QUIC CPU Performance

Can HTTP/3 be as efficient as HTTP/2 and HTTP 1.1?

SIGCOMM EPIQ 2020, Presented by Ian Swett

What are QUIC and HTTP/3?

QUIC is a transport

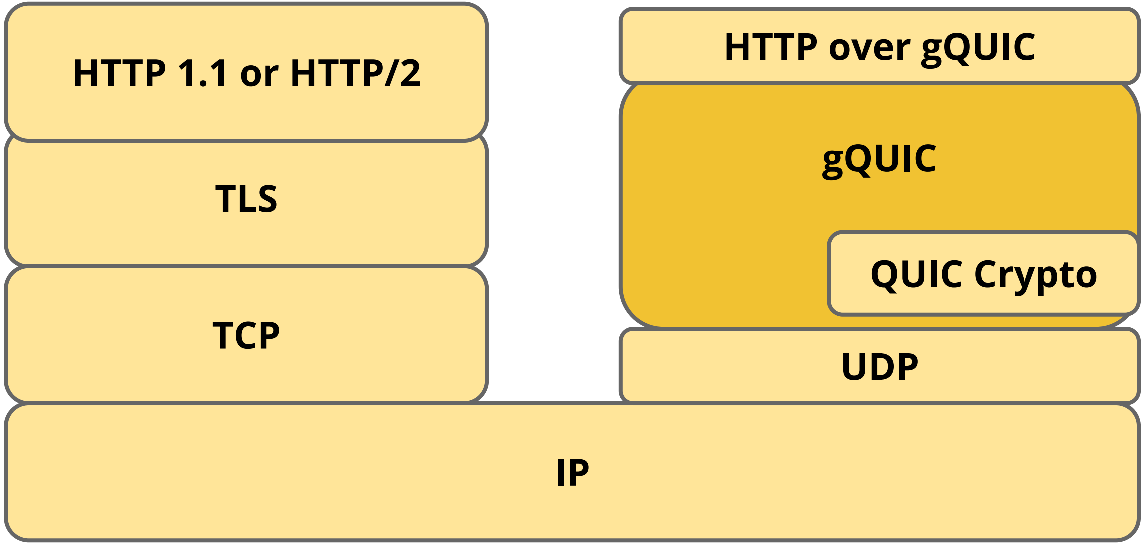
Always encrypted end-to-end
Multistreaming transport with no head of line blocking
0RTT connection establishment
Better loss recovery and flexible congestion control
Supports mixing reliable and unreliable transport features
Improved privacy and reset resistance
Connection migration

QUIC is an alternative to TCP+TLS that provides reliable data delivery

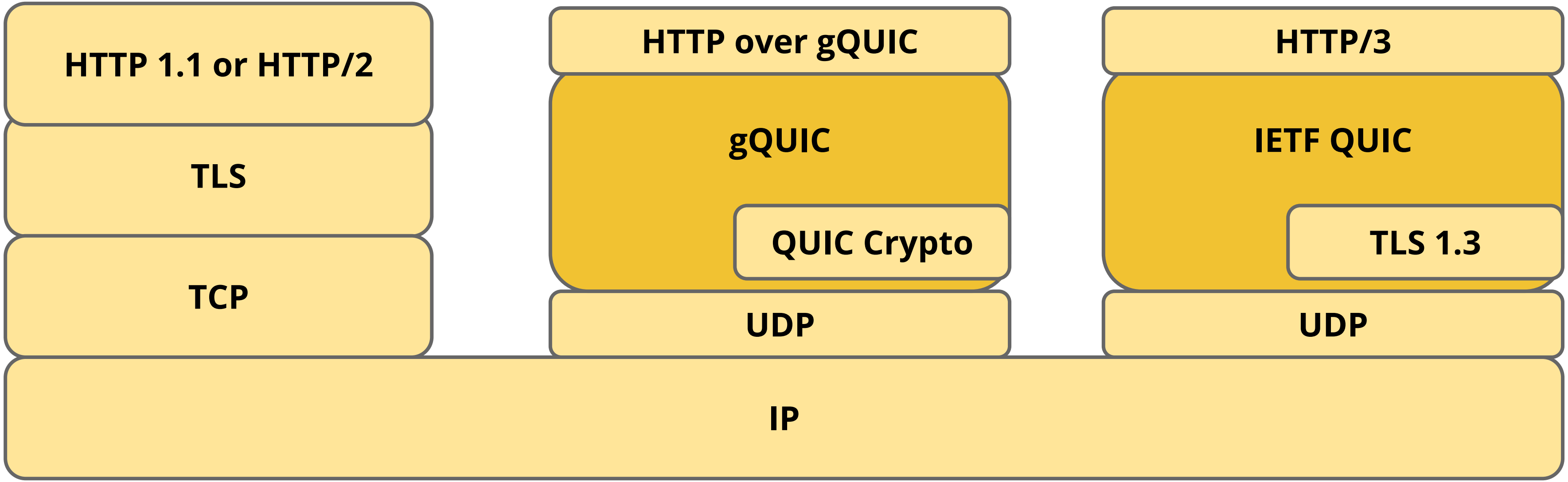
HTTP over QUIC (aka gQUIC)

QUIC

HTTP/2-like framing using HPACK



HTTP/3: The next version of HTTP



QUIC Status

IETF:

specifications in-progress, RFCs likely in 2021

Implementations:

Apple, Facebook, Fastly, Firefox, F5, Google, Microsoft ...

Server deployments have been going on for a while

Akamai, Cloudflare, Facebook, Fastly, Google …

Clients are at different stages of deployment

Chrome, Firefox, Edge, Safari

iOS, MacOS

Chrome experimenting in Stable

Background

Target Workload: DASH video streaming

Status Quo: HTTP 1.1 over TLS

DASH clients send a sequence of HTTP requests for audio and video segments

Adjustable bitrate(ABR) algorithm decided what format to request

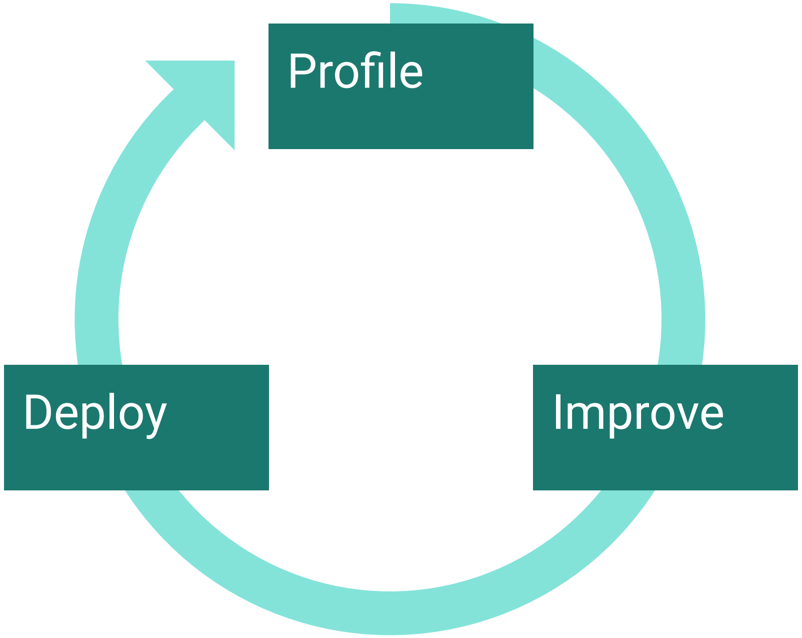
Key Objectives: Improved quality of experience, high CPU efficiency, MORE QUIC!

CPU: January 2017 at 2x HTTPS 1.1

Early implementations were **3.5x**

Obvious fixes reduced this to **2x**

- Don't call costly functions multiple times
- No allocations in the data path
- Minimize copies
- Workload specific datastructures



Challenge: Keeping QUIC running

Currently supports 4 gQUIC versions and 3 IETF QUIC drafts, including 2 invariants

QUIC was 1/3rd of Google's egress!

A bit like changing the tires while driving

Extra Challenges

Library used by two internal server binaries, Chromium and Envoy

- Lots of interfaces and visitors

Very ‘flexible’

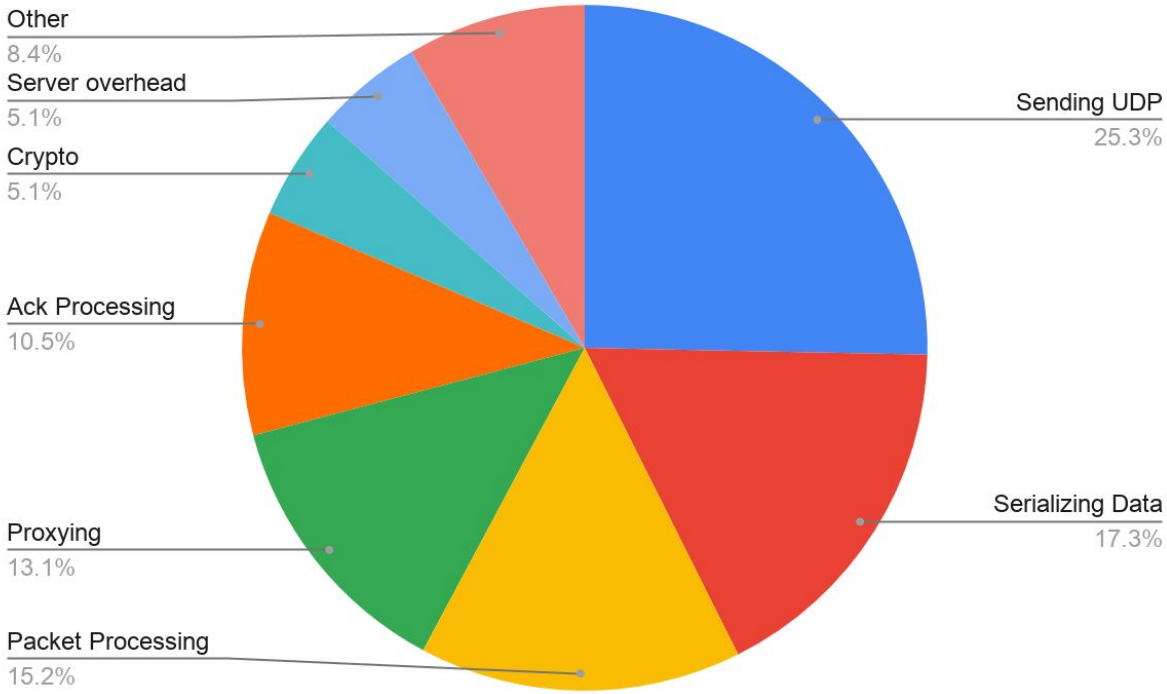
- 4 congestion controllers, 3 crypto handshakes,
- MANY experimental options

Originally written without CPU efficiency in mind

CPU: January 2017 at 2x

Only sendmsg and one memcpy are obviously costly

Other CPU users are tiny



CPU rules of thumb

Register	**1 cycle**	**~32**
L1 Cache	**1-3 cycles**	**32k**
Branch Misprediction	**~10 cycles**	
L2 Cache	**~10 cycles**	**128k-256k**
L3 Cache	**~100 cycles**	**1MB/core**
Main Memory	**250 cycles**	**Huge**

Spatial locality and **temporal locality** matter!

Modern Compilers and CPUs try to hide this

Compilers

- Inlining functions
- Reordering instructions
- De-virtualization

CPU

- Cache prefetch
- Branch prediction

Goal: make these optimizations **easier** or **possible**

Prefetch and predictors reward close, consistent access

Sending and Receiving UDP

Why is sending and receiving so important?

UDP sending is 25% of the CPU in our workload

>50% in some environments and benchmarks

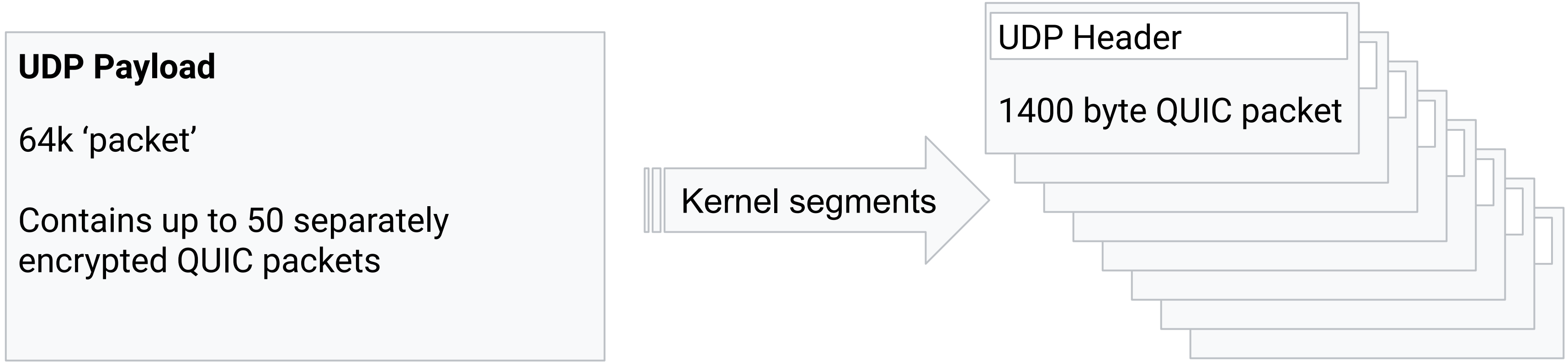
UDP sendmsg is up to **3.5x** the cycle/byte of TCP in Linux*

UDP sendmmsg only saves a syscall per packet vs sendmsg

Has very few restrictions, multiple destinations, etc

Sending UDP Packets: UDP GSO in Linux

UDP GSO is **7%** faster than TCP GSO**



Pacing sent **1 UDP packet** at once, had to make it **bursty**

Sending UDP Packets: kernel bypass

Bypassing some of the the kernel **can** be faster than UDP sockets on Linux

- DPDK is full kernel bypass
- AF_XDP is a new kernel API as fast as DPDK*
- Google has a software NIC**

Cons: Increased complexity, escalated privileges, dedicated machines

Alternately, everything in the kernel can be fast***

Sending UDP Packets: UDP GSO with hardware offload

Hardware offload is now much more common and provides another **2-3x**

Mellanox mlx5, Intel ixgbem, likely others

Cumulative acceleration is ~10x ideally and 5x in typical cases

=> 50% CPU usage(worst case) => 5% CPU usage => 2x improvement

GSO with hardware offload can be the best of both worlds

Sending UDP Packets: UDP GSO with pacing offload

Pacing offload can enable larger sends (patchset)
    ie: 16 packets instead of 4 packets

The API and implementation are not yet finalized
    Currently 1 to 15ms increments
    => If you’re interested in using it, please provide feedback and/or benchmarks

GSO with pacing and hardware offload is very promising

Receiving UDP Packets

mmap RX_RING was much faster

recvmmsg performance improved over time, now comparable

Using a BPF to steer by QUIC connection ID avoids thread hopping

UDP GRO(patch) improves receive CPU 35%

Detailed Optimizations

Fast path common cases

Observation: Packets are sent in order and most packets arrive in order

- Ack processing
- Data receipt
- Bulk data transmission

Optimizing for 1 STREAM frame/packet saved **5%** alone!

Efficiently Writing Data

Old: On every send, a packet data-structure copied all frames and data
Packets were retransmitted, not data or frames

New: Move data ownership to streams
Enabled bulk application writes
Eliminated a buffer allocation per packet
Buffers remain contiguous
Allowed the application to transfer data ownership

Makes QUIC more like TCP!

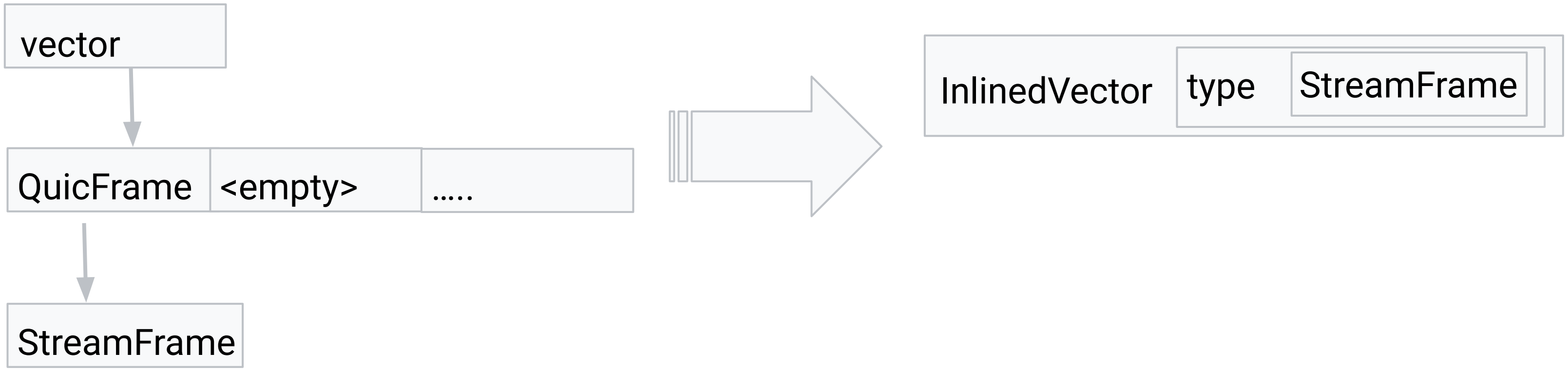
Increasing memory locality

Eliminate pointer chasing and virtual methods

Place all connection state in a single arena

Inline commonly used fields

Example



Send fewer ACKs

Acknowledgement processing is expensive on servers
Sending packets is expensive, particularly on mobile clients

BBR works well, because it's rate-based

Critical(25% reduction) to achieving parity with TCP in Quicly benchmarks

IETF draft: draft-iyengar-quic-delayed-ack

TCP already creates 'stretch ACKs'

Feedback Directed Optimization (aka FDO)

Code shared with Chromium ➪ lots of interfaces

FDO can de-virtualize and prefetch

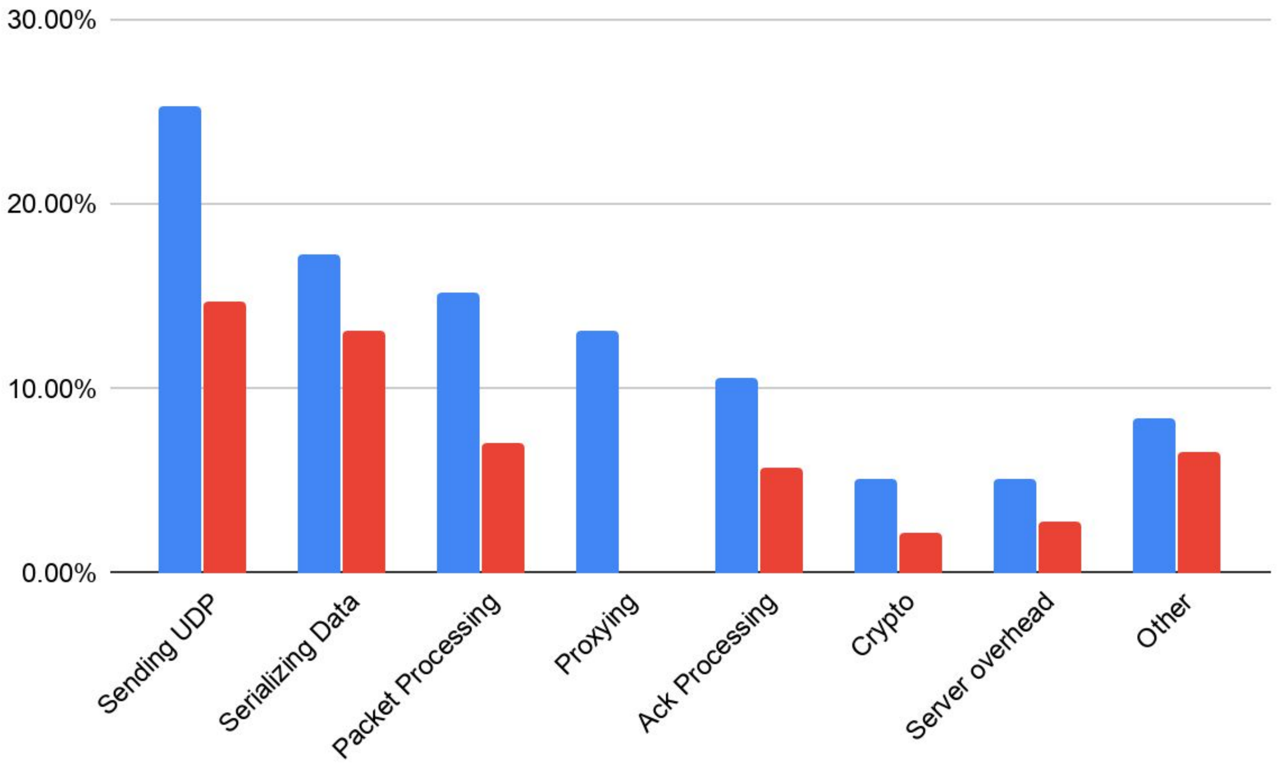
Userspace enables experimentation & flexibility ➪ great monitoring, analysis tools

FDO discovers tracing is unused >99% of the time

ThinLTO for cross-module optimization

15% CPU savings

Q4 2017 vs Today



What is the future?

Sending and Receiving UDP: Wider GSO support

Fast UDP send and receive APIs for more platforms

- Android, Windows, iOS...
- Hardware GSO widely supported : As fast as TCP TSO

Sending UDP: Crypto offload

"*Making QUIC Quicker with NIC Offload*"

Once UDP send are fast, symmetric Crypto is ~30% of CPU

Offload on the receive side enables reordering in the NIC

Open Question: What is the right API?

Open Question: Is QUIC offload worthwhile?

TSO has mixed benefits, especially at lower bandwidths

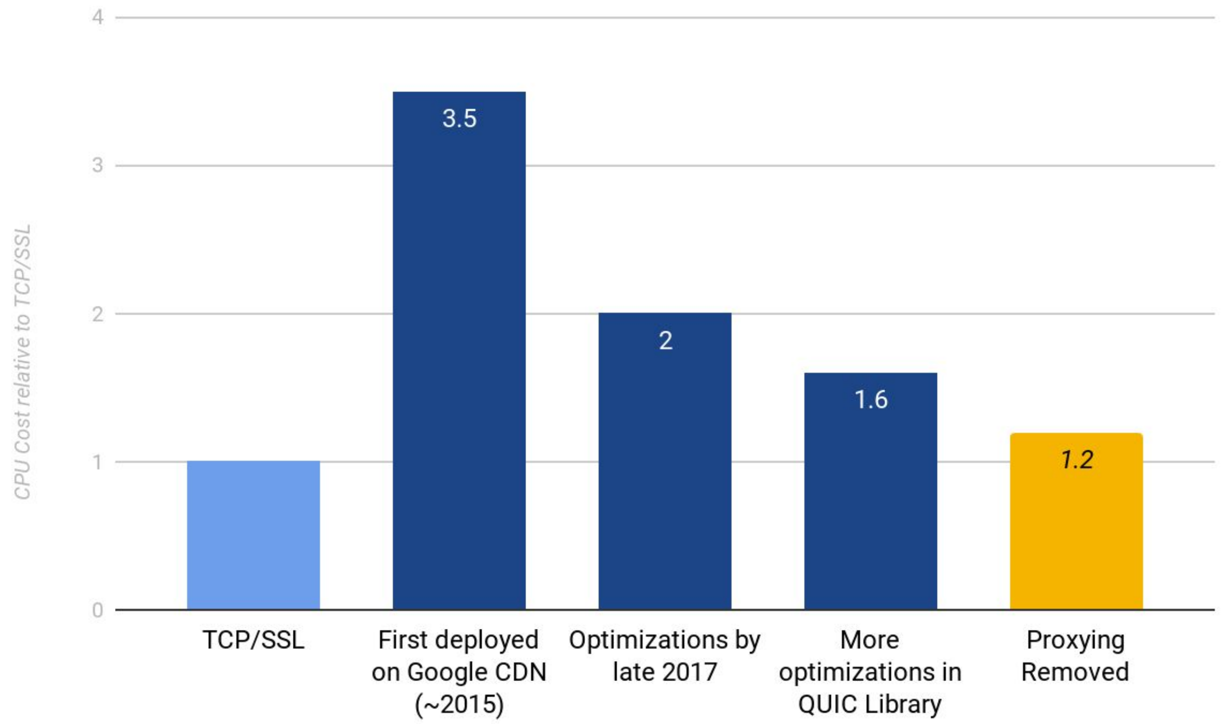
With symmetric offload, QUIC should be as fast as kTLS

IETF QUIC: Optimizing header encryption

IETF QUIC adds header protection, requiring 2-pass encryption
- Encrypts header bits and the packet number for privacy
- Small encryption operations are MUCH more expensive than bulk

Known Optimizations
- Encrypt multiple headers in one pass (WinQUIC, Litespeed)
- Calculate header protection in parallel (PicoTLS Fusion)
  - PicoTLS Benchmarks: 1, 2

Will HTTP/3 be more efficient than HTTP/1?



Questions?

IETF WG Page
Base IETF drafts: transport, recovery, tls, http, qpack, invariants
Chromium QUIC Code: cs.chromium.org
Chromium QUIC page: www.chromium.org/quic
Profiling a warehouse scale computer paper
QUIC SIGCOMM Tutorial