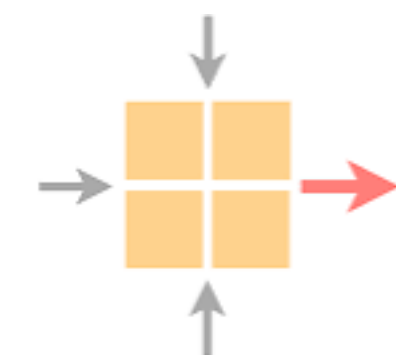


Application-Layer Parsing in eBPF

Laurin Brandner, Ayush Mishra, Sebastiano Miano,
Aurojit Panda, Gianni Antichi, Laurent Vanbever



eBPF is all the rage!

Cilium, Calico, Katran

redirect traffic in the kernel

iptables

allows dynamic matching with eBPF

DeepFlow, Hubble

collect runtime metrics

These projects do not handle L7 in eBPF

Instead, they fall back to user space for L7 support

Cilium, Calico, Katran

redirect traffic in the kernel

iptables

allows dynamic matching with eBPF

DeepFlow, Hubble

collect runtime metrics

There is a need for L7 support in eBPF

Container networks

are commonly configured with L7 policies (e.g. request authorization)

Firewalls

mitigate ever-increasing L7 DDoS attacks (e.g. Rapid Reset)

Observability tools

parse L7 payloads to monitor increasingly distributed applications

Is Application-Layer Parsing Possible in eBPF?

Enforcing Application-Layer Policies in eBPF

Laurin Brandner 
ETH Zürich
laurinb@ethz.ch

Ayush Mishra 
ETH Zürich
aymishra@ethz.ch

Sebastiano Miano 
Nvidia
smiano@nvidia.com

Aurojit Panda 
NYU
apanda@cs.nyu.edu

Gianni Antichi 
Politecnico di Milano
gianni.antichi@polimi.it

Laurent Vanbever 
ETH Zürich
lvanbever@ethz.ch

Abstract—Service meshes have recently emerged as the de-facto standard for deploying microservices. Conceptually, they provide a uniform abstraction for inter-process communication (IPC) between services by implementing common networking mechanisms—such as encryption, routing, and load balancing—and by allowing these mechanisms to be configured and composed through high-level policies. Supporting these policies, however, comes with a significant performance cost, since service meshes interpose proxies (“sidecars”) on the data path between every service.

This paper presents Beeline, a fast path for service meshes which can enforce the vast majority of application-layer policies seen in the wild directly in kernel space. Given high-level policies, Beeline automatically synthesizes an eBPF-based data plane which enforces them in the kernel. Beeline accelerates existing microservices without any code modification, and transparently falls back to existing service proxies (the slow path) for the few unsupported policies.

We fully implemented Beeline, with support for both TLS and HTTP/2. Compared to state-of-the-art service meshes, Beeline reduces the median request latency of realistic applications by up to 6× while sustaining 3× more throughput.

Index Terms—eBPF, service mesh, service proxy, HTTP

I. INTRODUCTION

Modern data center applications are no longer monolithic: they are assembled from swarms of microservices [1], [2], [3], [4] or “pods”, stitched together by service meshes [5], [6]. Service meshes simplify development by abstracting away the networking layer, making applications easier to deploy and manage. This approach has fueled the widespread adoption of platforms such as Istio [7] and Linkerd [8].

A key component of these service meshes is the service proxy, also known as “sidecar”. A service proxy acts as the data plane of the service mesh: it processes all the inter-pod traffic and enforces policies defined at the transport (L4) and/or application layer (L7). For example, Istio relies upon Envoy [9]—a popular service proxy—to enforce L4 load balancing or L7 request authorization.

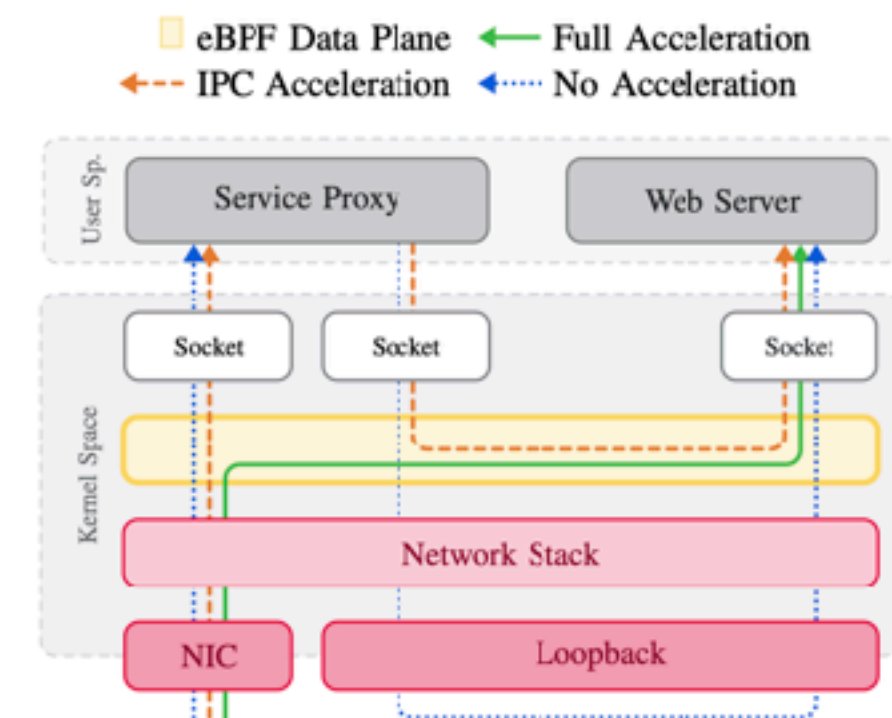


Figure 1: Service proxies typically route inter-pod traffic through the loopback device (blue line). State-of-the-art service meshes optimize IPC by rerouting the traffic using eBPF (orange line). Beeline enforces L7 policies in the kernel, eliminating the service proxy from the critical path (green line).

generality simplifies deployment but carries substantial performance overheads. (2) Service proxies increase the amount of inter-process communication (IPC), which in turn increases both CPU requirements and processing time. We visualize this extra cost in Figure 1 (see the blue line): each message that the web server receives is routed through the loopback device, traversing the network stack and crossing the user-kernel-boundary *three* times.

To reduce this overhead, state-of-the-art service meshes such as Cilium [11] and Calico [12] have mainly relied upon two techniques. The first technique aims at minimizing the IPC

It's possible!

And improves throughput of container networks with L7 policies by up to 3x.

Enforcing Application-Layer Policies in eBPF

Laurin Brandner 
ETH Zürich
laurinb@ethz.ch

Ayush Mishra 
ETH Zürich
aymishra@ethz.ch

Sebastiano Miano 
Nvidia
smiano@nvidia.com

Aurojit Panda 
NYU
apanda@cs.nyu.edu

Gianni Antichi 
Politecnico di Milano
gianni.antichi@polimi.it

Laurent Vanbever 
ETH Zürich
lvanbever@ethz.ch

Abstract—Service meshes have recently emerged as the de-facto standard for deploying microservices. Conceptually, they provide a uniform abstraction for inter-process communication (IPC) between services by implementing common networking mechanisms—such as encryption, routing, and load balancing—and by allowing these mechanisms to be configured and composed through high-level policies. Supporting these policies, however, comes with a significant performance cost, since service meshes interpose proxies (“sidecars”) on the data path between every service.

This paper presents Beeline, a fast path for service meshes which can enforce the vast majority of application-layer policies seen in the wild directly in kernel space. Given high-level policies, Beeline automatically synthesizes an eBPF-based data plane which enforces them in the kernel. Beeline accelerates existing microservices without any code modification, and transparently falls back to existing service proxies (the slow path) for the few unsupported policies.

We fully implemented Beeline, with support for both TLS and HTTP/2. Compared to state-of-the-art service meshes, Beeline reduces the median request latency of realistic applications by up to 6× while sustaining 3× more throughput.

Index Terms—eBPF, service mesh, service proxy, HTTP

I. INTRODUCTION

Modern data center applications are no longer monolithic: they are assembled from swarms of microservices [1], [2], [3], [4] or “pods”, stitched together by service meshes [5], [6]. Service meshes simplify development by abstracting away the networking layer, making applications easier to deploy and manage. This approach has fueled the widespread adoption of platforms such as Istio [7] and Linkerd [8].

A key component of these service meshes is the service proxy, also known as “sidecar”. A service proxy acts as the data plane of the service mesh: it processes all the inter-pod traffic and enforces policies defined at the transport (L4) and/or application layer (L7). For example, Istio relies upon Envoy [9]—a popular service proxy—to enforce L4 load balancing or L7 request authorization.

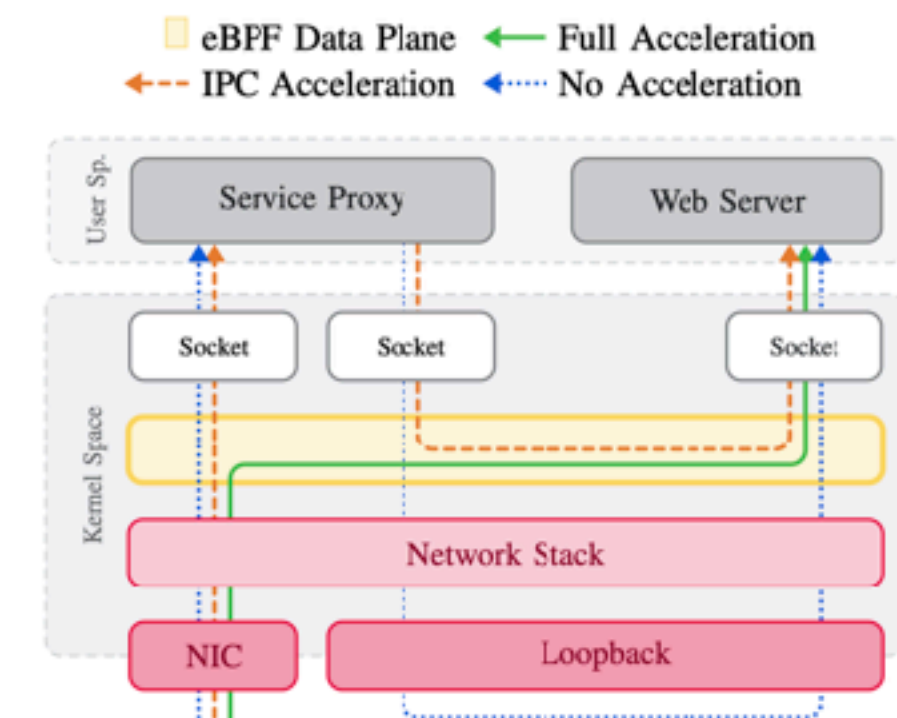


Figure 1: Service proxies typically route inter-pod traffic through the loopback device (blue line). State-of-the-art service meshes optimize IPC by rerouting the traffic using eBPF (orange line). Beeline enforces L7 policies in the kernel, eliminating the service proxy from the critical path (green line).

generality simplifies deployment but carries substantial performance overheads. (2) Service proxies increase the amount of inter-process communication (IPC), which in turn increases both CPU requirements and processing time. We visualize this extra cost in Figure 1 (see the blue line): each message that the web server receives is routed through the loopback device, traversing the network stack and crossing the user-kernel-boundary *three* times.

To reduce this overhead, state-of-the-art service meshes such as Cilium [11] and Calico [12] have mainly relied upon two techniques. The first technique aims at minimizing the IPC

It's possible!

And improves throughput of container networks with L7 policies by up to 3x.

... but how?

Many L7 protocols are deemed too complex to parse in eBPF

eBPF verification limits the offload-able complexity

Parsing L7 protocols often requires more logic and state than eBPF can afford

Two major challenges make parsing difficult

Because it often requires more state and complexity than eBPF can afford

Explicit data structure

need more logic to locate the relevant data segment in the buffer

Specialized encoding

need more logic to decode the message

Challenge 1: Explicit Data Structure

Require more state and logic to locate the relevant data segment

H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

Challenge 1: Explicit Data Structure

Require more state and logic to locate the relevant data segment

N	G	:	G	Z	I	P	\r	\n
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	C
O	N	T	E	N	T	-	T	Y

Challenge 1: Explicit Data Structure

HTTP/1.1 headers are case-insensitive

H	o	S	t	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

Challenge 1: Explicit Data Structure

HTTP/1.1 header fields may have optional whitespace

H	O	S	T	:			L	P
C	.	E	V	E	N	T	S	\r

Challenge 2: Specialized Encoding

HTTP/2 encodes header fields with a special type-length-value format

			12	10	L	P	C	.
E	V	E	N	T	S			

Challenge 2: Specialized Encoding

The type is variable-length, and also stores caching information

			?	10	L	P	C	.
E	V	E	N	T	S			

Challenge 2: Specialized Encoding

The length is also variable-length

			?	?	L	P	C	.
E	V	E	N	T	S			

Key Idea: Use a DFA to parse messages

This reduces the complexity in eBPF significantly

Locate relevant data segment

and trivially handles optional
whitespace or letter case

Operates on the encoded data

making decoding redundant

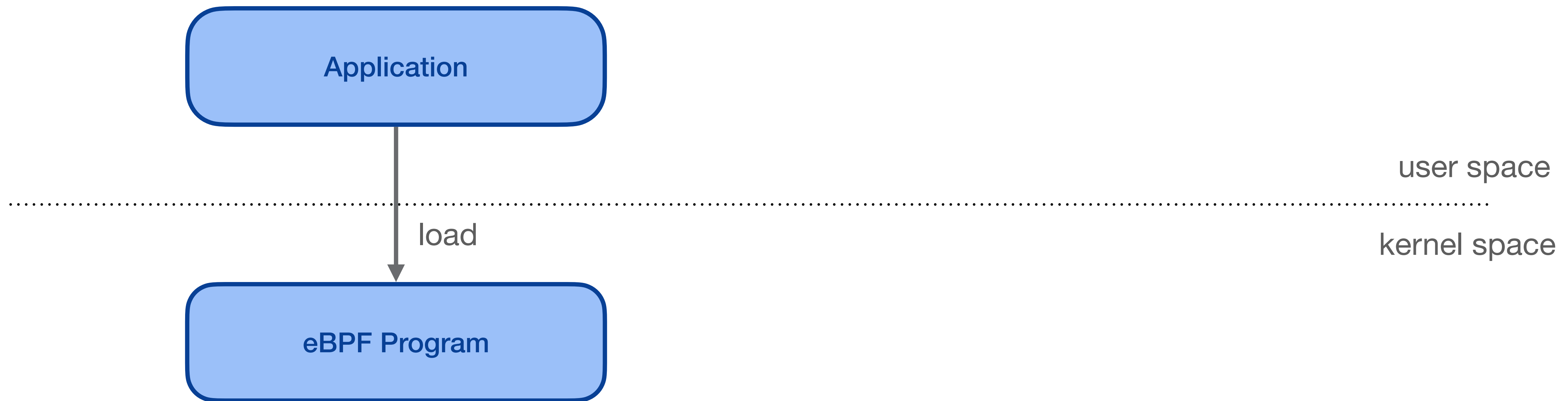
Beeper is an L7 parsing system for eBPF



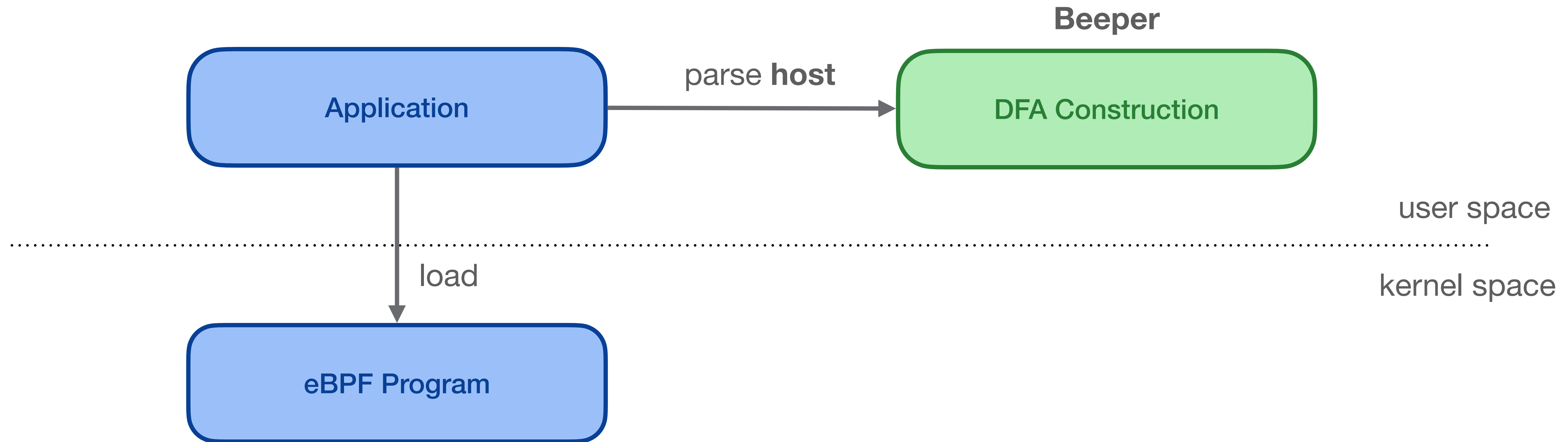
**Currently supports HTTP/1.1
and HTTP/2**

**Open-source and packaged
as a Rust crate**

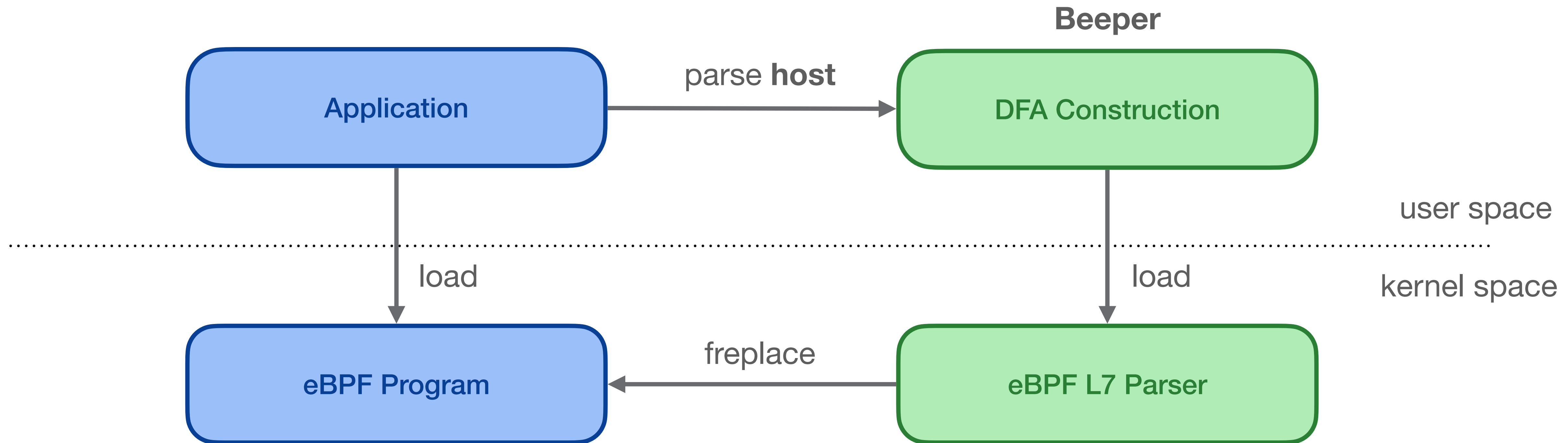
How to integrate Beeper into other programs?



Beeper constructs the DFA in user space

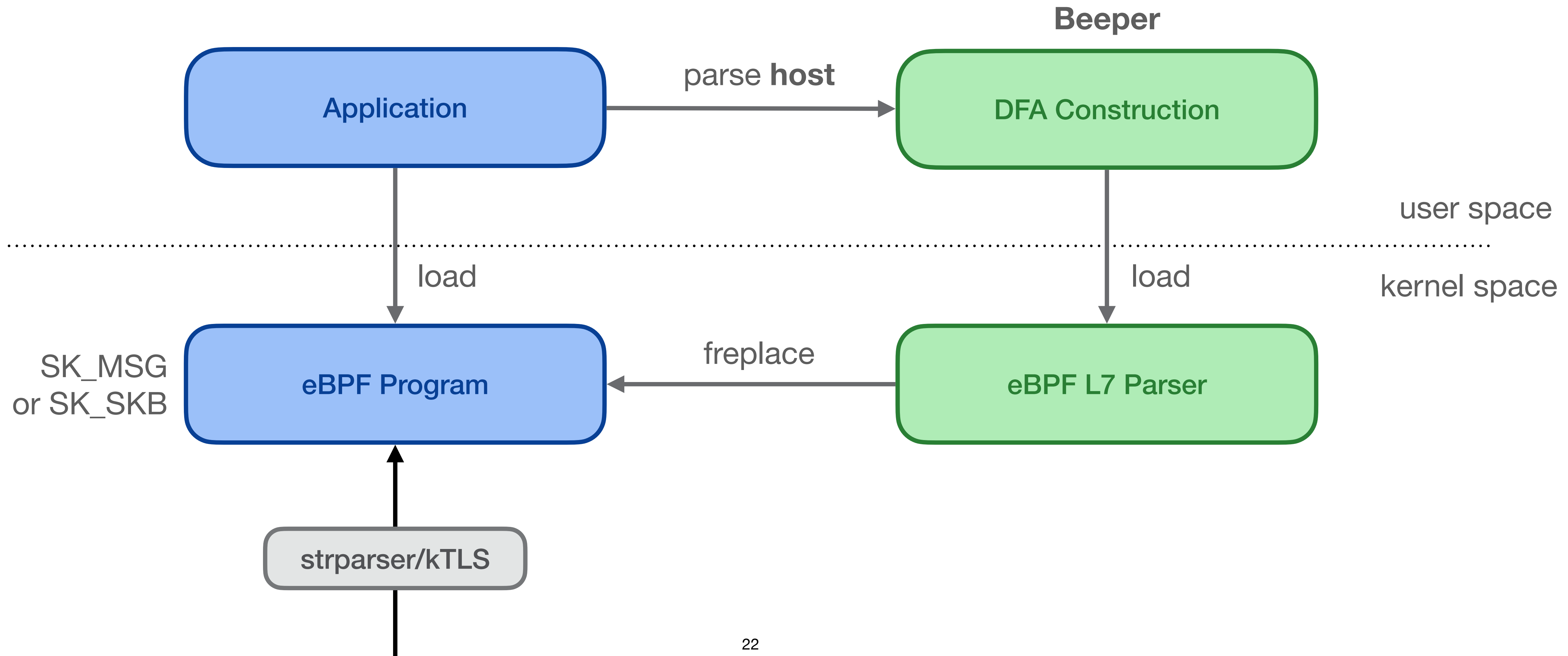


The eBPF parser integrates with freplace

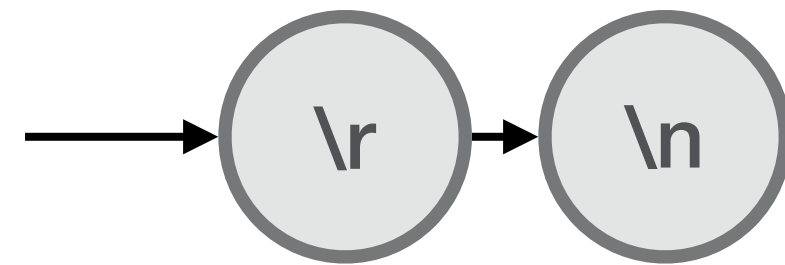


Beeper is most useful at the socket level

by integrating with stream parser and kTLS

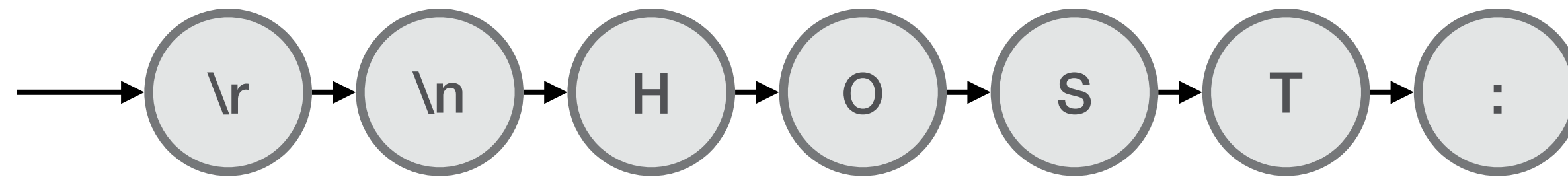


The DFA starts with CRLF to match a new HTTP/1.1 header



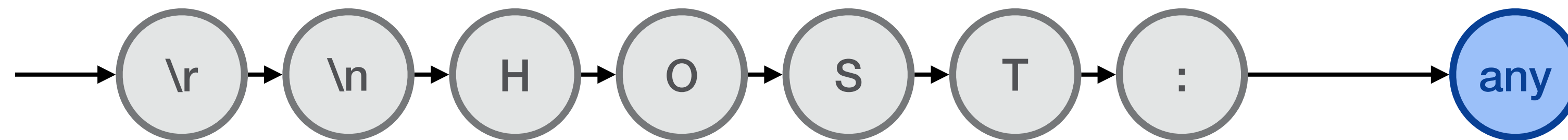
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

The following states match the header we're interested in



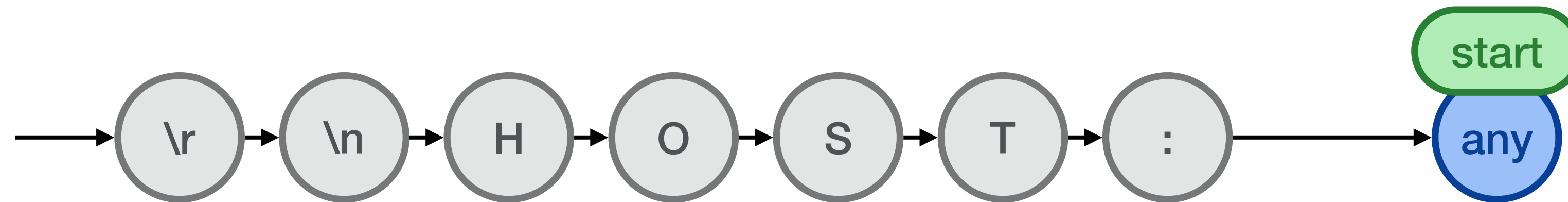
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

Whichever character follows next is the start of the header field



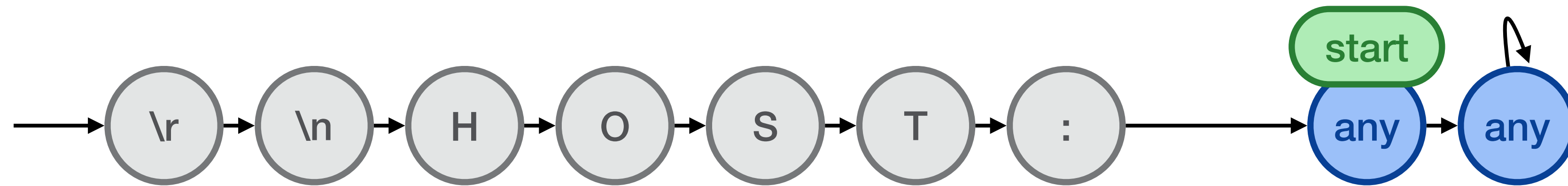
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

So we add an action that tells eBPF to save the current offset



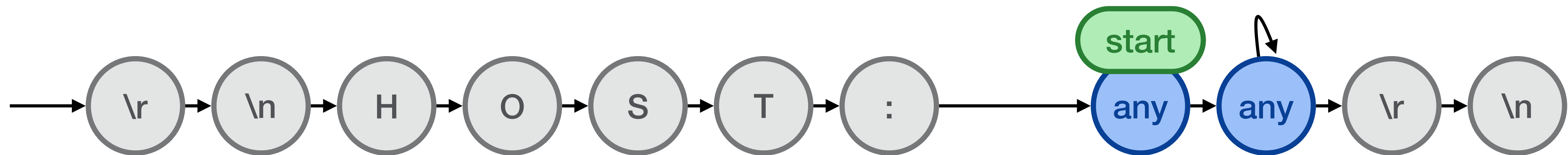
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

The following characters belong to the header



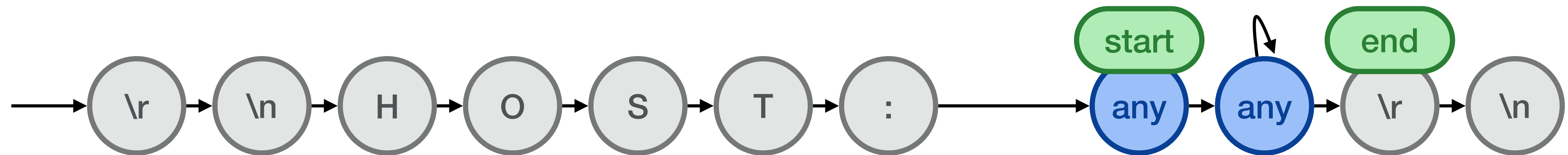
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

Until we find the next CRLF



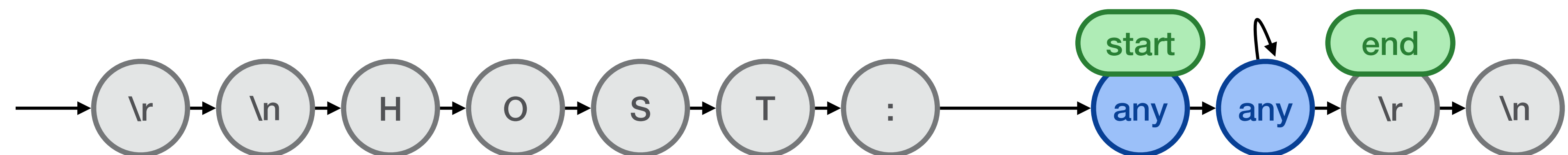
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

We add an action that stores the current offset



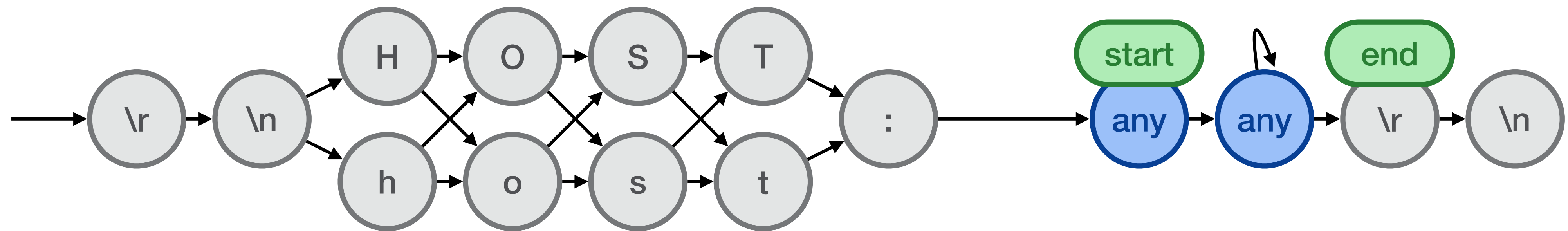
H	O	S	T	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

How to match case insensitively?



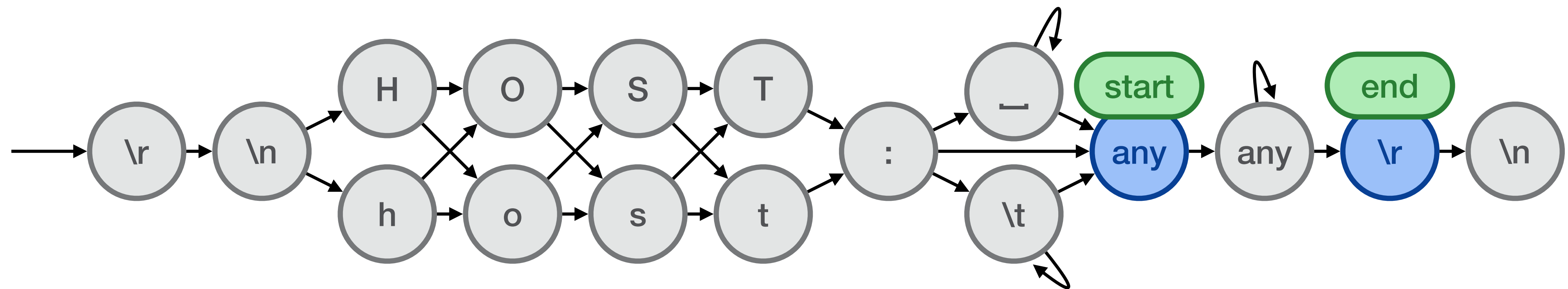
H	o	S	t	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

By adding optional transitions to the beginning of the pattern



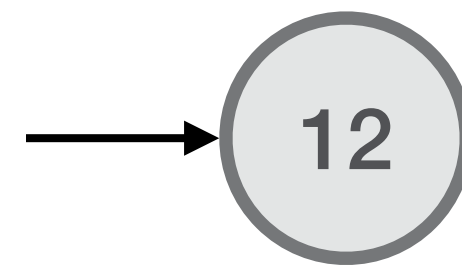
H	o	S	t	:	L	P	C	.
E	V	E	N	T	S	\r	\n	

This also makes it easy to handle optional whitespace



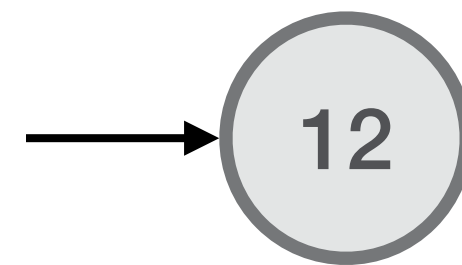
H	O	S	T	:	_	_	L	P
C	.	E	V	E	N	T	S	\r

The same approach makes it possible to parse encoded HTTP/2 headers



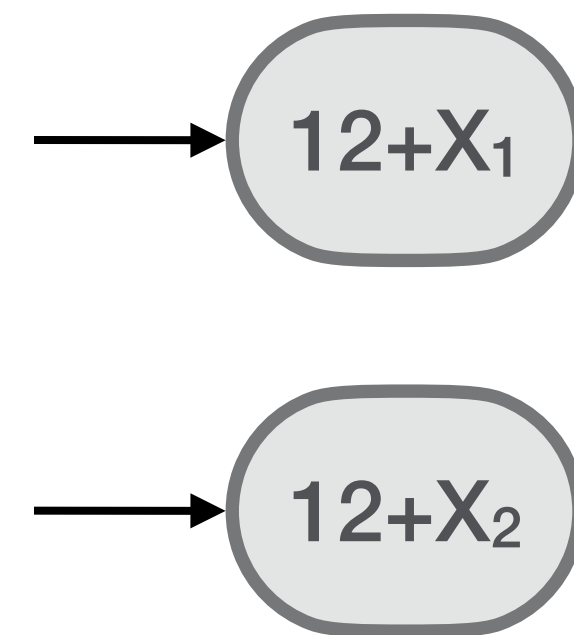
			12	10	L	P	C	.
E	V	E	N	T	S			

The type also contains caching information, making it hard to decode in eBPF



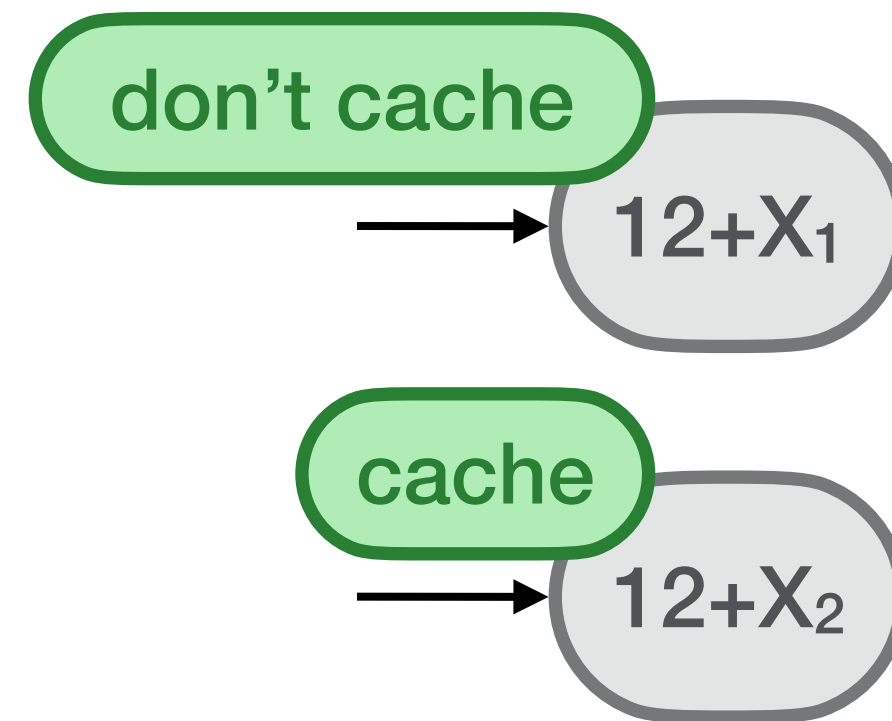
			?	10	L	P	C	.
E	V	E	N	T	S			

There's only a handful of types + caching behavior



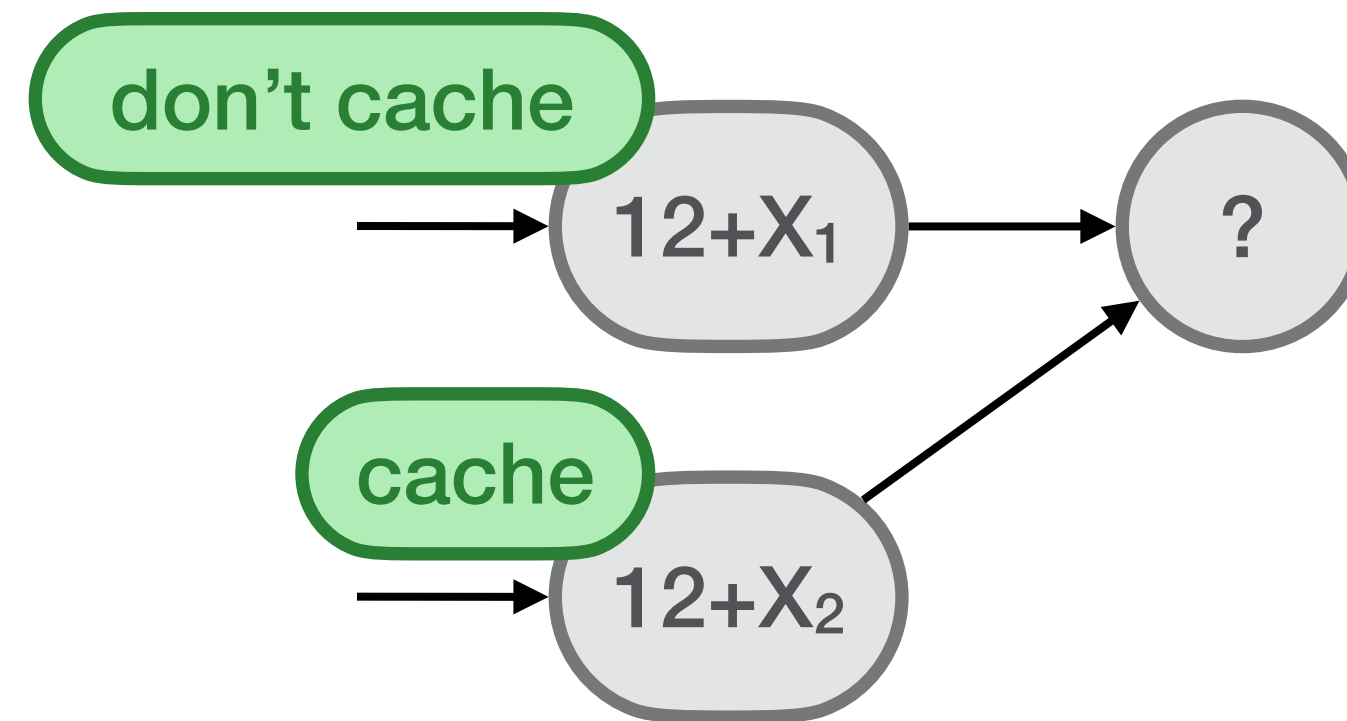
			?	10	L	P	C	.
E	V	E	N	T	S			

There's only a handful of types + caching behavior



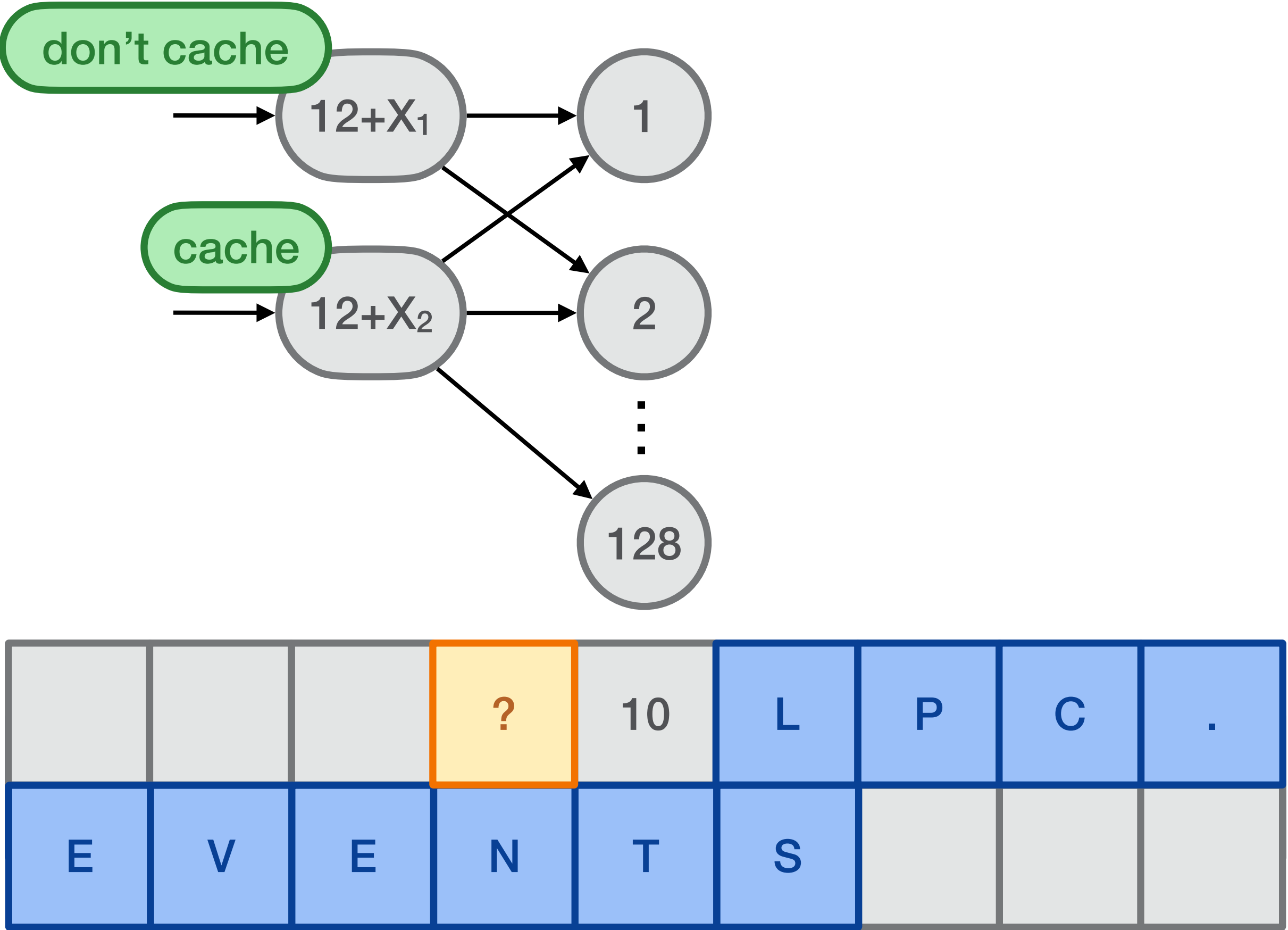
			?	10	L	P	C	.
E	V	E	N	T	S			

When constructing the DFA, we don't know what the length of the header will be

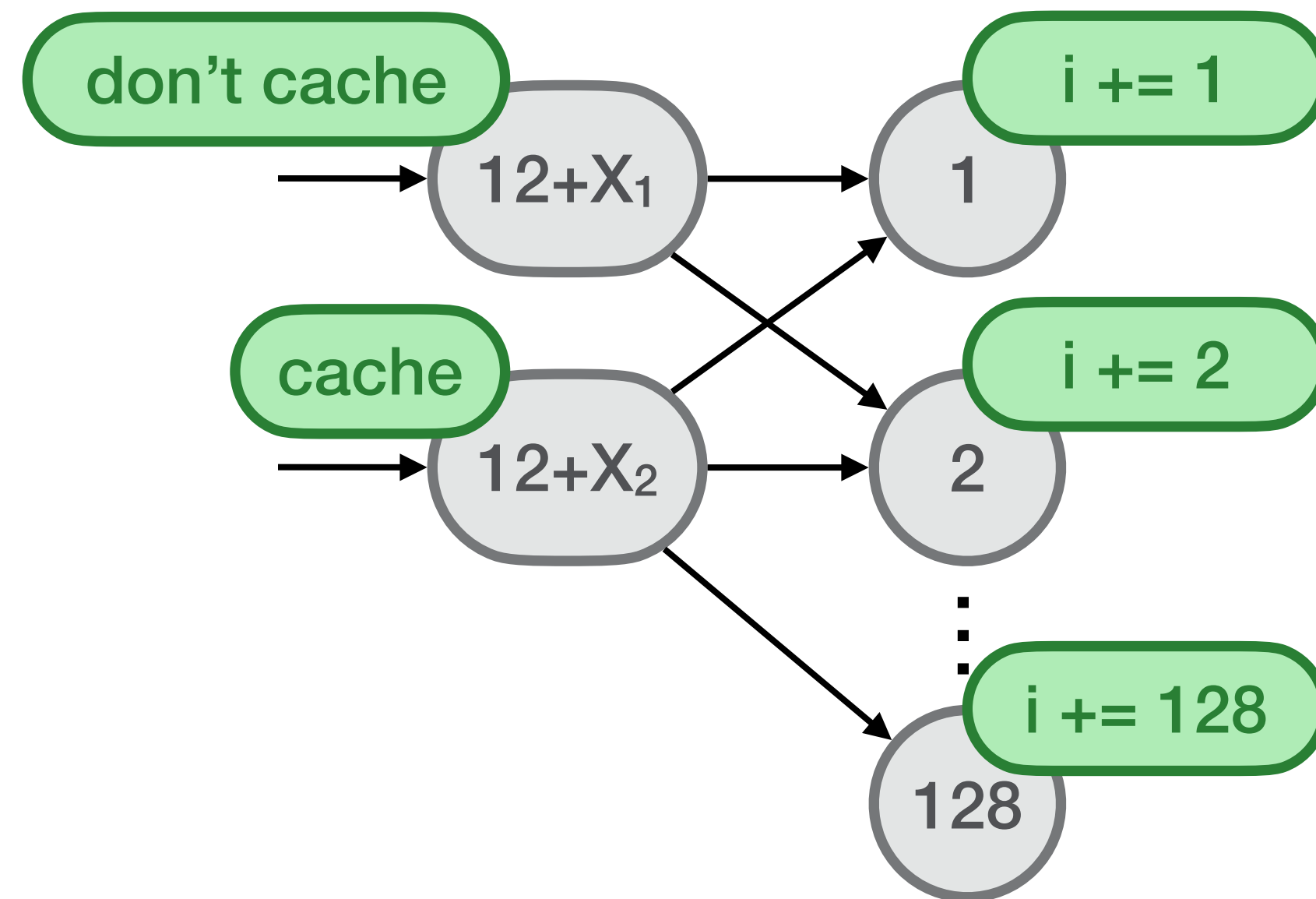


			?	10	L	P	C	.
E	V	E	N	T	S			

But we can iterate all possible bytes that can occur

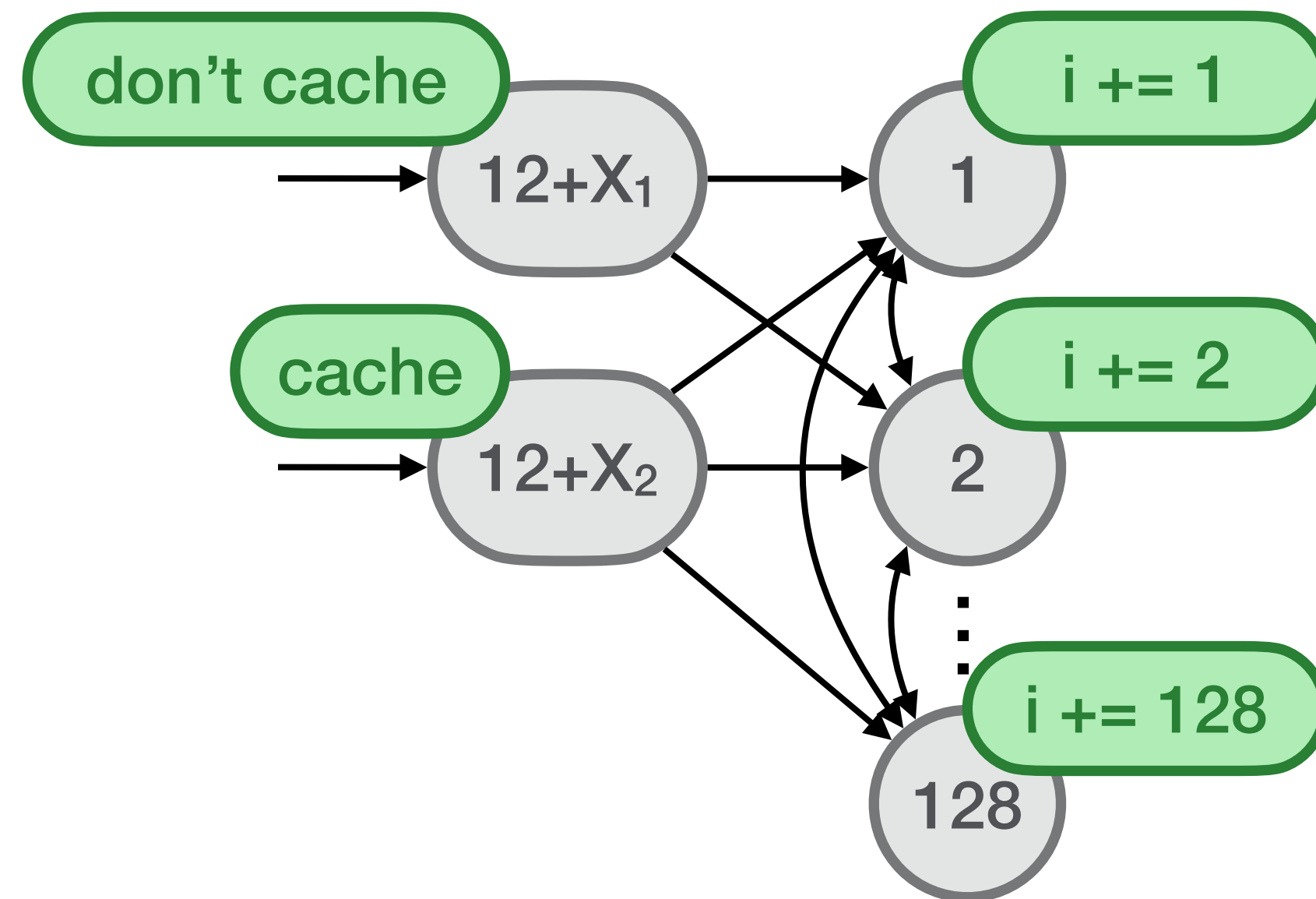


All of these states increase a length counter



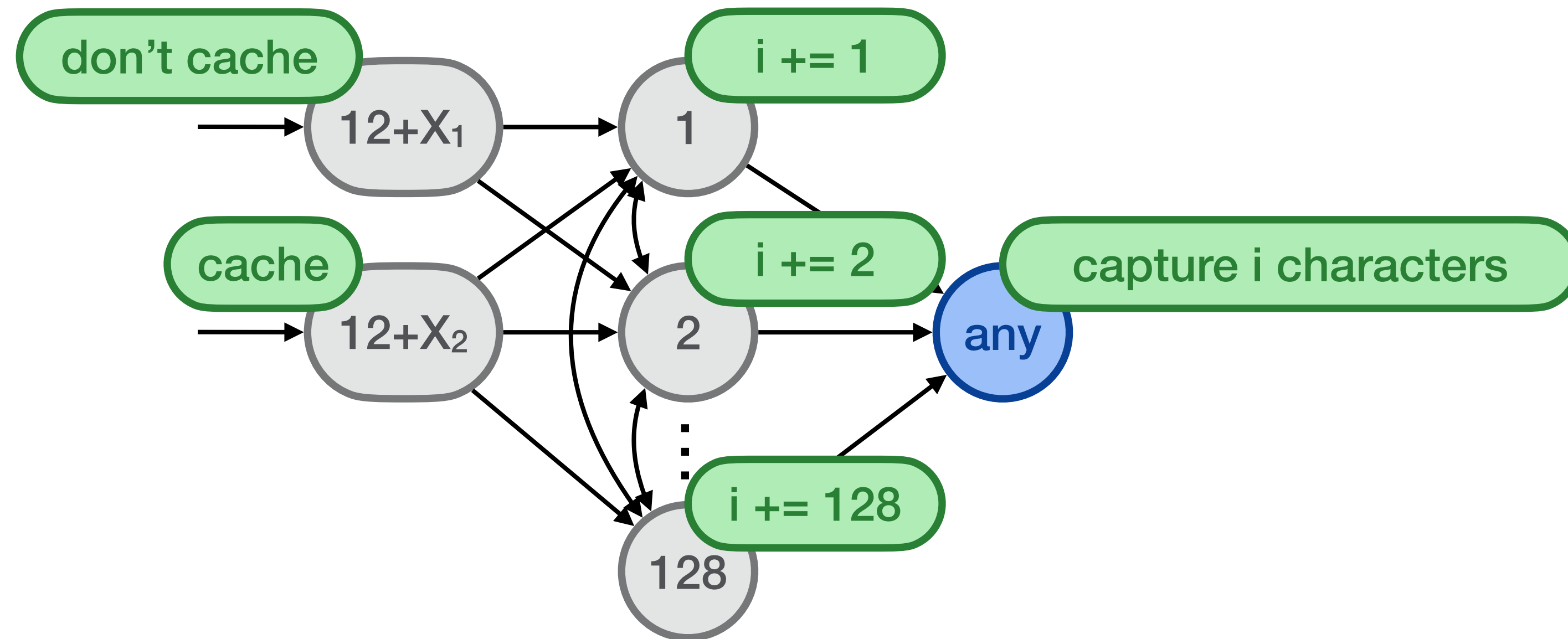
			?	10	L	P	C	.
E	V	E	N	T	S			

These states are interconnected to account for variable-length integers



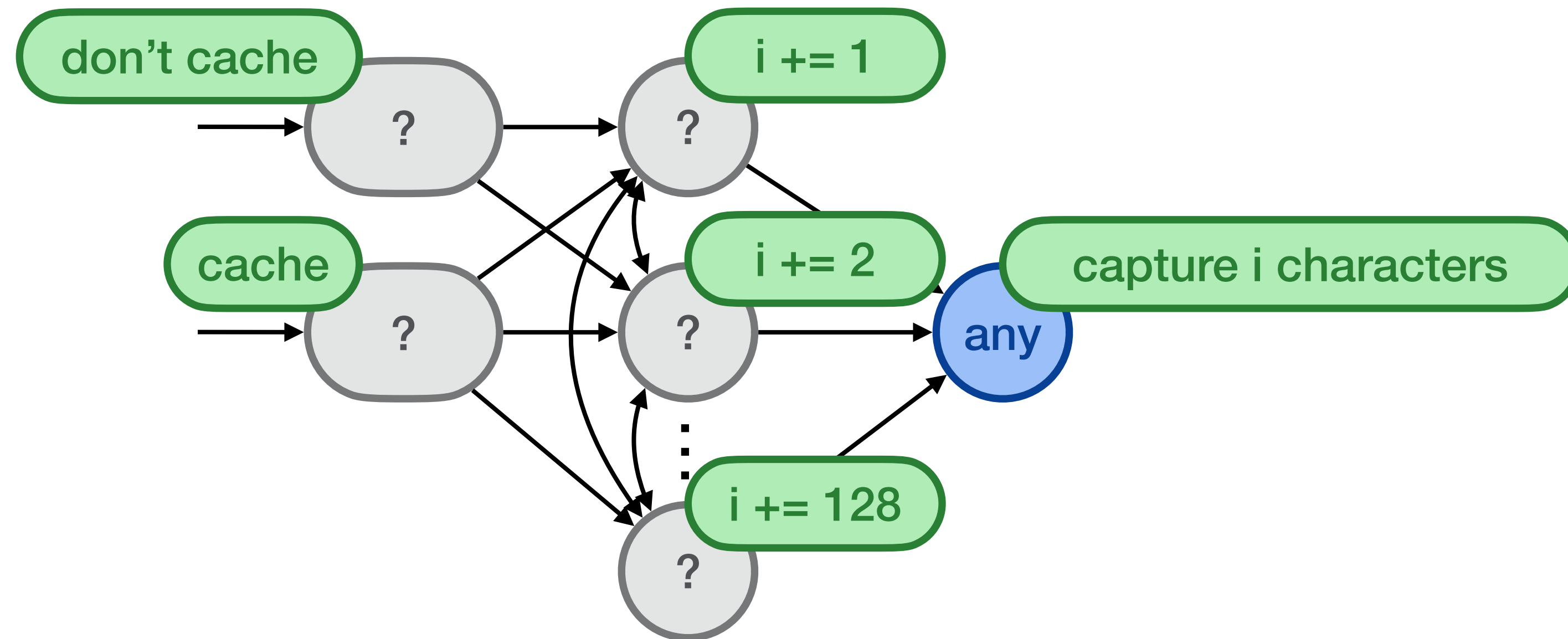
			?	10	L	P	C	.
E	V	E	N	T	S			

At this point we know how long the header will be



			?	10	L	P	C	.
E	V	E	N	T	S			

Remember that in eBPF, this data is encoded



			?	?	L	P	C	.
E	V	E	N	T	S			

What to use Beeper for?



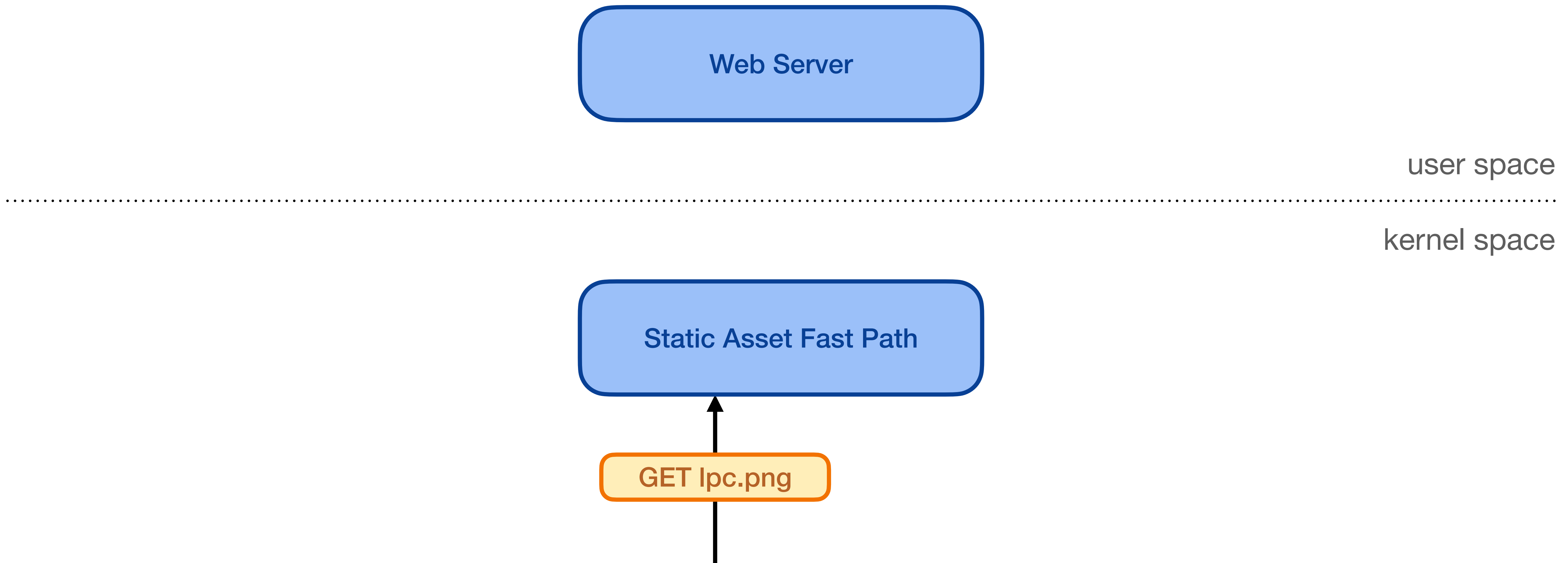
What to use Beeper for?



This talk: Accelerating a web server with eBPF

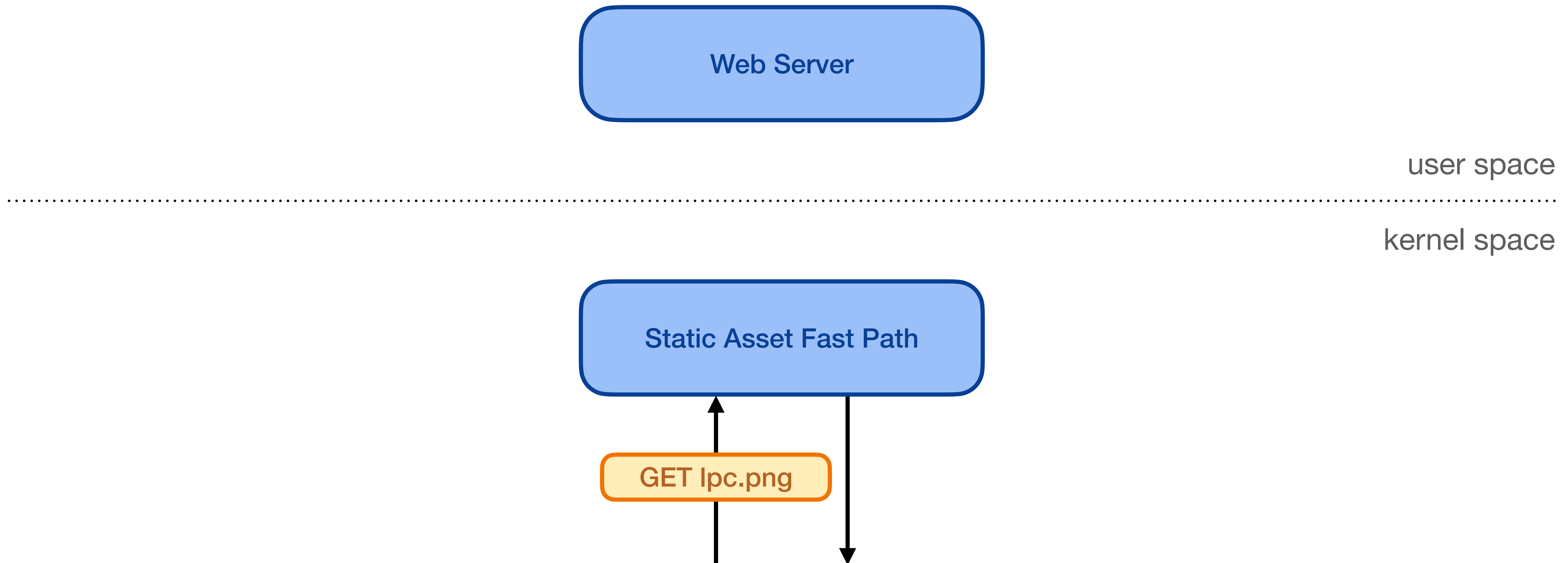
Serving static assets with eBPF

The fast path checks if it can serve this request from the kernel



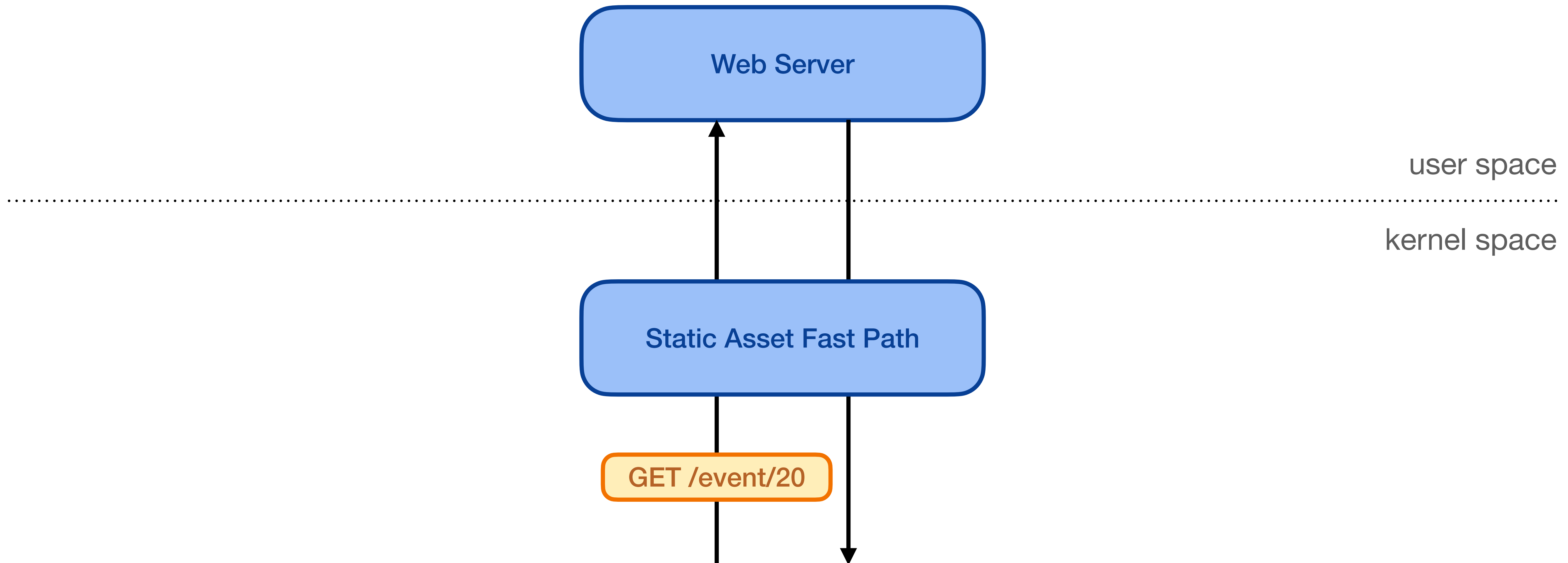
Serving static assets with eBPF

The fast path short-circuits the web server if it can



Serving static assets with eBPF

The fast path falls back to user space for dynamic content



Short-circuiting the web server introduces two problems

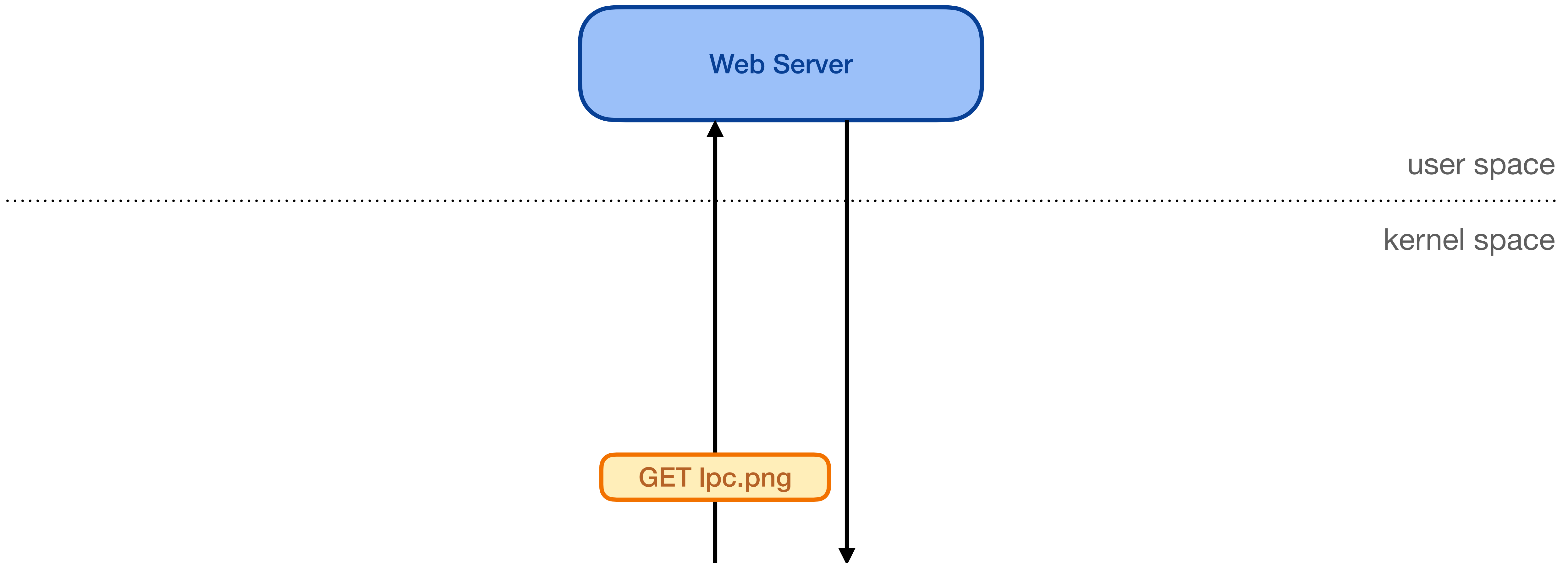
HTTP/2 caches headers

How to share this cache efficiently between eBPF and user space?

SK_SKB cannot respond with assets larger than 16KB

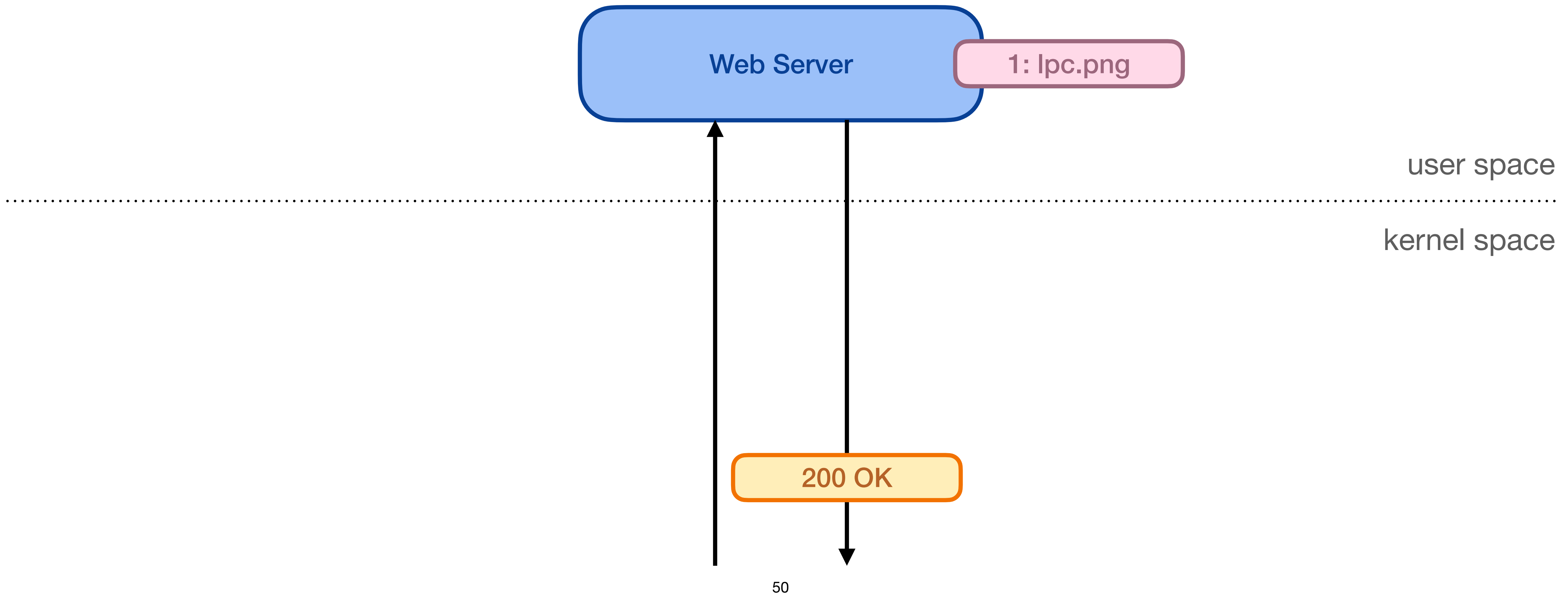
How to serve large assets to remote clients?

Challenge 1: HTTP/2 caches headers



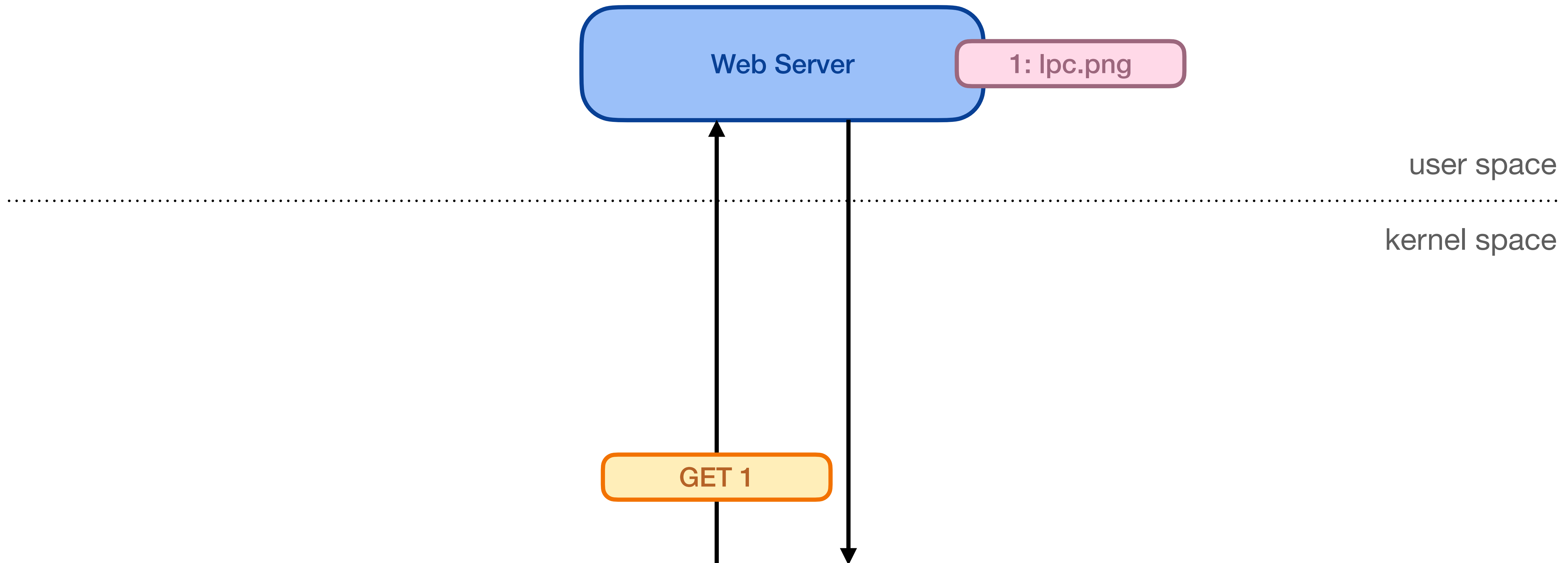
Challenge 1: HTTP/2 caches headers

The server caches the headers of each request



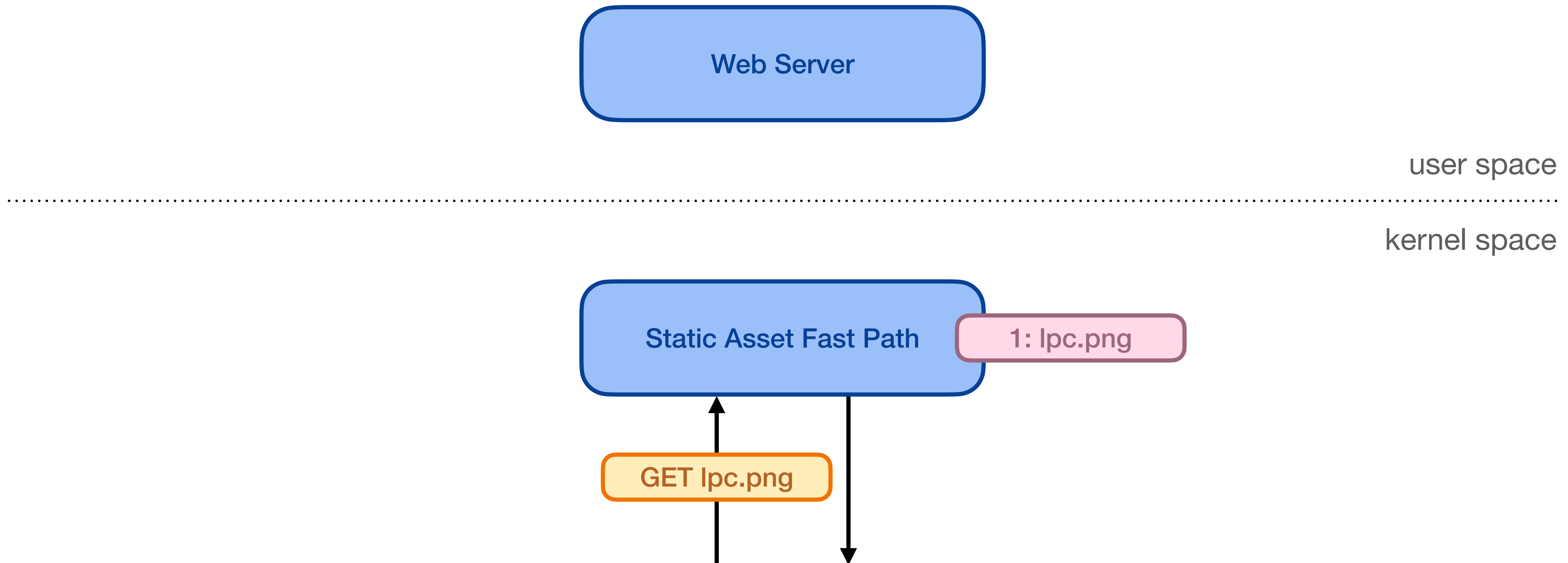
Challenge 1: HTTP/2 caches headers

The sender can now send header references to save bandwidth



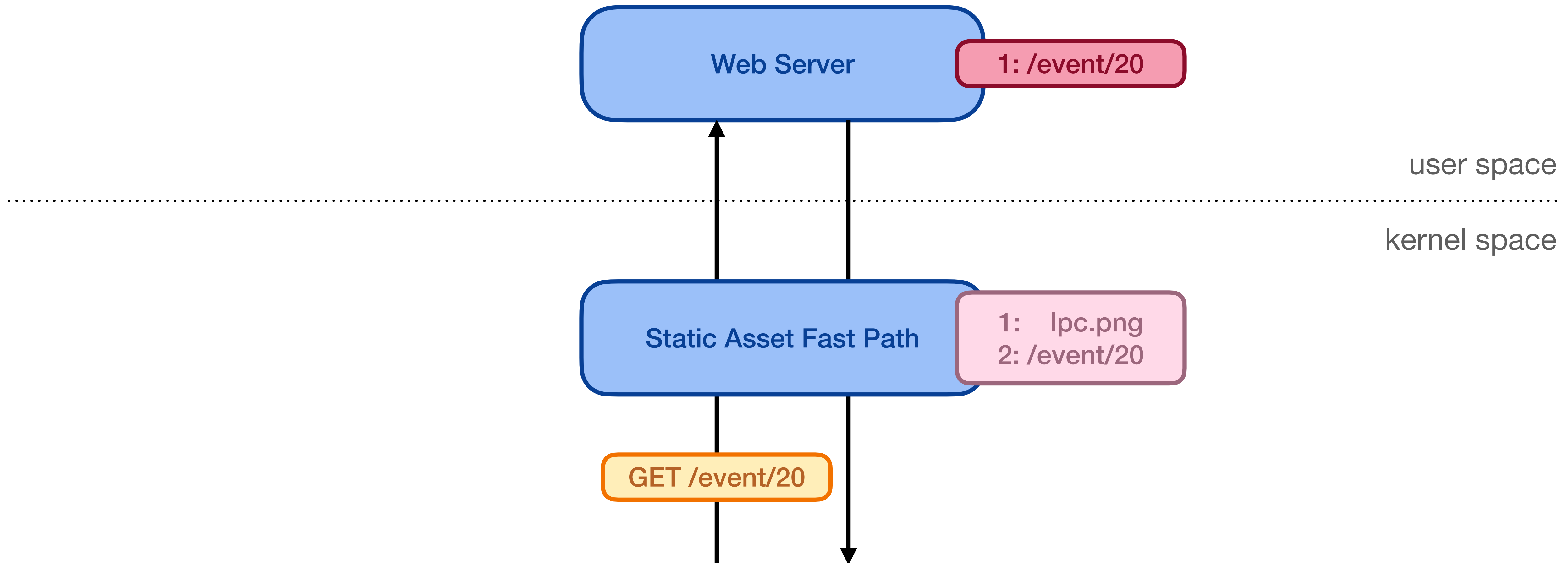
Challenge 1: HTTP/2 caches headers

The fast path falls back to user space if it cannot serve a request



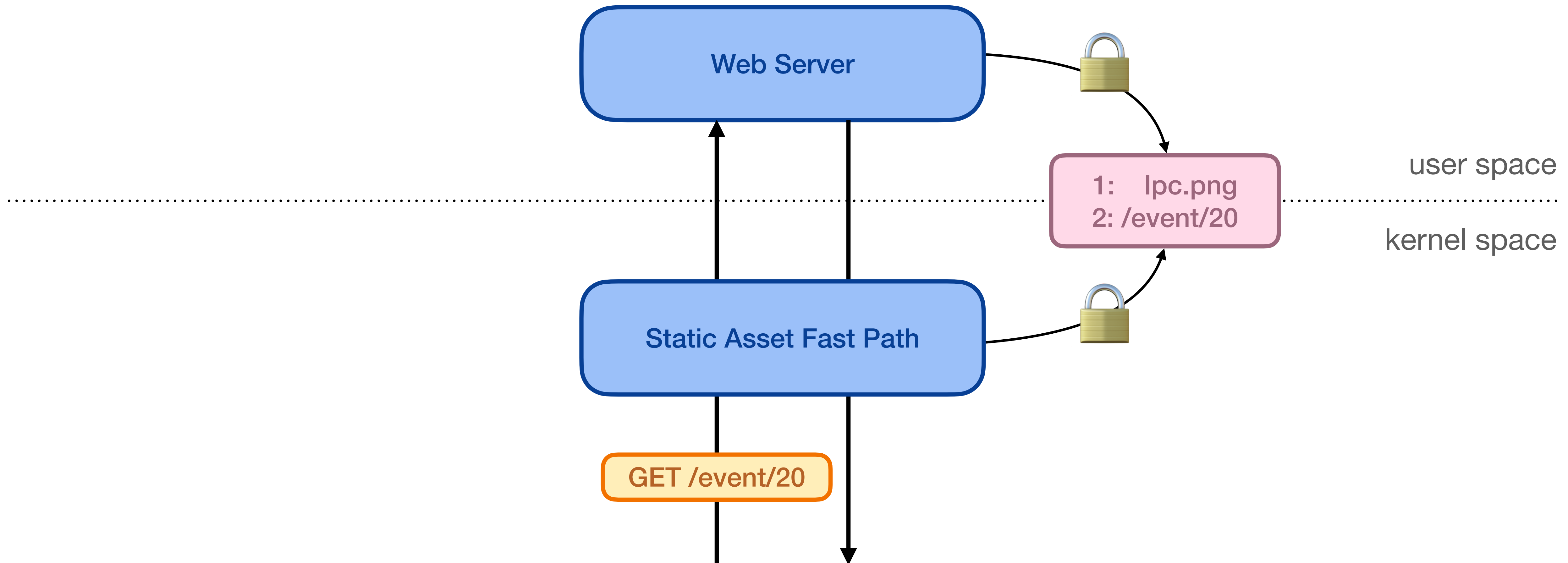
Challenge 1: HTTP/2 caches headers

But forwarding the request messes up the user space cache

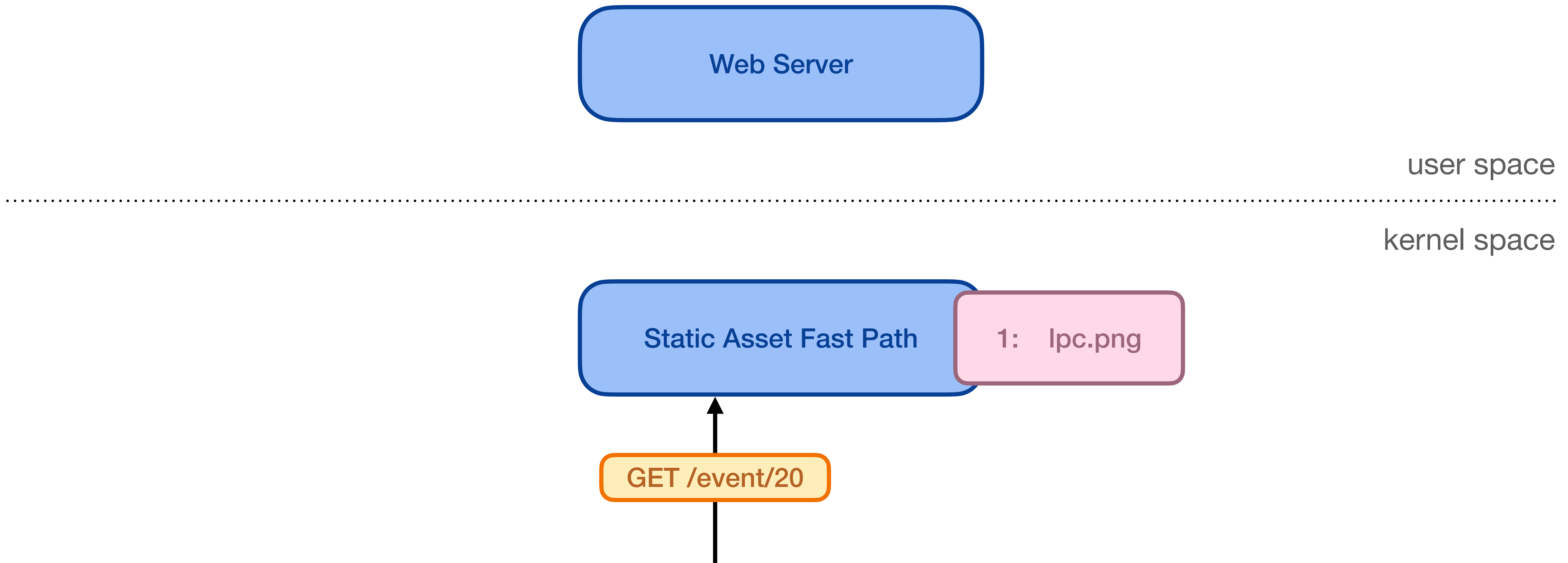


Challenge 1: HTTP/2 caches headers

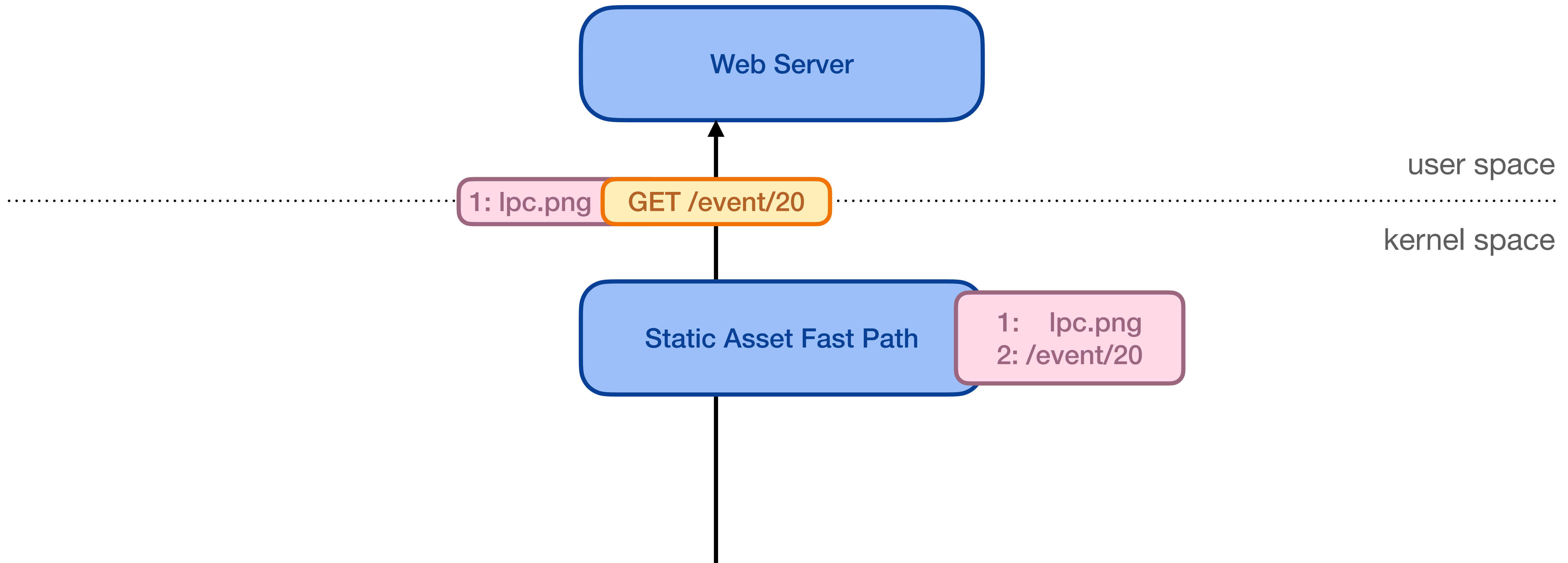
Using a shared map negates the benefits from the fast path



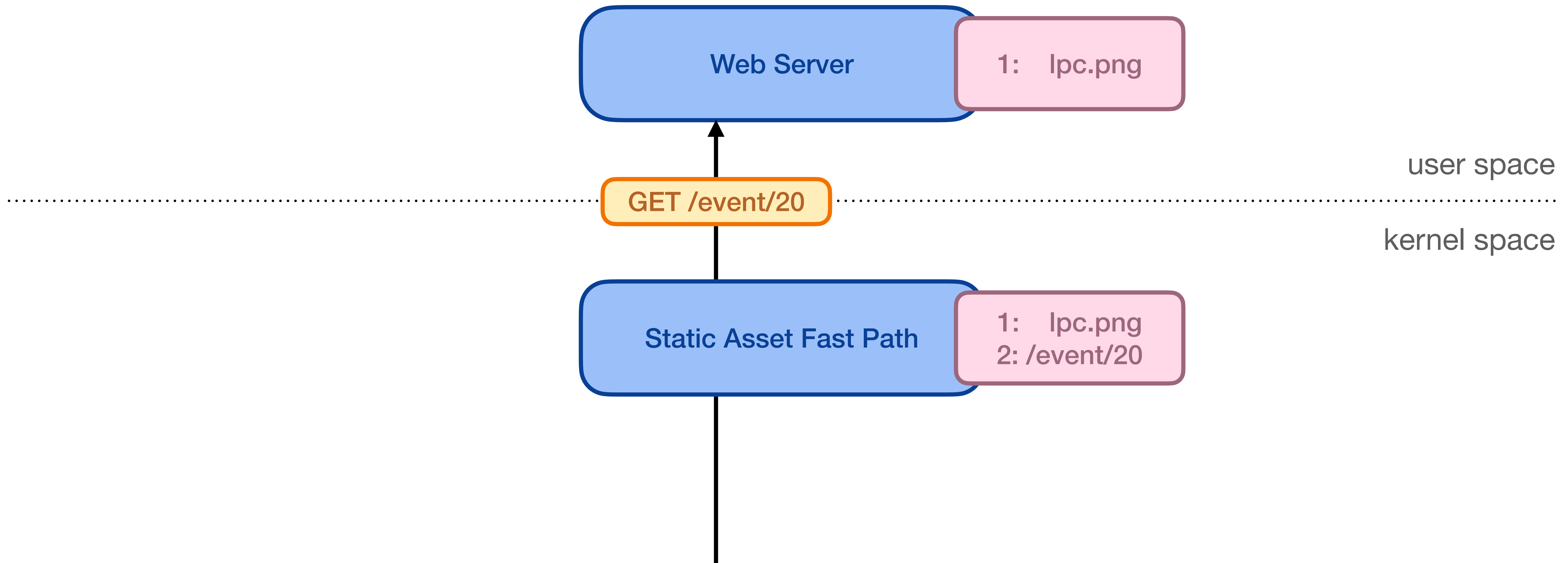
Solution: Prepending the cache changes to the request!



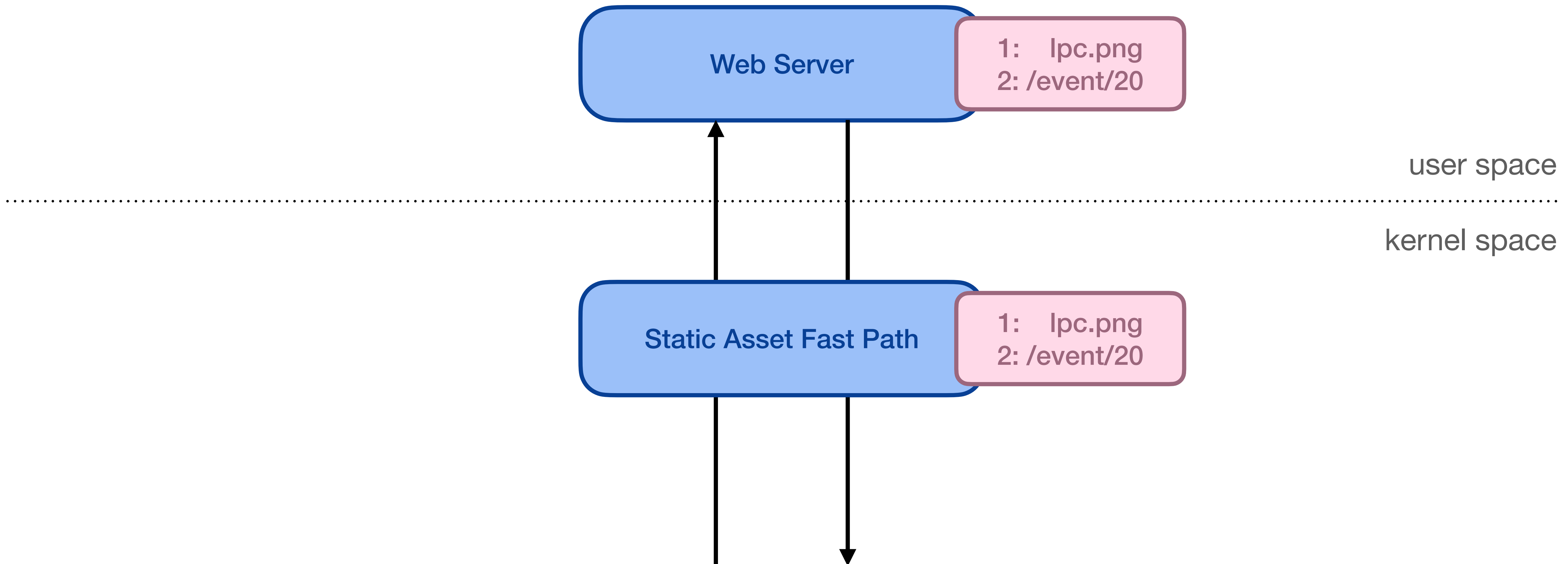
Solution: Prepending the cache changes to the request!



Solution: Prepending the cache changes to the request



Solution: Prepending the cache changes to the request



Challenge 2: SK_SKB cannot respond with assets larger than 16KB

bpf_skb_change_tail with a length of at most 4 pages

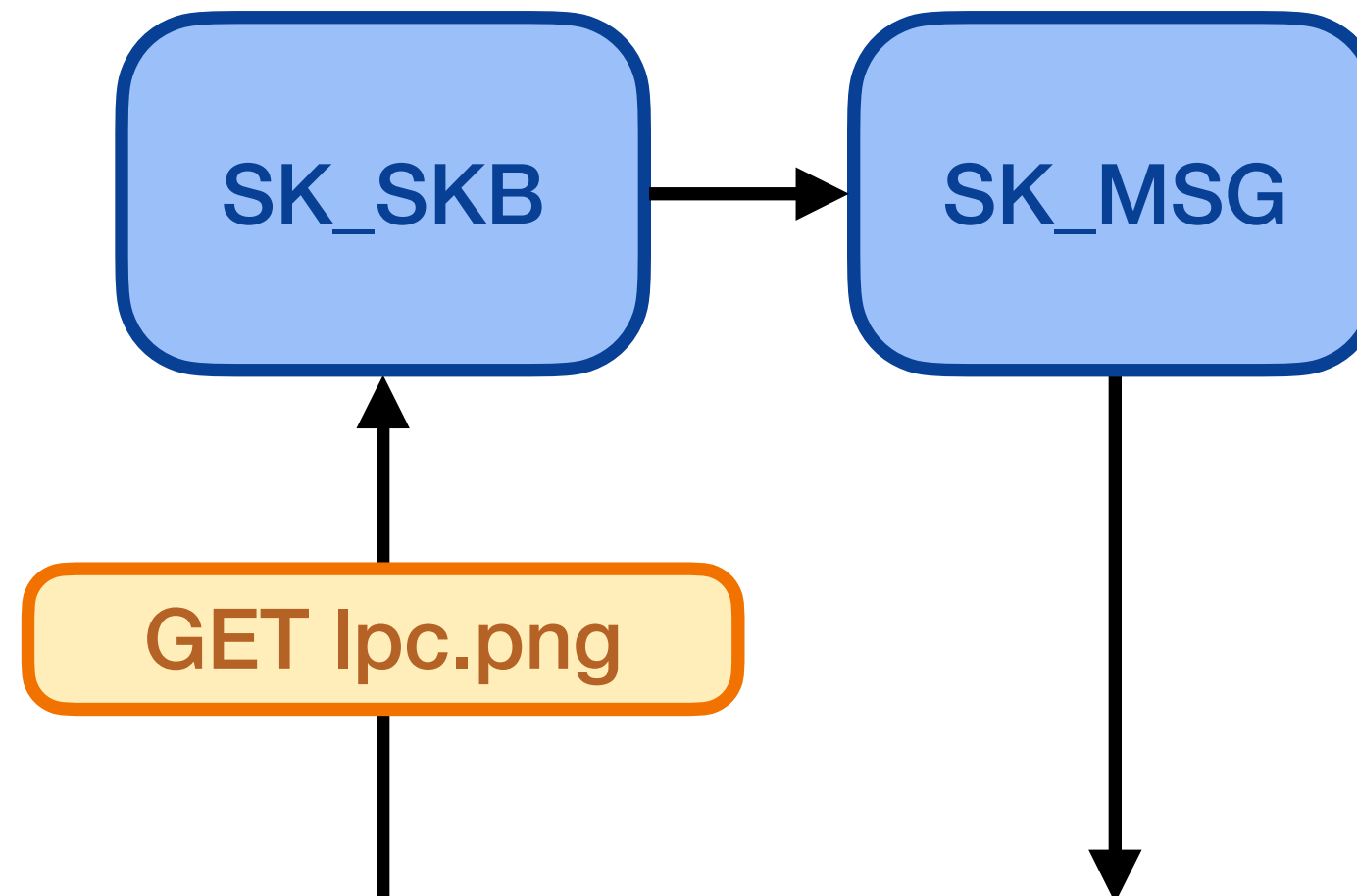
SK_MSG can respond with larger assets but is only called for local traffic

Solution: Forward ingress traffic from SK_SKB to SK_MSG

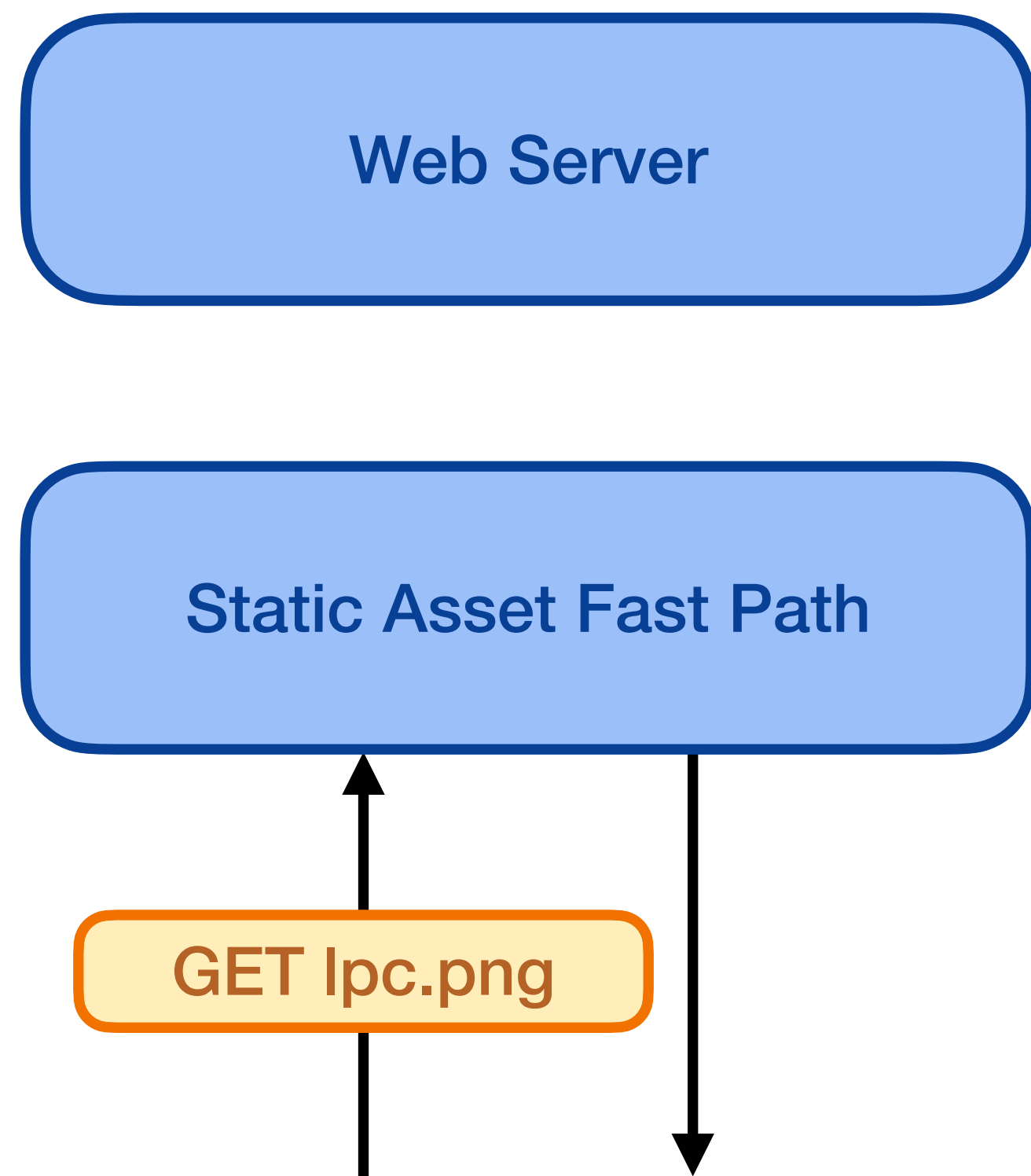


user space

kernel space



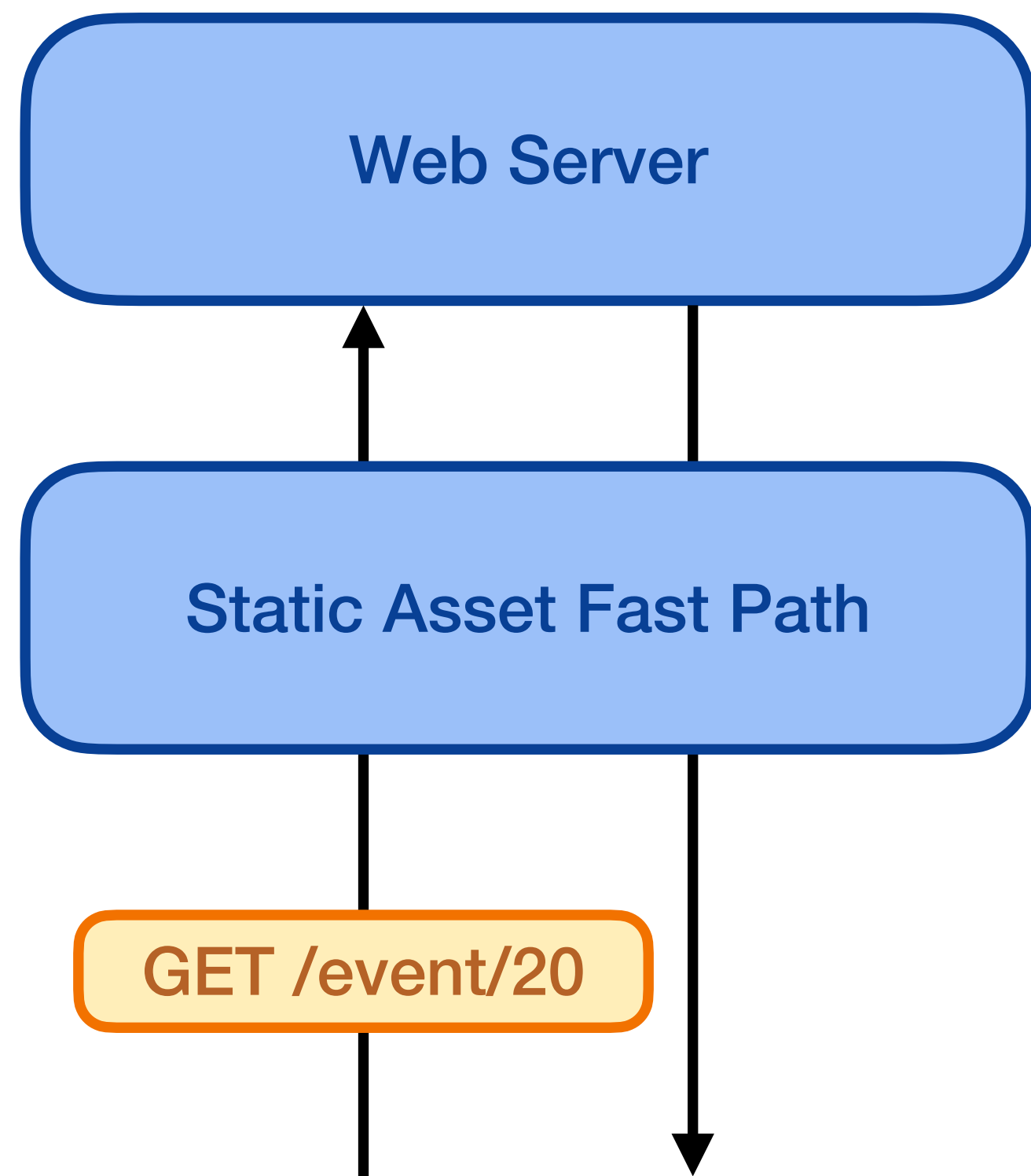
Benchmarks



1. How much faster is it to serve static files from eBPF?

2. What's the overhead when falling back to user space?

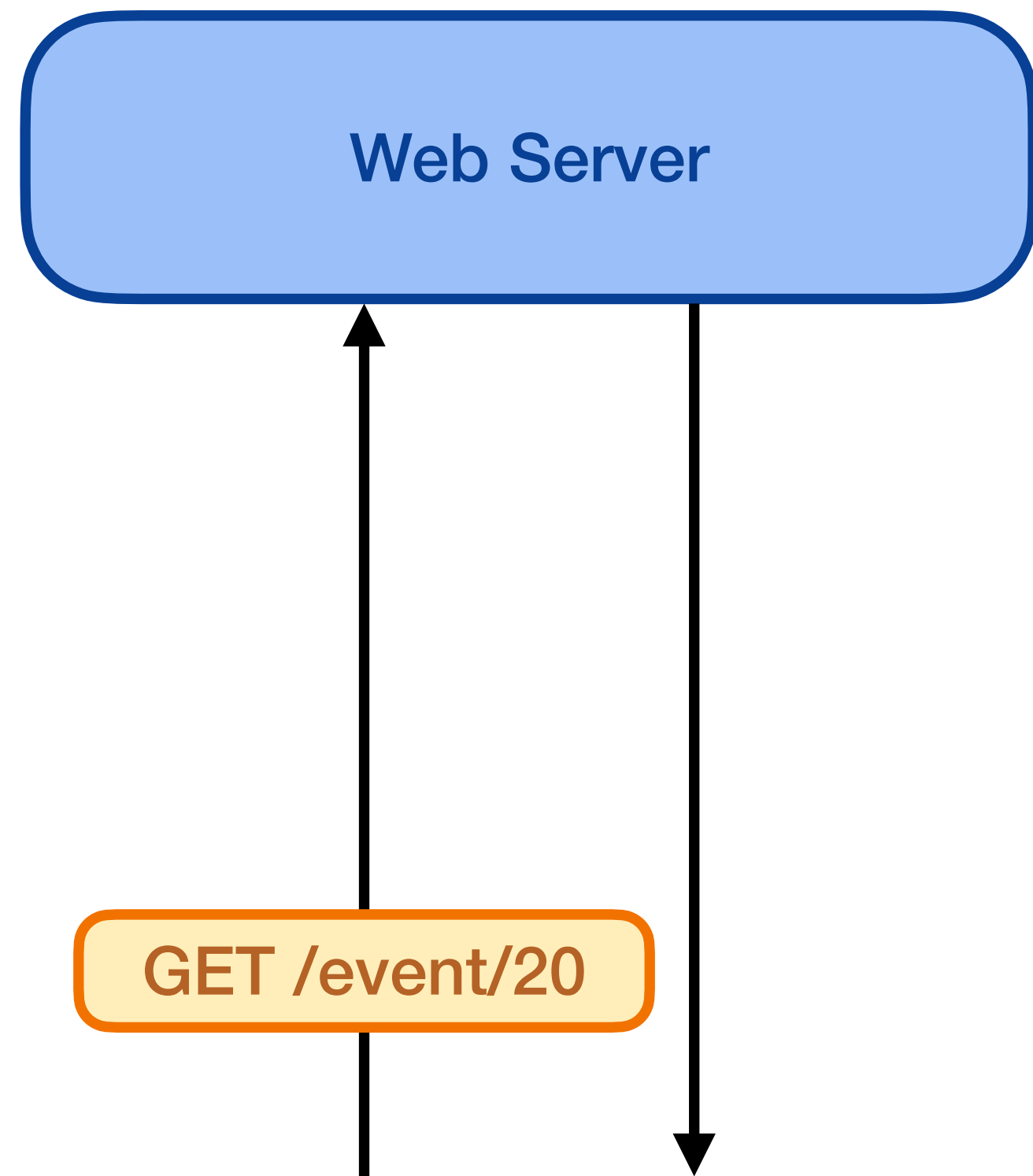
Benchmarks



1. How much faster is it to serve static files from eBPF?

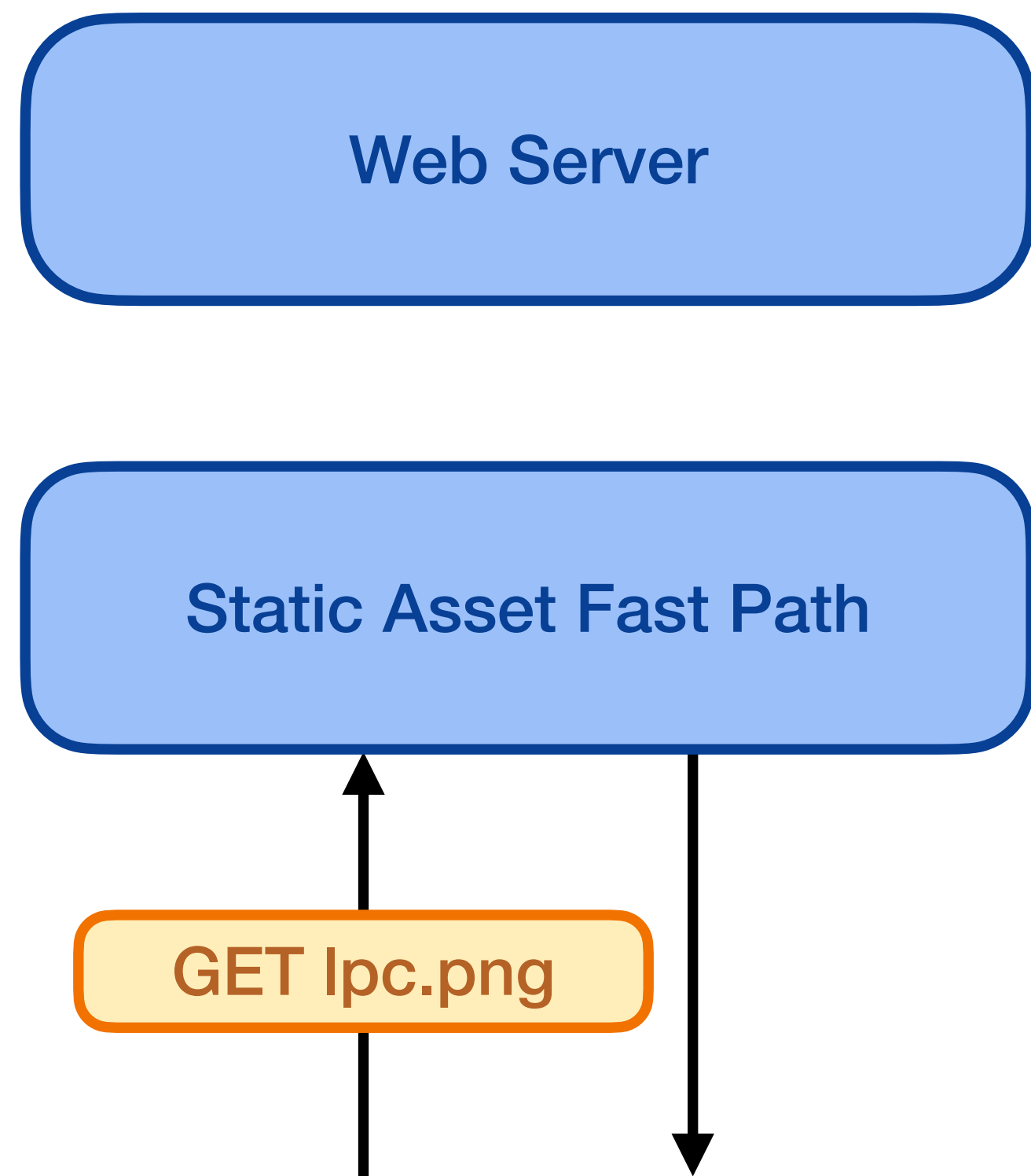
2. What's the overhead when falling back to user space?

Benchmarks



As baseline, we use the web server without the fast path

Benchmarks

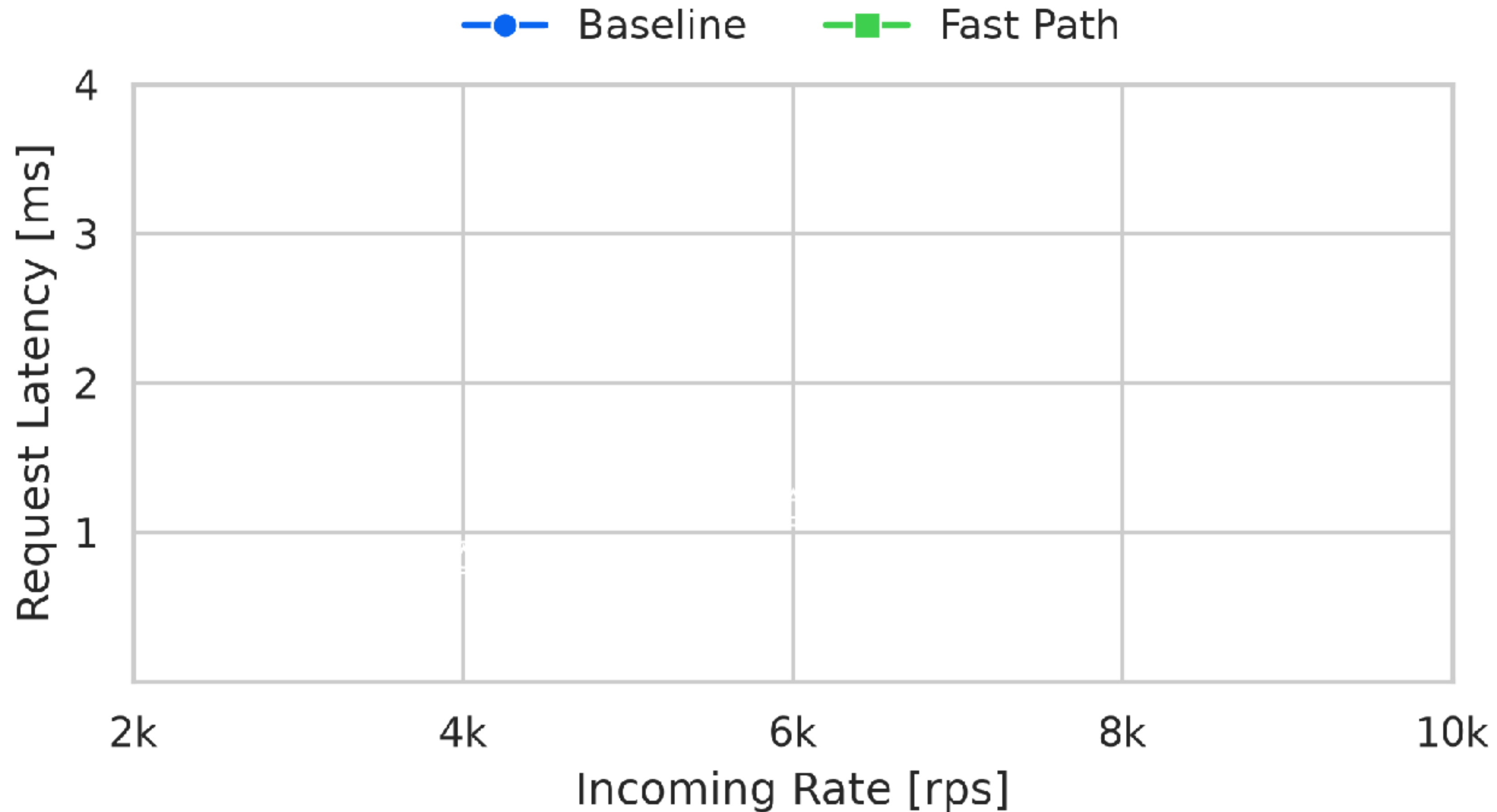


1. How much faster is it to serve static files from eBPF?

2. What's the overhead when falling back to user space?

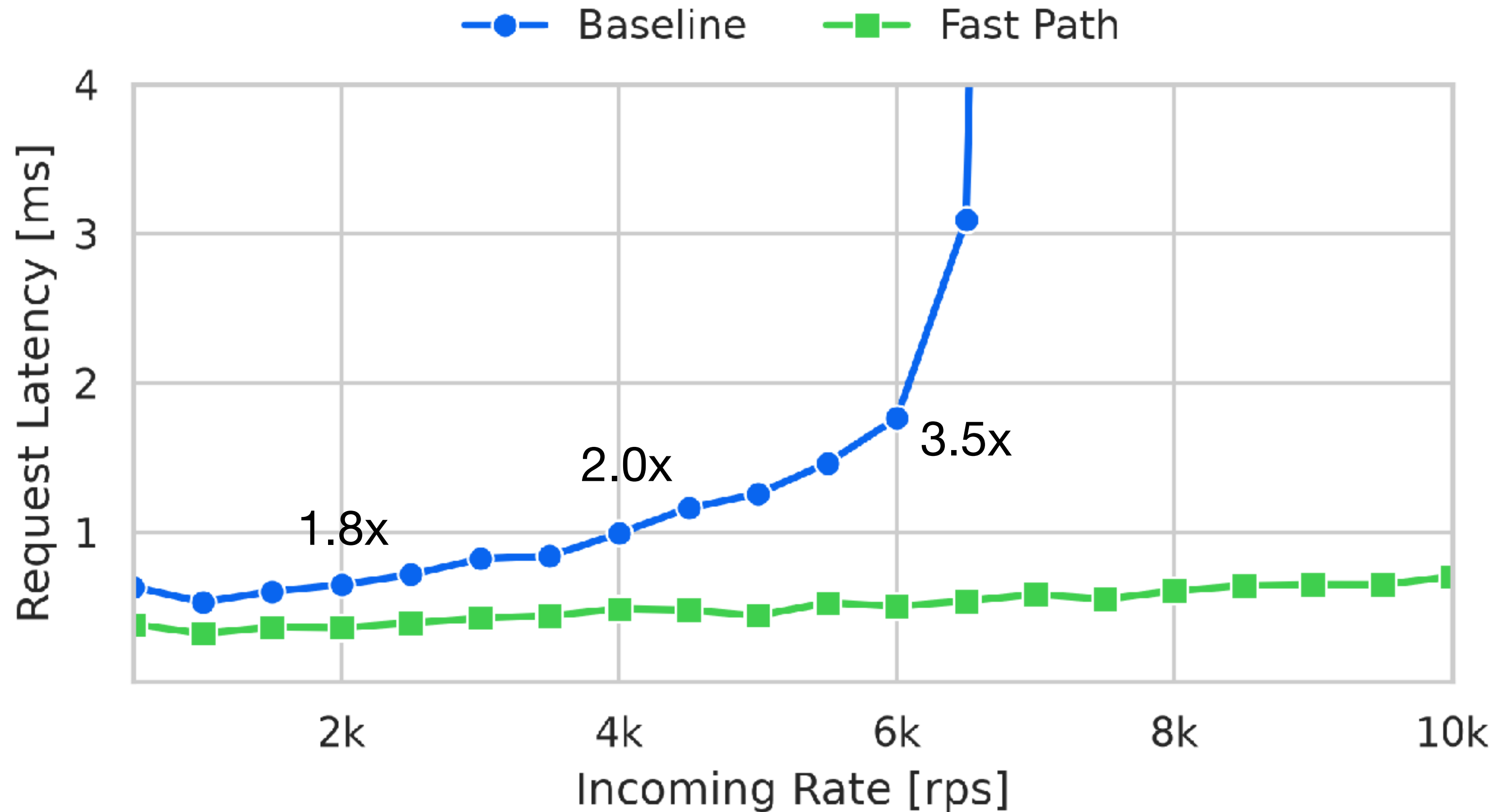
1. The Fast Path has lower request latency

when serving 32KB via HTTP/1.1



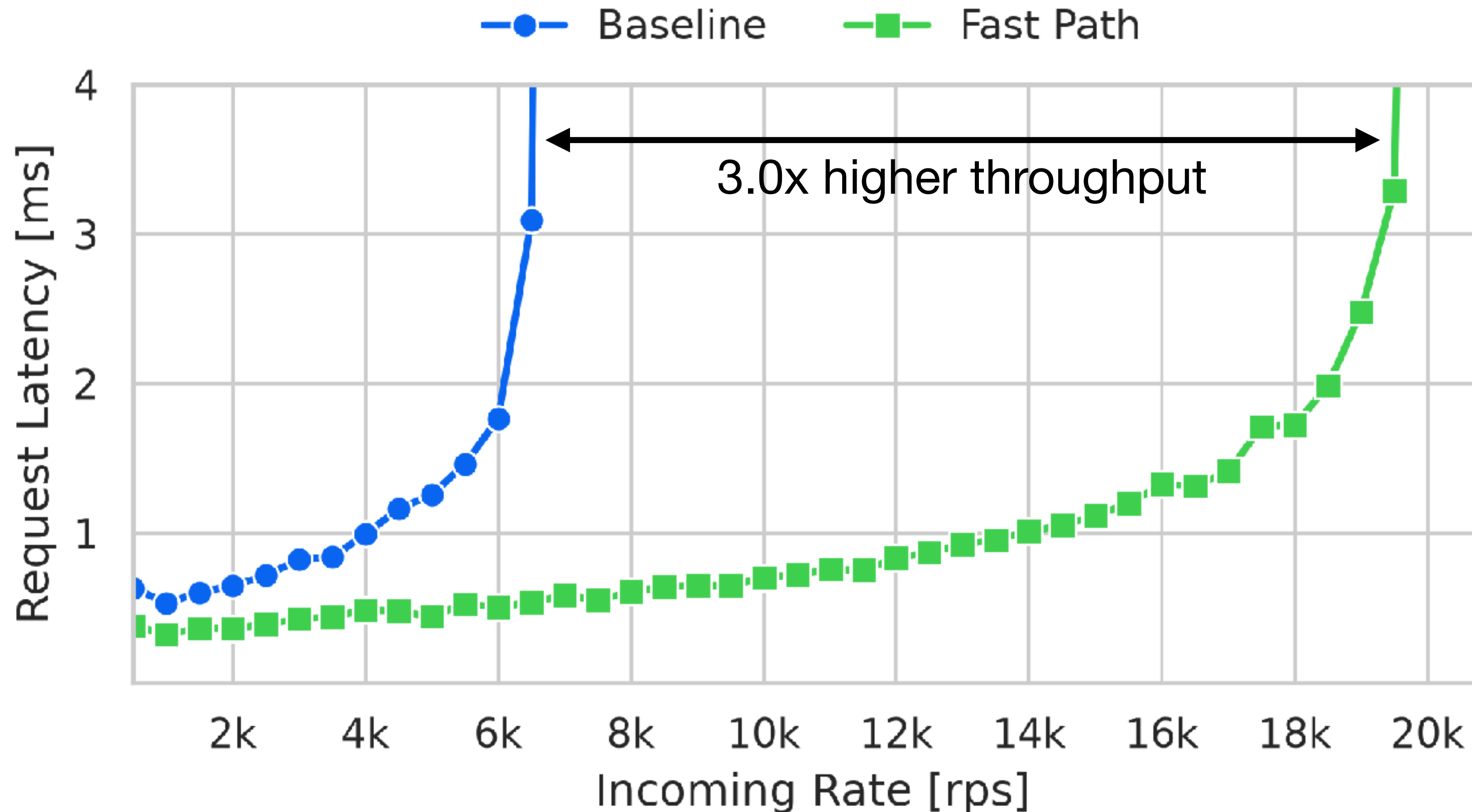
1. The Fast Path has lower request latency

when serving 32KB via HTTP/1.1



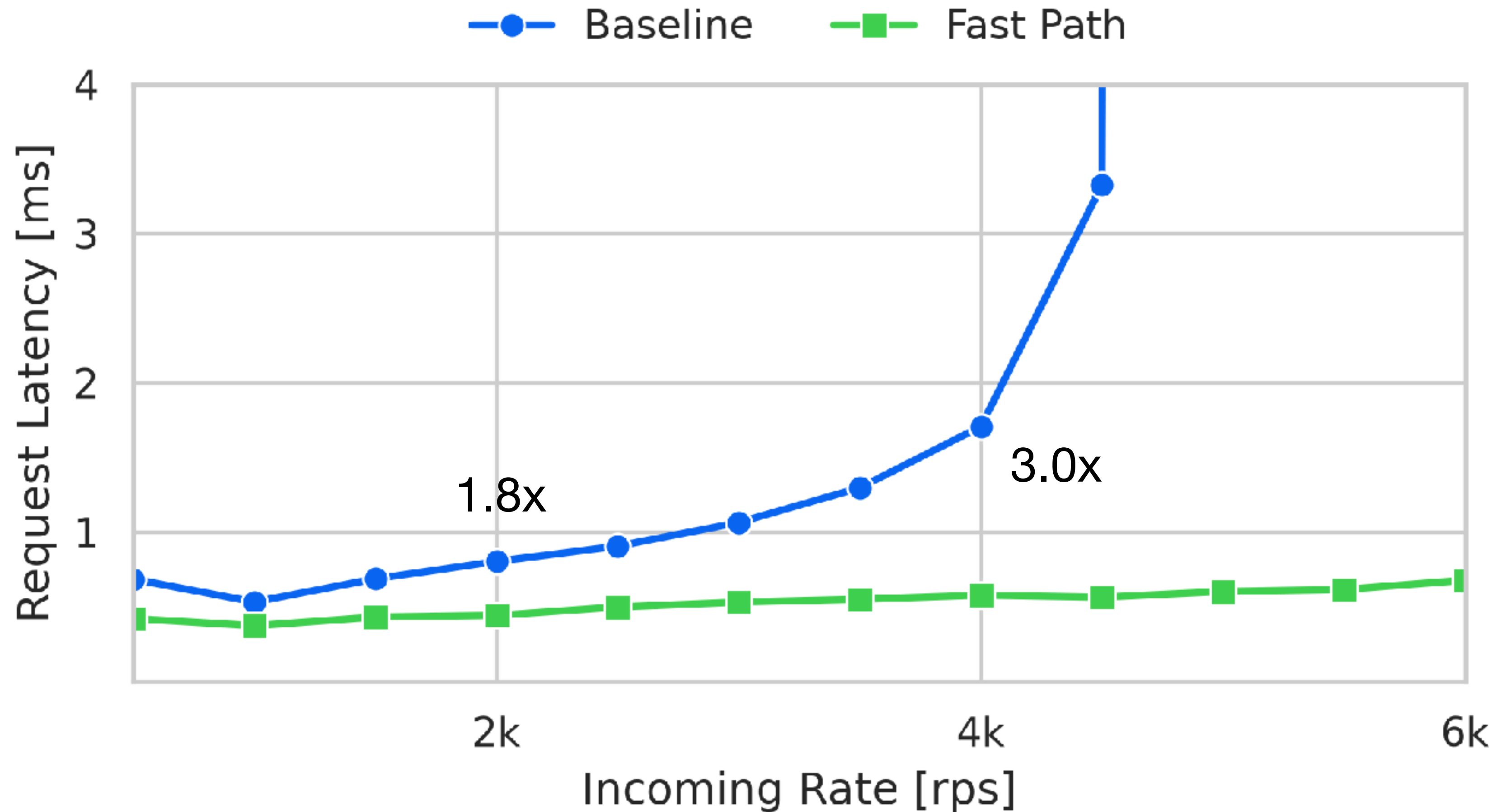
1. The Fast Path has higher throughput

when serving 32KB via HTTP/1.1



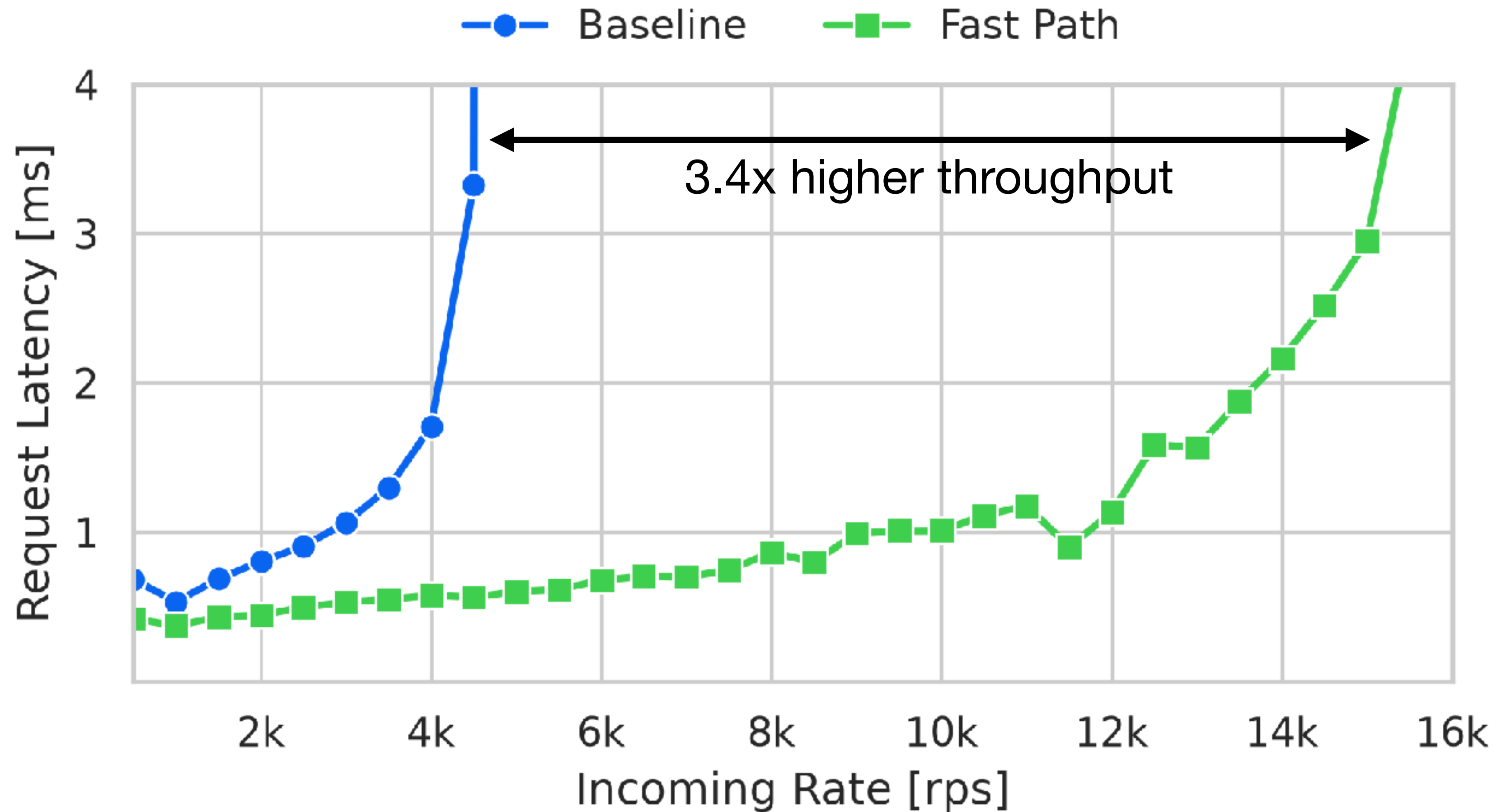
1. The Fast Path has lower request latency

when serving 32KB via HTTP/2

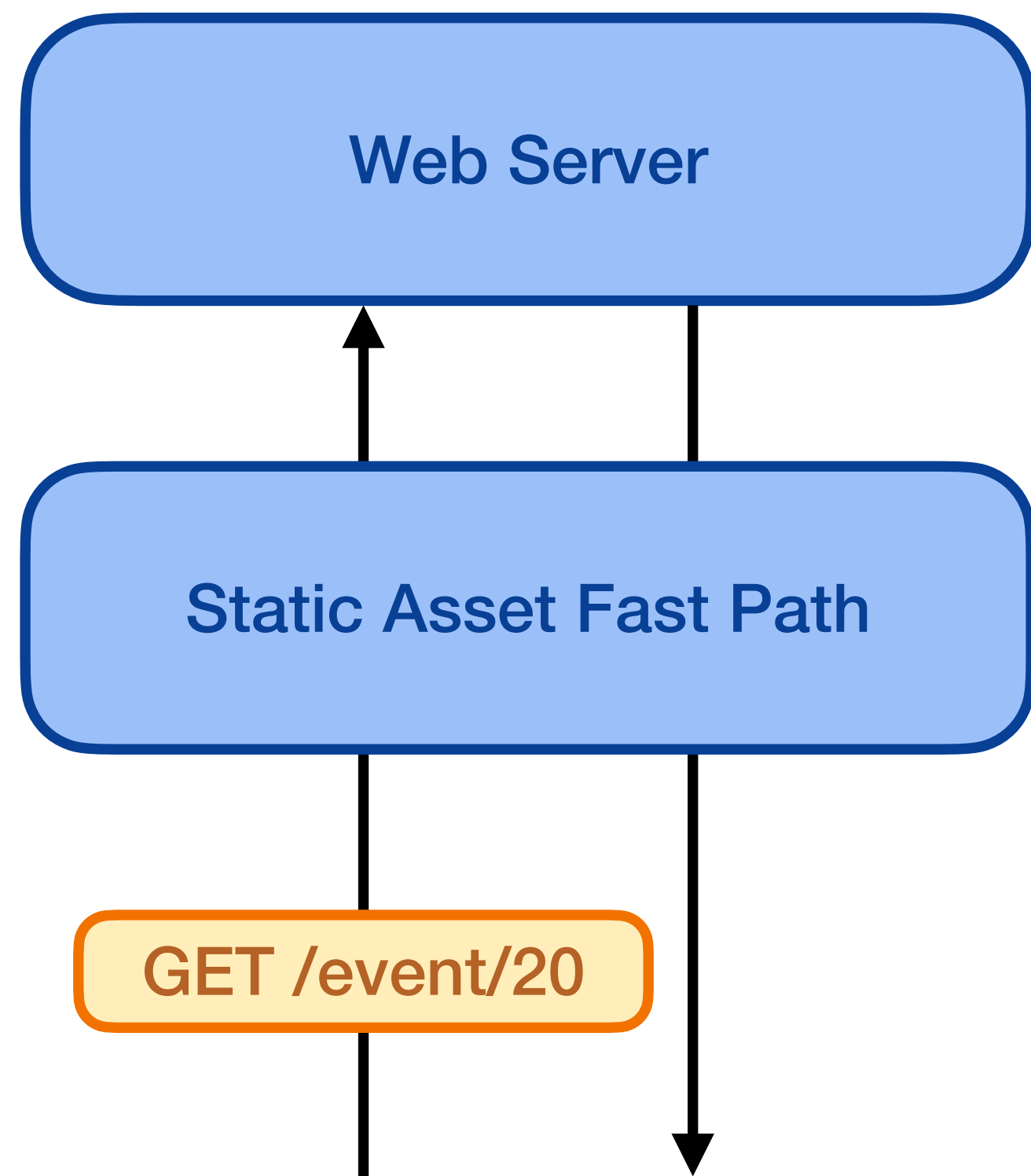


1. The Fast Path has higher throughput

when serving 32KB via HTTP/2



Benchmarks

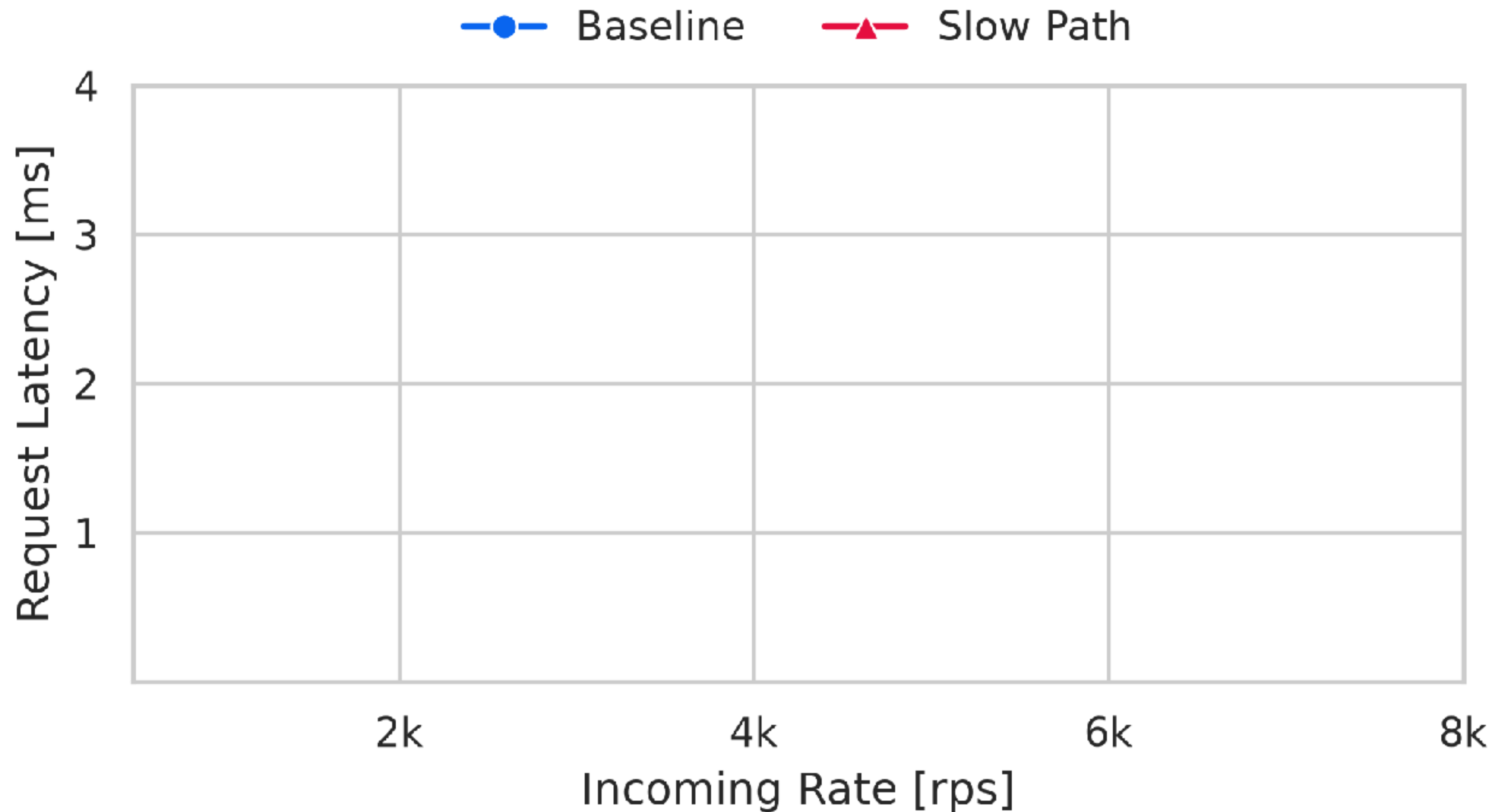


1. How much faster is it to serve static files from eBPF?

2. What's the overhead when falling back to user space?

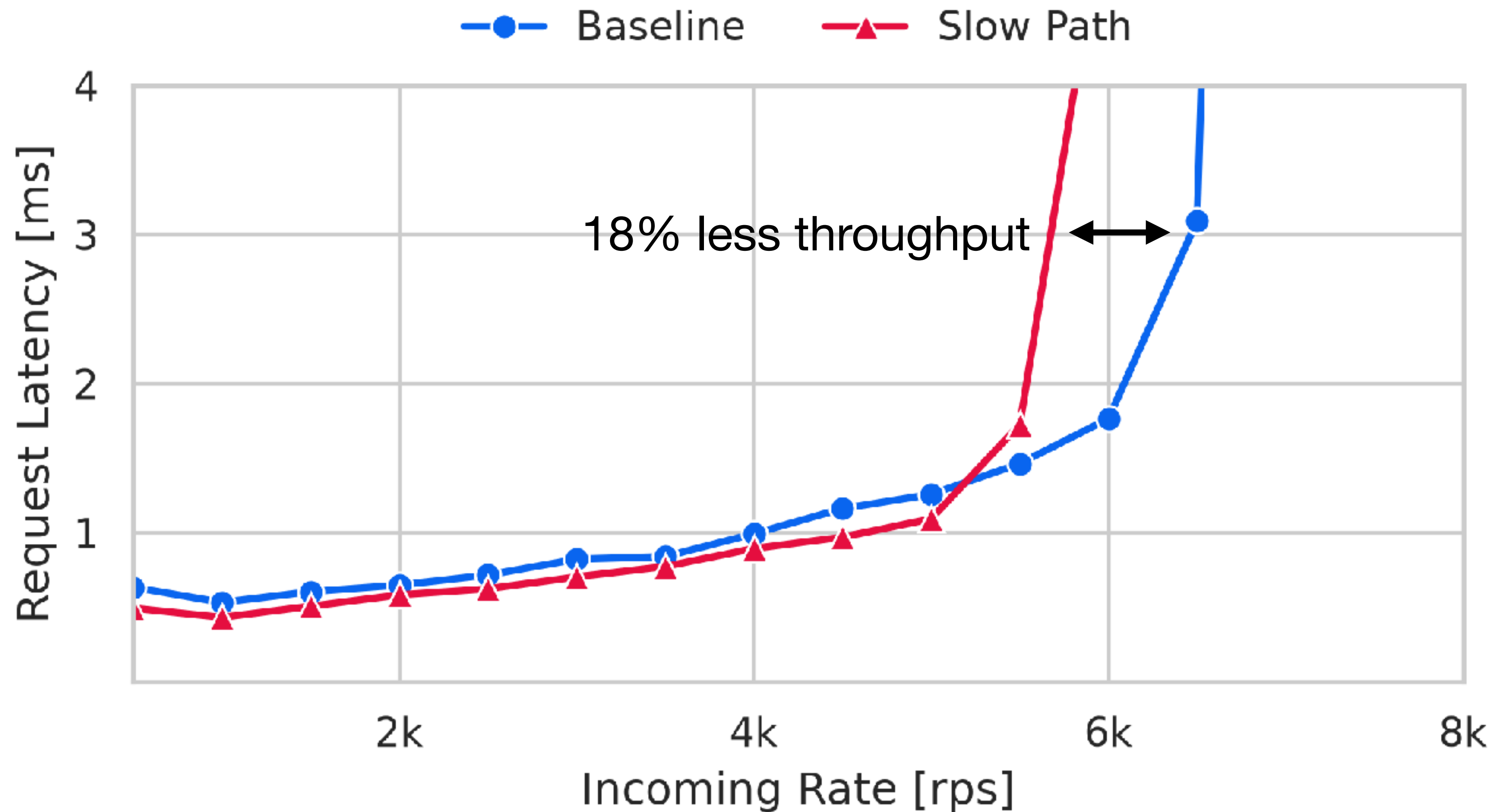
2. The Slow Path saturates earlier

when serving 32KB via HTTP/1.1



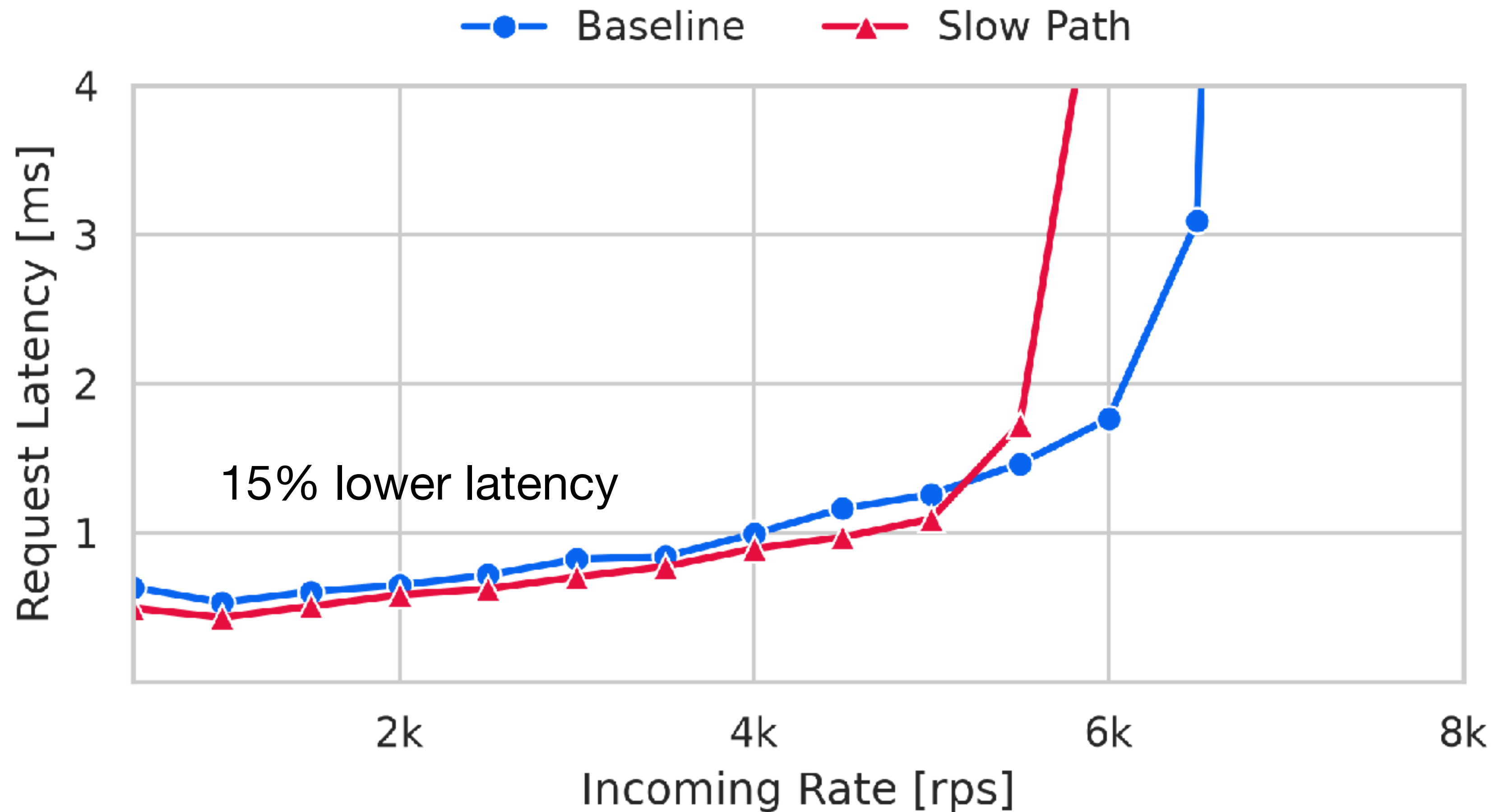
2. The Slow Path saturates earlier

when serving 32KB via HTTP/1.1



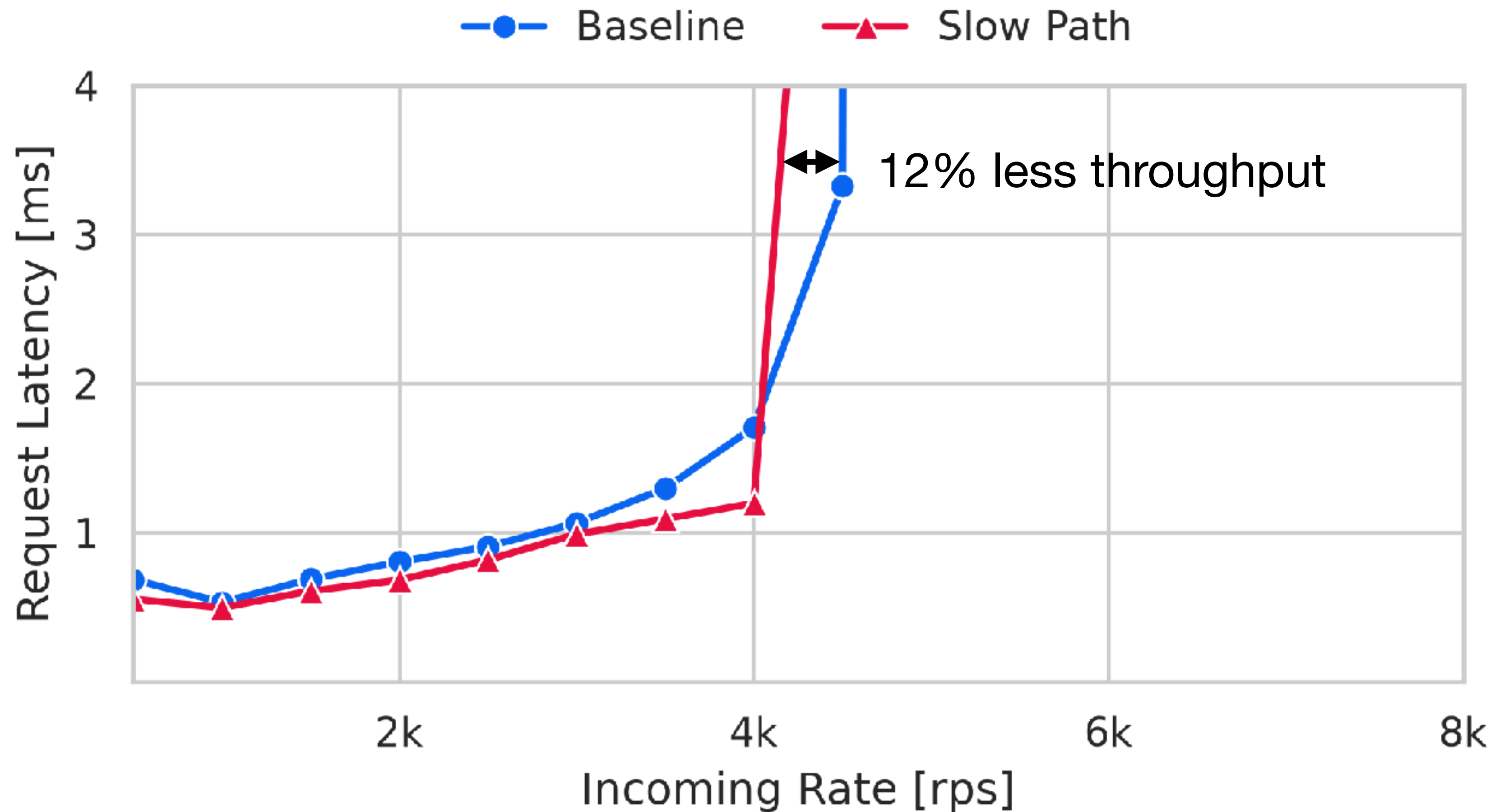
2. The Slow Path improves request latency

when serving 32KB via HTTP/1.1



2. The Slow Path saturates earlier

when serving 32KB via HTTP/2



Application-Layer Parsing in eBPF Is **Practical**

You can implement all of these use cases with Beeper!



Container networks with L7 policies

Application-layer Firewalls

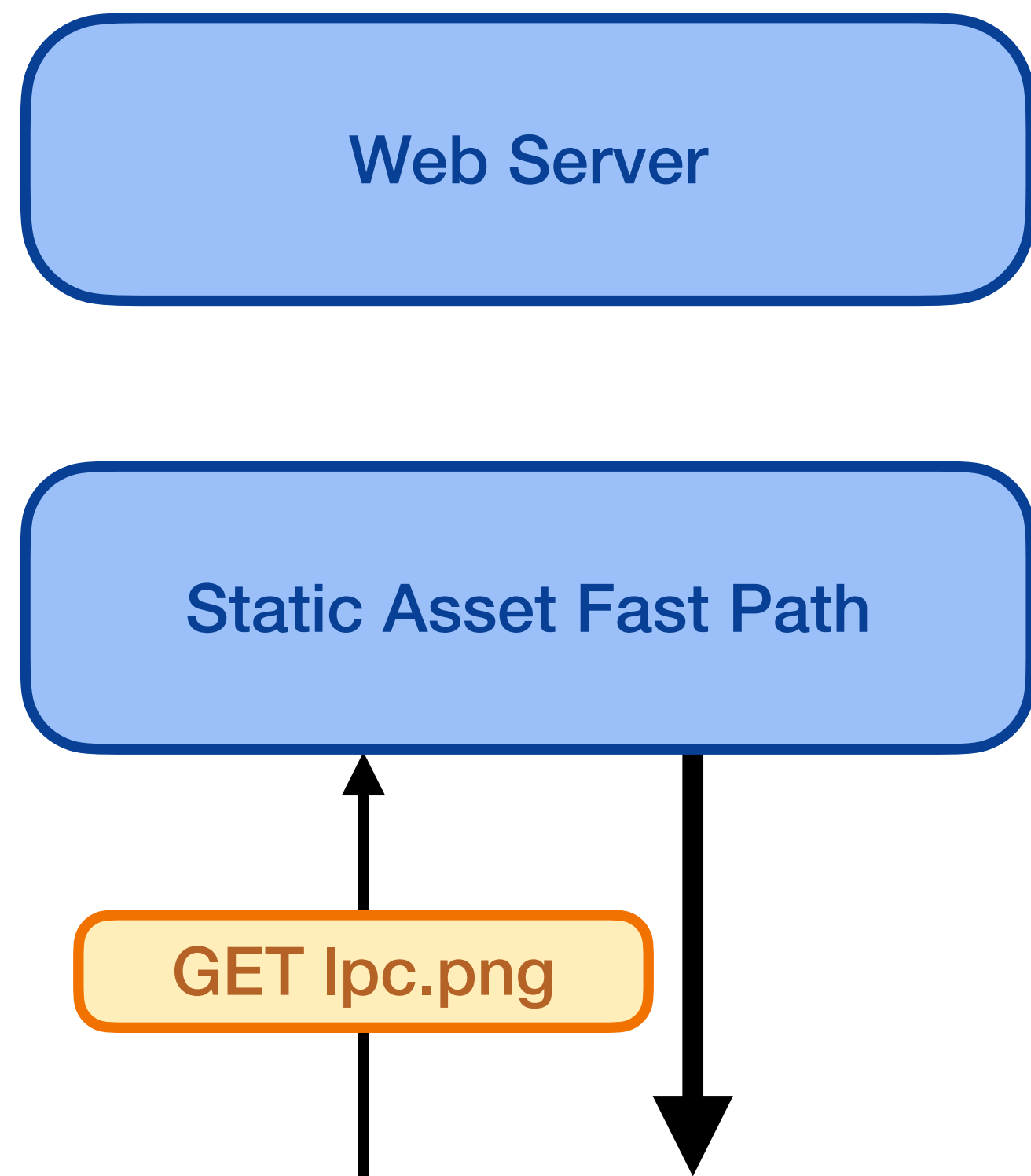
Application-layer metrics for observability tools

Thanks for listening!



github.com/lerboe/beeper

Benchmarks



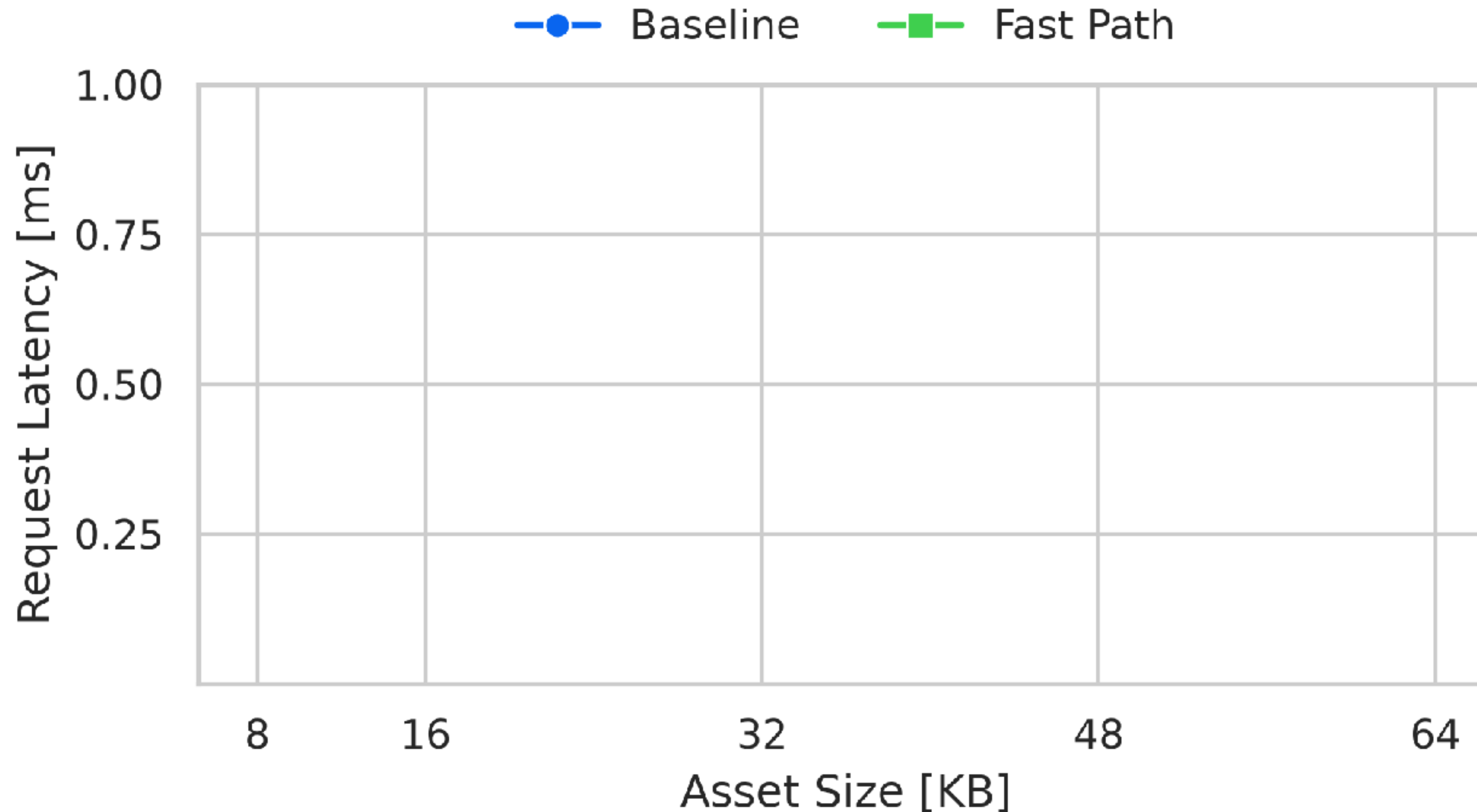
1. How much faster is it to serve static files from eBPF?

2. What's the overhead when falling back to user space?

3. How does this scale with the asset size?

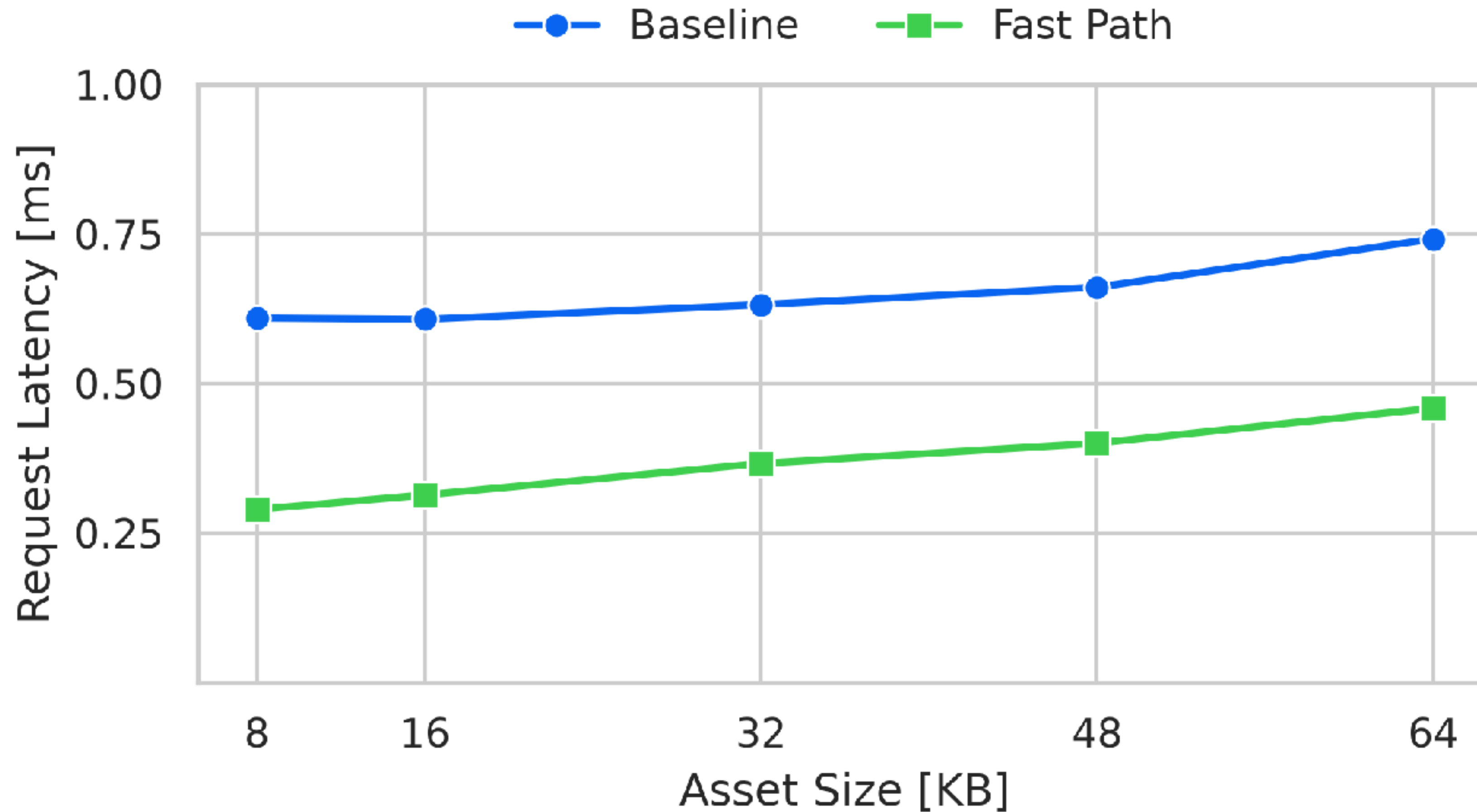
3. Both approaches scale similarly

when serving 2000 RPS via HTTP/1.1



3. Both approaches scale similarly

when serving 2000 RPS via HTTP/1.1



3. Both approaches scale similarly

when serving 2000 RPS via HTTP/2

