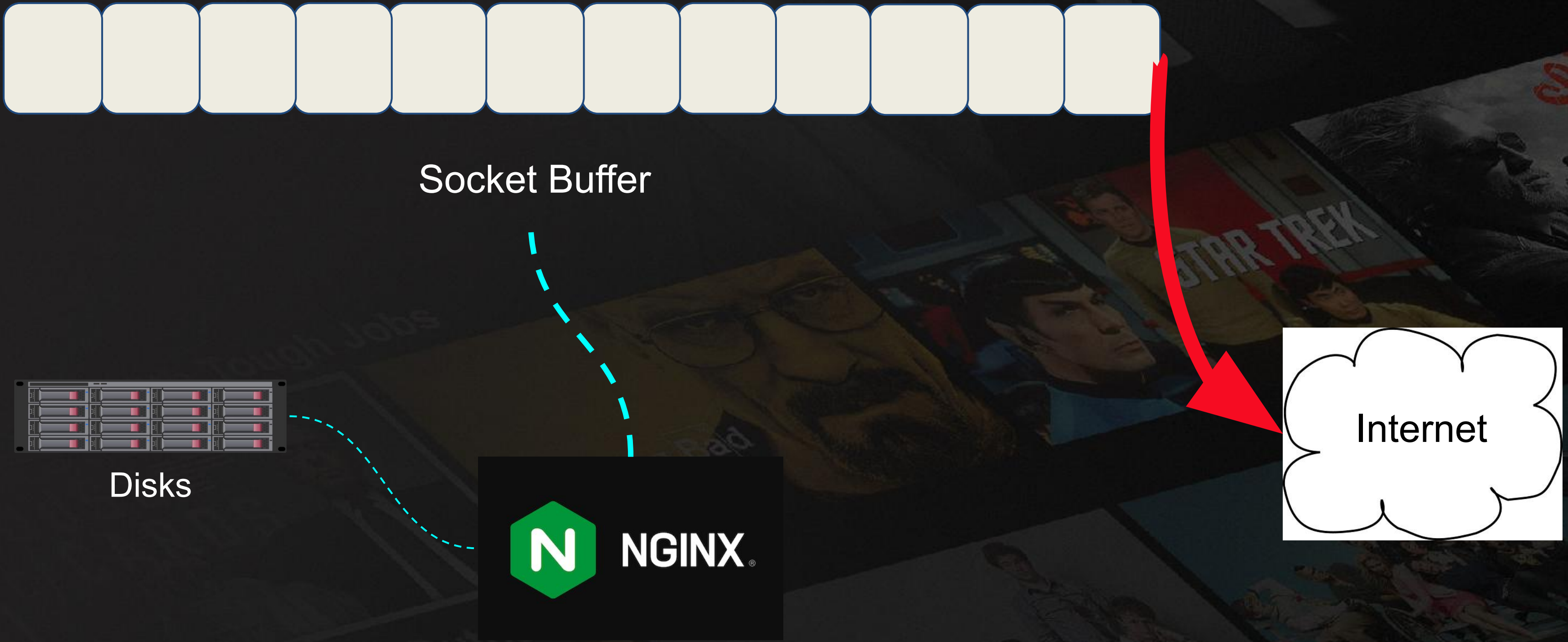
Socket Buffer

Disks

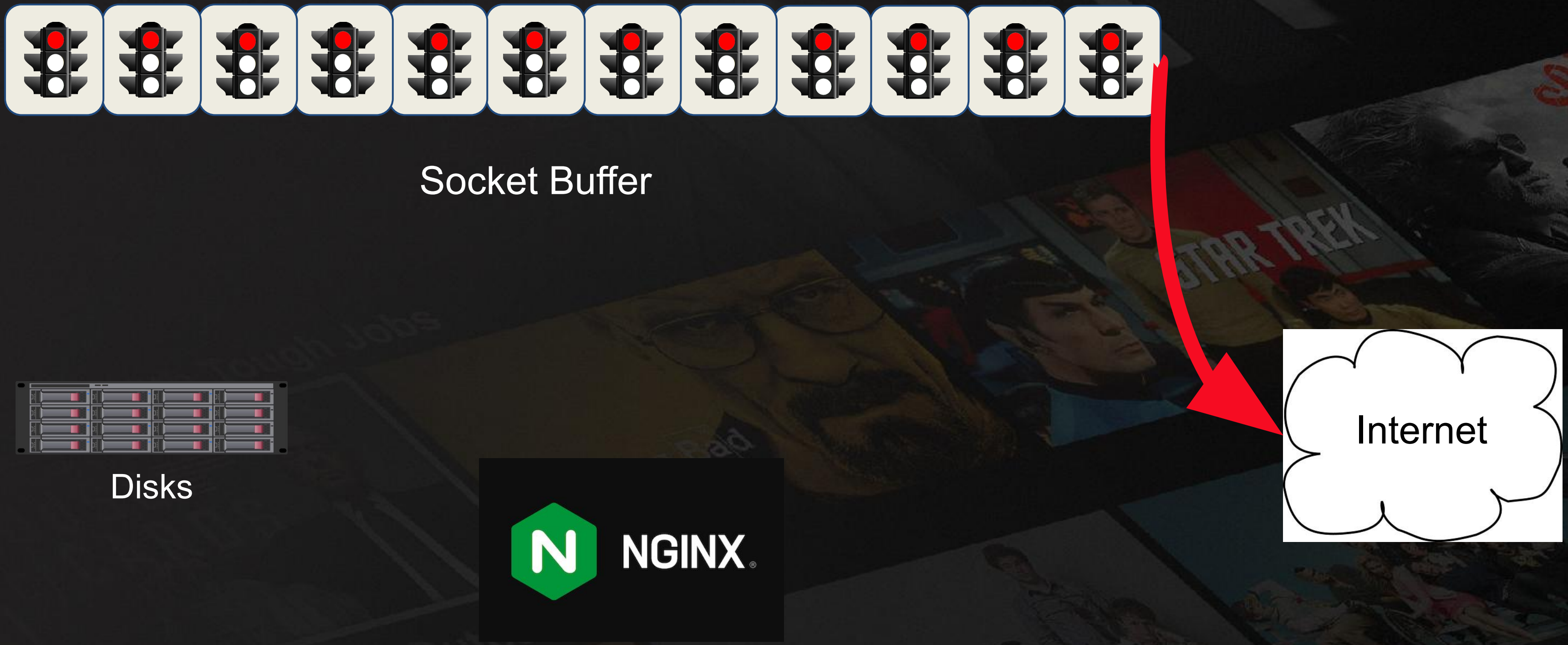
Internet

NETFLIX

Asynchronous sendfile



Asynchronous sendfile



Socket Buffer

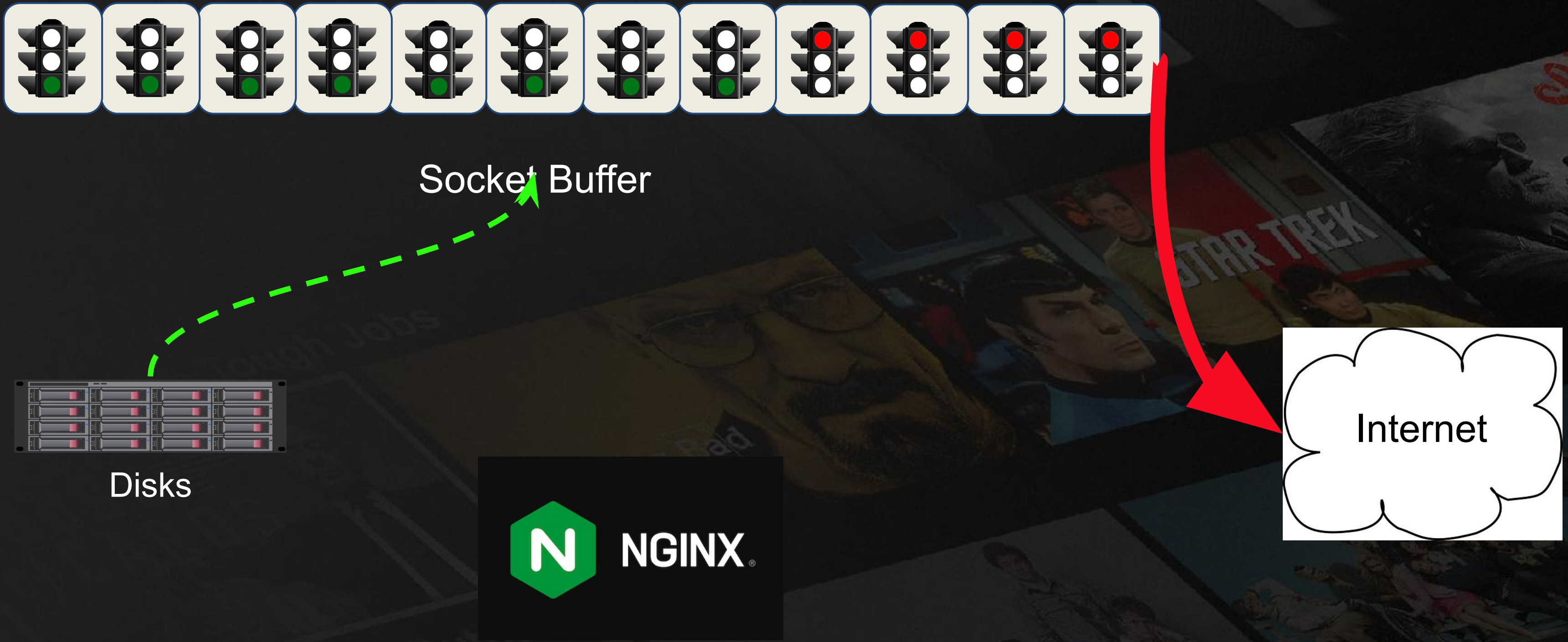
Disks

NGINX

Internet

NETFLIX

Asynchronous sendfile

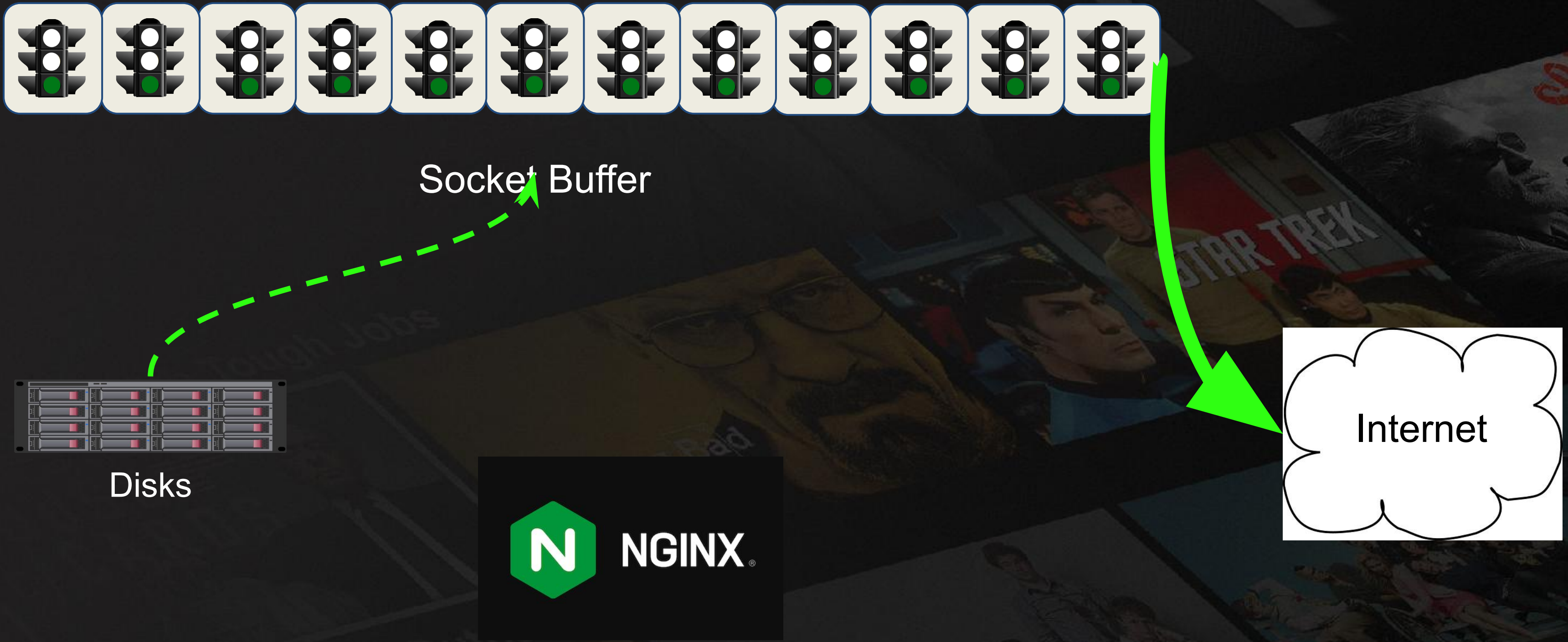


Socket Buffer

Disks

Internet

Asynchronous sendfile



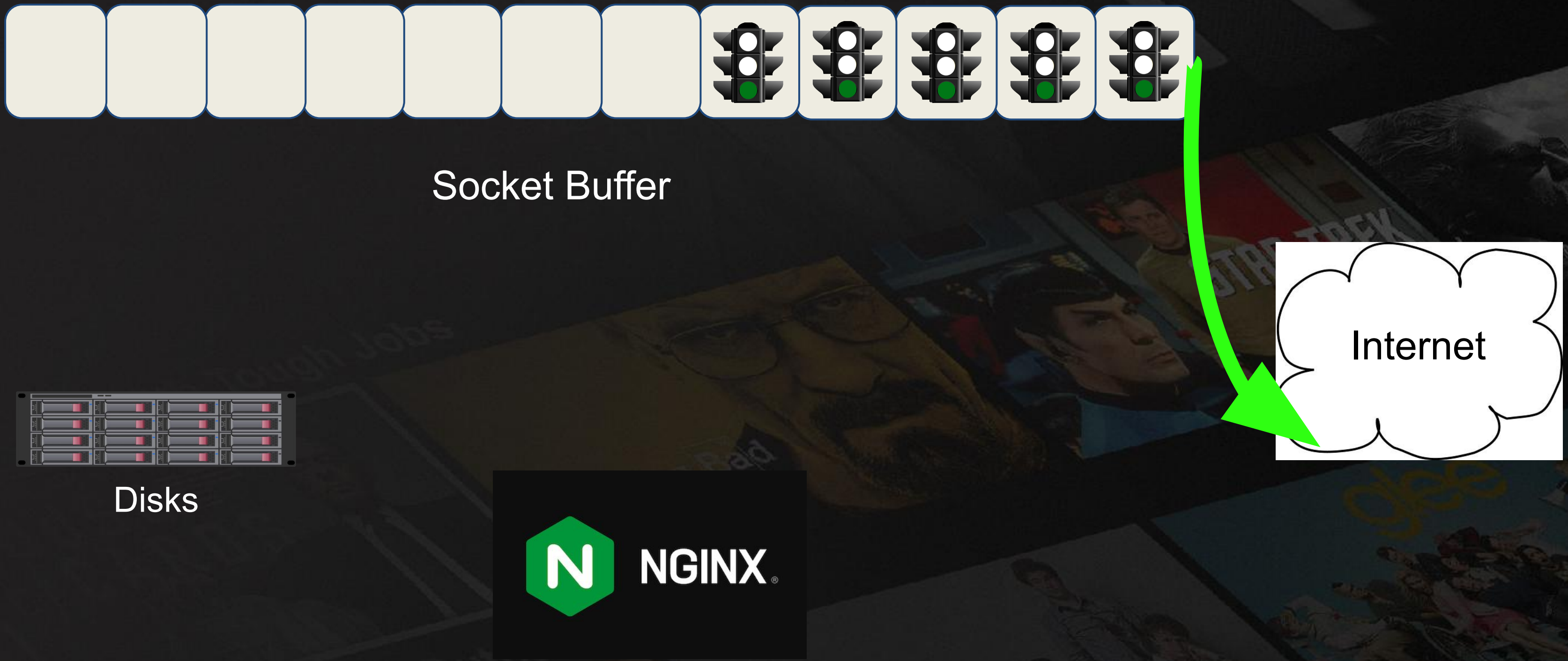
Socket Buffer

Disks

NGINX

Internet

Asynchronous sendfile

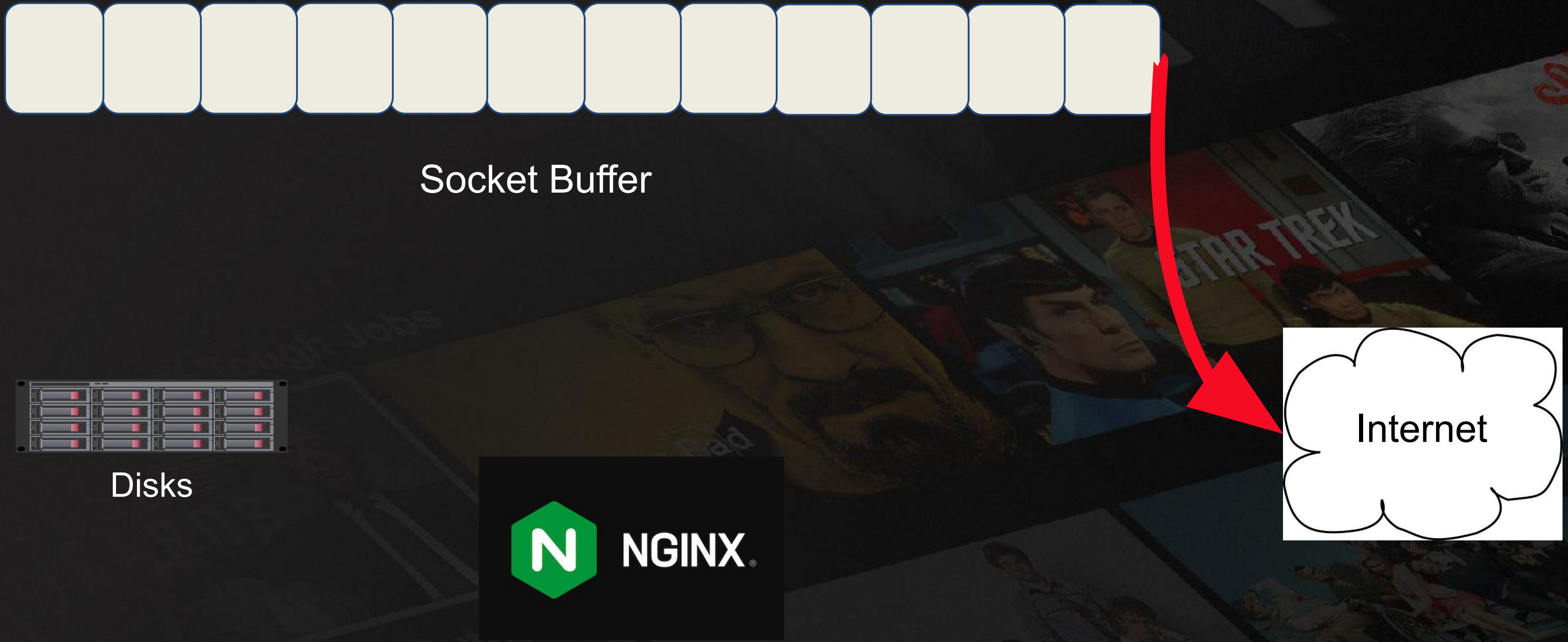


Socket Buffer

Internet

Disks

Asynchronous sendfile



Socket Buffer

Disks

Internet

What is kTLS?

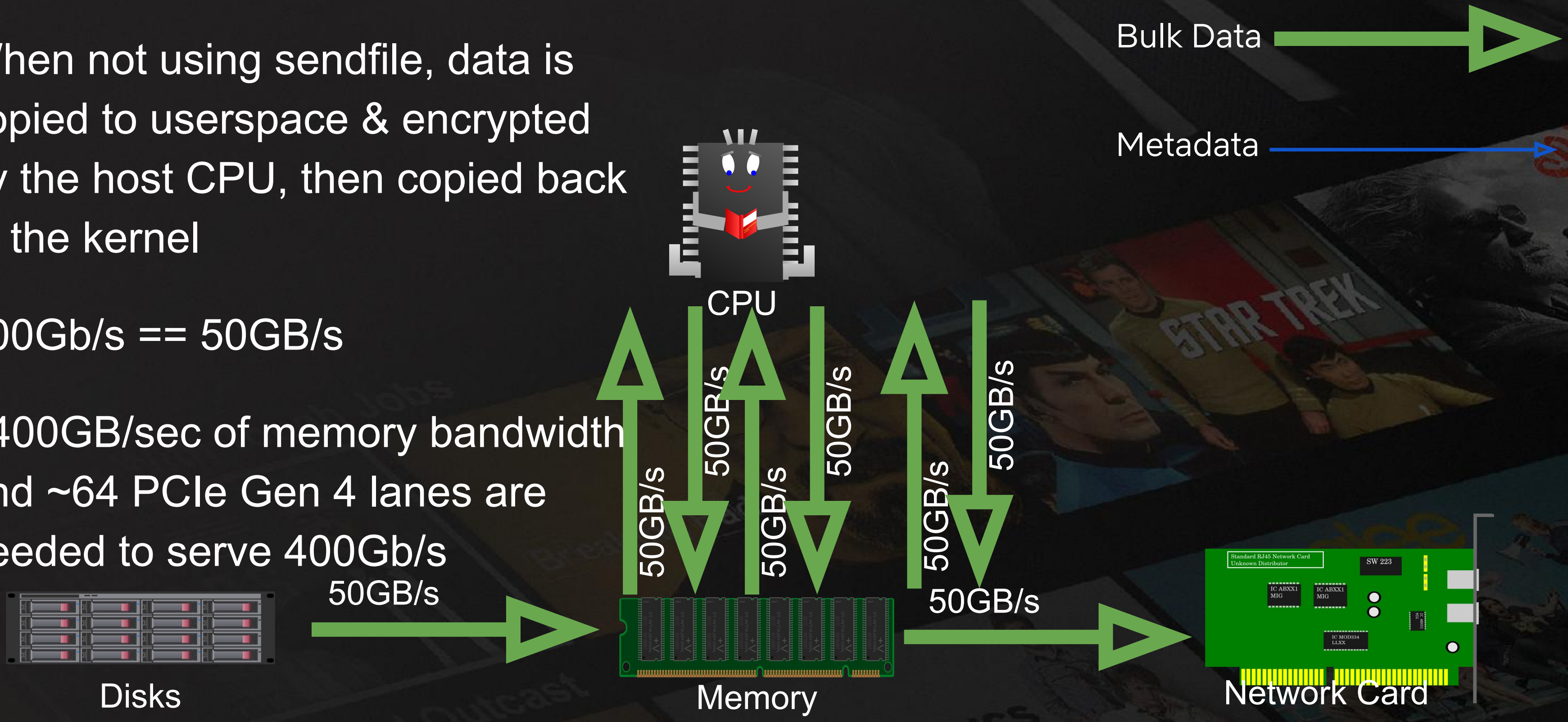
- Bulk crypto is moved into the kernel
    - Handshakes are still done in userspace
    - Required for async sendfile based dataflow with no copies or context switches.
- Doing crypto in the kernel almost quadruples CPU efficiency
- Originated at Netflix

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

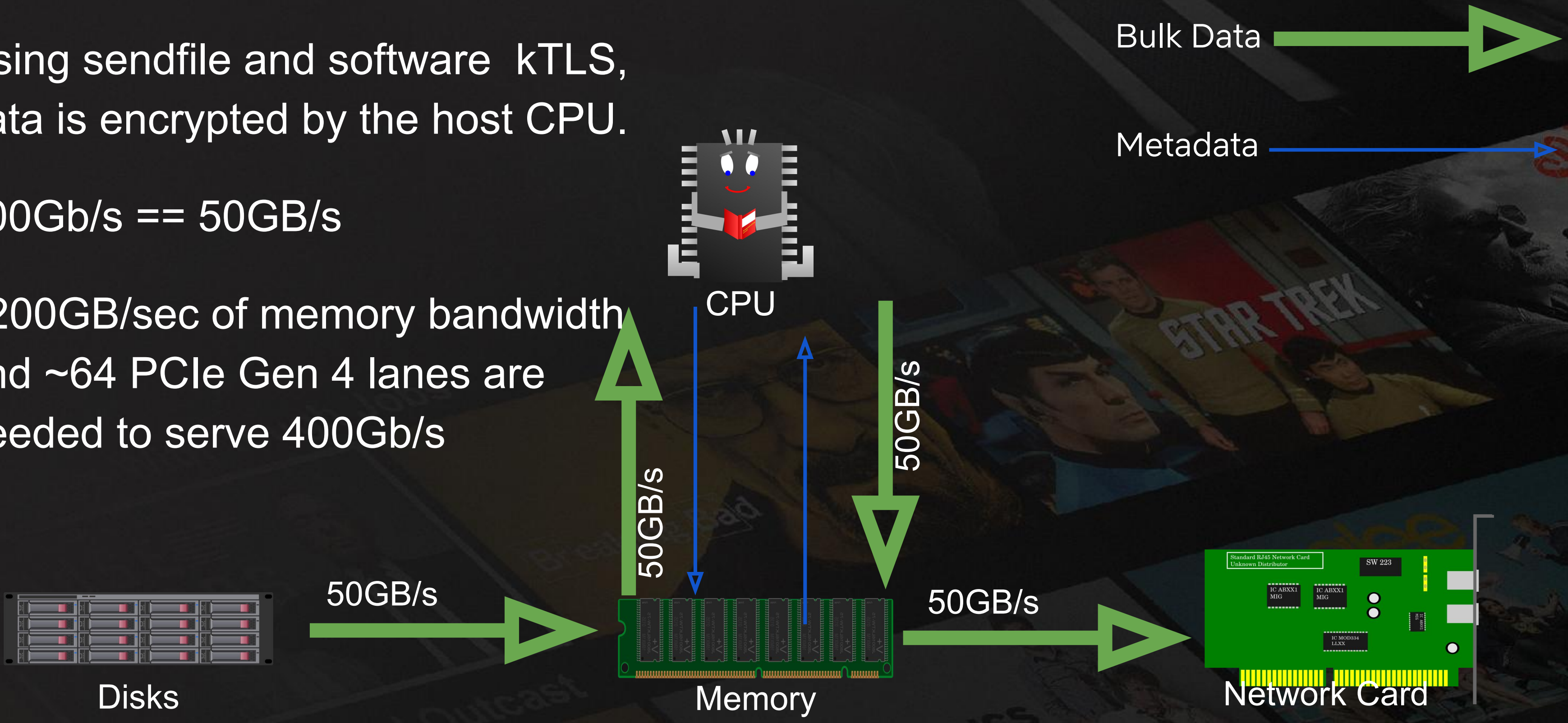


Netflix 400Gb/s Video Serving Data Flow

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



Bulk Data

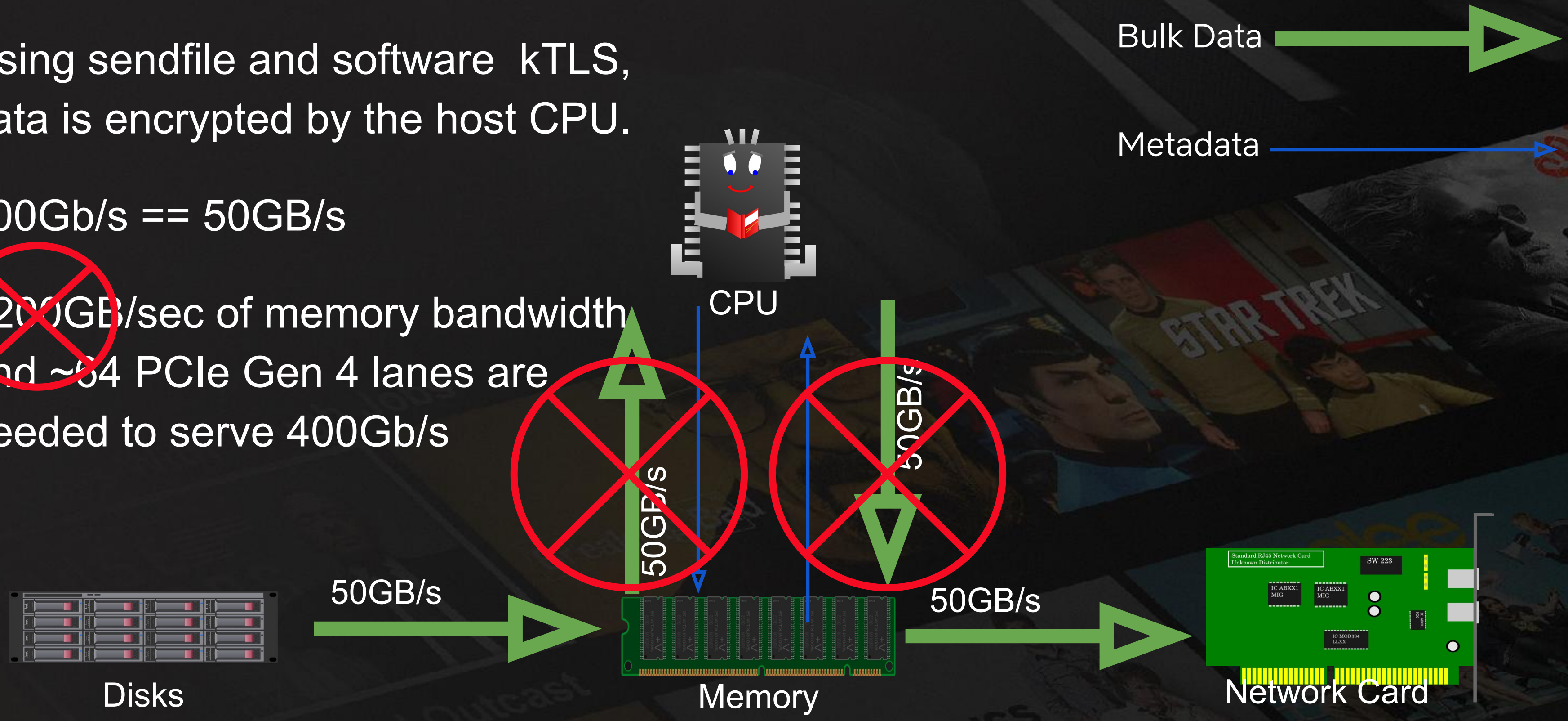
Metadata

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

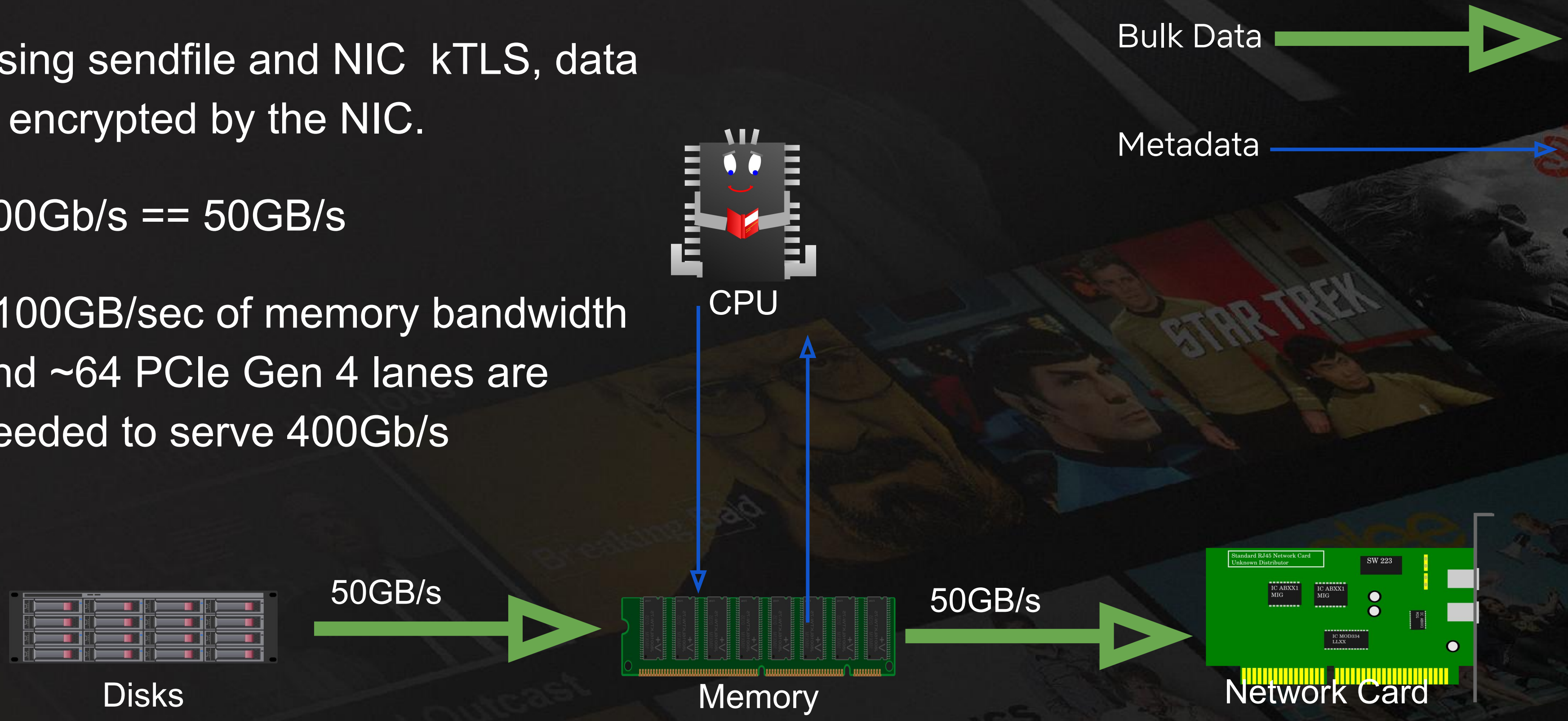


Netflix 400Gb/s Video Serving Data Flow

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

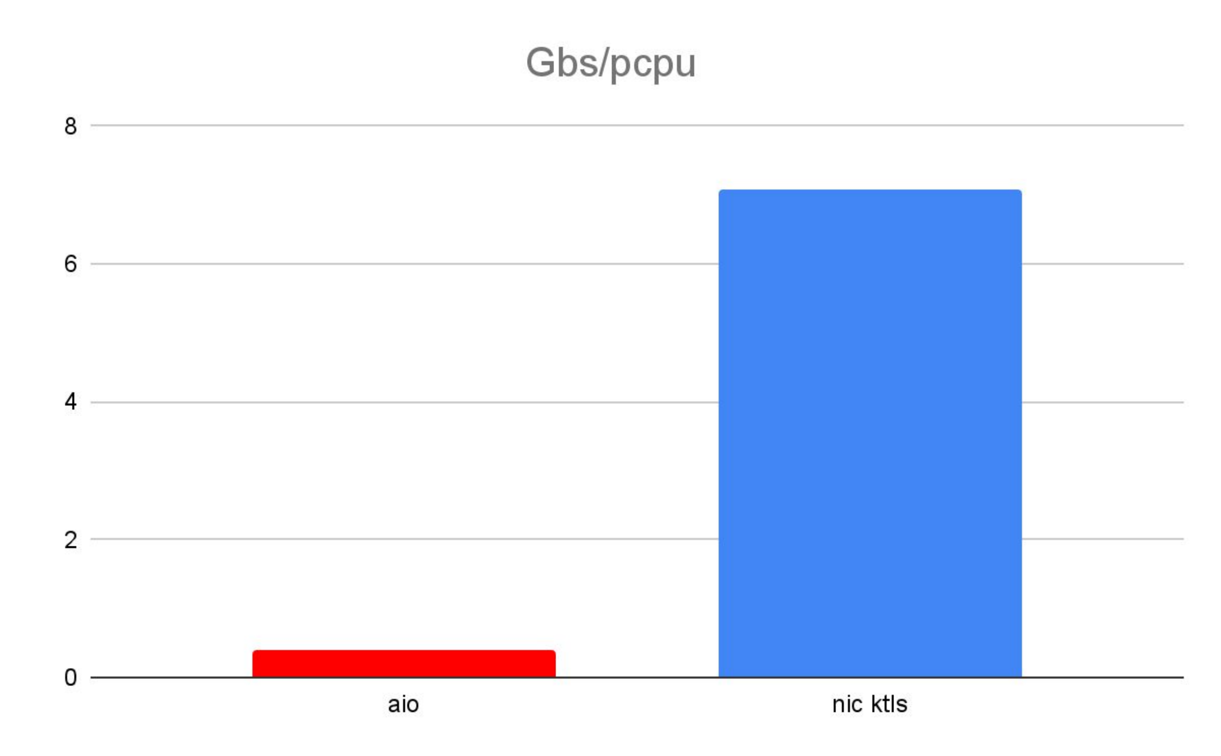


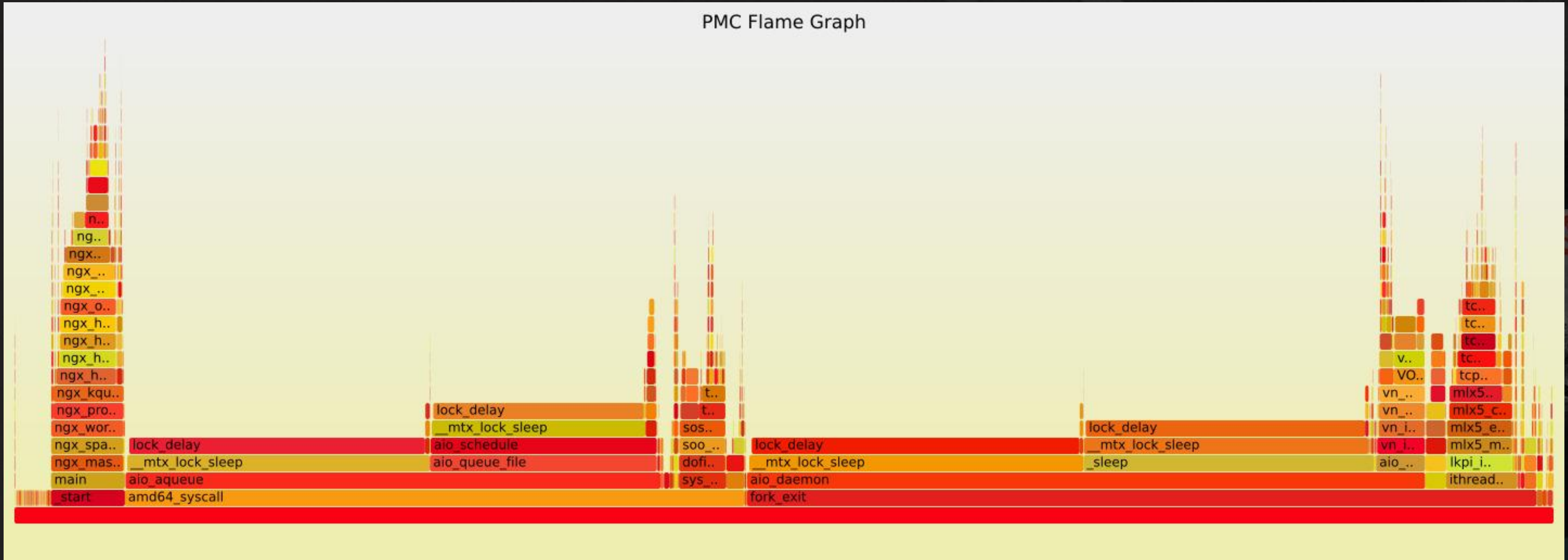
Disable kTLS (and async sendfile)

- 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.

NETFLIX

Disable kTLS (and async sendfile)



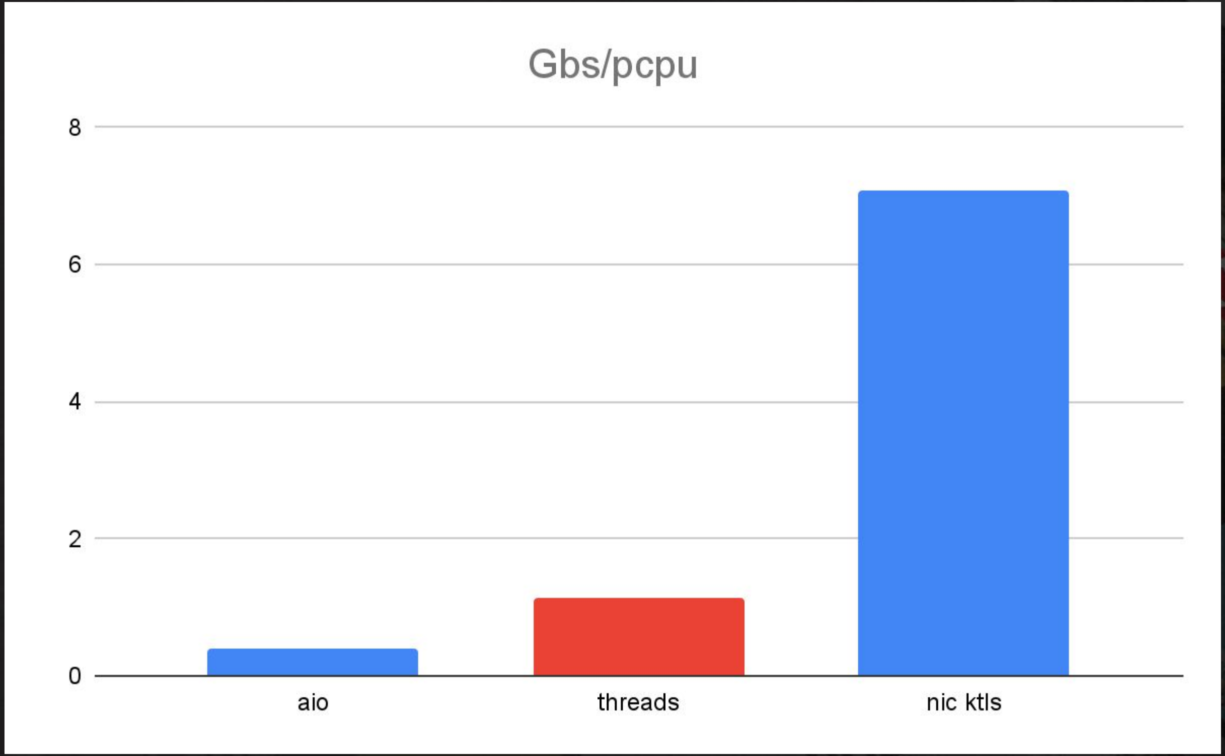


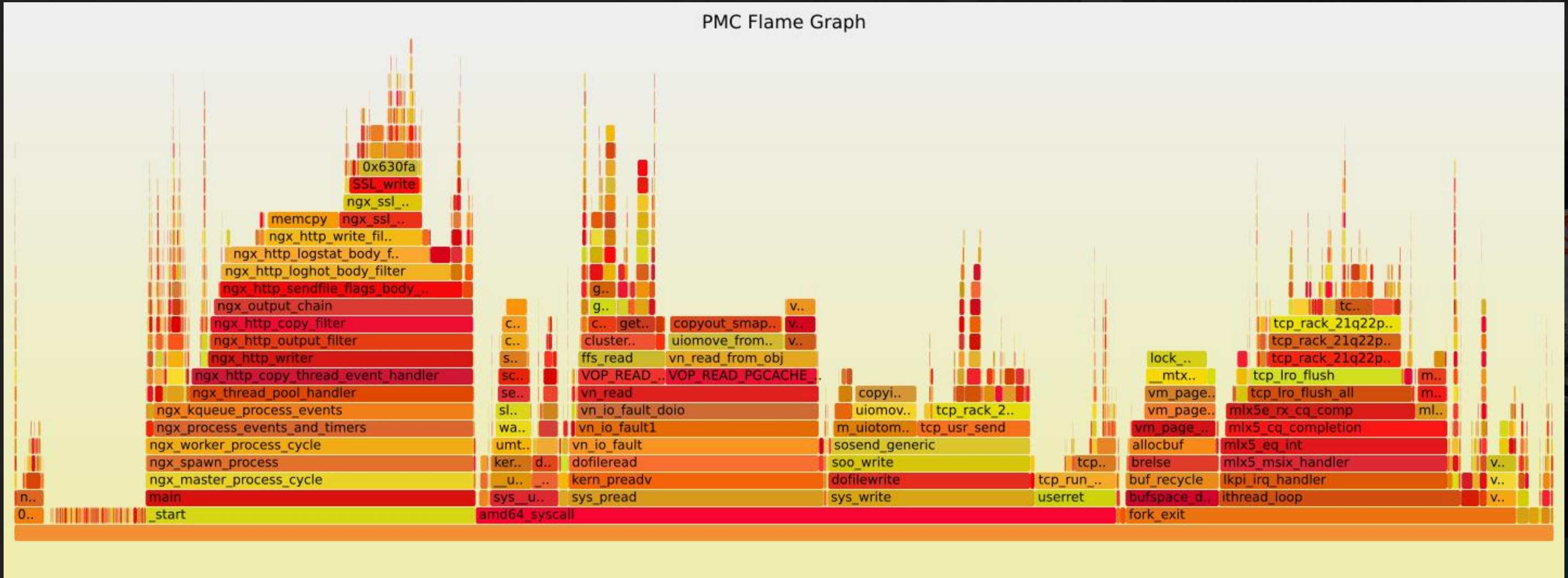
PMC Flame Graph

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 ngix
    - Crypto in userspace SSL
    - Extra memcpy in nginx for SSL
    - Copy in data to kernel from nginx

Disable kTLS (and async sendfile)



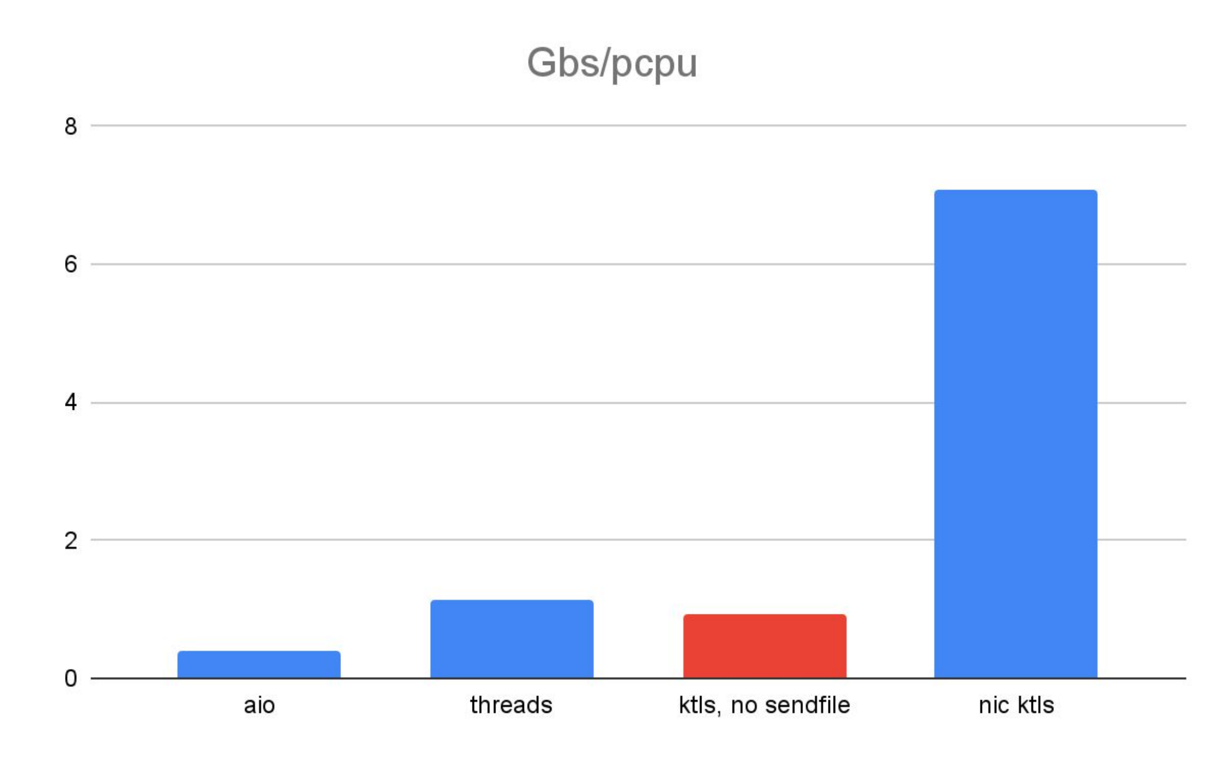


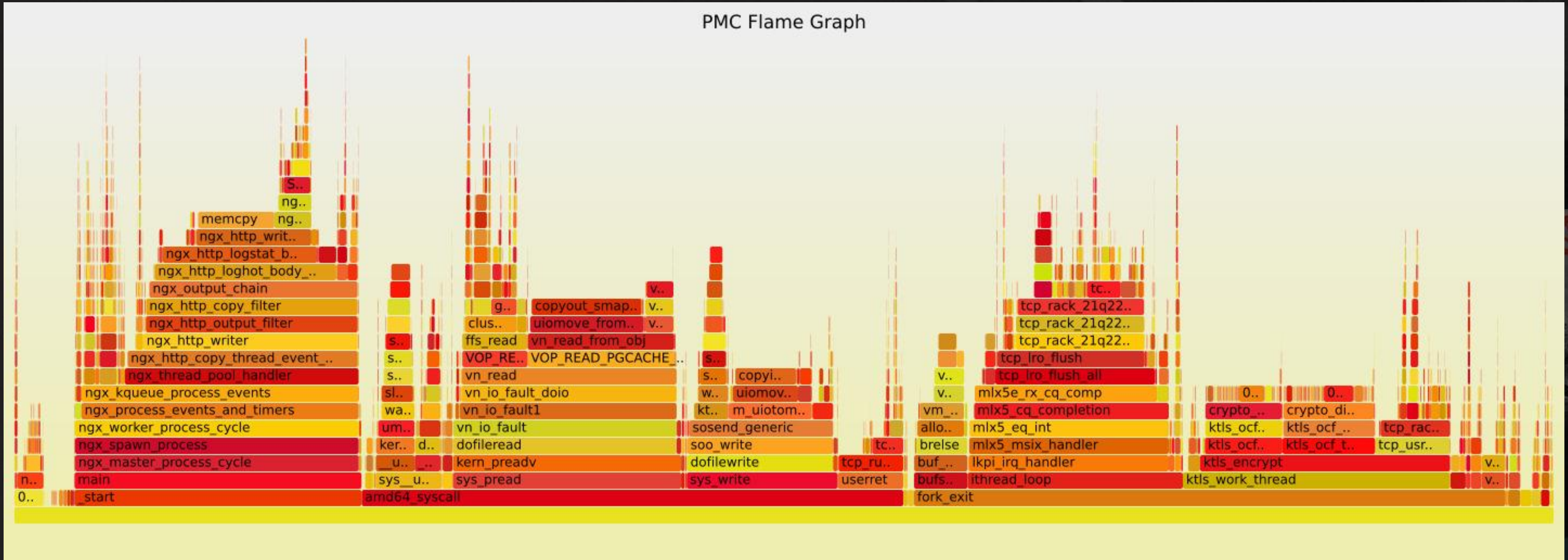
PMC Flame Graph

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 ngix
    - Extra memcpy in nginx for SSL
    - Copy in data to kernel from nginx
    - Crypto in-place in kernel

Disable sendfile (but use kTLS)

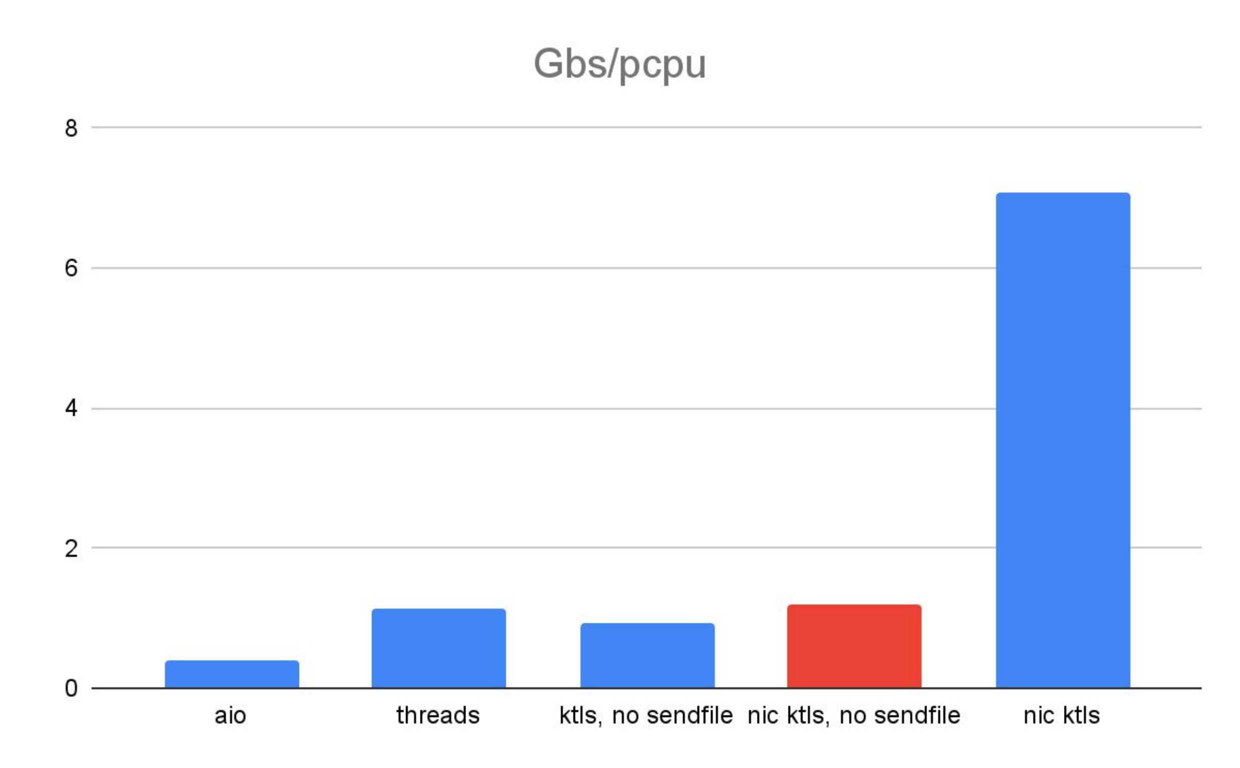


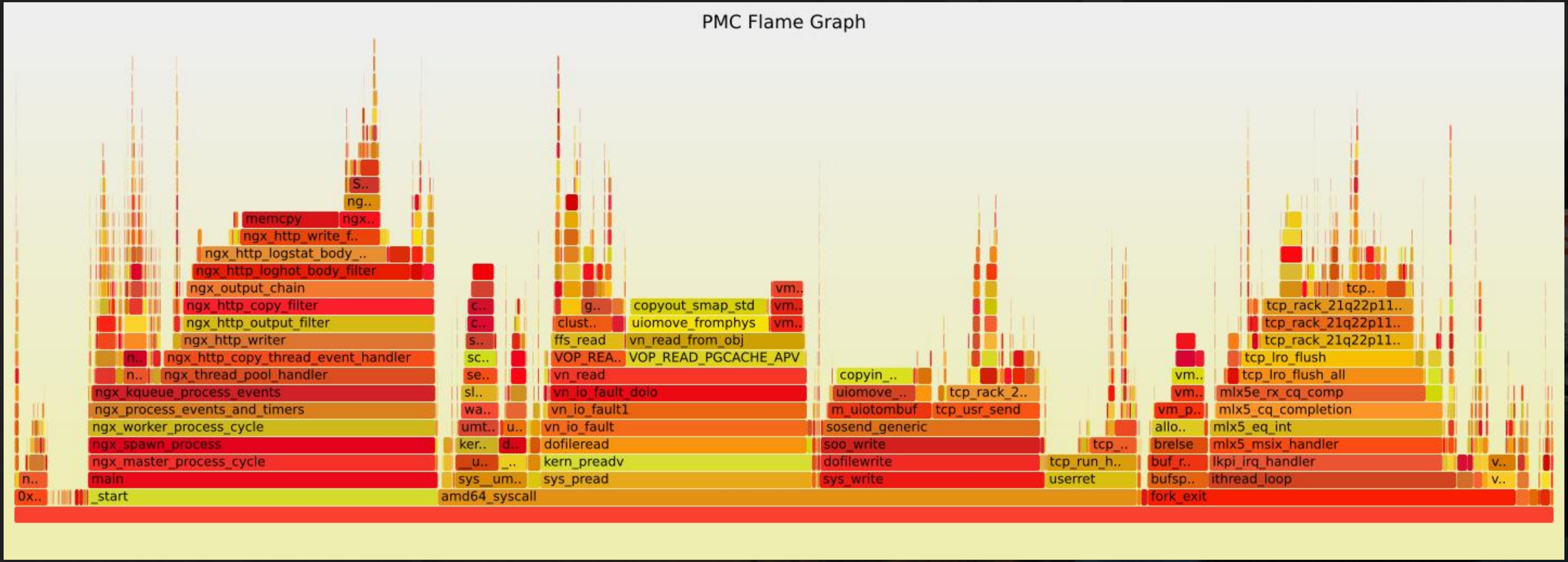

PMC Flame Graph
S..
ng..
memcpy
ng..
ngx_http_writ..
ngx_http_logstat_b..
ngx_http_loghot_body_..
ngx_output_chain
ngx_http_copy_filter
ngx_http_output_filter
ngx_http_writer
ngx_http_copy_thread_event_..
ngx_thread_pool_handler
ngx_kqueue_process_events
ngx_process_events_and_timers
ngx_worker_process_cycle
ngx_spawn_process
ngx_master_process_cycle
main
_start
n..
0..
s..
s..
s..
sl..
wa..
um..
ker..
d..
_u..
sys_u..
amd64_syscall
g..
copyout_smap..
v..
v..
clus..
uiomove_from..
v..
ffs_read
vn_read_from_obj
VOP_RE..
VOP_READ_PGCACHE_..
vn_read
vn_io_fault_doio
vn_io_fault1
vn_io_fault
dofileread
kern_preadv
sys_pread
s..
s..
copyi..
w..
uiomov..
kt..
m_uiotom..
sosend_generic
soo_write
dofilewrite
sys_write
tc..
tcp_ru..
userret
v..
v..
vm_..
allo..
brelse
buf_..
bufs..
fork_exit
tc..
tcp_rack_21q22..
tcp_rack_21q22..
tcp_rack_21q22..
tcp_lro_flush
tcp_lro_flush_all
mlx5e_rx_cq_comp
mlx5_cq_completion
mlx5_eq_int
mlx5_msix_handler
lkpi_irq_handler
ithread_loop
0..
0..
crypto_..
crypto_di..
ktls_ocf..
ktls_ocf_..
tcp_rac..
ktls_ocf..
ktls_ocf_t..
tcp_usr..
ktls_encrypt
v..
ktls_work_thread
v..

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 ngix
    - 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)




PMC Flame Graph

ISA-L

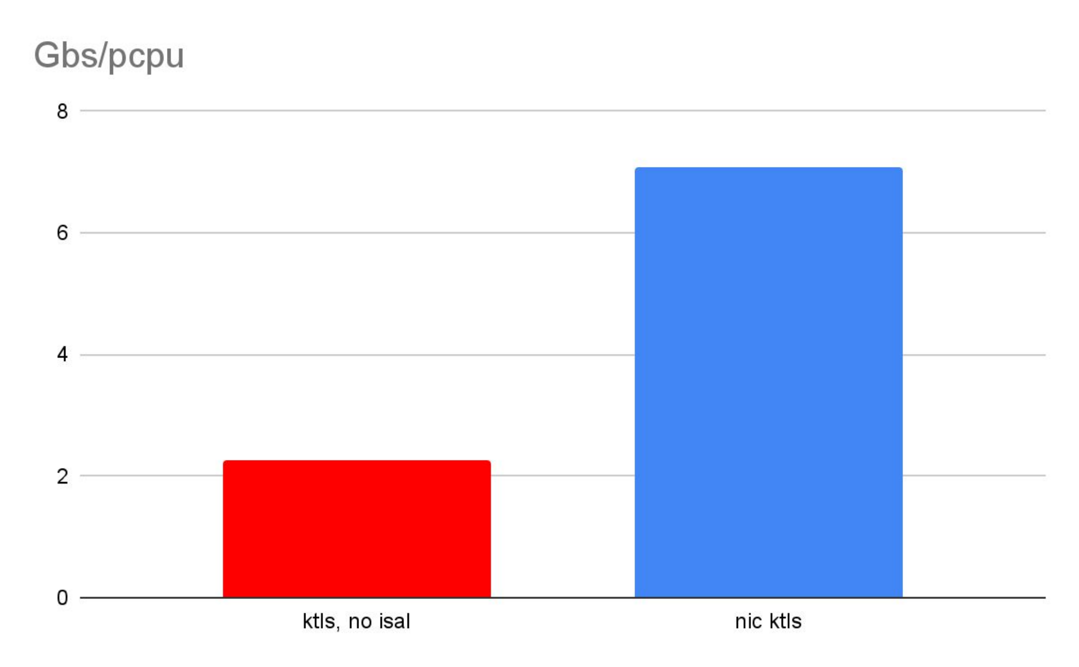
- Intel Intelligent Storage Acceleration Library
  - In ports as security/isal-kmod
  - Works well on AMD CPUs as well as Intel

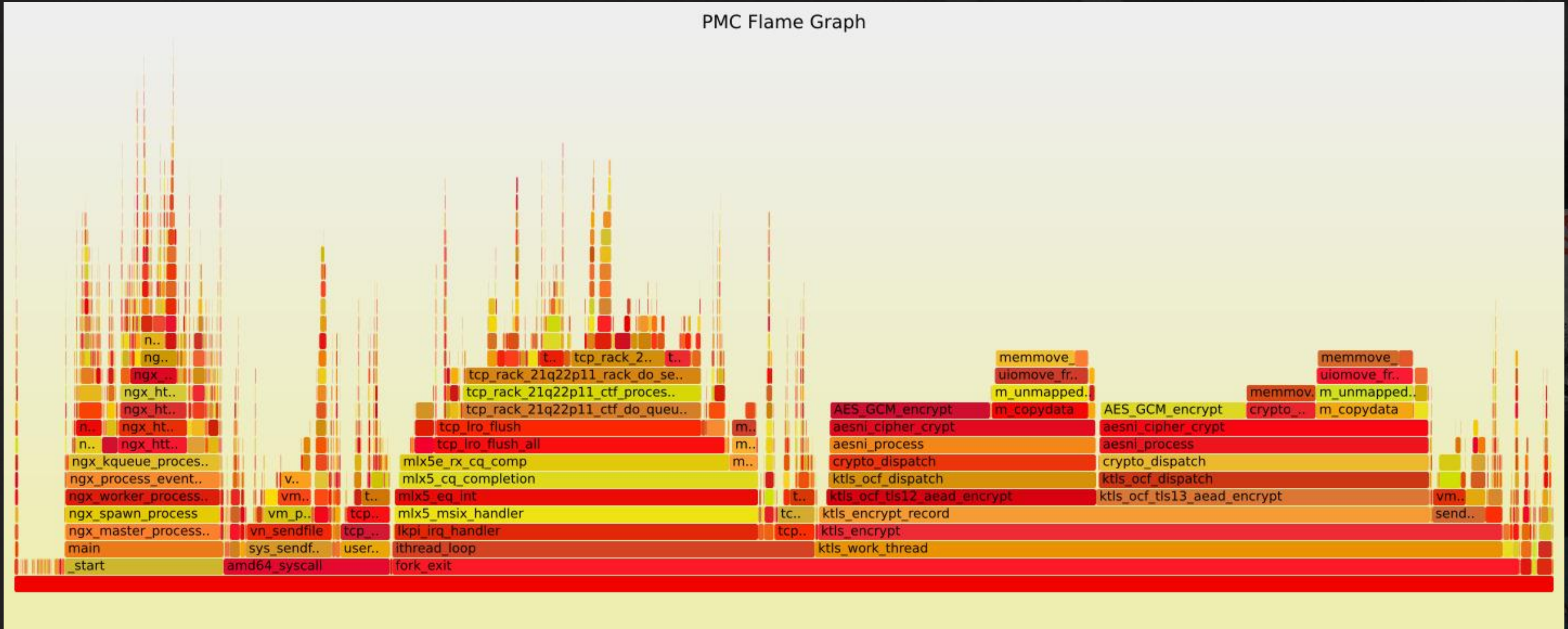
- Highly optimized accelerated AES block ciphers
  - Has options to use non-temporal instructions, which avoids read-modify-write cache miss when storing crypto results

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



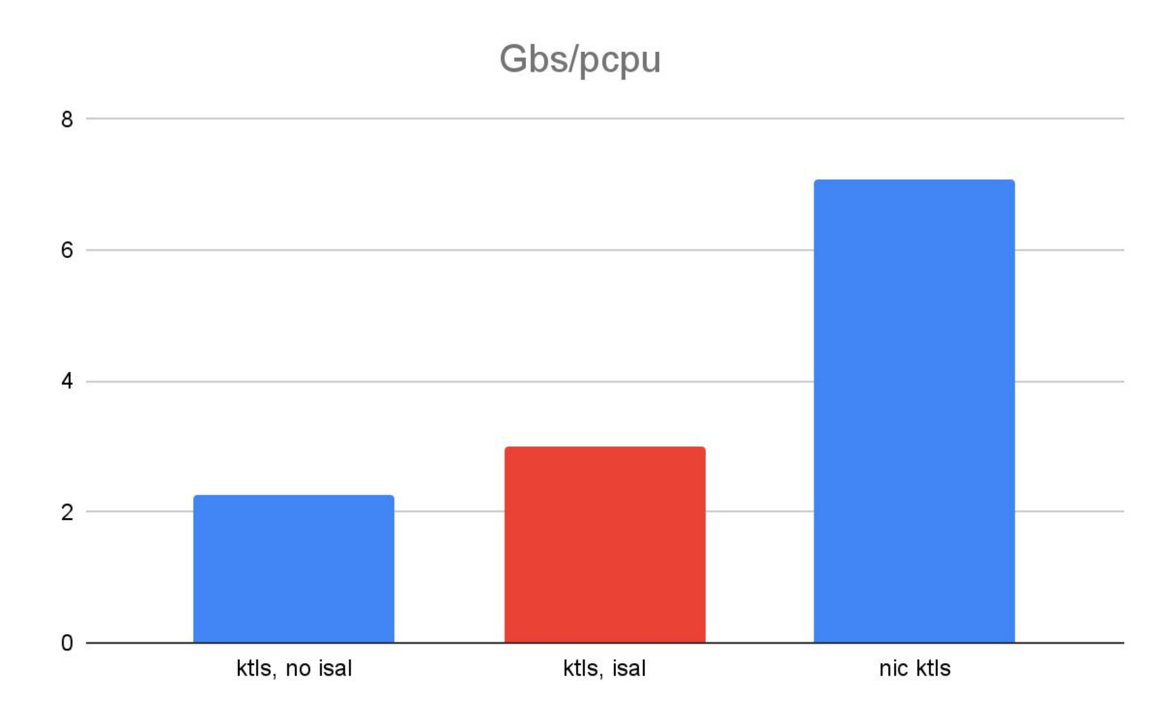

PMC Flame Graph

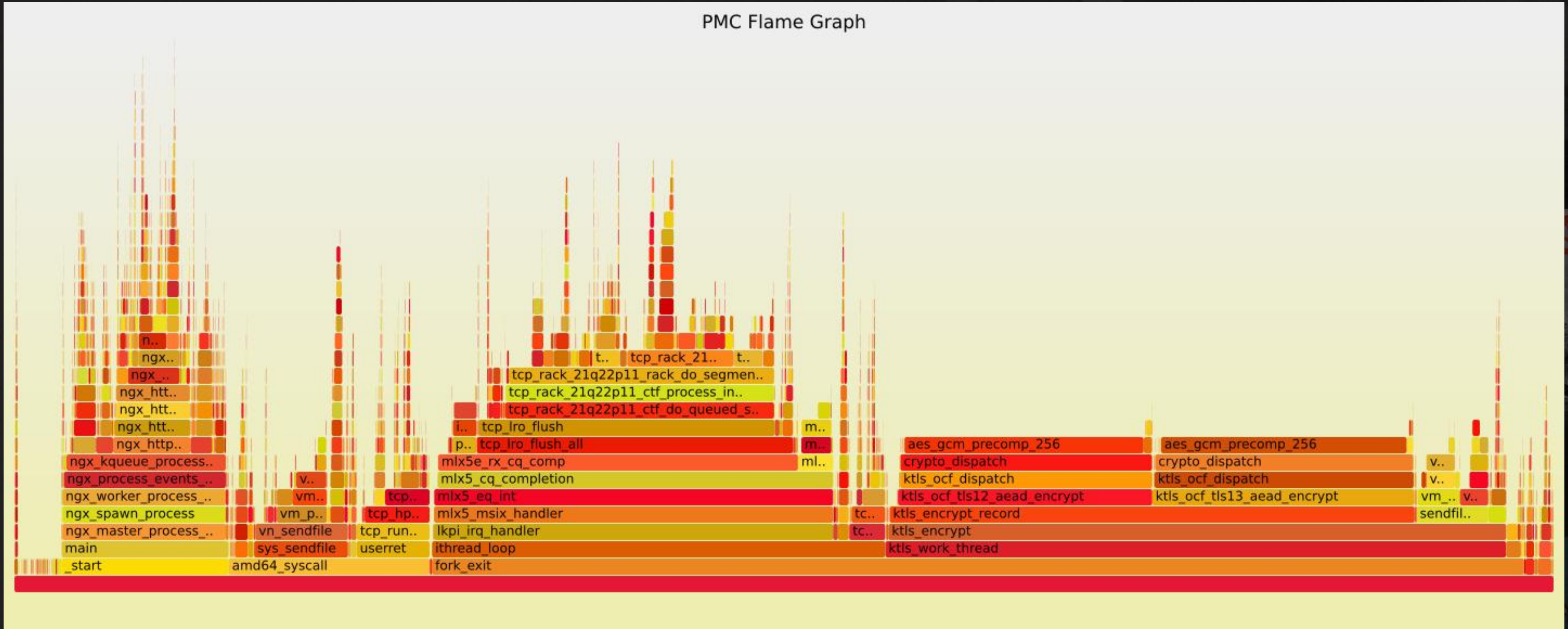
NETFLIX

Enable Sendfile & kTLS

- 240Gb/s, 80% CPU
    - CPU / Memory bound in ISA-L crypto

Enable Sendfile & kTLS

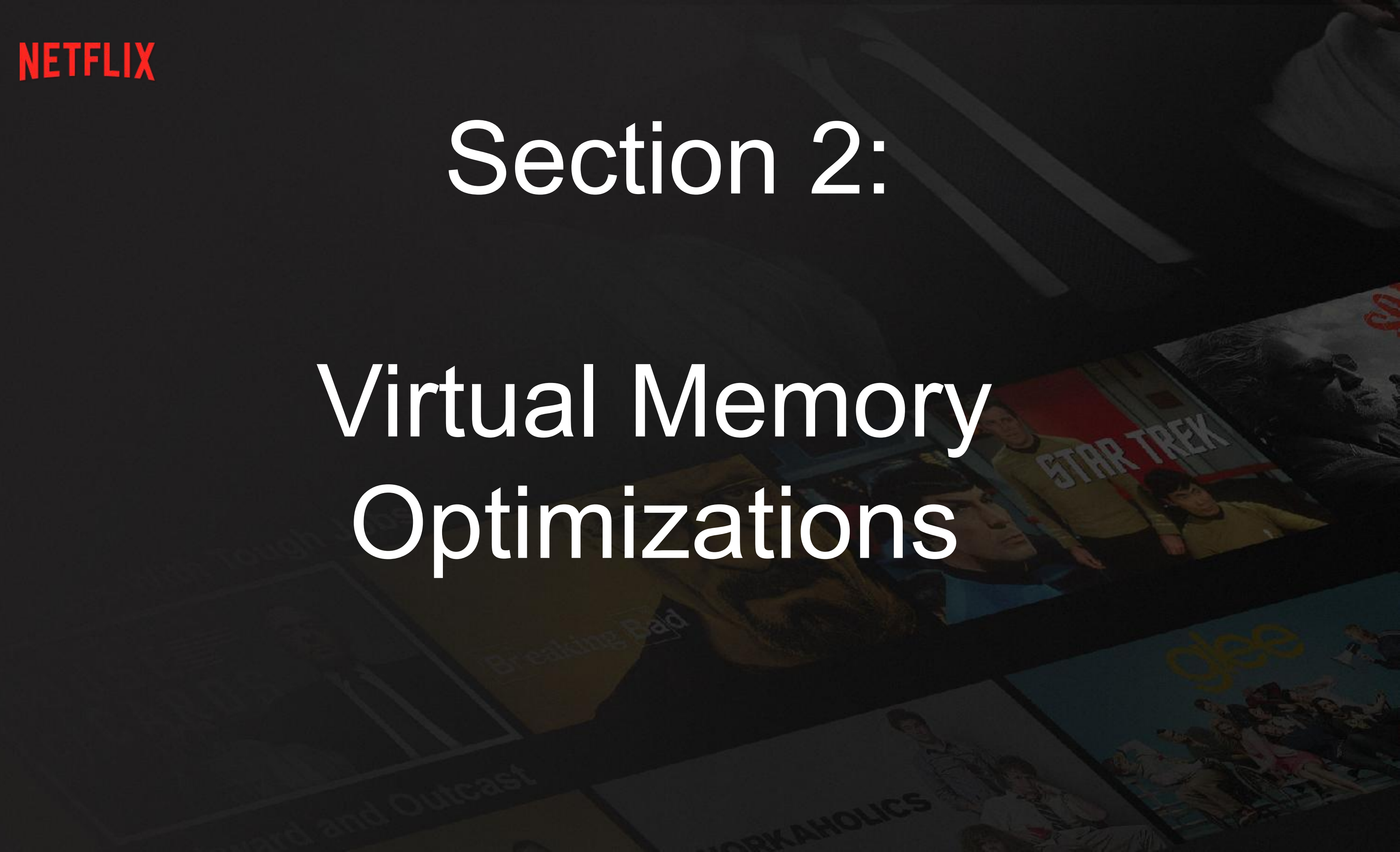




PMC Flame Graph

Section 2:

Virtual Memory Optimizations

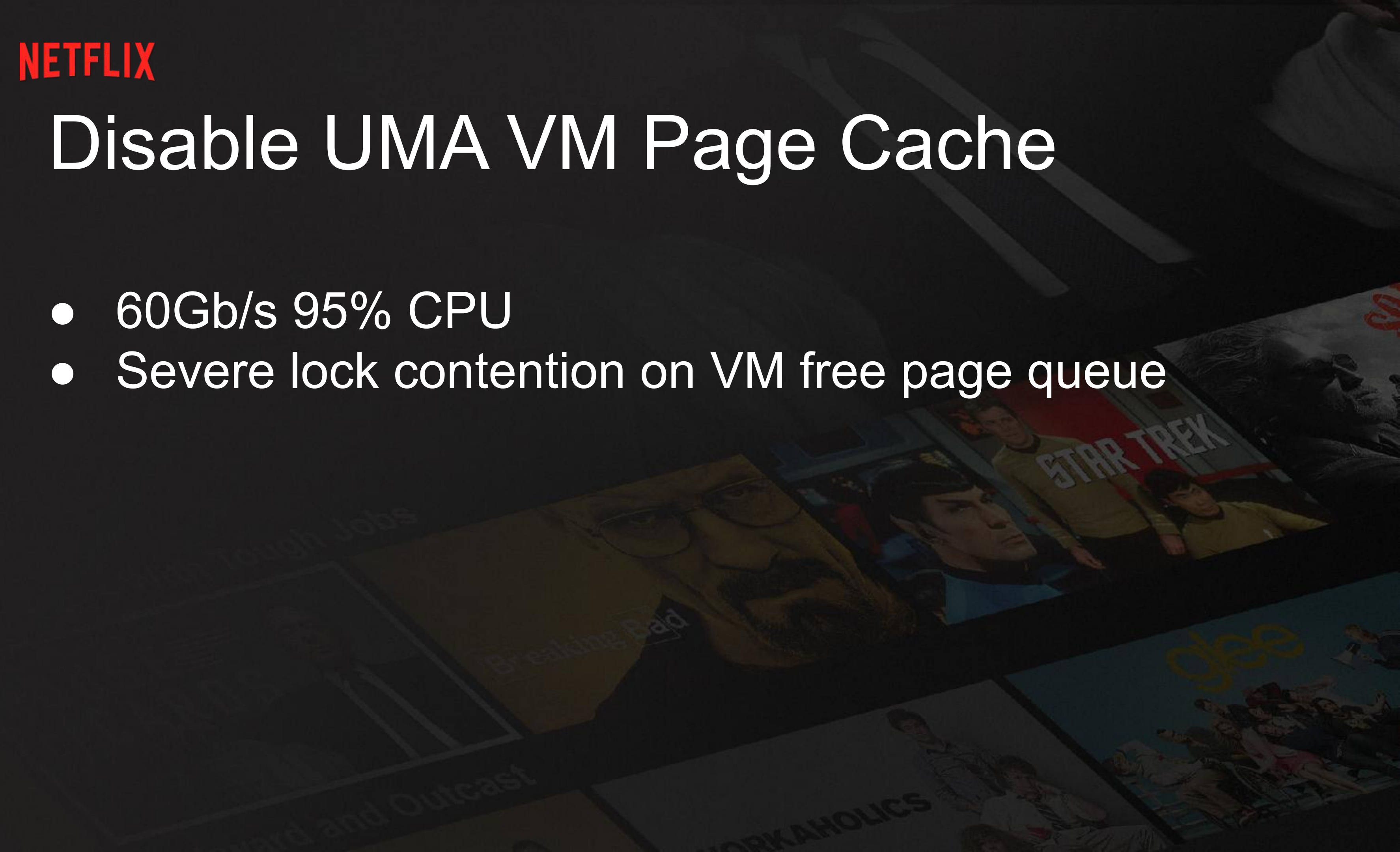


UMA VM Page Cache

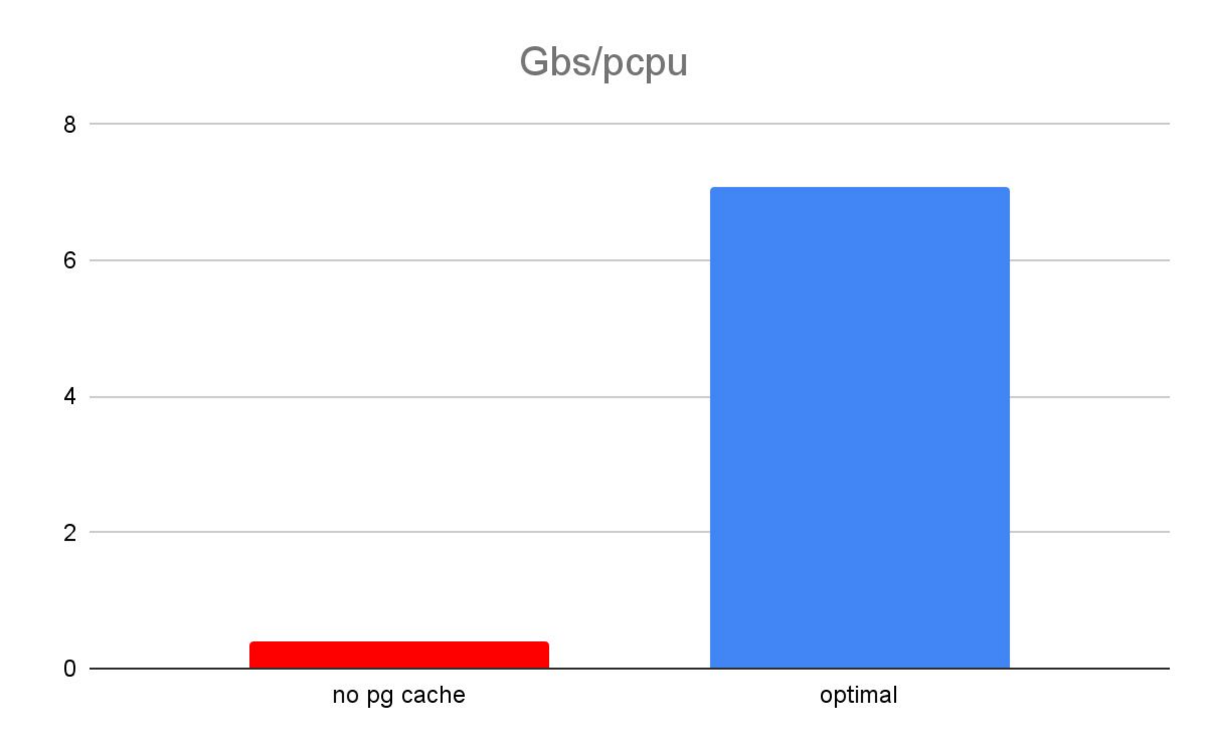
- A per-cpu pool of free pages that can be accessed locklessly
- Managed via UMA (Universal Memory Allocator)
- Only works for free pages, not pages that are recycled into the inactive or active page queues

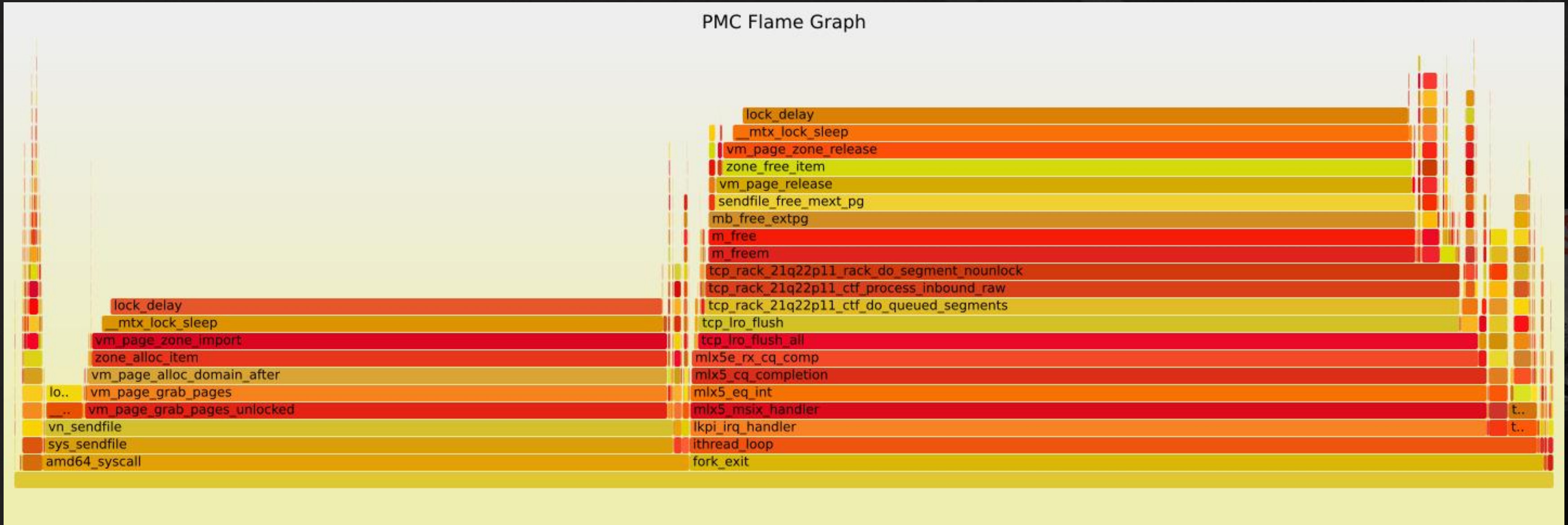
Disable UMA VM Page Cache

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



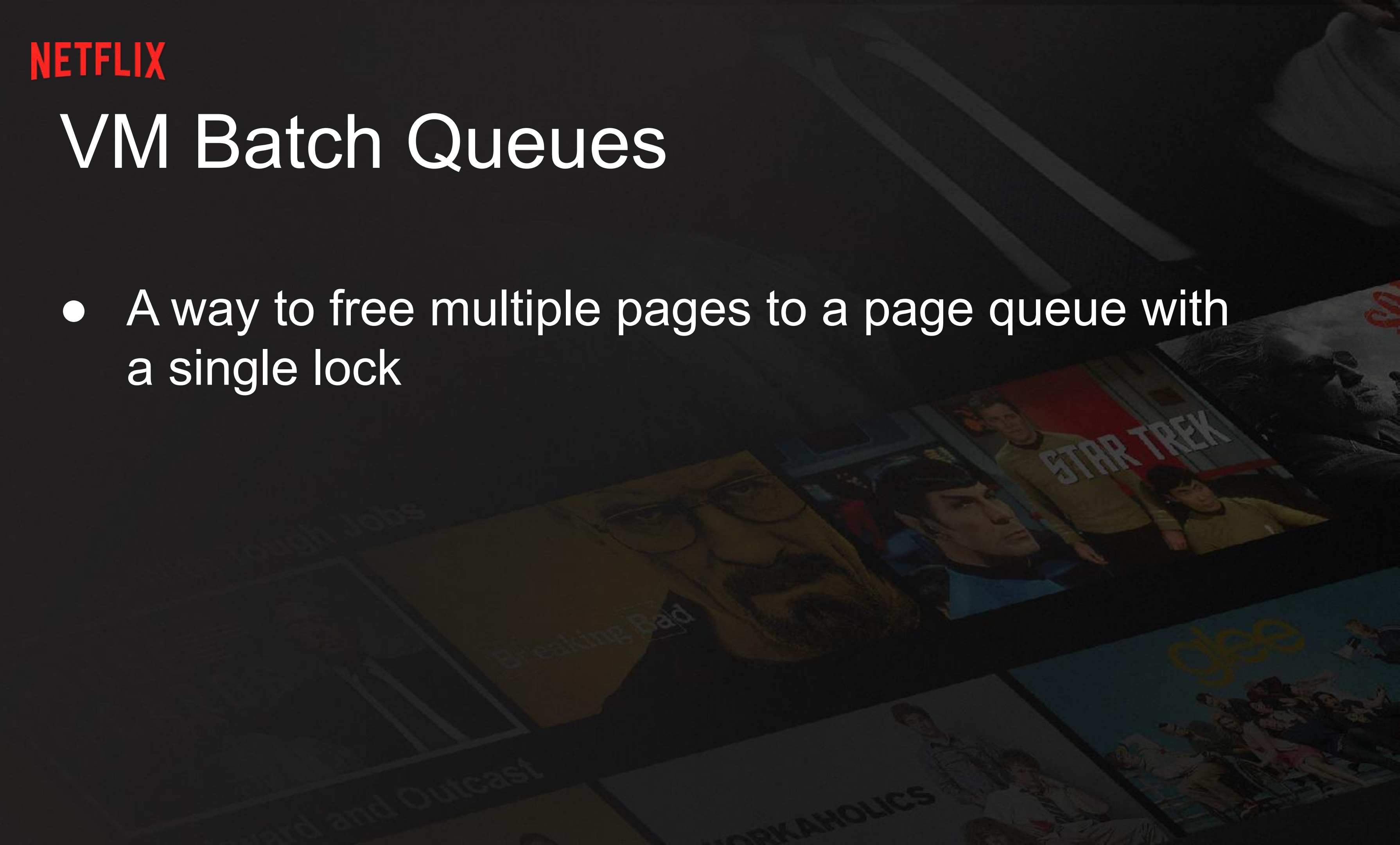
UMA VM Page Cache




PMC Flame Graph
lock_delay
__mtx_lock_sleep
vm_page_zone_import
zone_alloc_item
vm_page_alloc_domain_after
lo..
vm_page_grab_pages
_..
vm_page_grab_pages_unlocked
vn_sendfile
sys_sendfile
amd64_syscall
lock_delay
__mtx_lock_sleep
vm_page_zone_release
zone_free_item
vm_page_release
sendfile_free_mext_pg
mb_free_extpg
m_free
m_freem
tcp_rack_21q22p11_rack_do_segment_nounlock
tcp_rack_21q22p11_ctf_process_inbound_raw
tcp_rack_21q22p11_ctf_do_queued_segments
tcp_lro_flush
tcp_lro_flush_all
mlx5e_rx_cq_comp
mlx5_cq_completion
mlx5_eq_int
mlx5_msix_handler
t..
lkpi_irq_handler
t..
ithread_loop
fork_exit

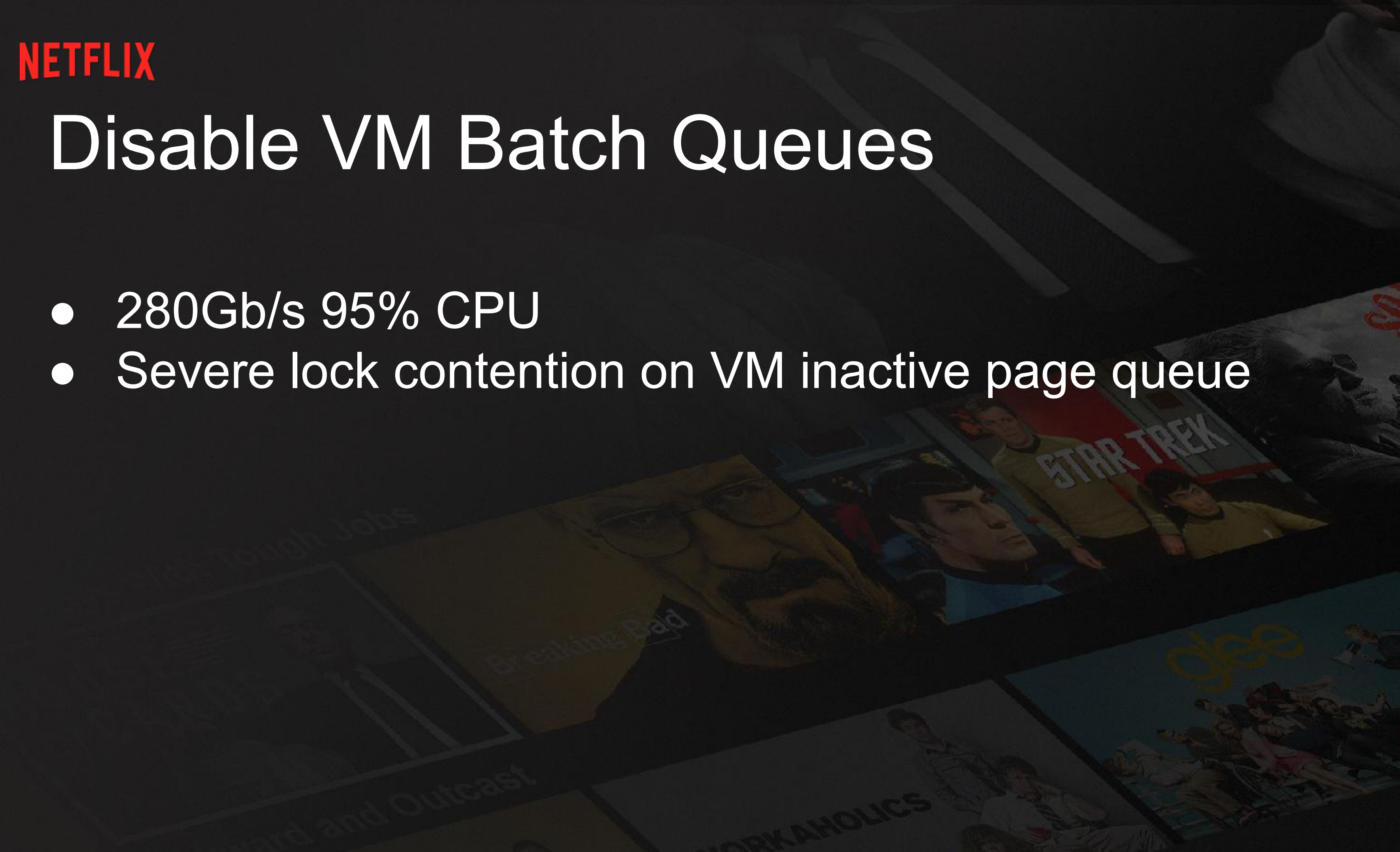
VM Batch Queues

- A way to free multiple pages to a page queue with a single lock



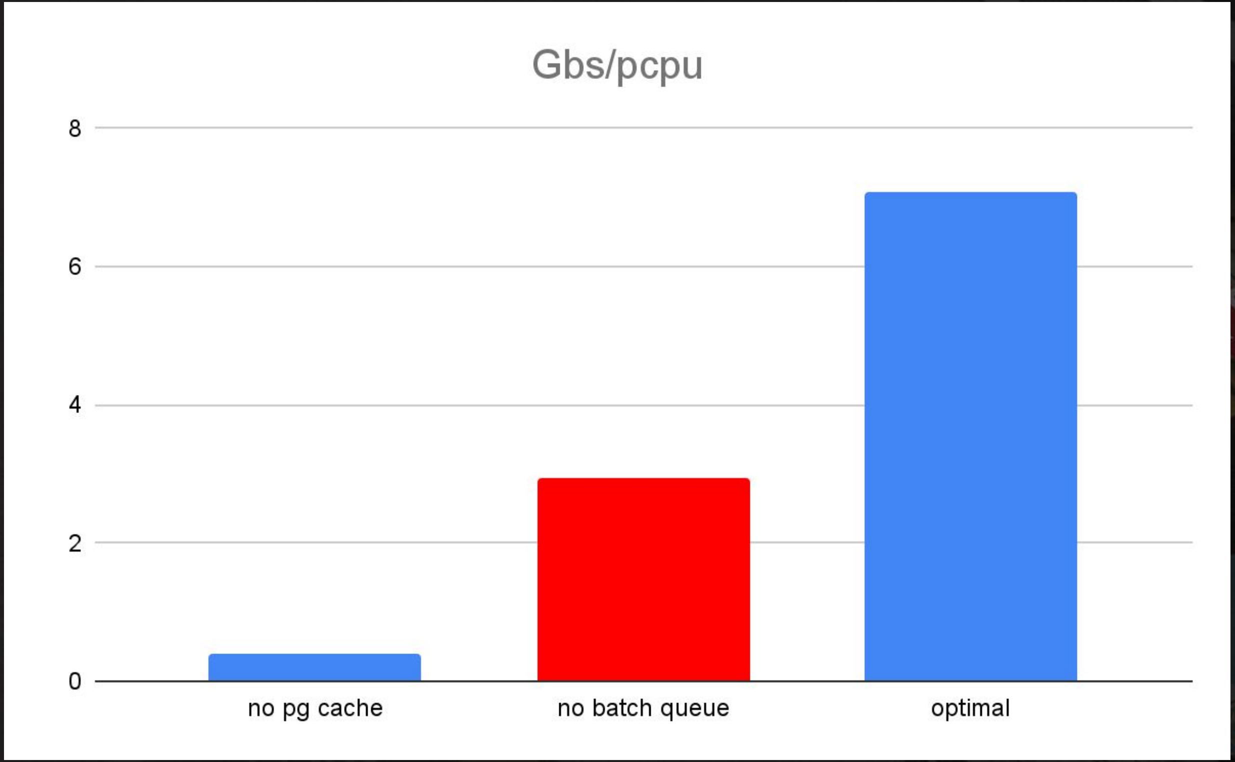
Disable VM Batch Queues

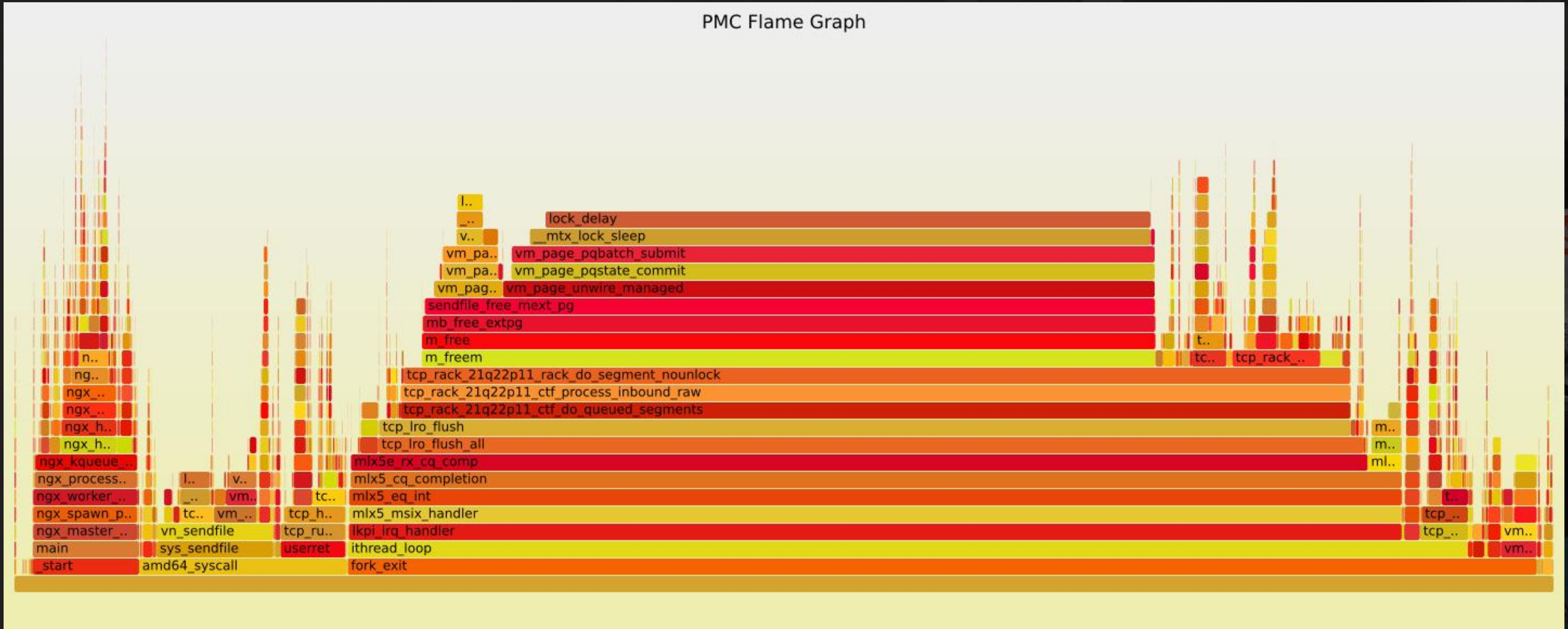
- 280Gb/s 95% CPU
- Severe lock contention on VM inactive page queue



NETFLIX

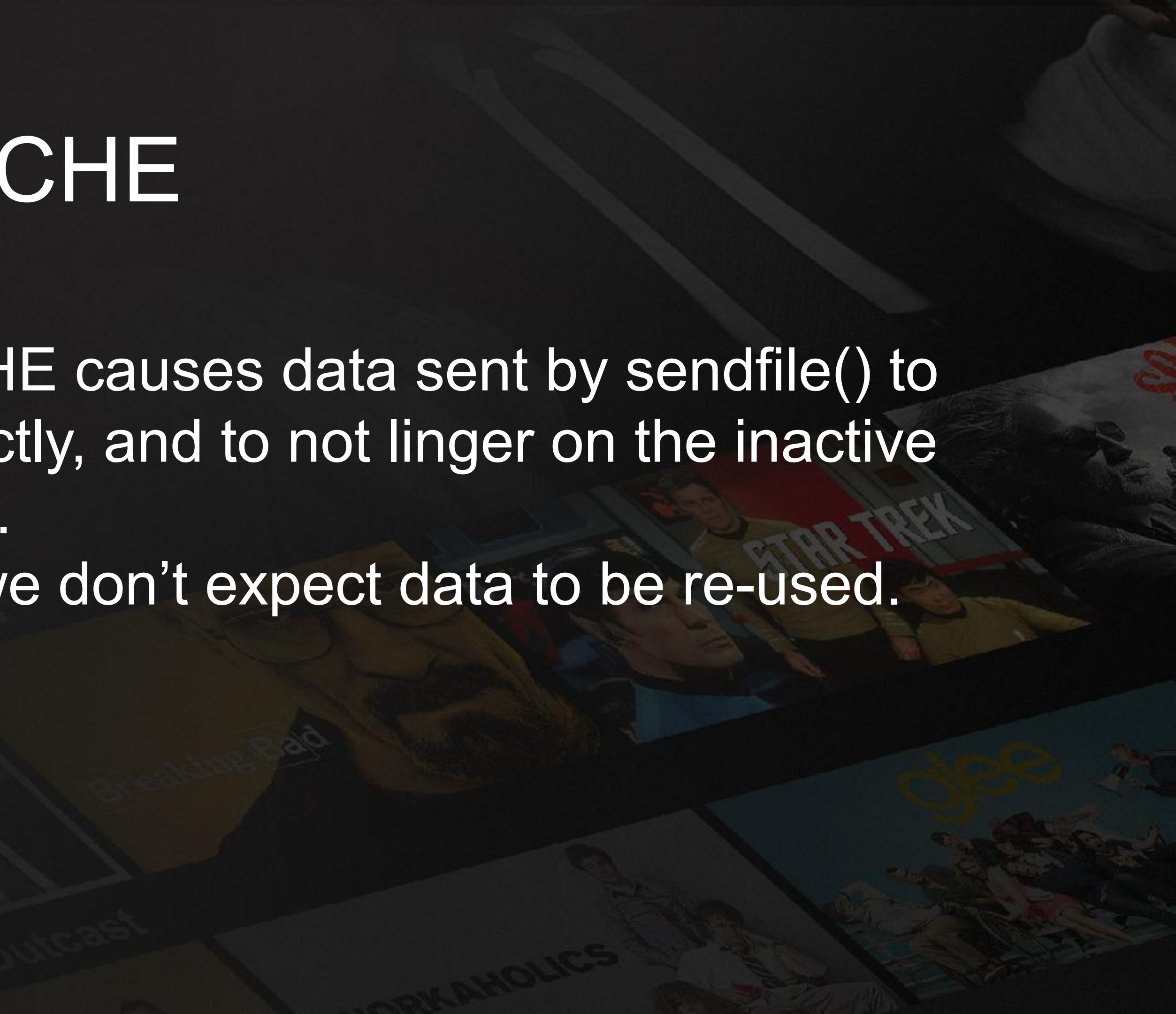
VM Batch Queues




PMC Flame Graph
l..
_..
v..
lock_delay
__mtx_lock_sleep
vm_pa..
vm_page_pqbatch_submit
vm_pa..
vm_page_pqstate_commit
vm_pag..
vm_page_unwire_managed
sendfile_free_mext_pg
mb_free_extpg
m_free
t..
m_freem
tc..
tcp_rack_..
n..
ng..
tcp_rack_21q22p11_rack_do_segment_nounlock
ngx_..
tcp_rack_21q22p11_ctf_process_inbound_raw
ngx_..
tcp_rack_21q22p11_ctf_do_queued_segments
ngx_h..
tcp_lro_flush
m..
ngx_h..
tcp_lro_flush_all
m..
ngx_kqueue_..
mlx5e_rx_cq_comp
ml..
ngx_process..
l..
v..
mlx5_cq_completion
ngx_worker_..
_..
vm..
tc..
mlx5_eq_int
t..
ngx_spawn_p..
tc..
vm_..
tcp_h..
mlx5_msix_handler
tcp_..
ngx_master_..
vn_sendfile
tcp_ru..
lkpi_irq_handler
tcp_..
vm..
main
sys_sendfile
userret
ithread_loop
vm..
_start
amd64_syscall
fork_exit

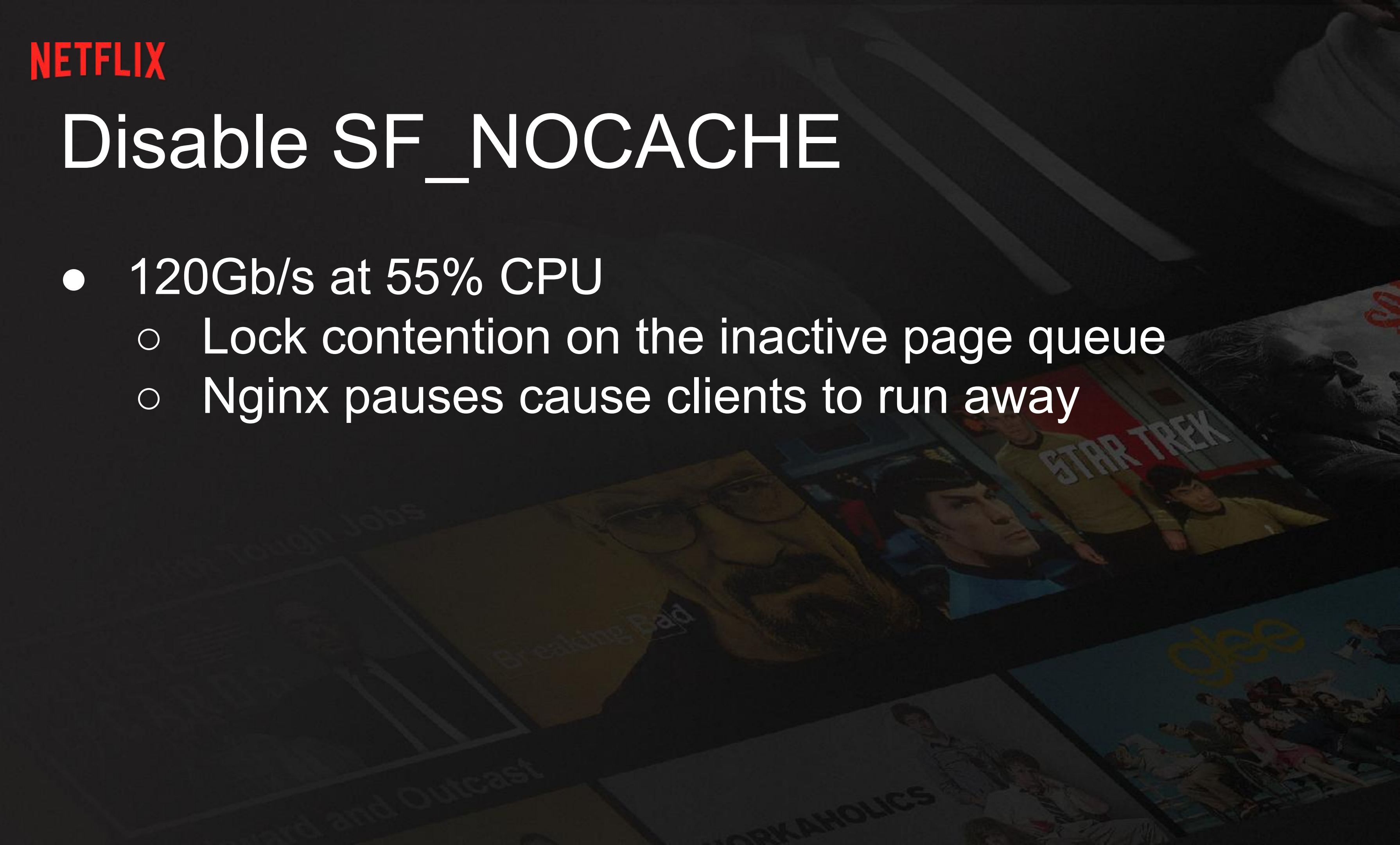
SF_NOCACHE

- 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.

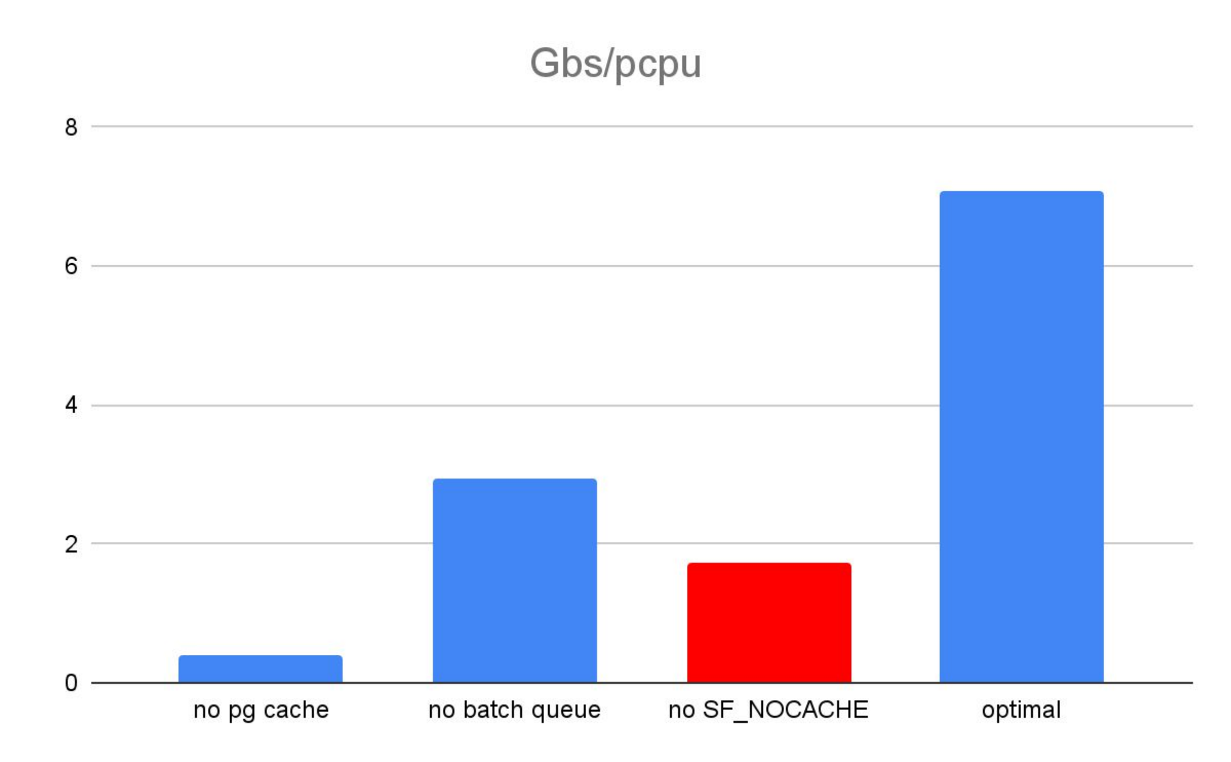


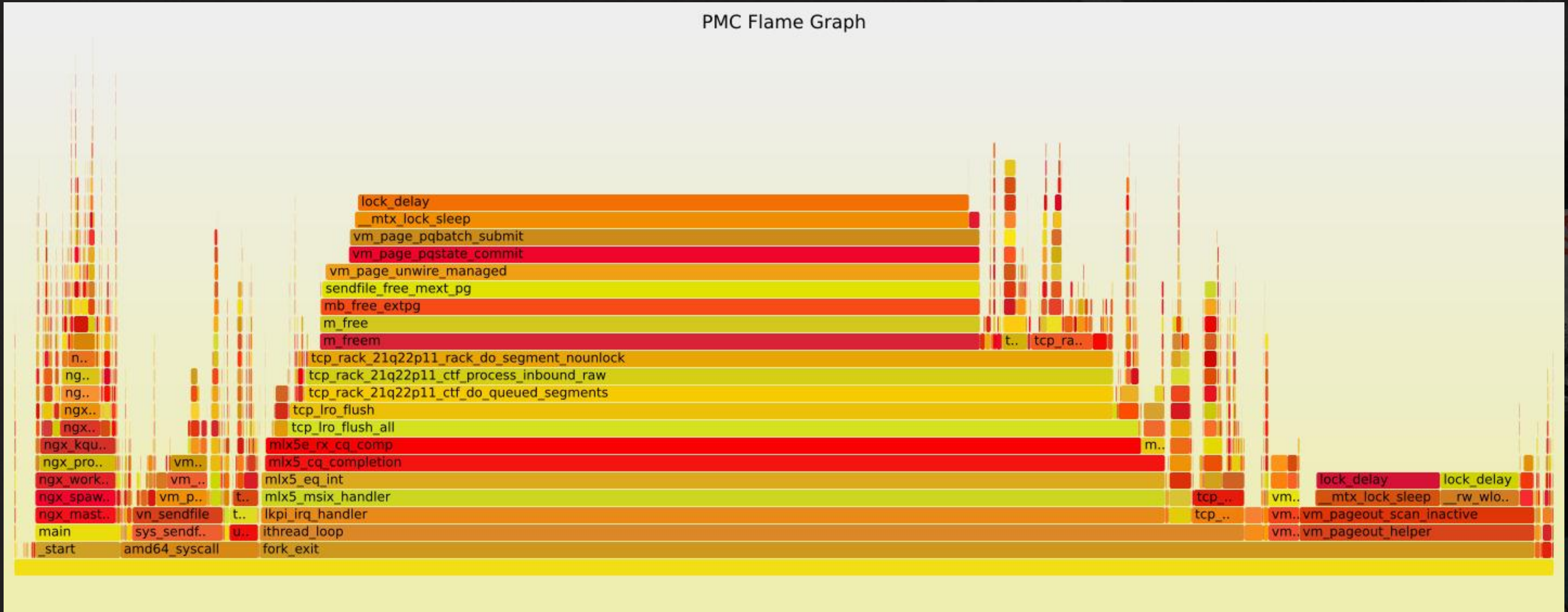
Disable SF_NOCACHE

- 120Gb/s at 55% CPU
  - Lock contention on the inactive page queue
  - Nginx pauses cause clients to run away



Disable SF_NOCACHE

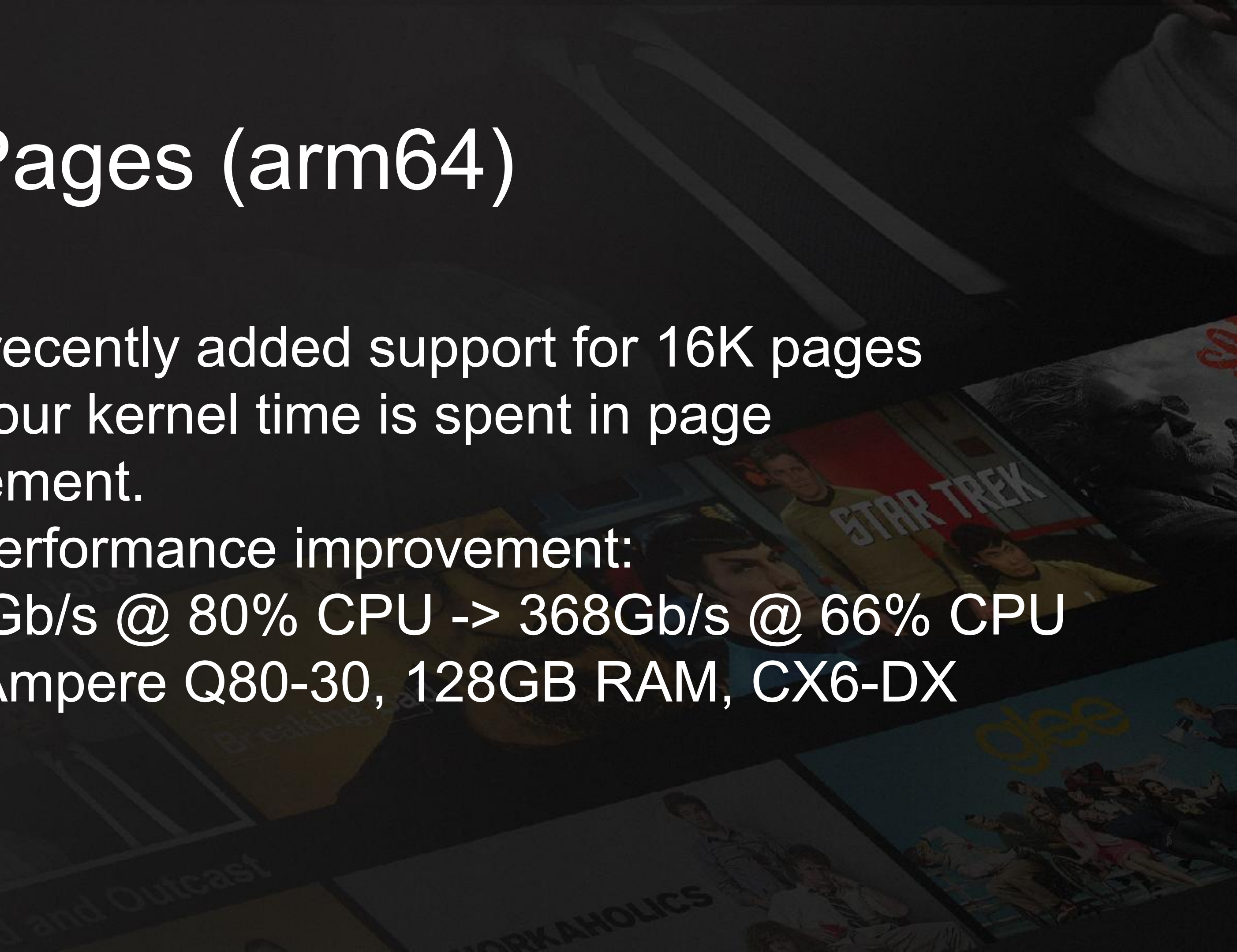



PMC Flame Graph
lock_delay
__mtx_lock_sleep
vm_page_pqbatch_submit
vm_page_pqstate_commit
vm_page_unwire_managed
sendfile_free_mext_pg
mb_free_extpg
m_free
m_freem
t..
tcp_ra..
n..
tcp_rack_21q22p11_rack_do_segment_nounlock
ng..
tcp_rack_21q22p11_ctf_process_inbound_raw
ng..
tcp_rack_21q22p11_ctf_do_queued_segments
ngx..
tcp_lro_flush
ngx..
tcp_lro_flush_all
ngx_kqu..
mlx5e_rx_cq_comp
m..
ngx_pro..
vm..
mlx5_cq_completion
ngx_work..
vm_..
mlx5_eq_int
lock_delay
lock_delay
ngx_spaw..
vm_p..
t..
mlx5_msix_handler
tcp_..
vm..
__mtx_lock_sleep
__rw_wlo..
ngx_mast..
vn_sendfile
t..
lkpi_irq_handler
tcp_..
vm..
vm_pageout_scan_inactive
main
sys_sendf..
u..
ithread_loop
vm..
vm_pageout_helper
_start
amd64_syscall
fork_exit

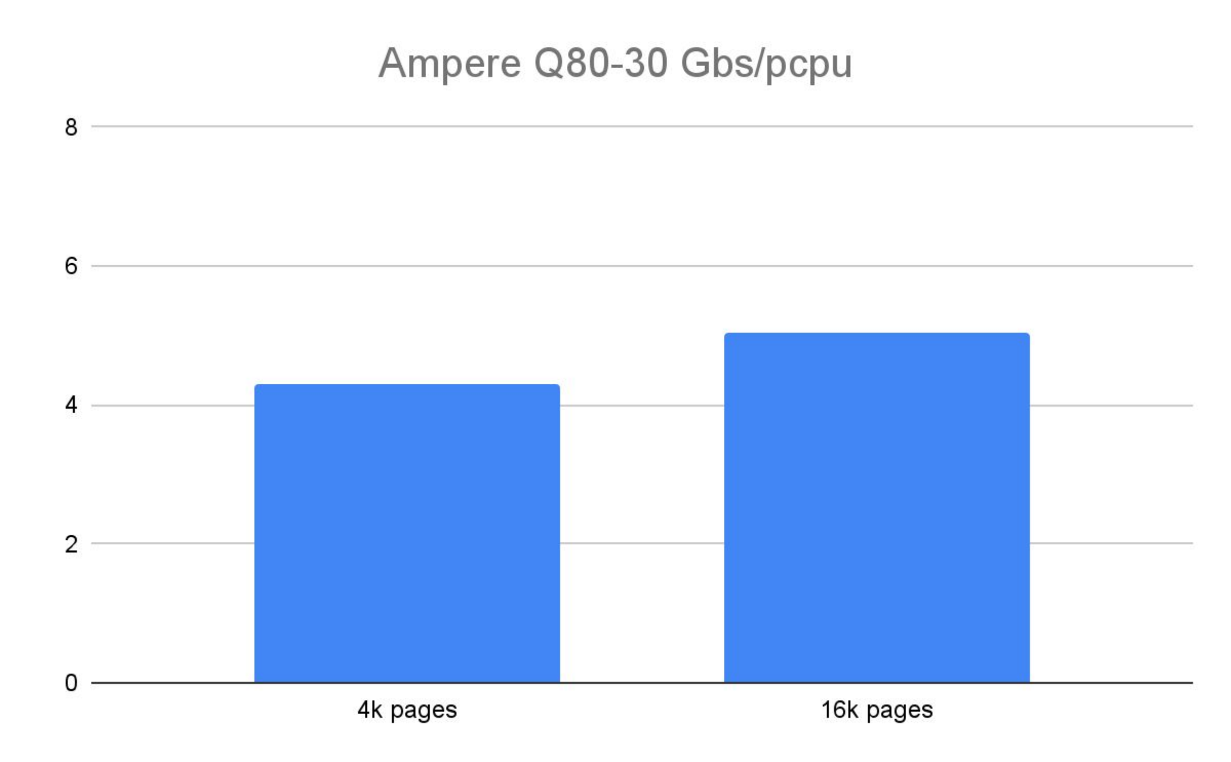
NETFLIX

16KB Pages (arm64)

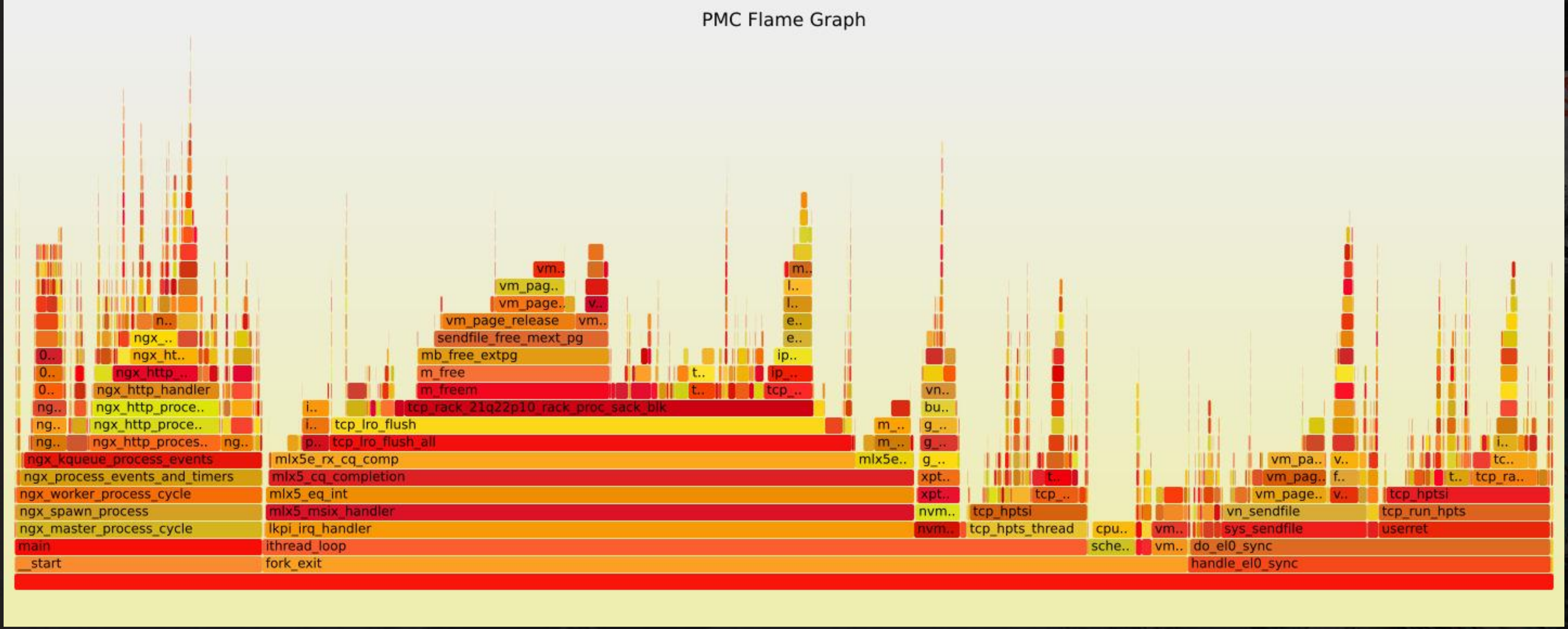
- Arm64 recently added support for 16K pages
- A lot of our kernel time is spent in page management.
- Large performance improvement:
  - 345Gb/s @ 80% CPU -> 368Gb/s @ 66% CPU
    - Ampere Q80-30, 128GB RAM, CX6-DX



16K Pages

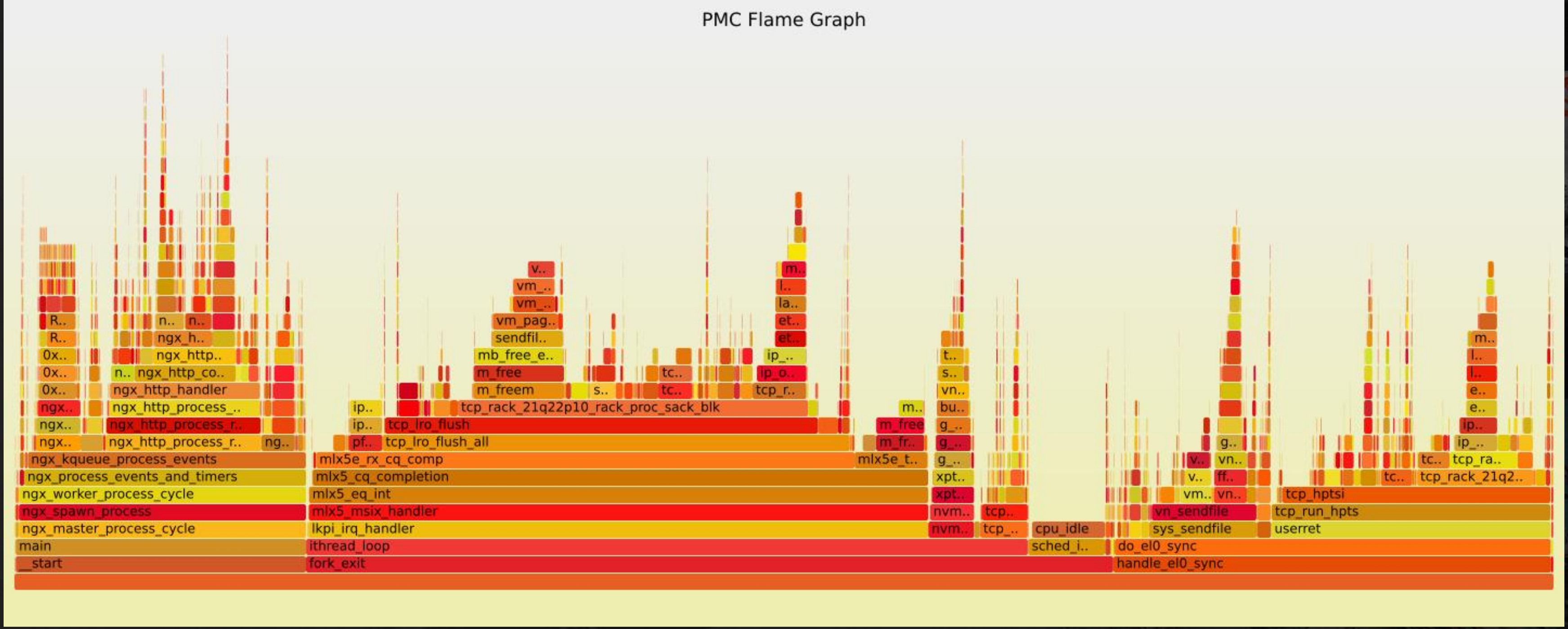


4K Pages



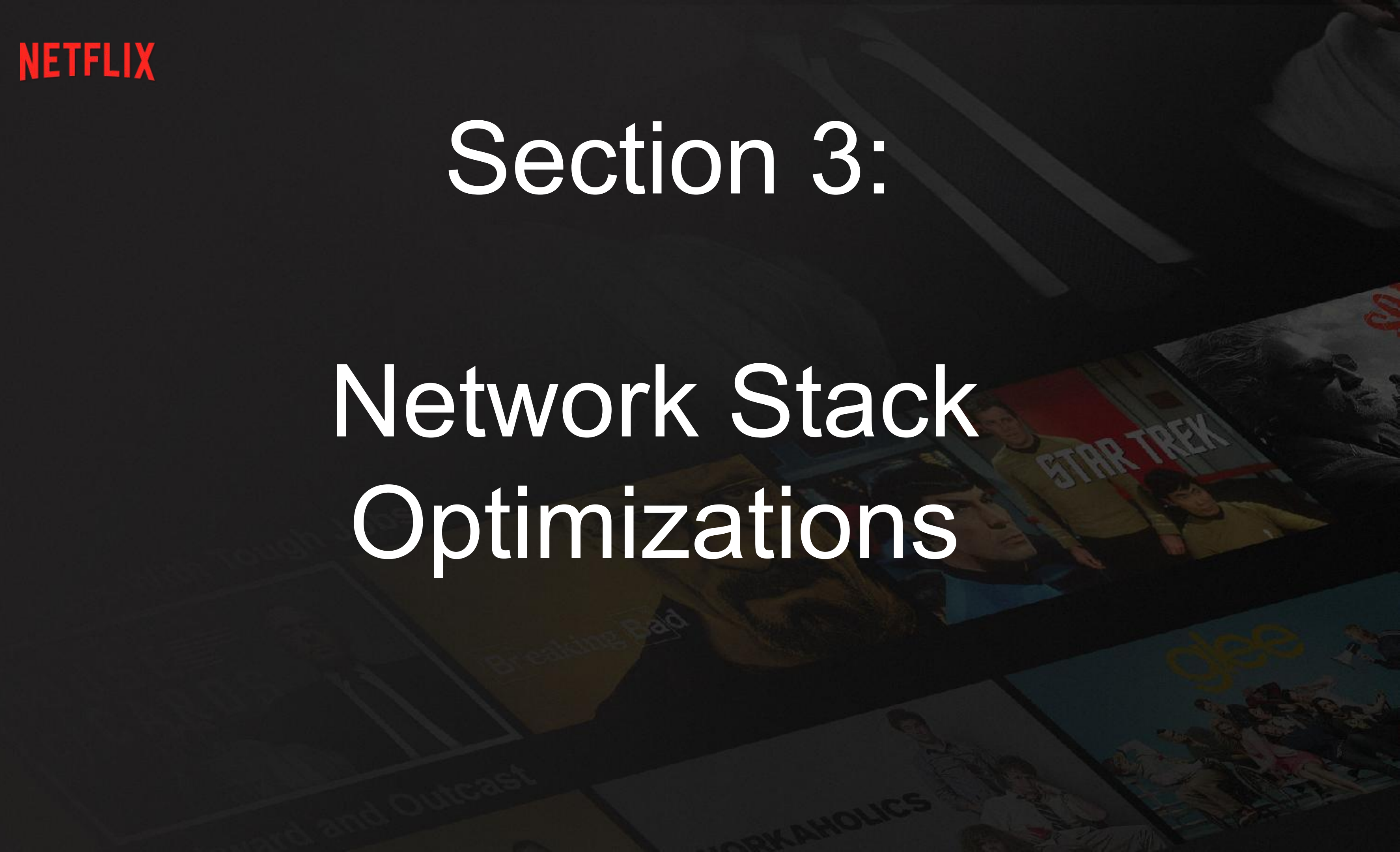
PMC Flame Graph

16K Pages


PMC Flame Graph
_start
main
ngx_master_process_cycle
ngx_spawn_process
ngx_worker_process_cycle
ngx_process_events_and_timers
ngx_kqueue_process_events
ngx_http_process_r..
ngx_http_process_r..
ngx_http_process_..
ngx_http_handler
ngx_http_co..
ngx_http..
ngx_h..
fork_exit
ithread_loop
lkpi_irq_handler
mlx5_msix_handler
mlx5_eq_int
mlx5_cq_completion
mlx5e_rx_cq_comp
tcp_lro_flush_all
tcp_lro_flush
tcp_rack_21q22p10_rack_proc_sack_blk
m_freem
m_free
mb_free_e..
sendfil..
vm_pag..
mlx5e_t..
cpu_idle
sched_i..
handle_el0_sync
do_el0_sync
sys_sendfile
vn_sendfile
userret
tcp_run_hpts
tcp_hptsi
tcp_rack_21q2..
tcp_ra..

Section 3:

Network Stack Optimizations



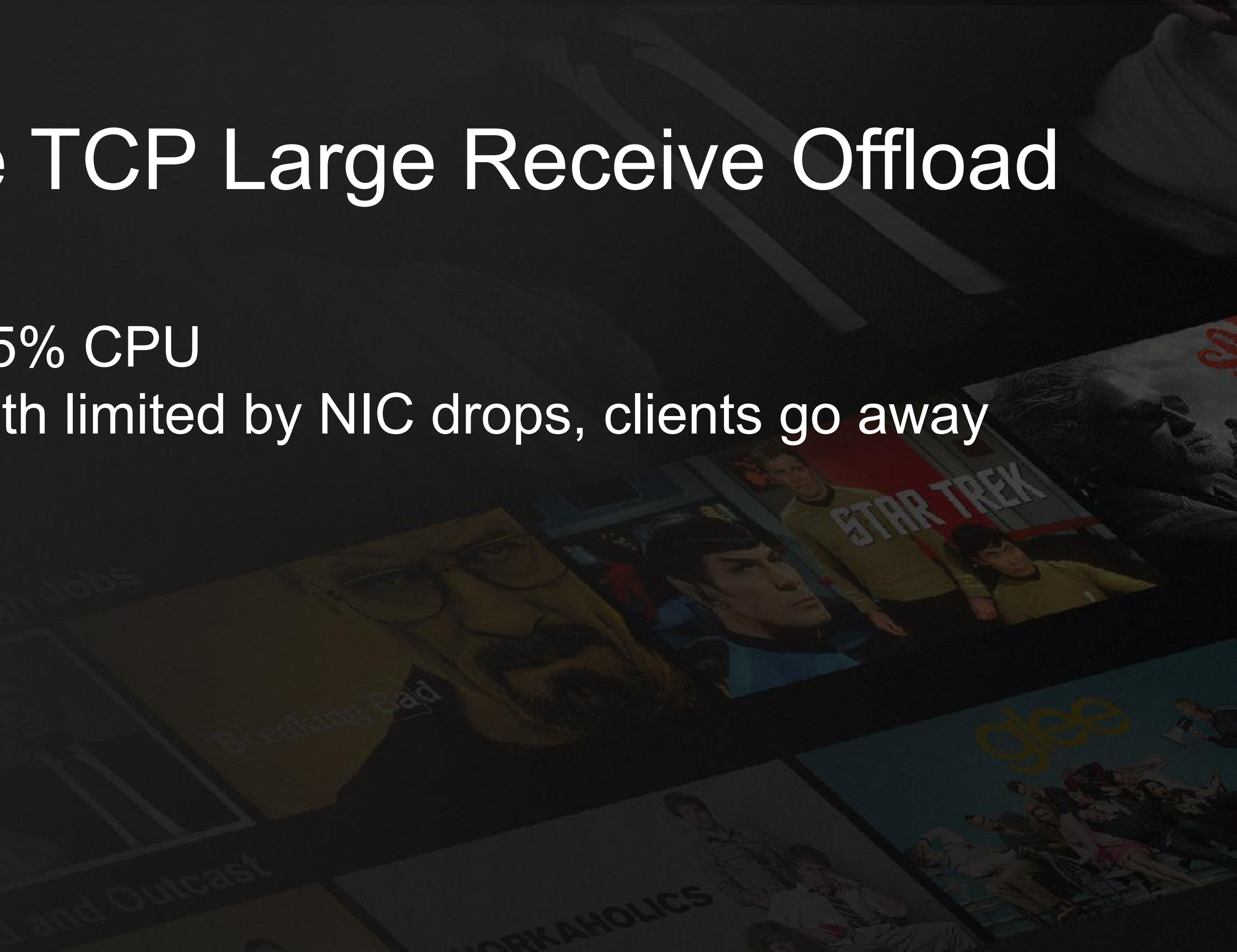
TCP Large Receive Offload (LRO)

- 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.

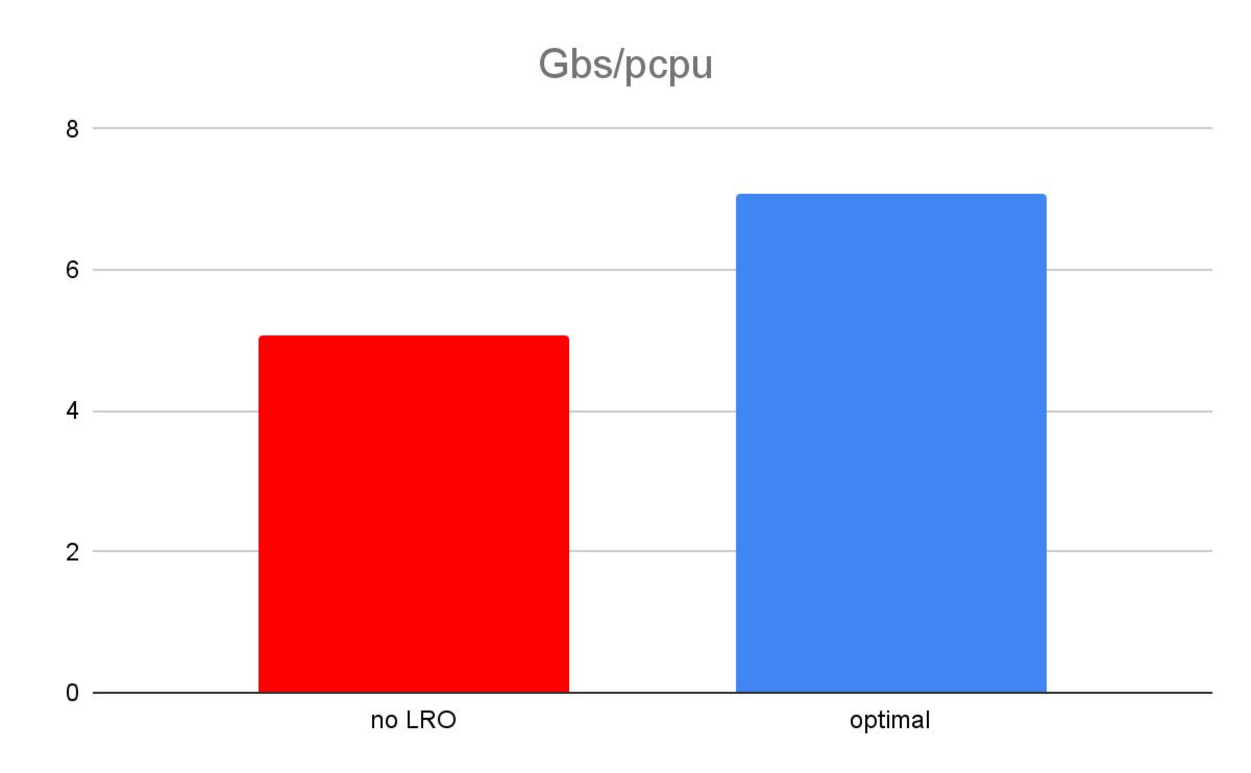
NETFLIX

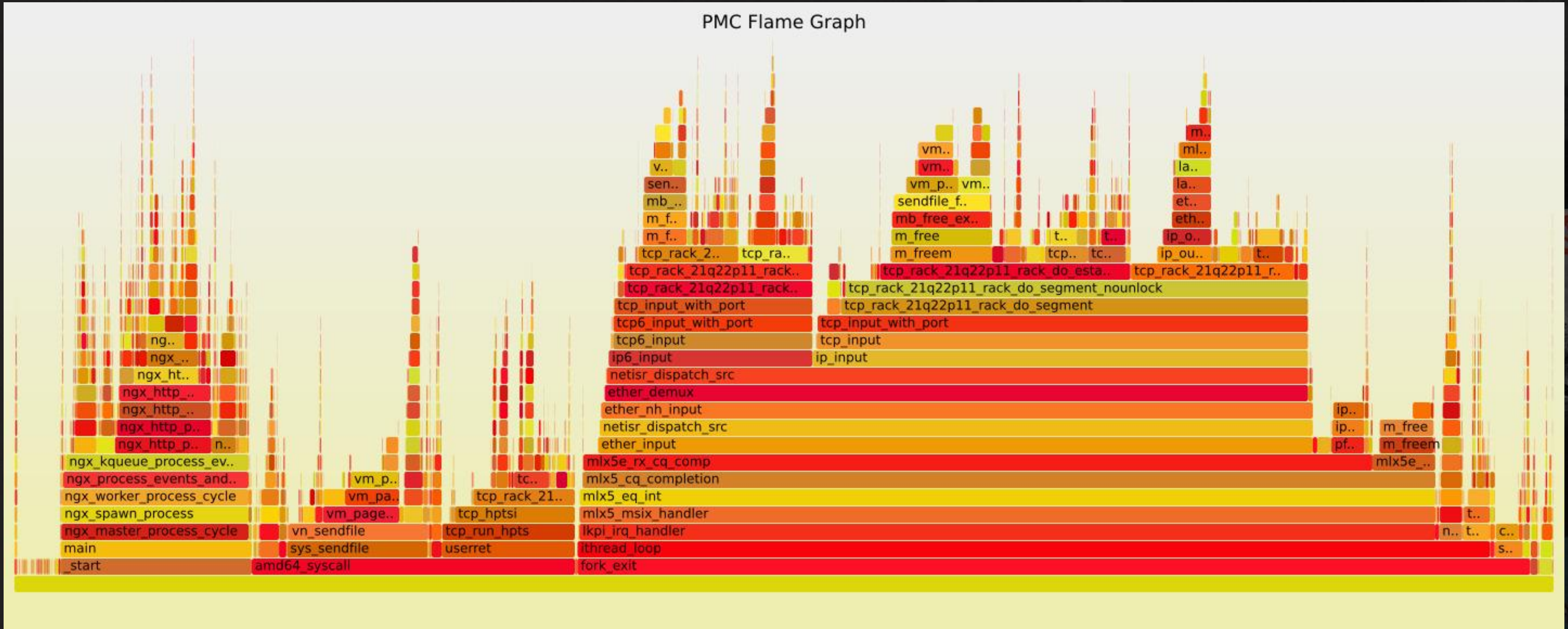
Disable TCP Large Receive Offload

- 330G 65% CPU
  - Health limited by NIC drops, clients go away



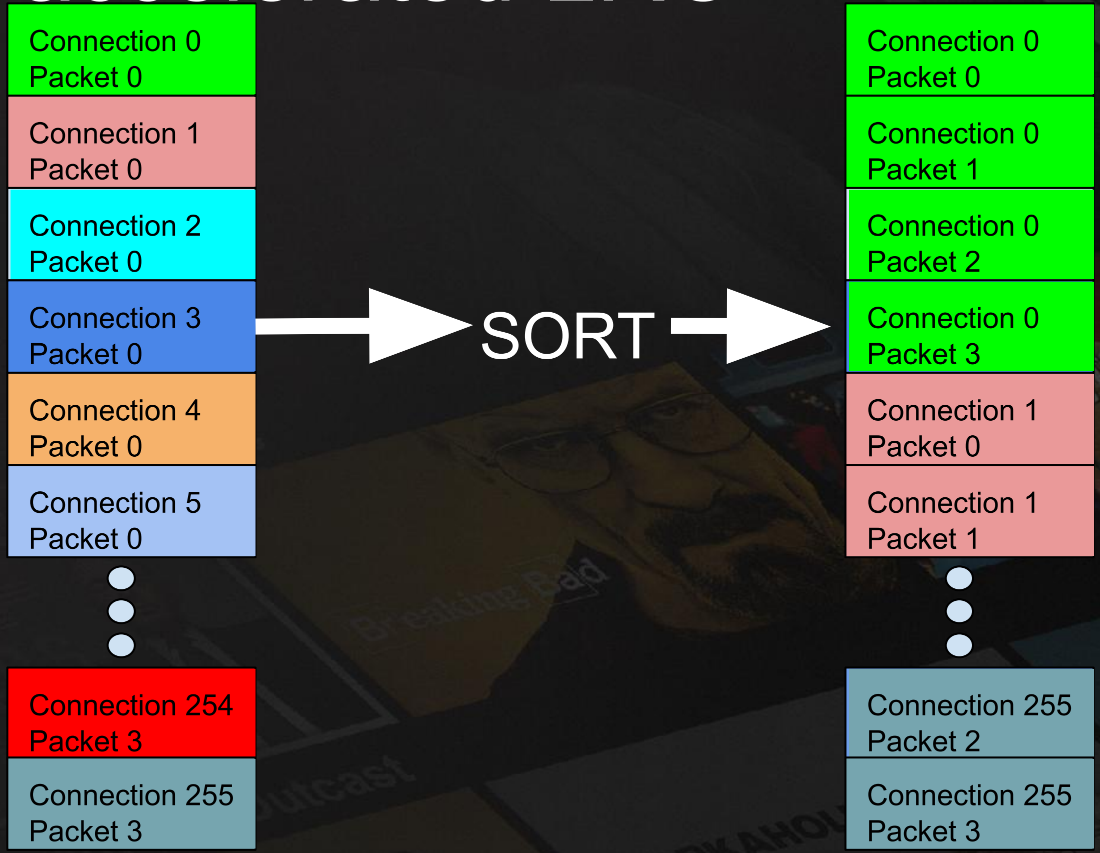
TCP Large Receive Offload





PMC Flame Graph

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

SORT

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

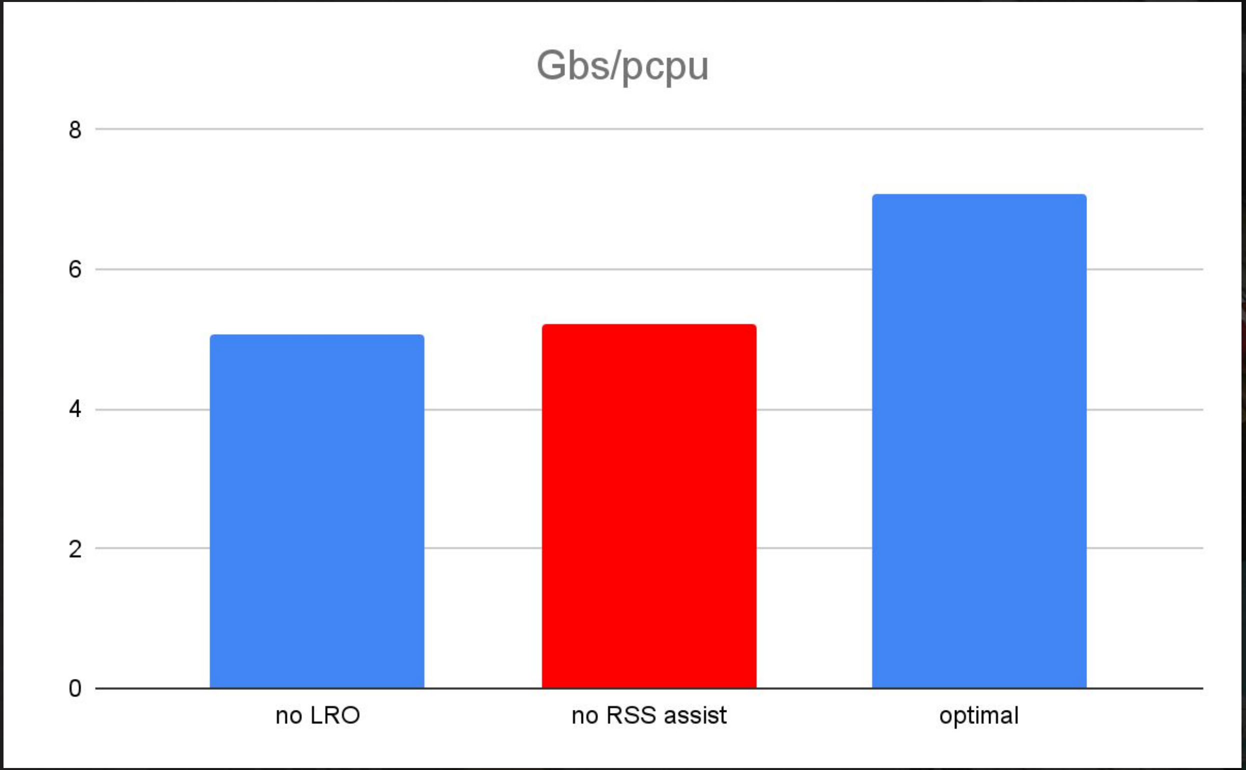
Connection 255
Packet 3

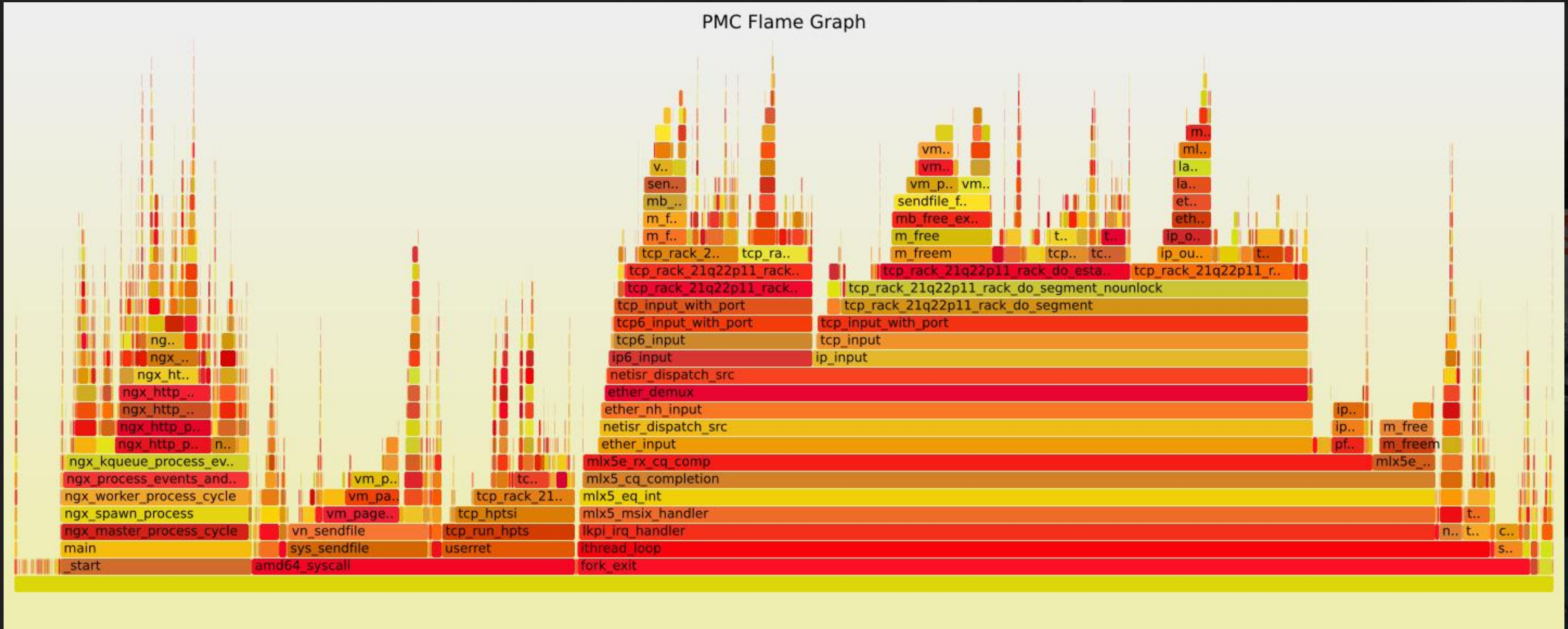
Disable RSS accelerated LRO

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

NETFLIX

RSS accelerated LRO





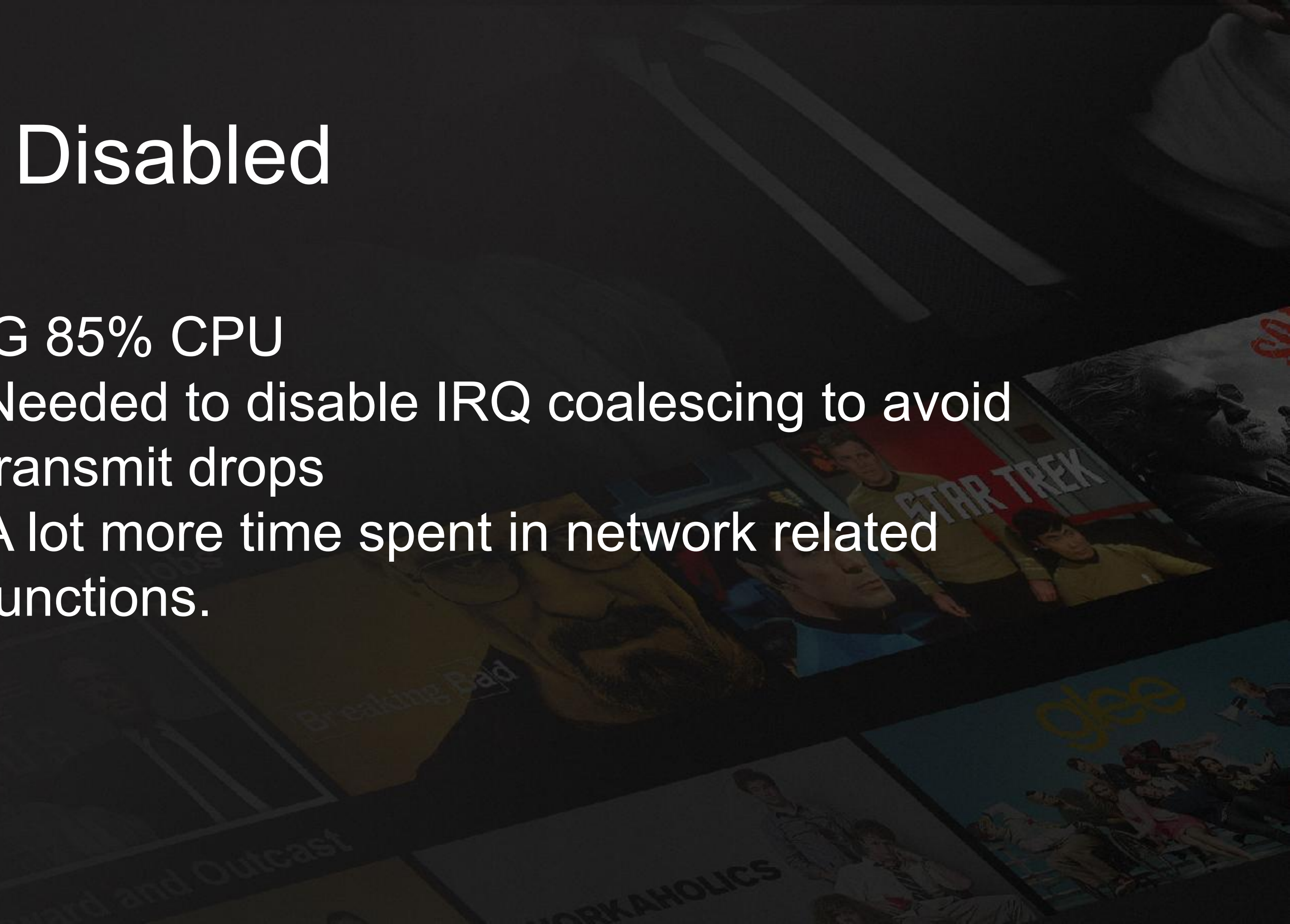
PMC Flame Graph

TCP Large Send Offload (TSO)

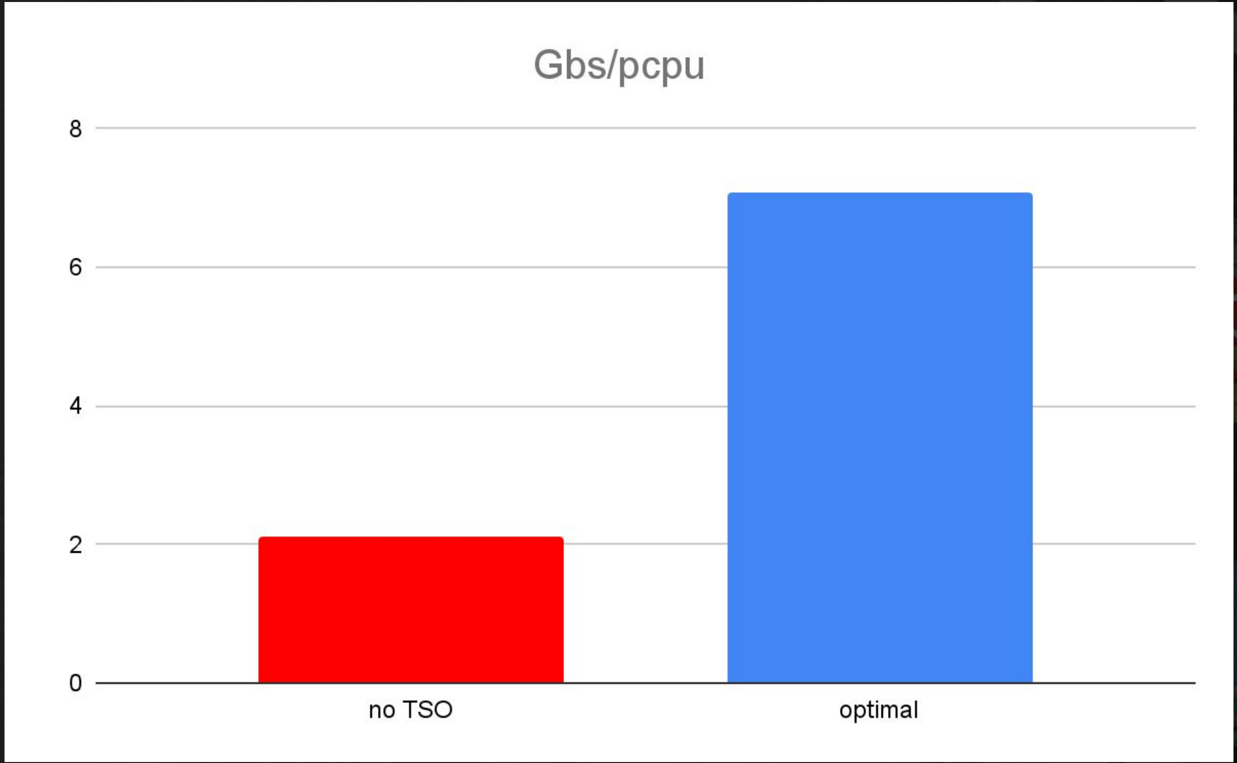
- Like LRO, we reduce the number of trips through the network stack.
- Rather than sending 2 (or 8 or 43) packets to the NIC, we send one. NIC breaks (segments) it into 2 (or 8 or 43) packets on the wire.
- Avoids having to allocate headers for each, look up ethernet addresses, and interact with NIC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

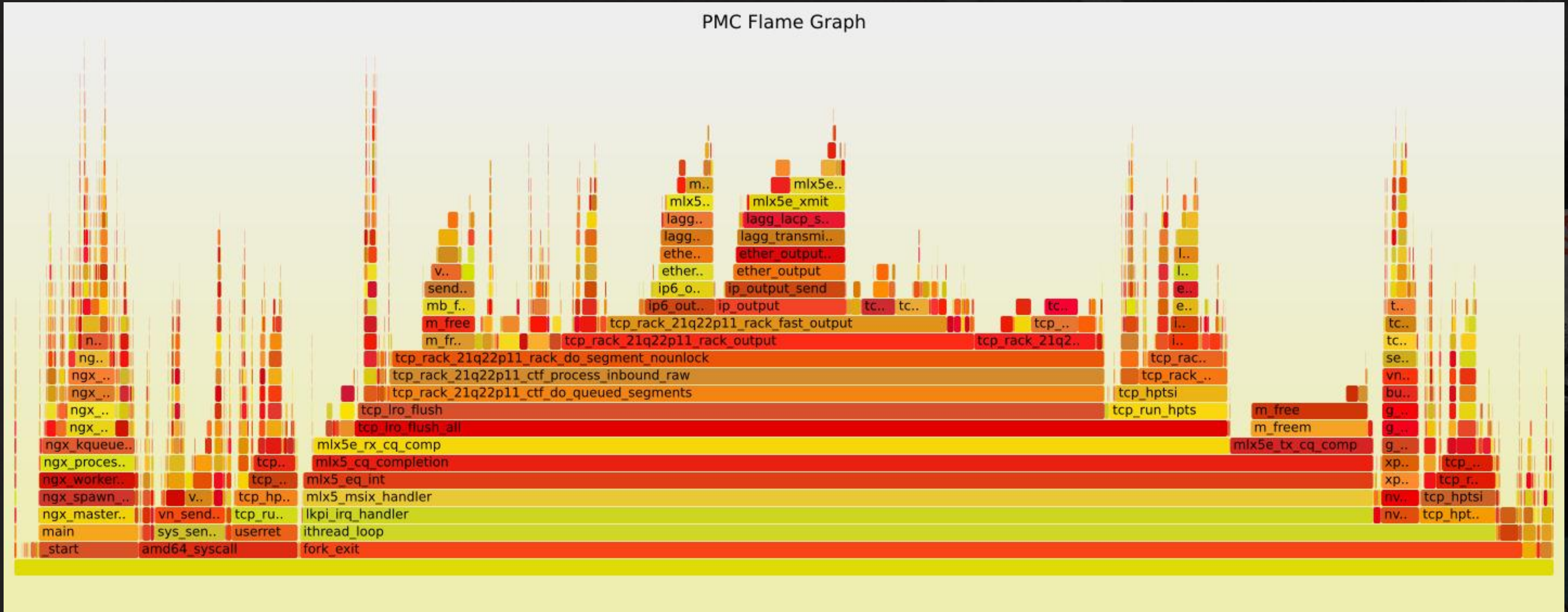
TSO Disabled

- 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)



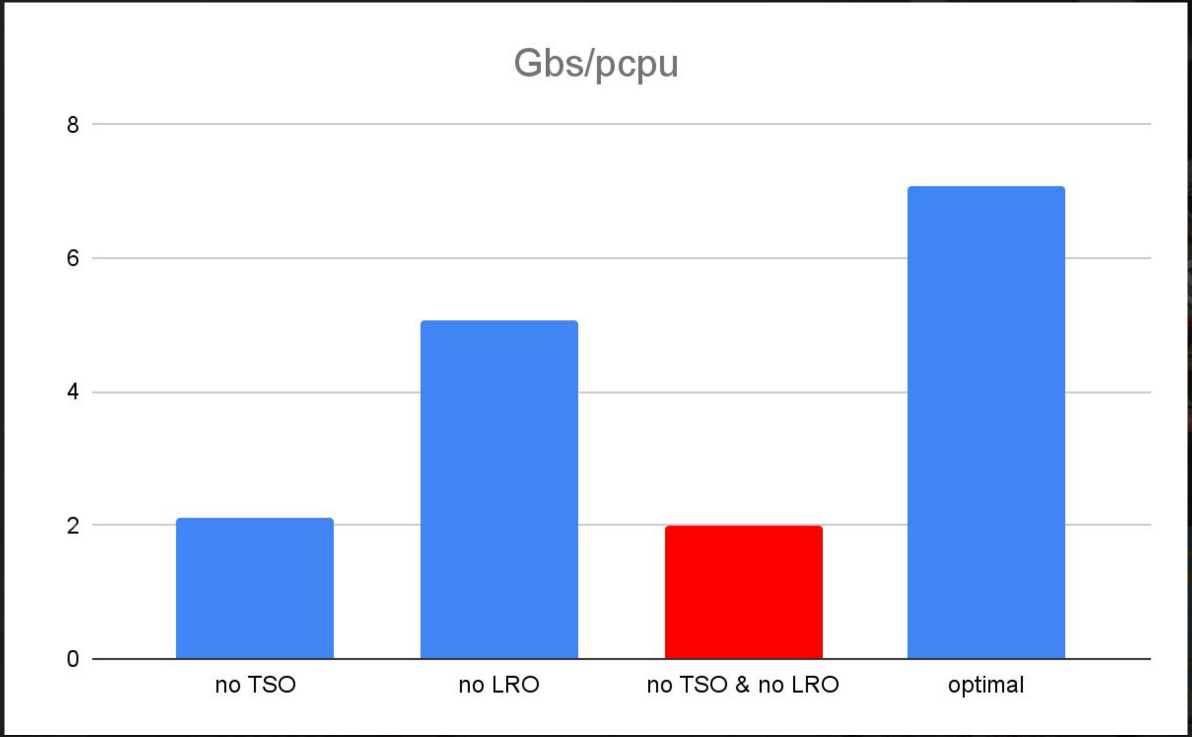

PMC Flame Graph
m..
mlx5e..
mlx5..
mlx5e_xmit
lagg..
lagg_lacp_s..
lagg..
lagg_transmi..
ethe..
ether_output..
v..
ether..
ether_output
l..
l..
send..
ip6_o..
ip_output_send
e..
t..
mb_f..
ip6_out..
ip_output
tc..
tc..
tc..
e..
m_free
tcp_rack_21q22p11_rack_fast_output
tcp_..
i..
tc..
n..
m_fr..
tcp_rack_21q22p11_rack_output
tcp_rack_21q2..
i..
tc..
ng..
tcp_rack_21q22p11_rack_do_segment_nounlock
tcp_rac..
se..
ngx_..
tcp_rack_21q22p11_ctf_process_inbound_raw
tcp_rack_..
vn..
ngx_..
tcp_rack_21q22p11_ctf_do_queued_segments
tcp_hptsi
bu..
ngx_..
tcp_lro_flush
tcp_run_hpts
m_free
g..
ngx_..
tcp_lro_flush_all
m_freem
g..
ngx_kqueue..
mlx5e_rx_cq_comp
mlx5e_tx_cq_comp
g..
ngx_proces..
tcp..
mlx5_cq_completion
xp..
tcp_..
ngx_worker..
tcp_..
mlx5_eq_int
xp..
tcp_r..
ngx_spawn_..
v..
tcp_hp..
mlx5_msix_handler
nv..
tcp_hptsi
ngx_master..
vn_send..
tcp_ru..
lkpi_irq_handler
nv..
tcp_hpt..
main
sys_sen..
userret
ithread_loop
_start
amd64_syscall
fork_exit

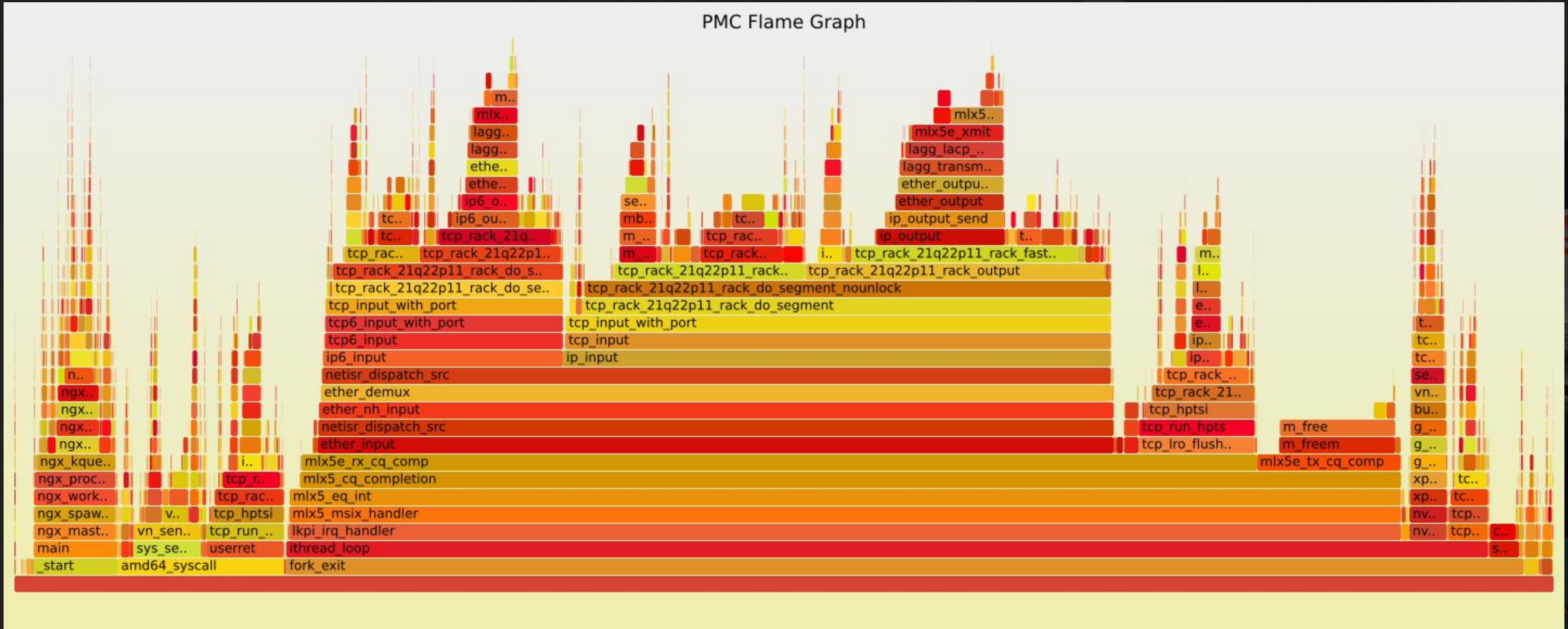
NETFLIX

Disable TSO and LRO

- 170G 85% CPU
    - Needed to disable IRQ coalescing to avoid transmit drops

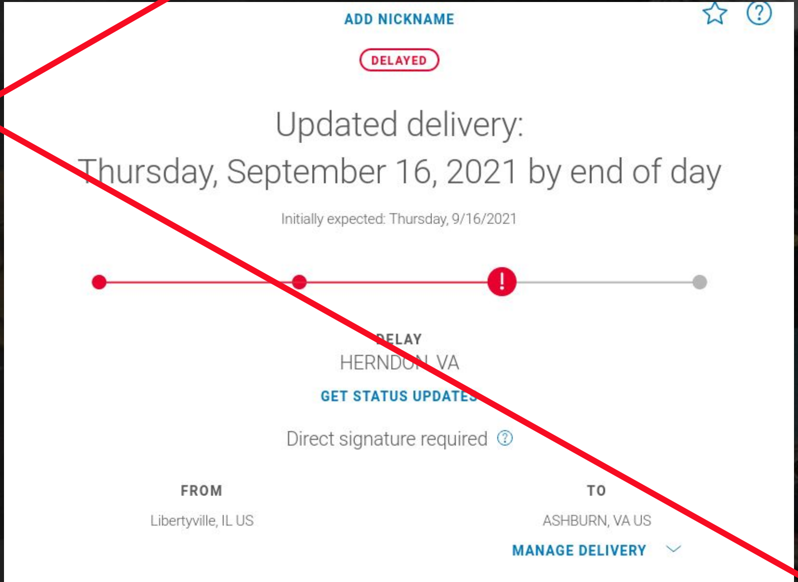
TCP Large Send Offload (TSO) and LRO




PMC Flame Graph

But wait, there's …. not … more..

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



Updated delivery:
Thursday, September 16, 2021 by end of day

Initially expected: Thursday, 9/16/2021

DELAY
HERNDON, VA

Direct signature required

FROM
Libertyville, IL US

TO
ASHBURN, VA US

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

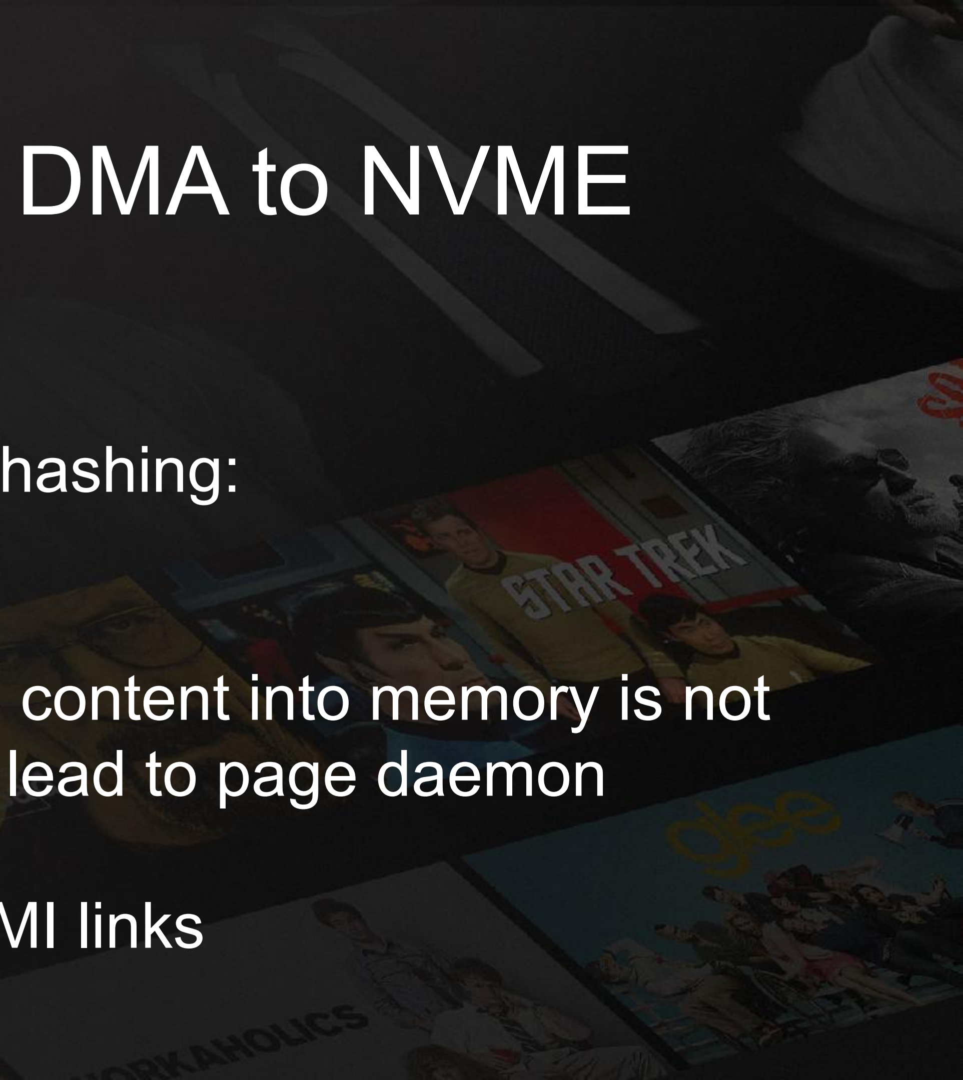
NETFLIX

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!

NETFLIX

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.

NETFLIX

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

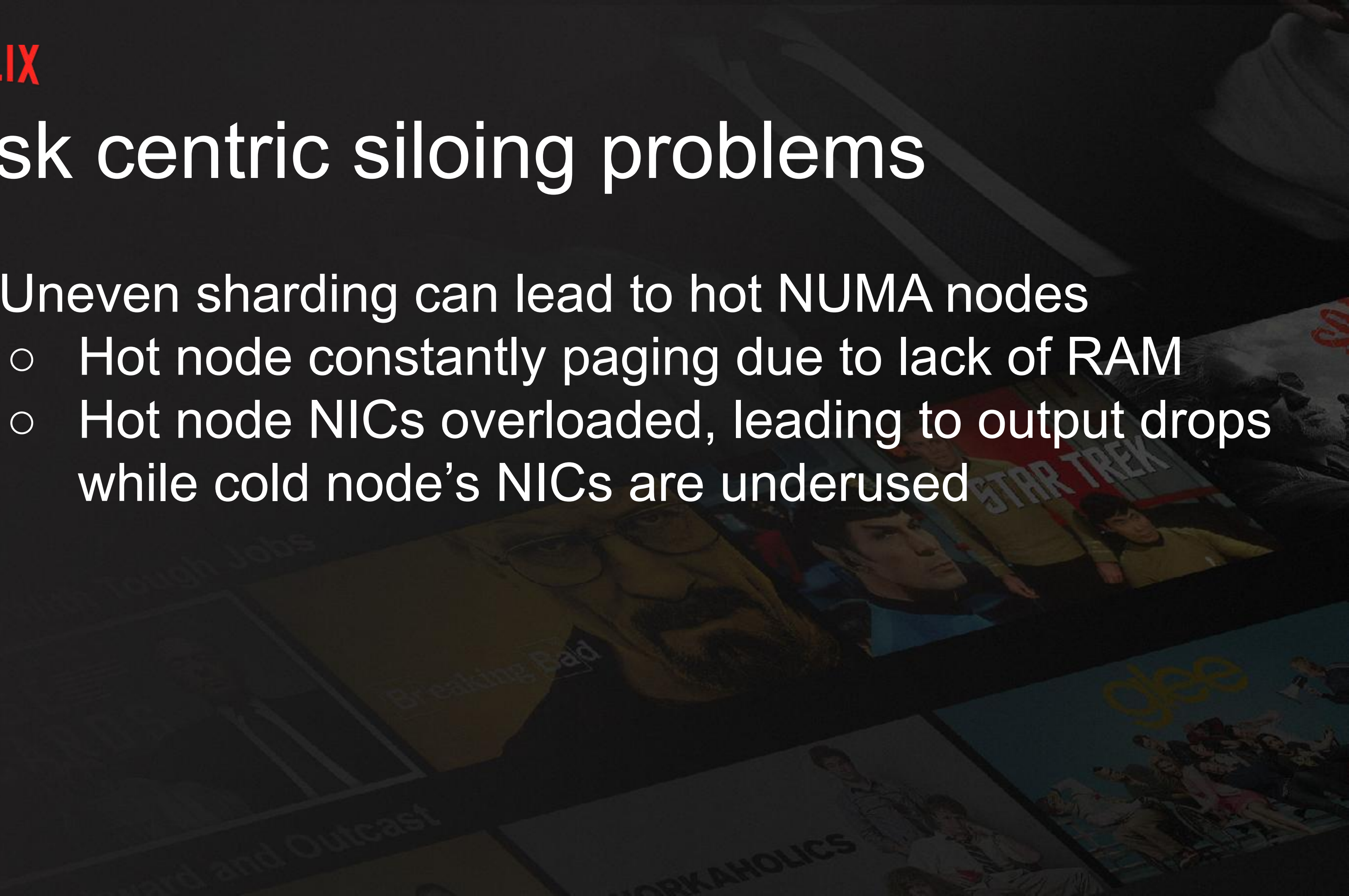
NETFLIX

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



NETFLIX

“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)