Imperial College London

MEng Individual Project

Accelerating data centre communication patterns using eBPF

Alex Constantin-Gómez
Under supervision of Marios Kogias

When the kernel becomes the communication bottleneck

- Large-scale distributed applications in data centres rely on 100s or 1000s machines connected over a network.

When the kernel becomes the communication bottleneck

- Large-scale distributed applications in data centres rely on 100s or 1000s machines connected over a network.
- This leads to a huge volume of network messages, incurring a significant amount of:
    - User-kernel crossings (via system calls)
    - Context switches
    - In-kernel network stack traversals

When the kernel becomes the communication bottleneck

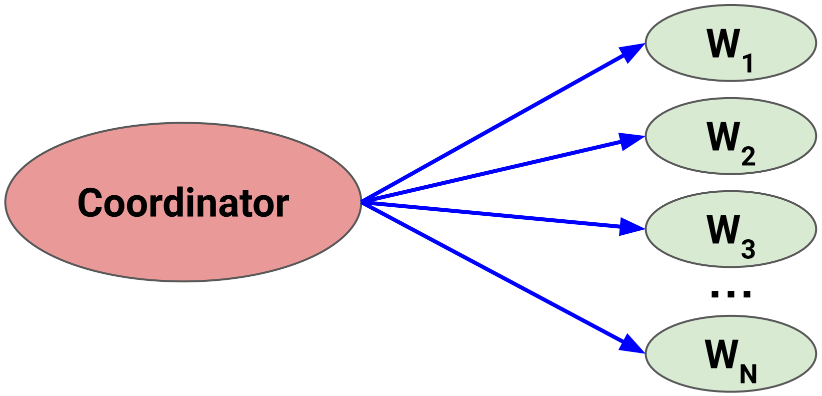
- Large-scale distributed applications in data centres rely on 100s or 1000s machines connected over a network.
- This leads to a huge volume of network messages, incurring a significant amount of:
  - User-kernel crossings (via system calls)
  - Context switches
  - In-kernel network stack traversals
- The default Linux kernel network stack is designed for a small number of long-lived connections, rather than a large number of small messages.
  - High per-packet overhead

When the kernel becomes the communication bottleneck

- This problem is further exaggerated when high-speed NICs are used
  - This moves the communication bottleneck to the kernel
- This is especially true for applications following *scatter-gather* workloads which generate a large number of messages.
- Scatter-gather communication is a very common pattern in data centre applications (also known as fan-out/fan-in pattern in “*The tail at scale”* article [1])

Scatter-gather communication

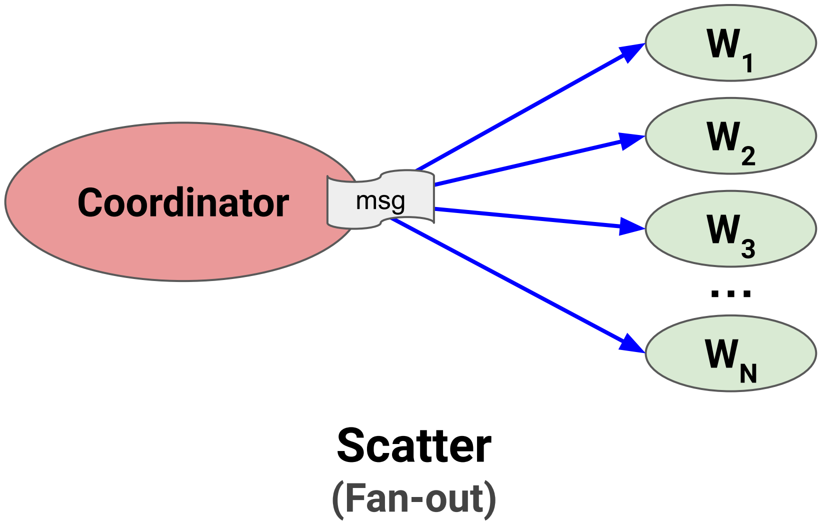
- The *scatter-gather* pattern refers to a single coordinator node scattering a message to multiple worker nodes and then gathering their responses into a single result.
- Typically, the coordinator performs some aggregation logic on the responses to produce a final reduced result (aka *scatter-reduce*).



Scatter
(Fan-out)

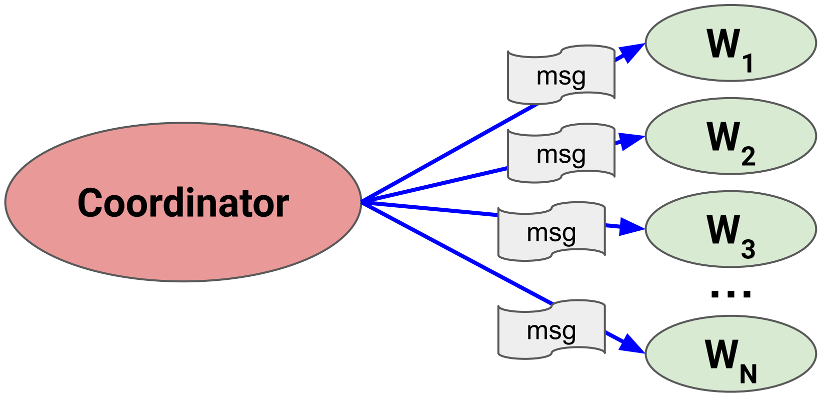
Scatter-gather communication

- The *scatter-gather* pattern refers to a single coordinator node scattering a message to multiple worker nodes and then gathering their responses into a single result.
- Typically, the coordinator performs some aggregation logic on the responses to produce a final reduced result (aka *scatter-reduce*).



Scatter-gather communication

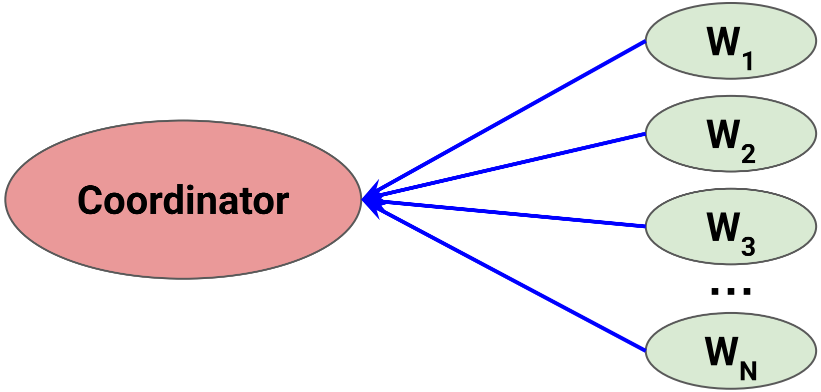
- The *scatter-gather* pattern refers to a single coordinator node scattering a message to multiple worker nodes and then gathering their responses into a single result.
- Typically, the coordinator performs some aggregation logic on the responses to produce a final reduced result (aka *scatter-reduce*).



Scatter
(Fan-out)

Scatter-gather communication

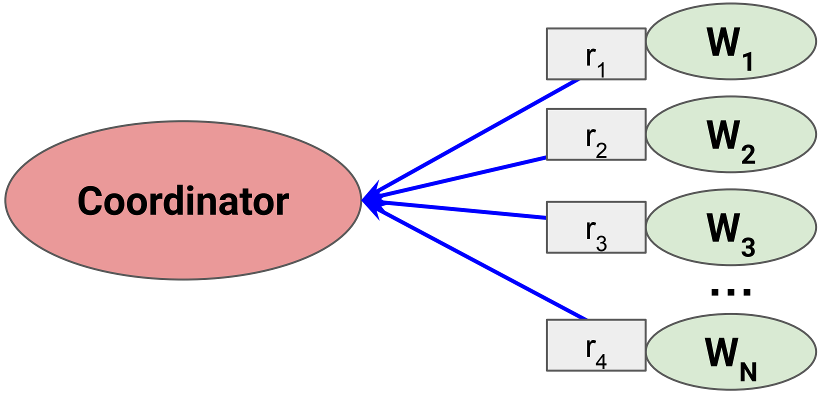
- The *scatter-gather* pattern refers to a single coordinator node scattering a message to multiple worker nodes and then gathering their responses into a single result.
- Typically, the coordinator performs some aggregation logic on the responses to produce a final reduced result (aka *scatter-reduce*).



Gather
(Fan-in)

Scatter-gather communication

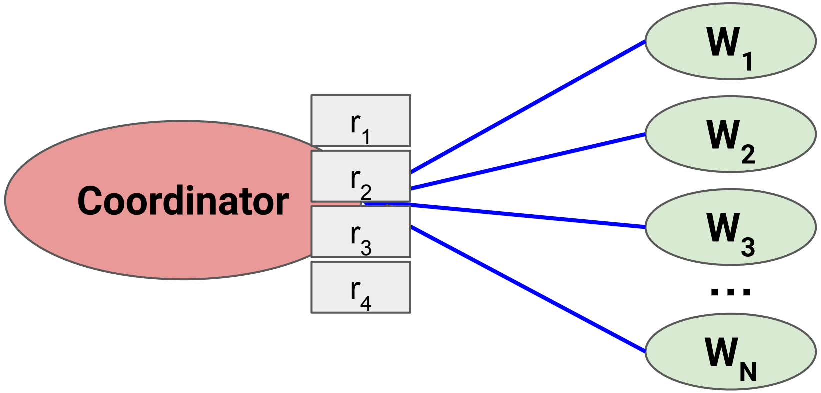
- The *scatter-gather* pattern refers to a single coordinator node scattering a message to multiple worker nodes and then gathering their responses into a single result.
- Typically, the coordinator performs some aggregation logic on the responses to produce a final reduced result (aka *scatter-reduce*).



Gather
(Fan-in)

Scatter-gather communication

- The *scatter-gather* pattern refers to a single coordinator node scattering a message to multiple worker nodes and then gathering their responses into a single result.
- Typically, the coordinator performs some aggregation logic on the responses to produce a final reduced result (aka *scatter-reduce*).



Gather
(Fan-in)

Scatter-gather communication

- Many applications follow scatter-gather communication:
  - Distributed machine learning training
  - Distributed graph processing
  - Distributed (network) file systems

Scatter-gather communication

- Many applications follow scatter-gather communication:
  - Distributed machine learning training
  - Distributed graph processing
  - Distributed (network) file systems

- **Problem:** scatter-gather communication incurs a large volume of messages.
  - ✗ Excessive number of system calls (and context switches)
  - ✗ Excessive number of kernel network stack traversals

Accelerating scatter-gather communication

- **Goal**: accelerate the performance of scatter-gather workloads using readily-available technologies on commodity Linux machines with minimal deployment overhead.
    - ✗ User-space networking and hardware-offloaded solutions not suitable.

Accelerating scatter-gather communication

- **Goal**: accelerate the performance of scatter-gather workloads using readily-available technologies on commodity Linux machines with minimal deployment overhead.
  - ✗ User-space networking and hardware-offloaded solutions not suitable.
- Our approach uses **eBPF** as a performance accelerator.

Accelerating scatter-gather communication

- **Goal**: accelerate the performance of scatter-gather workloads using readily-available technologies on commodity Linux machines with minimal deployment overhead.
  - ✗ User-space networking and hardware-offloaded solutions not suitable.

- Our approach uses **eBPF** as a performance accelerator.

- Presentation outline:
  a. Introduction to eBPF
  b. *sgbpf*, an eBPF-accelerated scatter-gather primitive
  c. Demonstration of the *sgbpf* API
  d. Performance evaluation

Introduction to eBPF

- eBPF is a technology that allows users to write code that runs inside the Linux kernel.

Introduction to eBPF

- eBPF is a technology that allows users to write code that runs inside the Linux kernel.
- Guarantees both:
  - **Safety**, via the eBPF static verifier
  - **Performance**, via JIT compilation

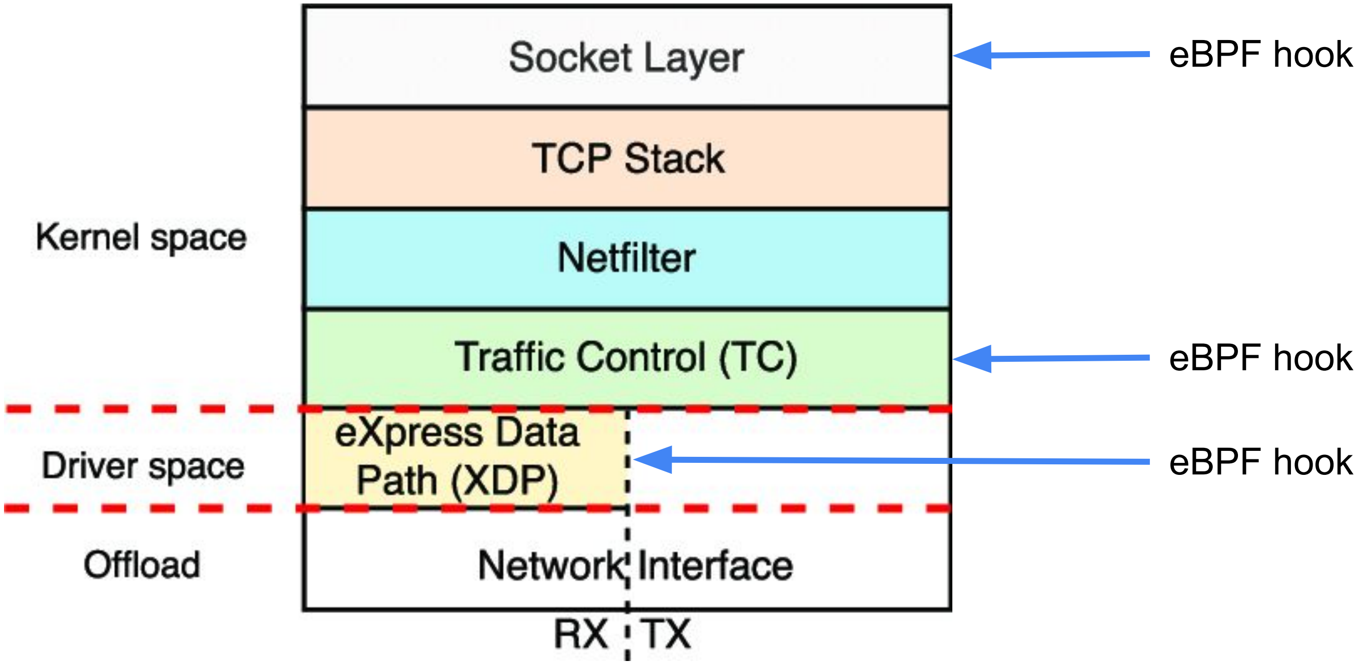
Introduction to eBPF

- eBPF is a technology that allows users to write code that runs inside the Linux kernel.
- Guarantees both:
    - **Safety**, via the eBPF static verifier
    - **Performance**, via JIT compilation
- Users write their code as eBPF programs, which are event-driven functions that are triggered on the arrival of a kernel event.
    - Programs are attached to a specific hook point inside the kernel
    - Written in a high level language like C and compiled to eBPF bytecode

Introduction to eBPF

- eBPF is a technology that allows users to write code that runs inside the Linux kernel.
- Guarantees both:
  - **Safety**, via the eBPF static verifier
  - **Performance**, via JIT compilation
- Users write their code as eBPF programs, which are event-driven functions that are triggered on the arrival of a kernel event.
  - Programs are attached to a specific hook point inside the kernel
  - Written in a high level language like C and compiled to eBPF bytecode
- State can be managed using eBPF maps
  - Program ⇆ program communication
  - User-space ⇆ program communication

Network hooks in eBPF



Using eBPF to accelerate network applications

- Applications such as Memcached [2] and Multi-Paxos consensus [3] have seen successful performance boosts with eBPF.

Using eBPF to accelerate network applications

- Applications such as Memcached [2] and Multi-Paxos consensus [3] have seen successful performance boosts with eBPF.
- By performing application logic before the kernel networking stack, high packet processing rates can be achieved by minimising user-kernel crossings.

Using eBPF to accelerate network applications

- Applications such as Memcached [2] and Multi-Paxos consensus [3] have seen successful performance boosts with eBPF.
- By performing application logic before the kernel networking stack, high packet processing rates can be achieved by minimising user-kernel crossings.
- Memcached example:
    - Fast replying: prepare a response packet in XDP to serve requests quickly by reading from a working copy of a cache in the kernel using eBPF maps.
- Multi-Paxos example:
    - Wait-on-quorums: count ACK messages from followers in the kernel and only wake up the leader application when enough ACKs have been received.

sgbpf: an eBPF-accelerated scatter-gather primitive

- *sgbpf* is a library that exposes a scatter-gather network primitive for UDP workloads.

sgbpf: an eBPF-accelerated scatter-gather primitive

- *sgbpf* is a library that exposes a scatter-gather network primitive for UDP workloads.
- Available as a C++ API which can be integrated into network applications.

sgbpf: an eBPF-accelerated scatter-gather primitive

- *sgbpf* is a library that exposes a scatter-gather network primitive for UDP workloads.
- Available as a C++ API which can be integrated into network applications.
- It uses eBPF under-the-hood to perform custom packet processing inside the kernel
  - Minimises system calls
  - Minimises network stack traversals

sgbpf: an eBPF-accelerated scatter-gather primitive

- *sgbpf* is a library that exposes a scatter-gather network primitive for UDP workloads.
- Available as a C++ API which can be integrated into network applications.
- It uses eBPF under-the-hood to perform custom packet processing inside the kernel
  - Minimises system calls
  - Minimises network stack traversals
- Supports several other features:
  - “All-gather” mode: broadcast final aggregated result to all workers
  - Configurable timeouts and completions policies
  - Multiple data delivery modes available to access aggregated data

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a `sendto()` syscall per worker.

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a `sendto()` syscall per worker.
- Inefficiencies:

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a `sendto()` syscall per worker.
- Inefficiencies:
    - ✗ Performs a syscall per worker (obviously)

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a **`sendto()`** syscall per worker.
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously)
  - ✗ Performs a user-to-kernel copy of the message per worker

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a **`sendto()`** syscall per worker.
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously)
  - ✗ Performs a user-to-kernel copy of the message per worker
  - ✗ Performs a network stack traversal per worker

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a `sendto()` syscall per worker.
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously)
  - ✗ Performs a user-to-kernel copy of the message per worker
  - ✗ Performs a network stack traversal per worker
- How can we do better?

Accelerating the scatter phase

- This is essentially a broadcast operation.
- Naive implementation incurs a `sendto()` syscall per worker.
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously)
  - ✗ Performs a user-to-kernel copy of the message per worker
  - ✗ Performs a network stack traversal per worker

- How can we do better?

eBPF

Accelerating the scatter phase

- With APIs such as `io_uring`, we can batch multiple I/O operations into a single syscall

Accelerating the scatter phase

- With APIs such as `io_uring`, we can batch multiple I/O operations into a single syscall
- Our approach with eBPF **tackles all three problems** in one go:

Accelerating the scatter phase

- With APIs such as `io_uring`, we can batch multiple I/O operations into a single syscall
- Our approach with eBPF **tackles all three problems** in one go:
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously) ← `io_uring` solves this
  - ✗ Performs a user-to-kernel copy of the message per worker
  - ✗ Performs a network stack traversal per worker

Accelerating the scatter phase

- With APIs such as `io_uring`, we can batch multiple I/O operations into a single syscall
- Our approach with eBPF **tackles all three problems** in one go:
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously) ← `io_uring` solves this
  - ✗ Performs a user-to-kernel copy of the message per worker
  - ✗ Performs a network stack traversal per worker

eBPF solves all of the above

Accelerating the scatter phase

- With APIs such as `io_uring`, we can batch multiple I/O operations into a single syscall
- Our approach with eBPF **tackles all three problems** in one go:
- Inefficiencies:
  - ✗ Performs a syscall per worker (obviously) ← `io_uring` solves this
  - ✗ Performs a user-to-kernel copy of the message per worker
  - ✗ Performs a network stack traversal per worker

eBPF solves all of the above

Accelerating the scatter phase

- With eBPF, we can perform the broadcast inside the kernel

Accelerating the scatter phase

- With eBPF, we can perform the broadcast inside the kernel

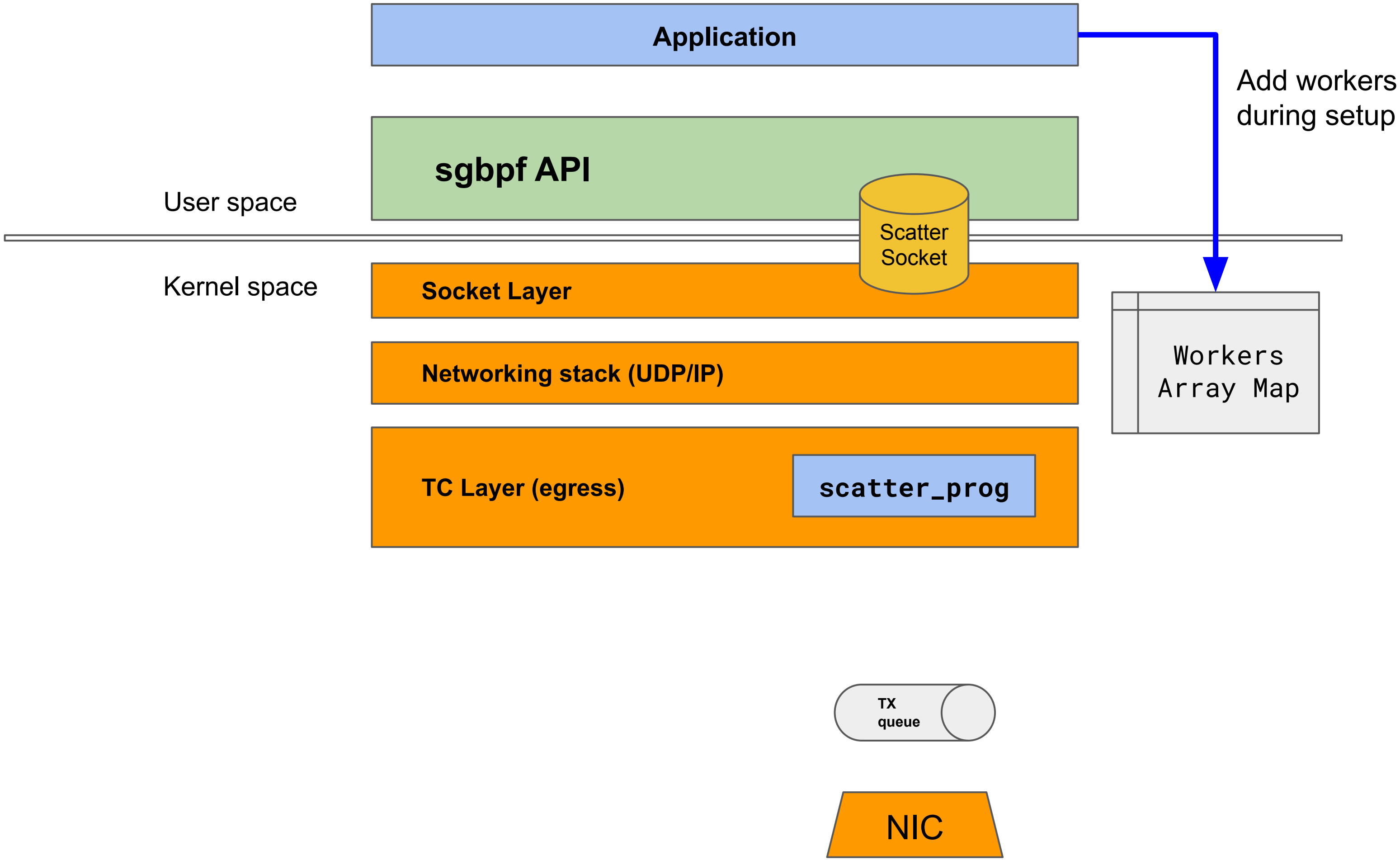
- Specifically, it occurs **after** the main networking stack

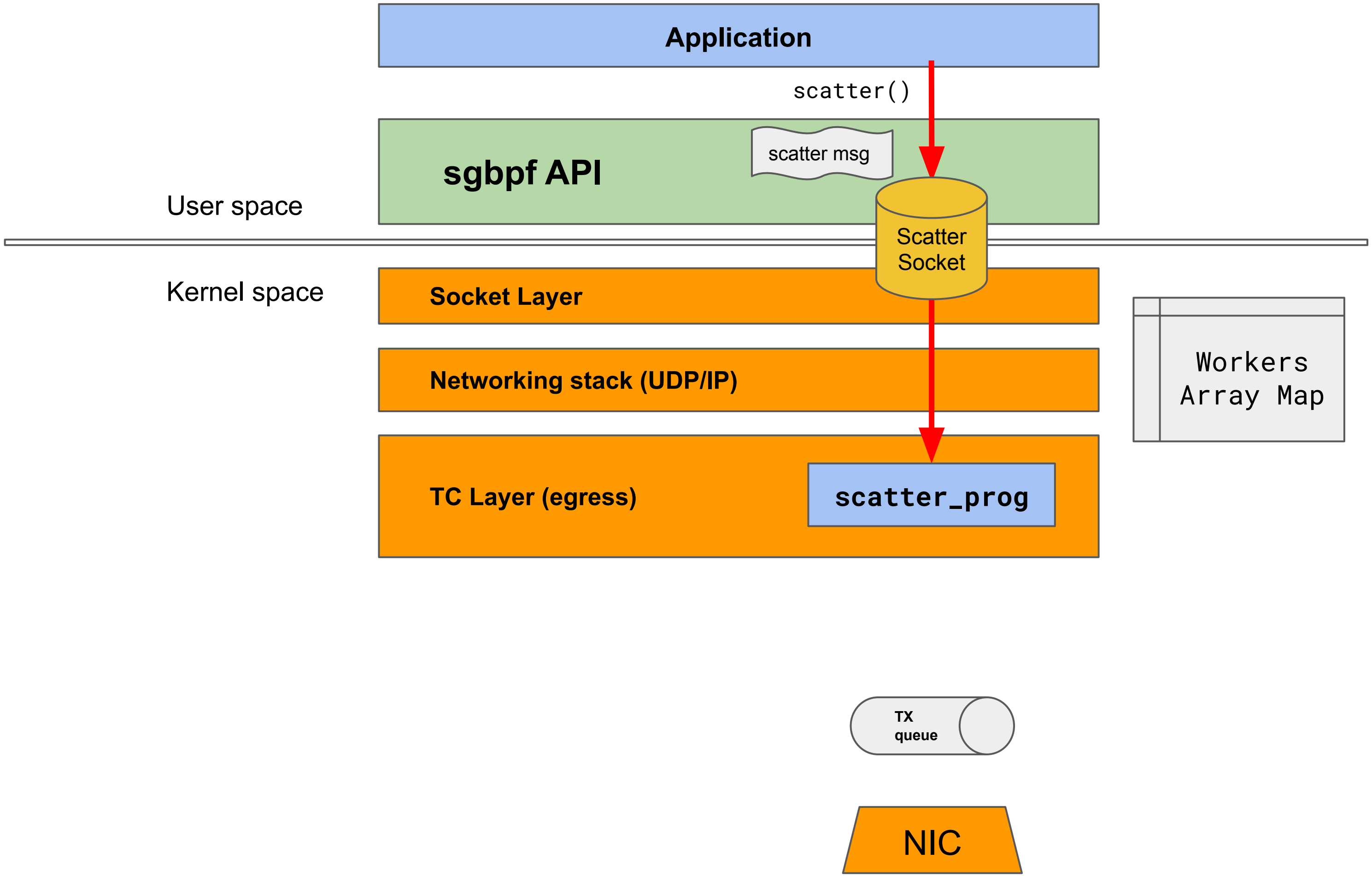
Accelerating the scatter phase

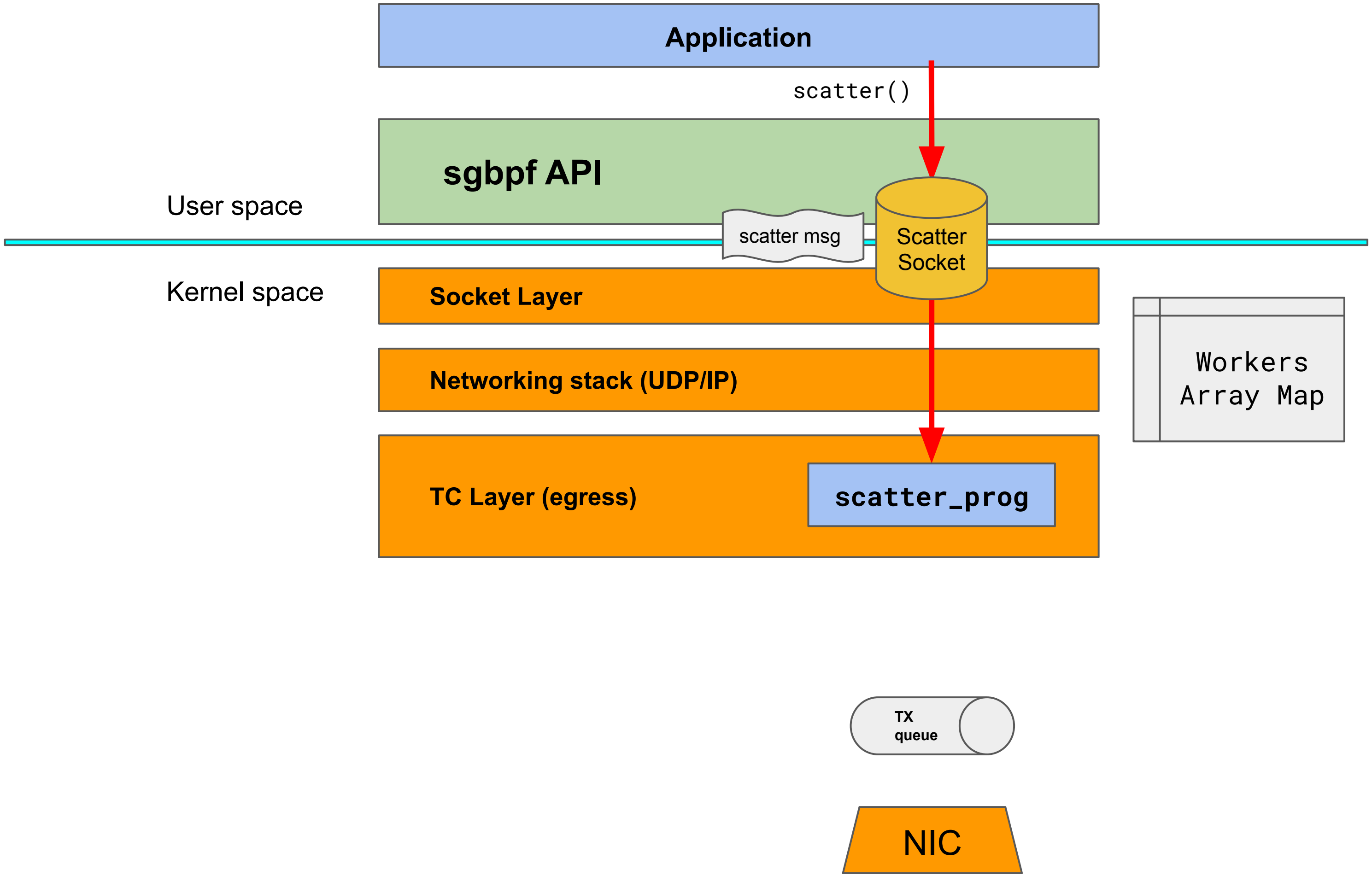
- With eBPF, we can perform the broadcast inside the kernel
- Specifically, it occurs **after** the main networking stack
- The cloned packets do not re-traverse the kernel networking stack

Accelerating the scatter phase

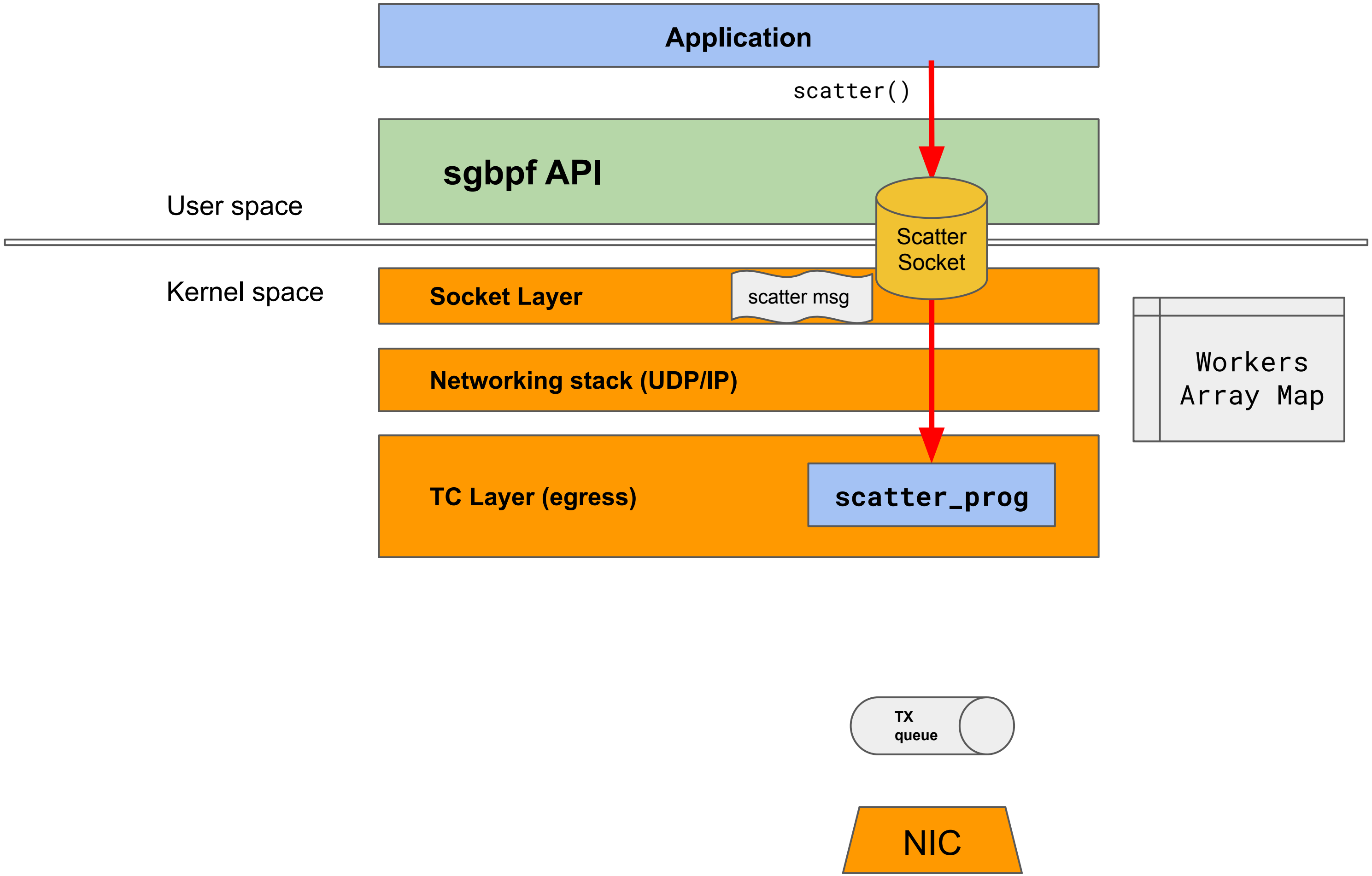
- With eBPF, we can perform the broadcast inside the kernel
- Specifically, it occurs **after** the main networking stack
- The cloned packets do not re-traverse the kernel networking stack
- Only a single message is written into the socket from the application to trigger the broadcast operation in the eBPF program
    - Hence a single network stack traversal is needed

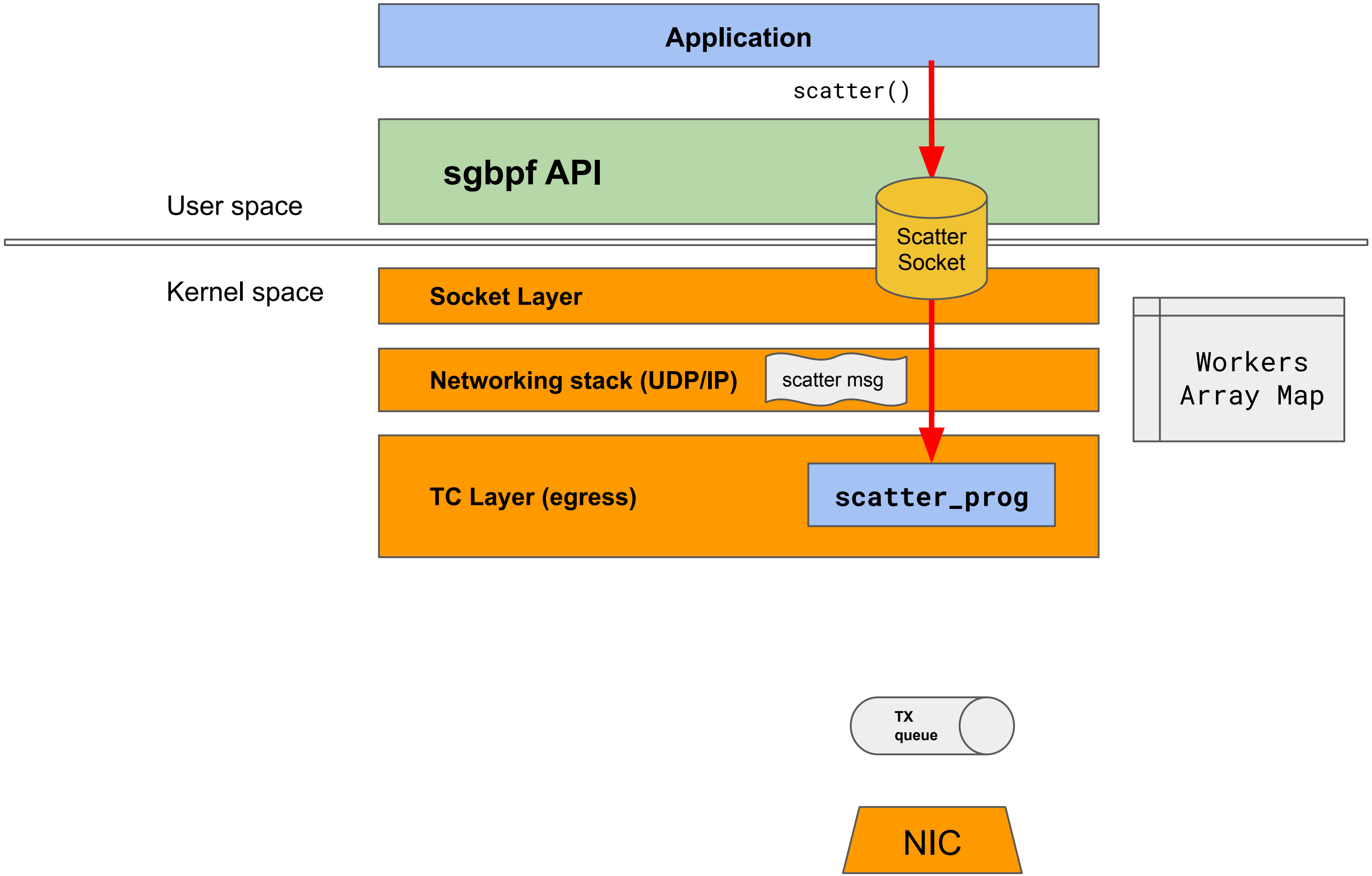

Application
Add workers
during setup
sgbpf API
User space
Scatter
Socket
Kernel space
Socket Layer
Workers
Array Map
Networking stack (UDP/IP)
TC Layer (egress)
scatter_prog
TX
queue
NIC


Application
scatter()
sgbpf API
scatter msg
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
TX queue
NIC

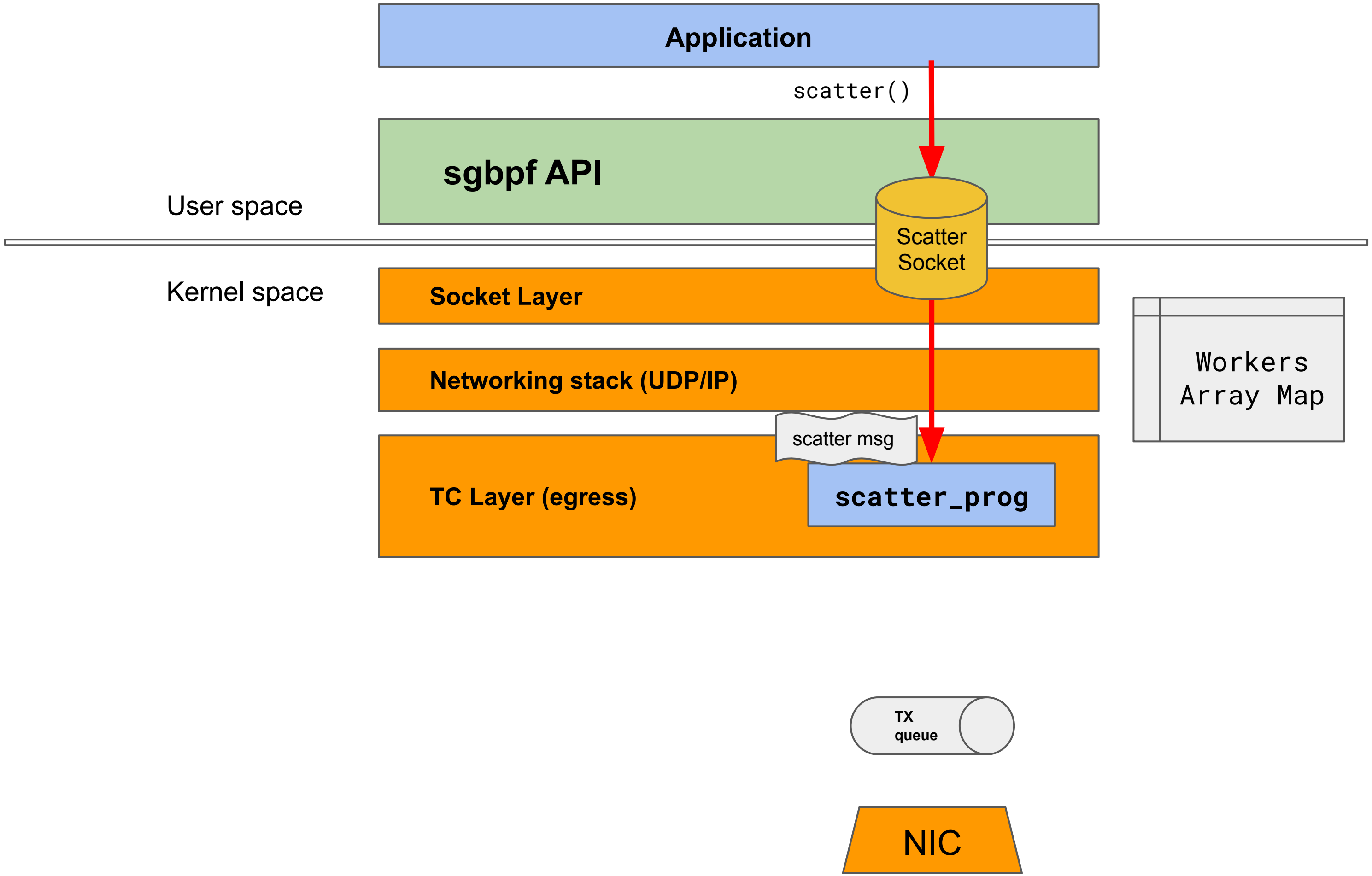


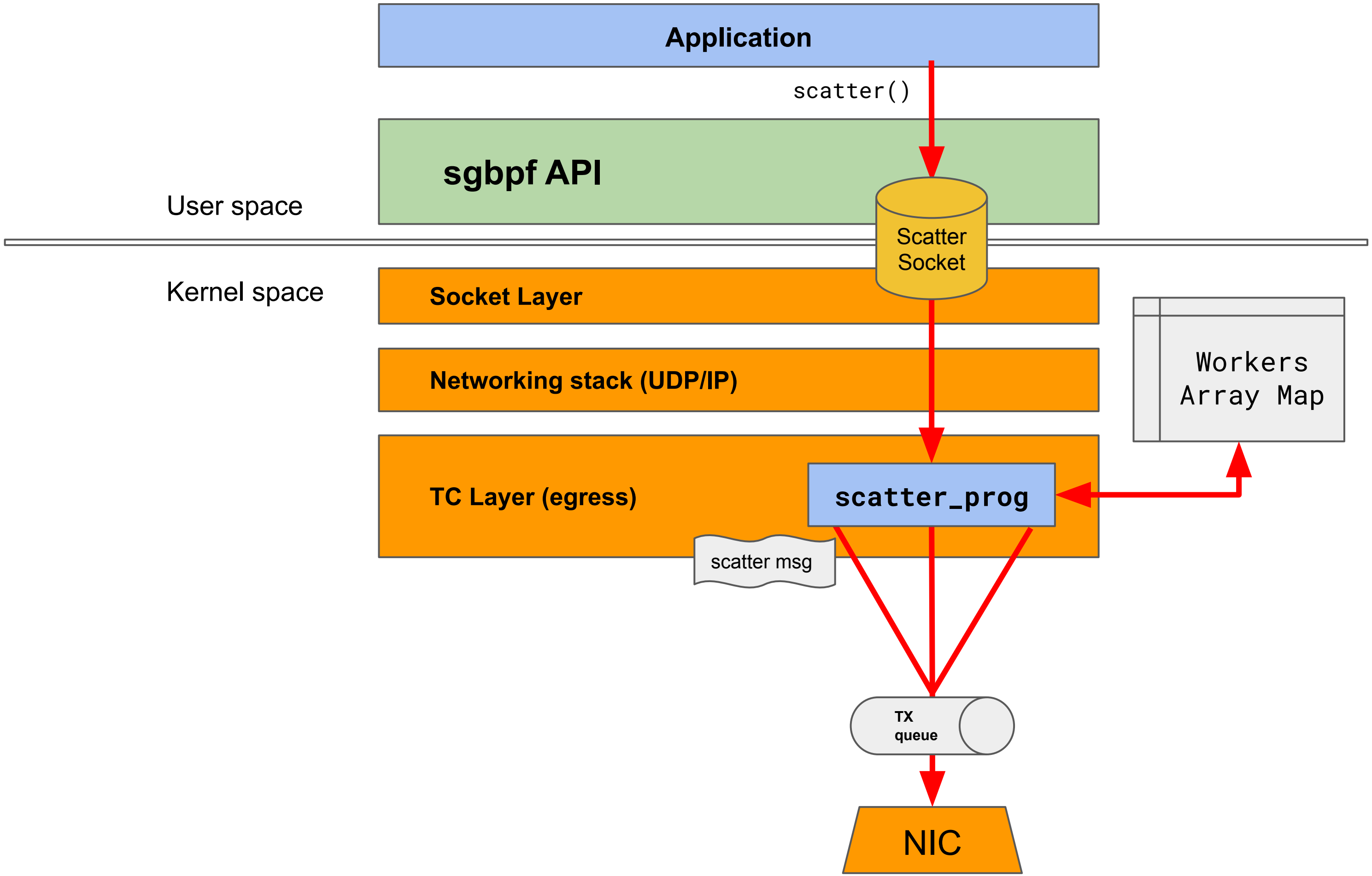
Application
scatter()
sgbpf API
User space
scatter msg
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
TX queue
NIC

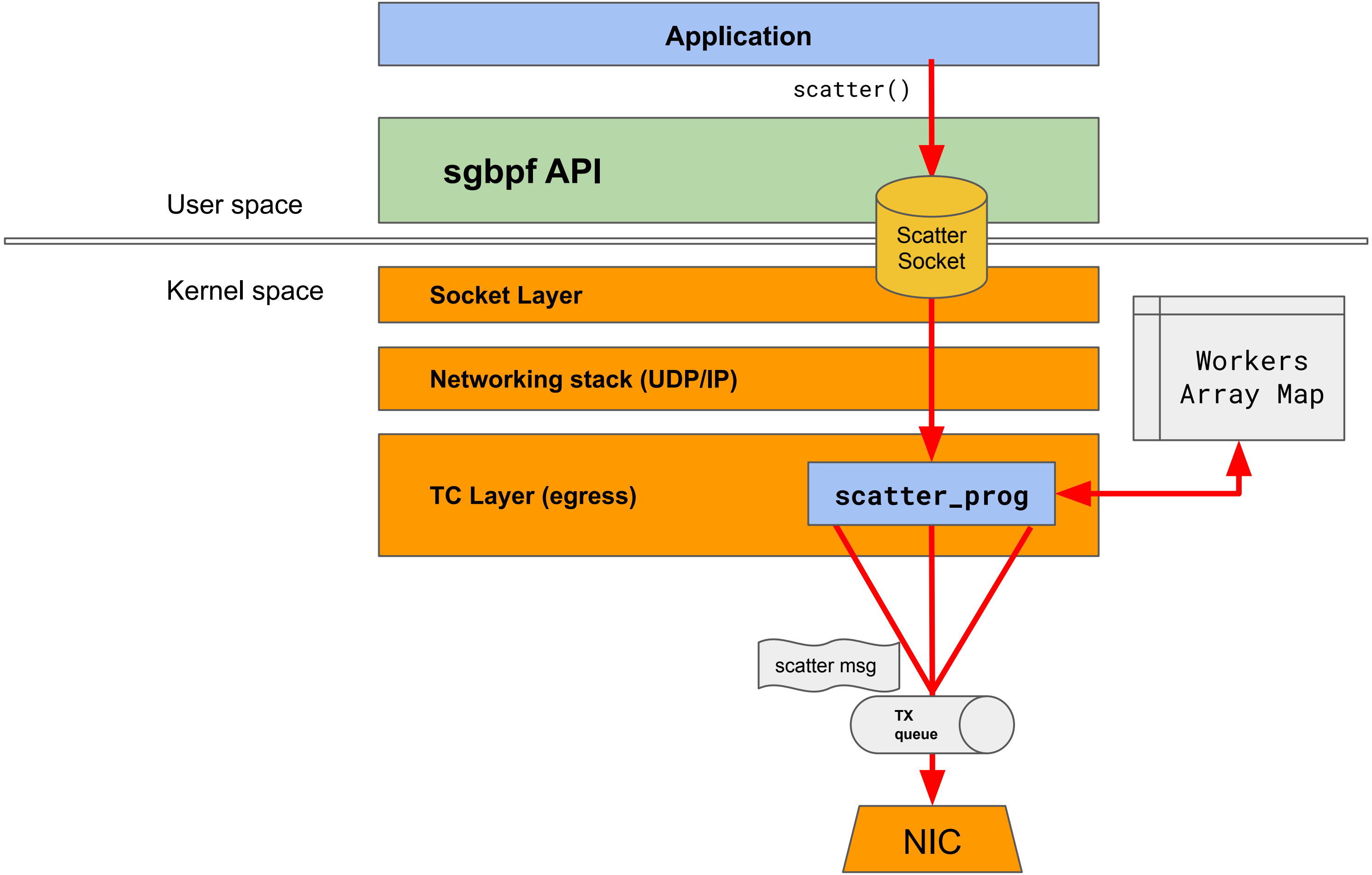

Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
scatter msg
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
TX queue
NIC

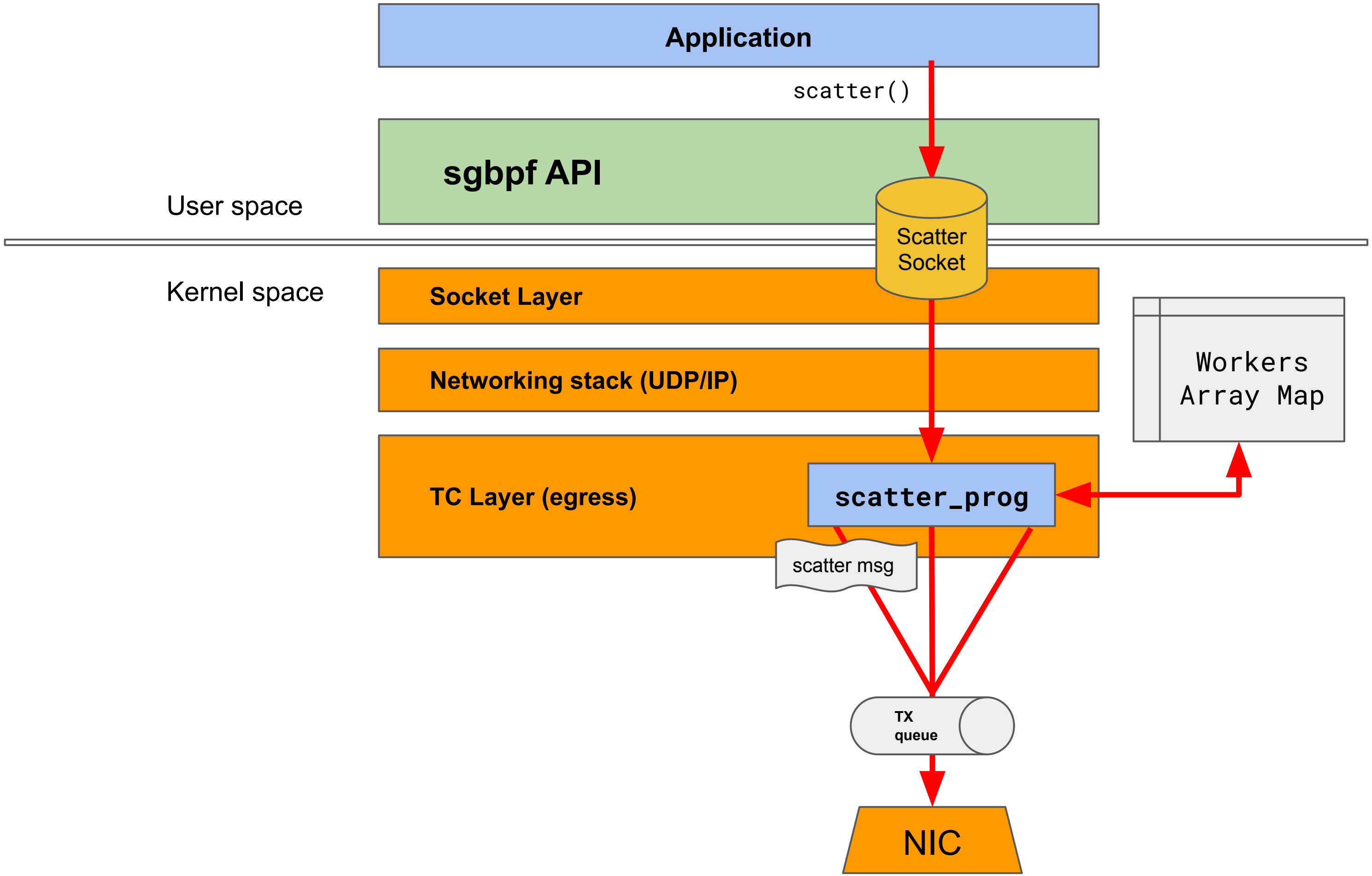


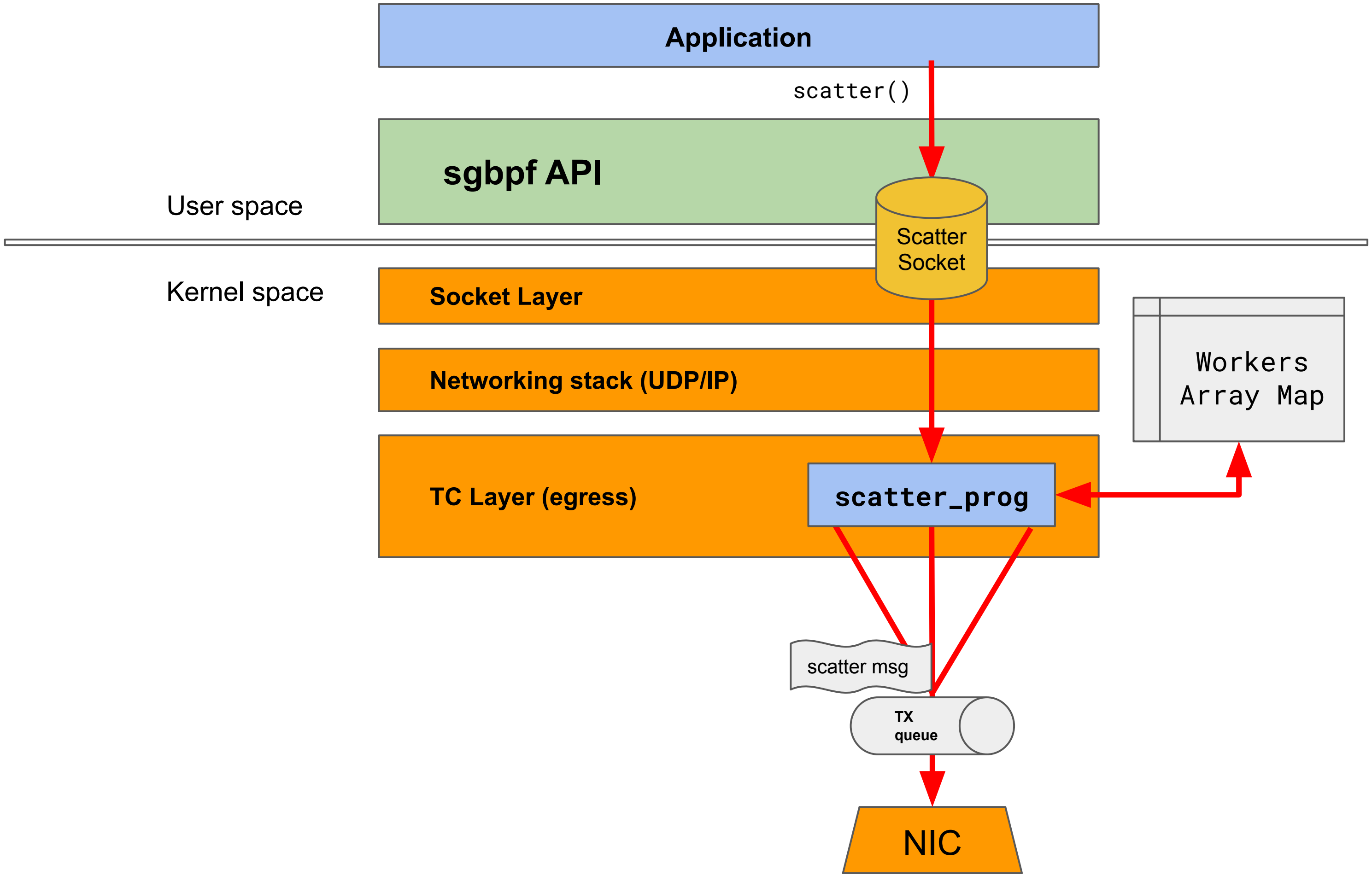
Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
scatter msg
TC Layer (egress)
scatter_prog
Workers Array Map
TX queue
NIC

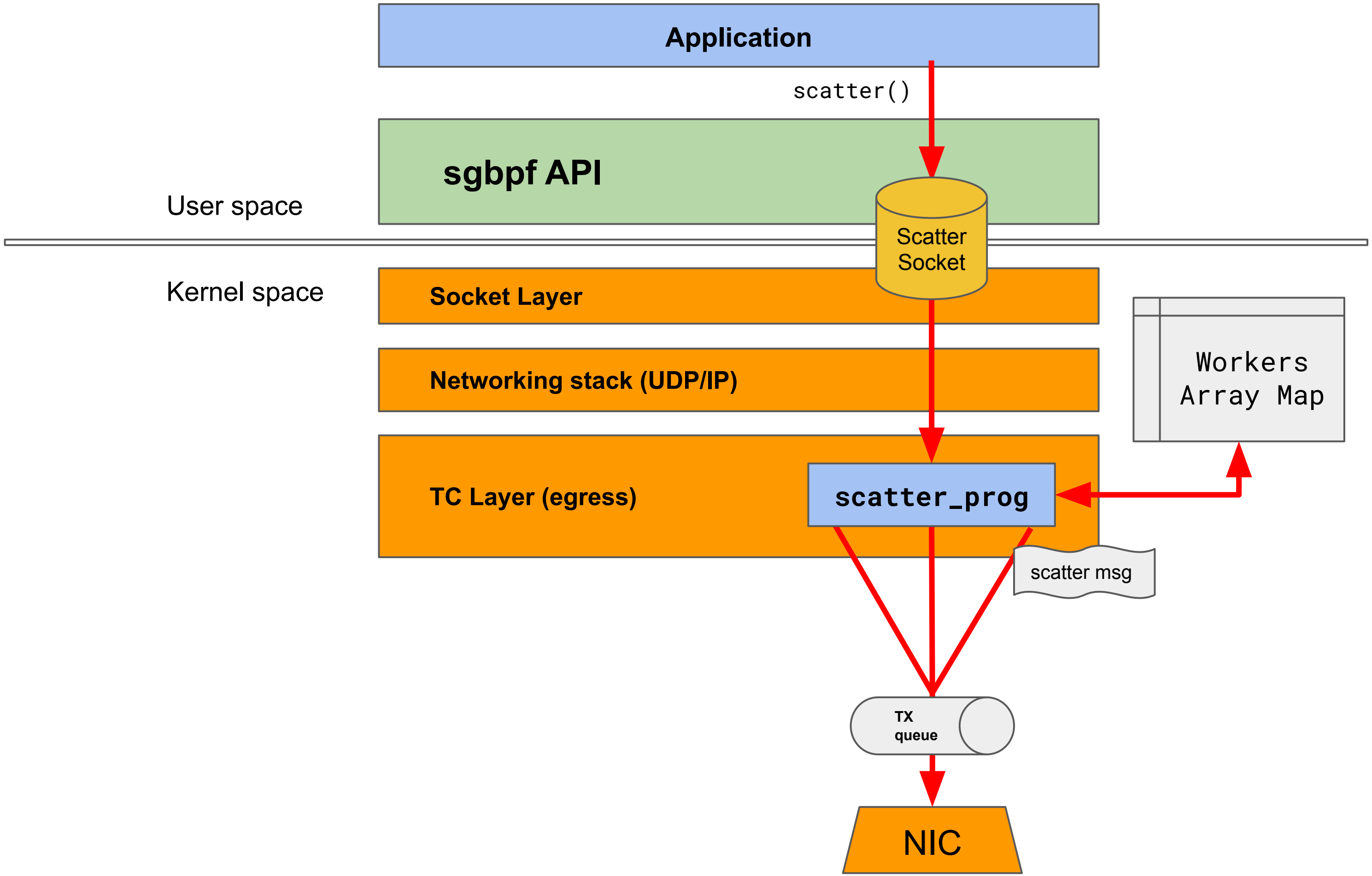

Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
scatter msg
TC Layer (egress)
scatter_prog
TX queue
NIC


Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
scatter msg
TX queue
NIC

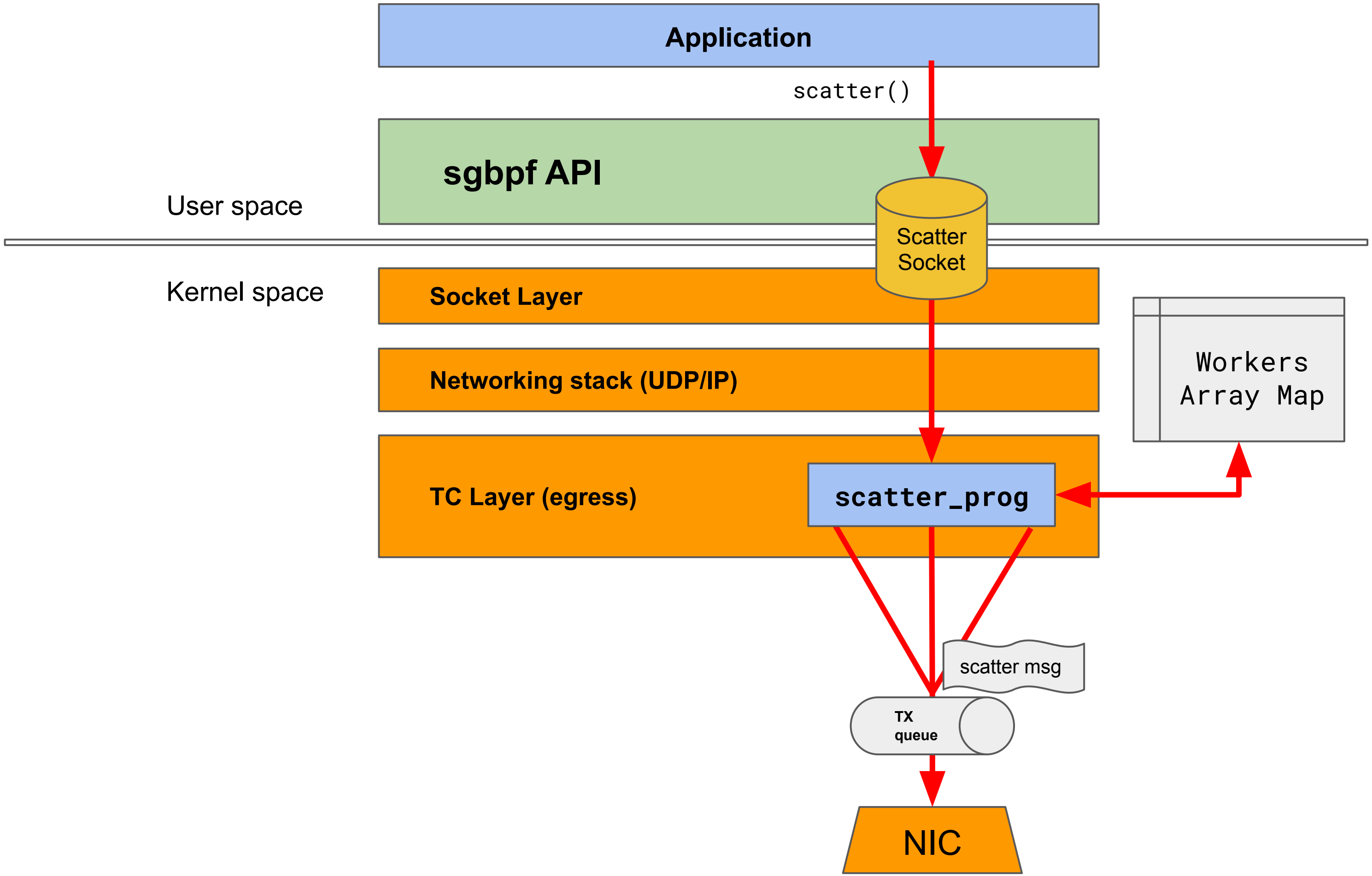

Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
TC Layer (egress)
scatter_prog
Workers Array Map
scatter msg
TX queue
NIC


Application
scatter()
sgbpf API
User space
Scatter
Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers
Array Map
TC Layer (egress)
scatter_prog
scatter msg
TX
queue
NIC


Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
scatter msg
TX queue
NIC



Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
scatter msg
TX queue
NIC



Application
scatter()
sgbpf API
User space
Scatter Socket
Kernel space
Socket Layer
Networking stack (UDP/IP)
Workers Array Map
TC Layer (egress)
scatter_prog
scatter msg
TX queue
NIC

Accelerating the scatter phase

- The eBPF-enabled version now requires only:
  - A single system call
  - A single traversal of the kernel networking stack

Accelerating the scatter phase

- The eBPF-enabled version now requires only:
  - A single system call
  - A single traversal of the kernel networking stack

- Good start, but the main source of overhead lies in the gather phase.

Accelerating the gather phase

- Can we process incoming responses as early as possible without incurring a kernel-to-user crossing per packet?

Accelerating the gather phase

- Can we process incoming responses as early as possible without incurring a kernel-to-user crossing per packet?

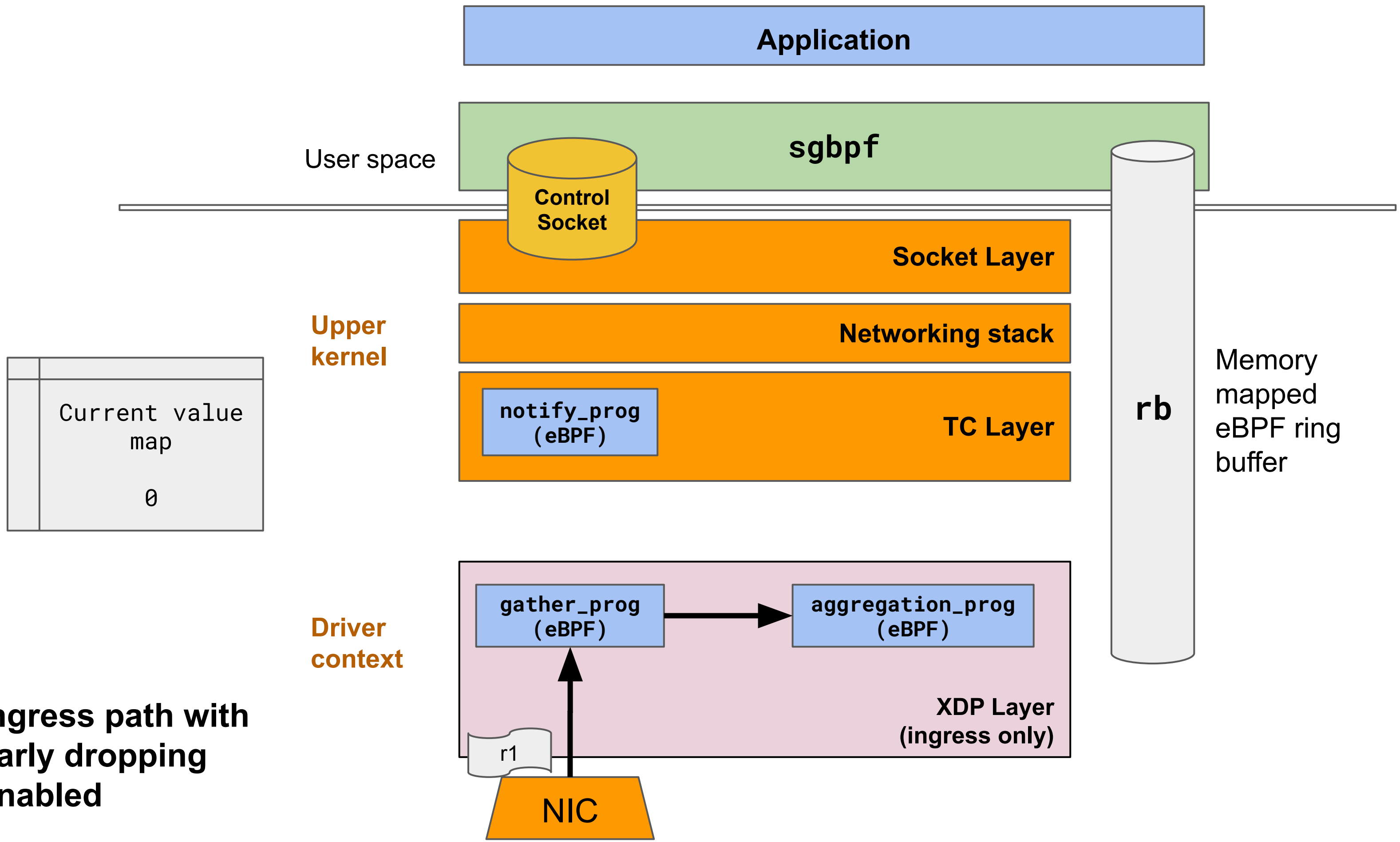
- **Idea:** execute the aggregation logic in the kernel using eBPF
  - Perform pre-stack packet processing to bypass the kernel networking stack
  - The application only gets the final aggregated value
  - Minimises the number of syscalls, number of copies and network stack traversals required

Accelerating the gather phase

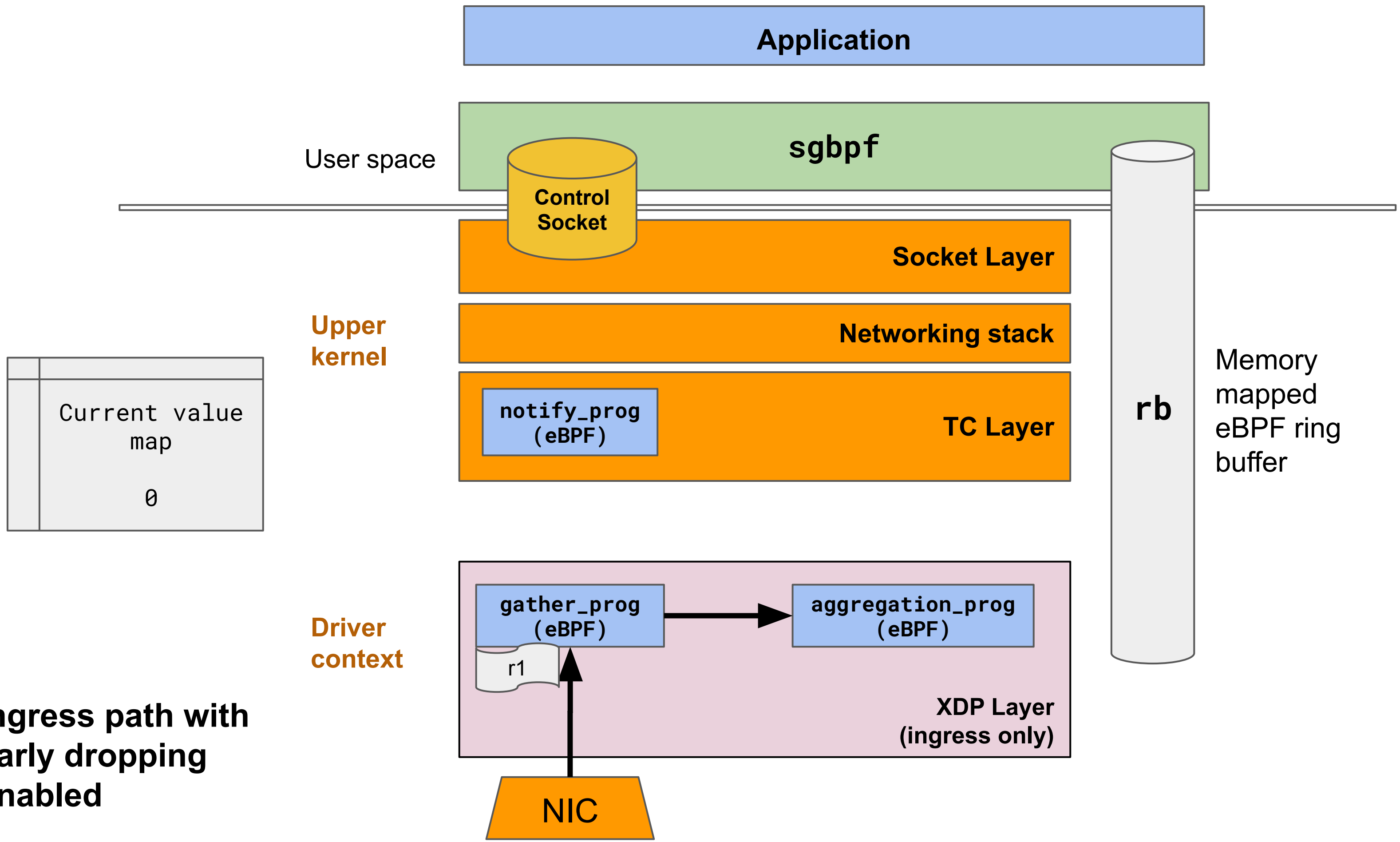
- *sgbpf* relies on pre-stack processing with XDP to aggregate the responses as early as possible
- The aggregation logic is a user-supplied eBPF XDP program
  - Allows flexibility for developer to define the semantics of the aggregation
  - However, must conform to the rules imposed by the eBPF static verifier
- The packet's fate after the aggregation is also specified by the developer:
  - Drop the packet, if no longer needed (referred to as early dropping - recommended)
  - Allow the packet, if needed in the application for further processing

Accelerating the gather phase

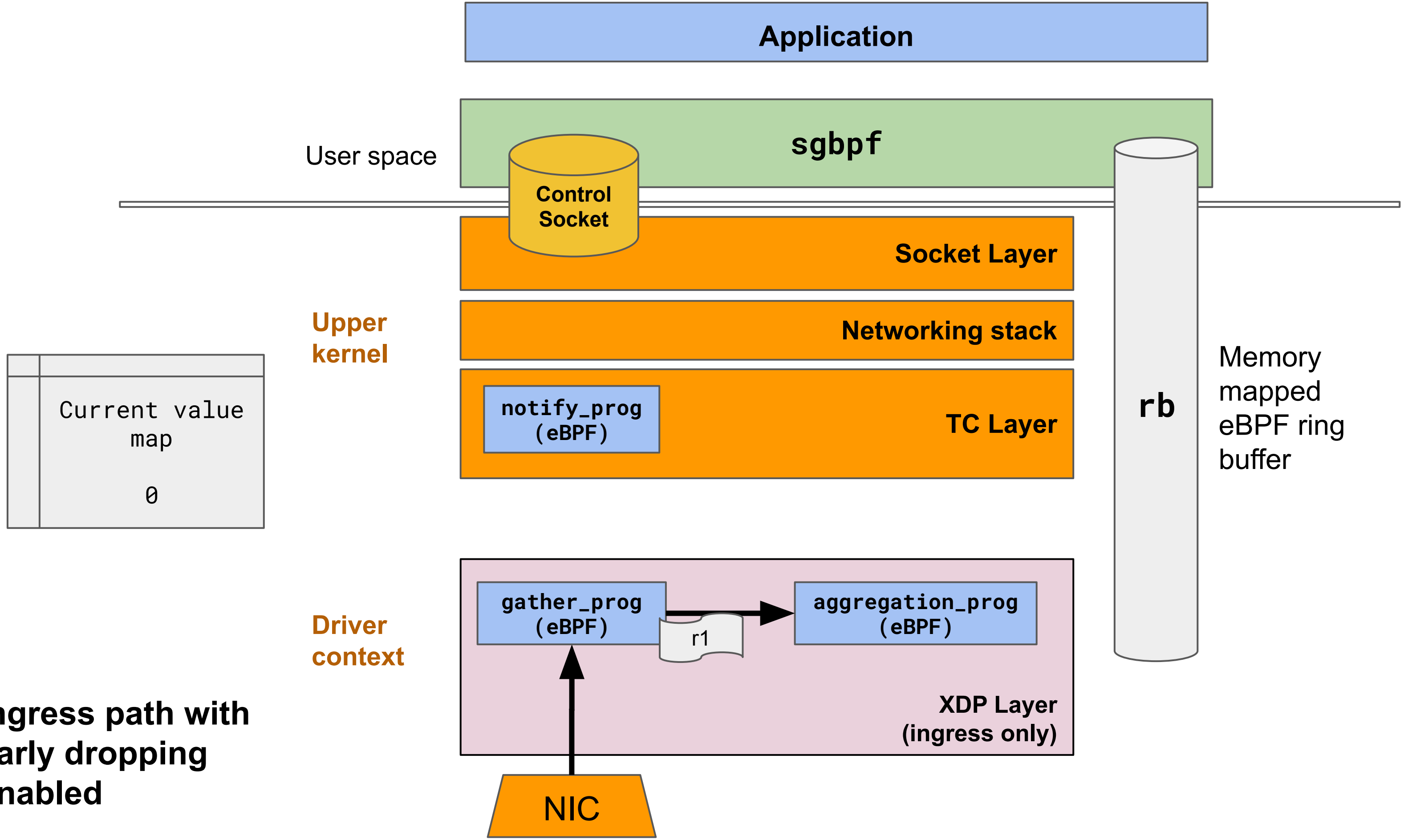
- Once all the worker responses have been aggregated, the final aggregated value is delivered to the user-space application.
- This takes place via one of the following communication channels:
  - **Control socket**: deliver data using a packet redirected into a specific socket monitored by the application
  - **Ring buffer**: use a shared `mmaped` buffer to deliver data to the application



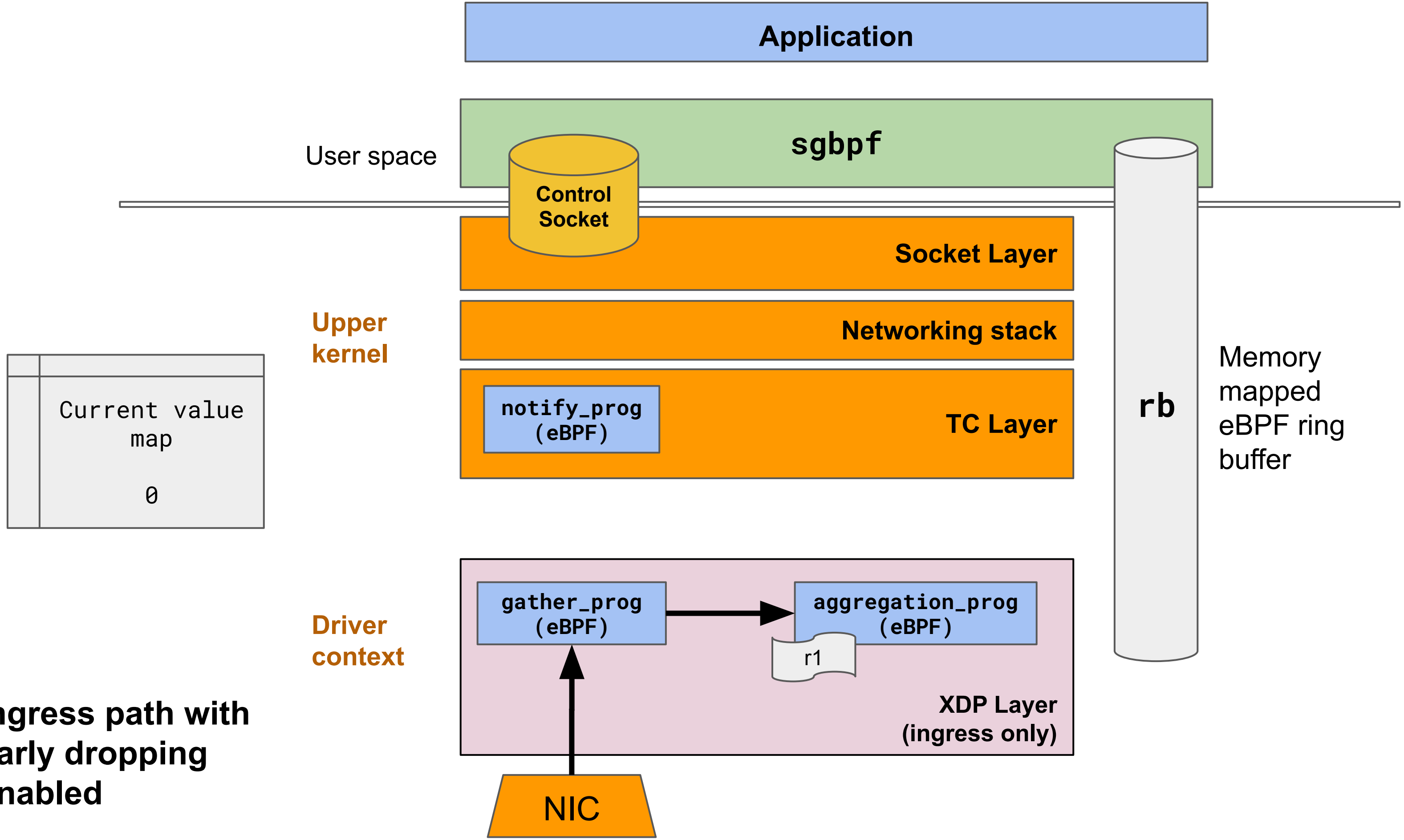
Ingress path with early dropping enabled



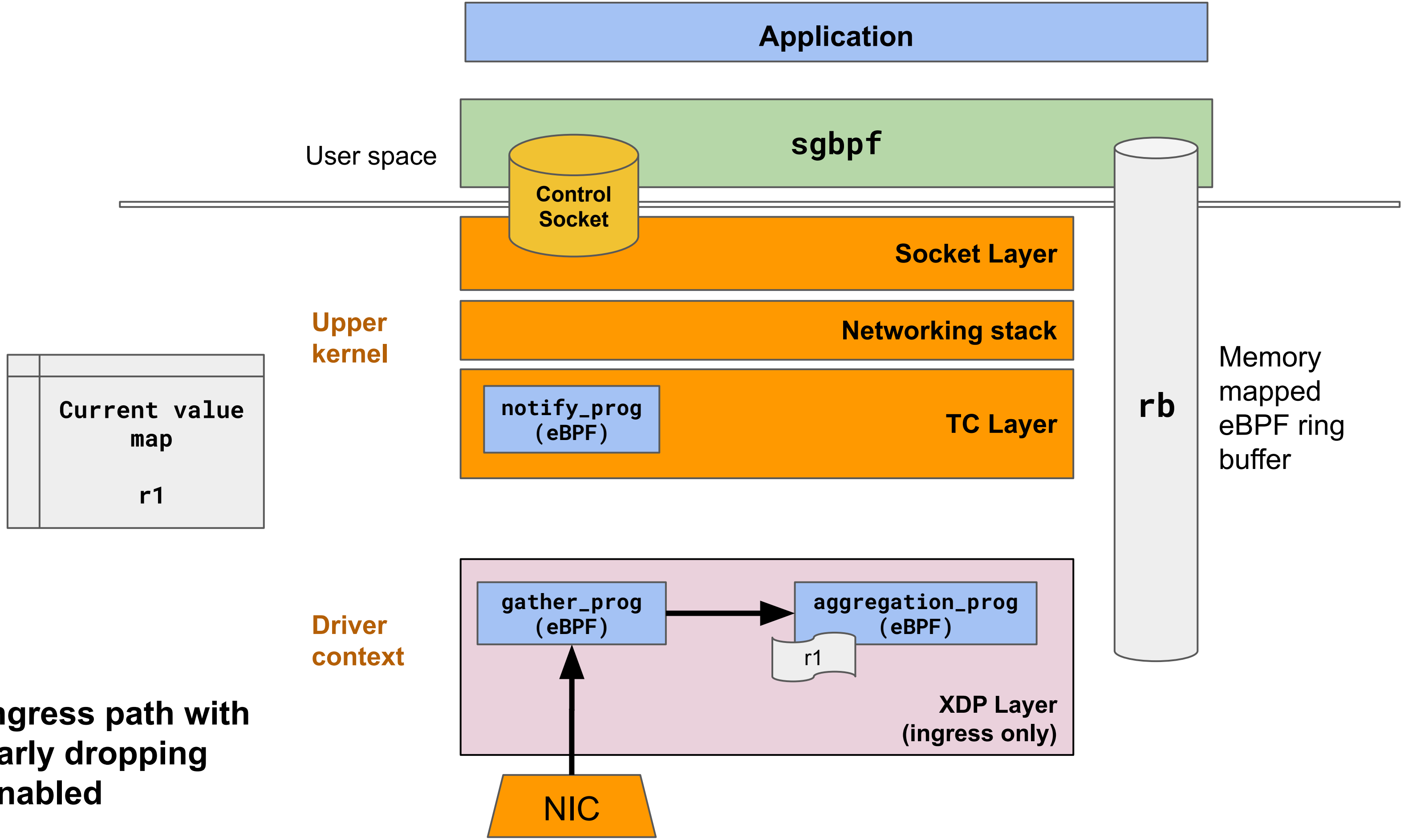
Ingress path with early dropping enabled



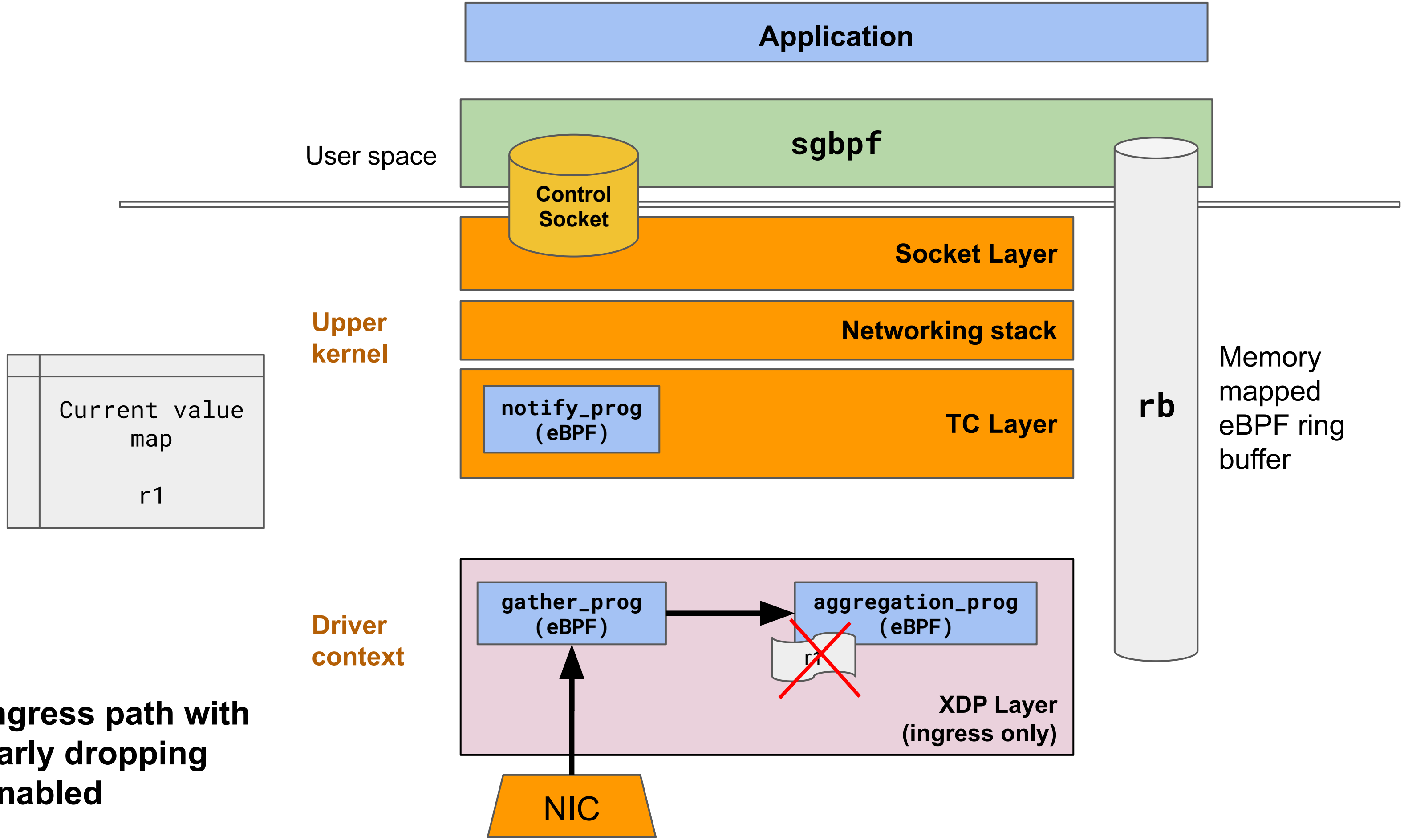
Ingress path with early dropping enabled



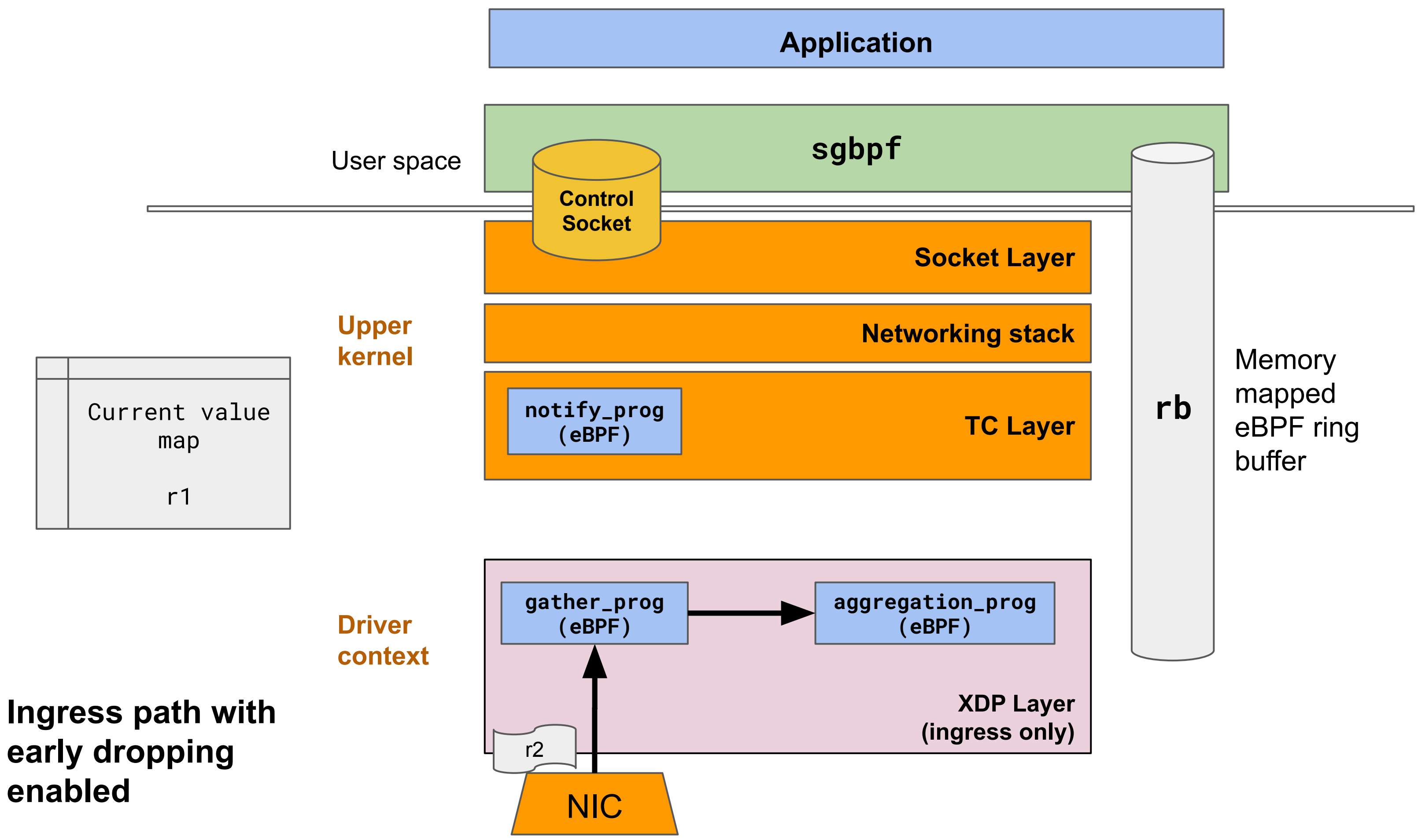
Ingress path with early dropping enabled

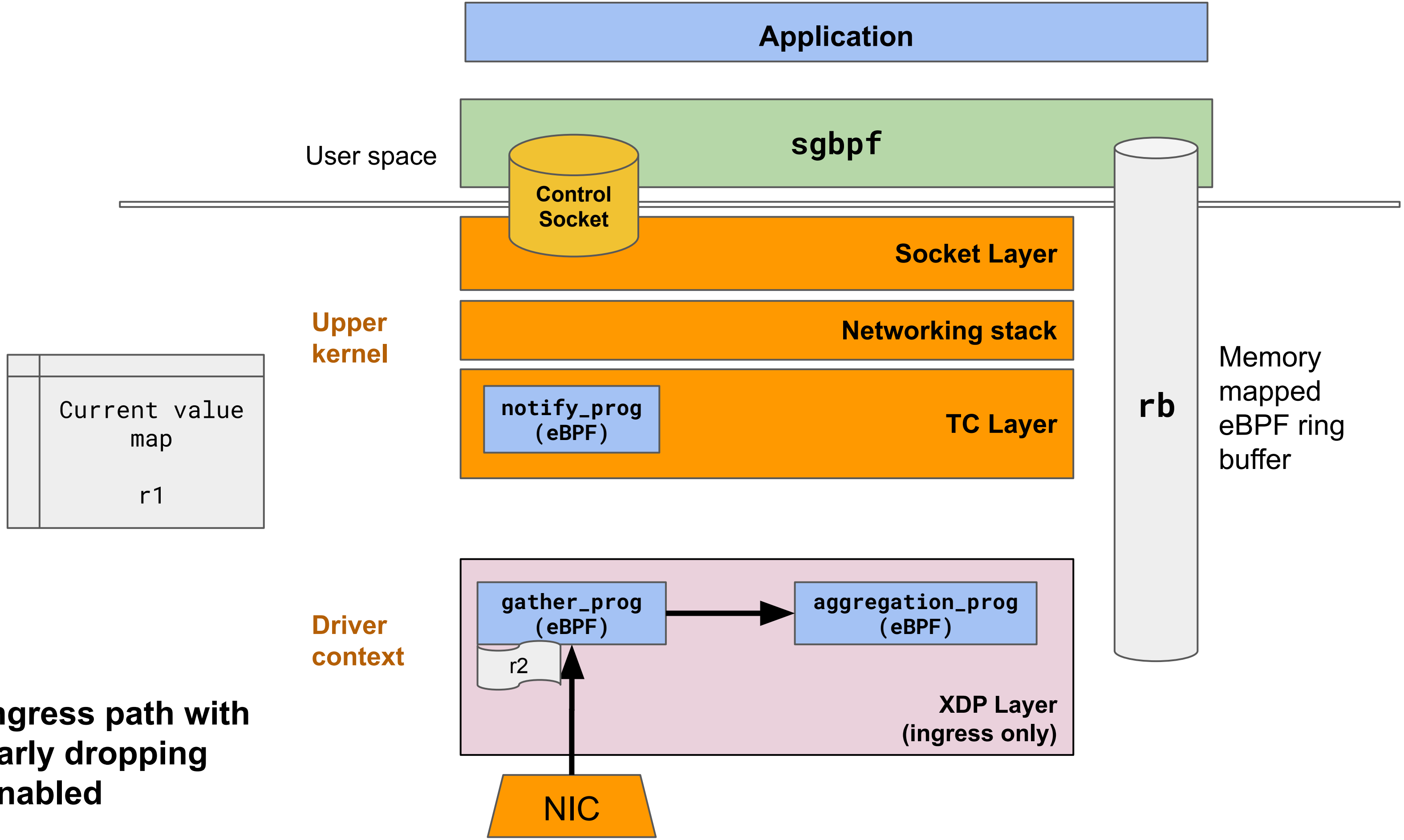


Ingress path with early dropping enabled

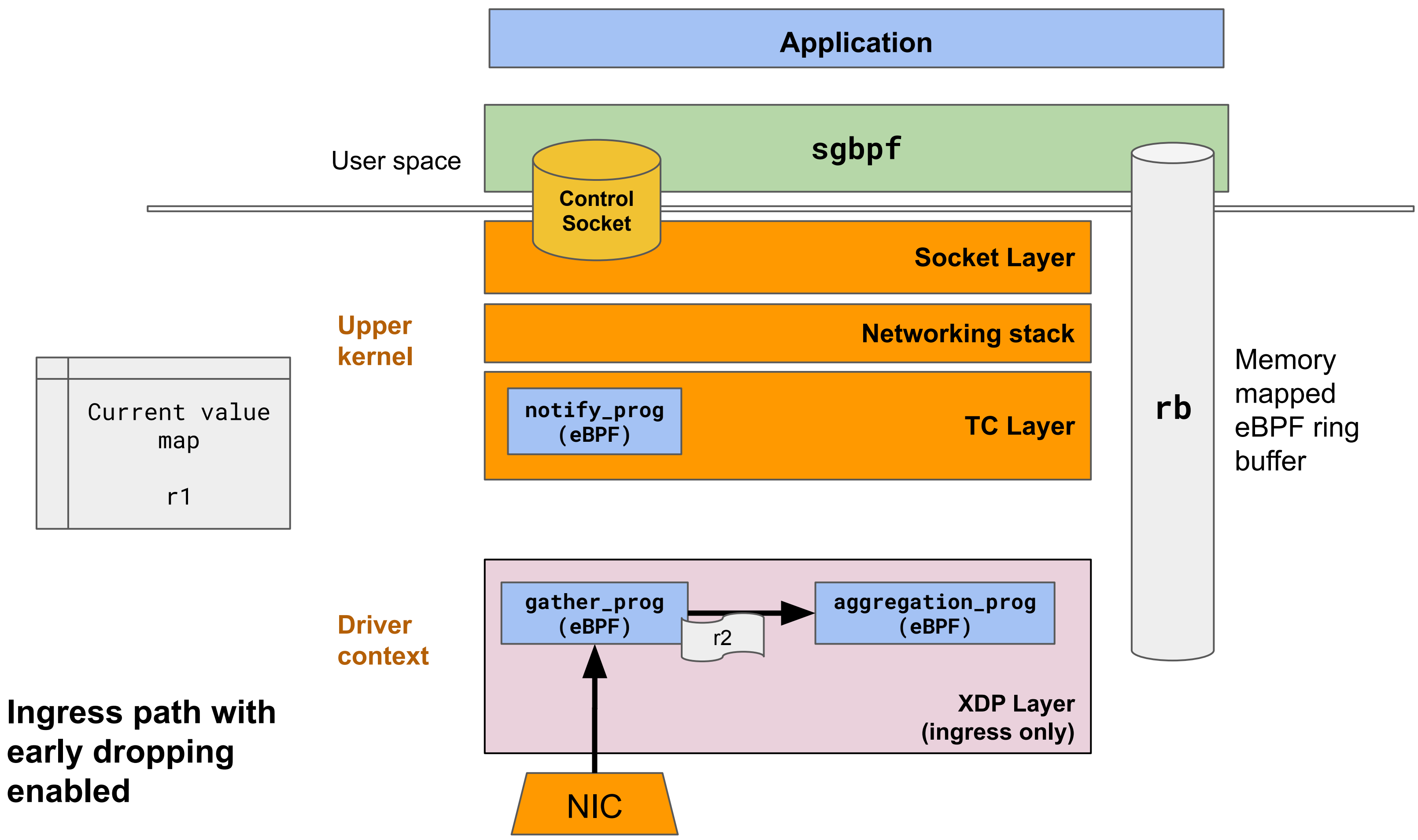


Ingress path with early dropping enabled

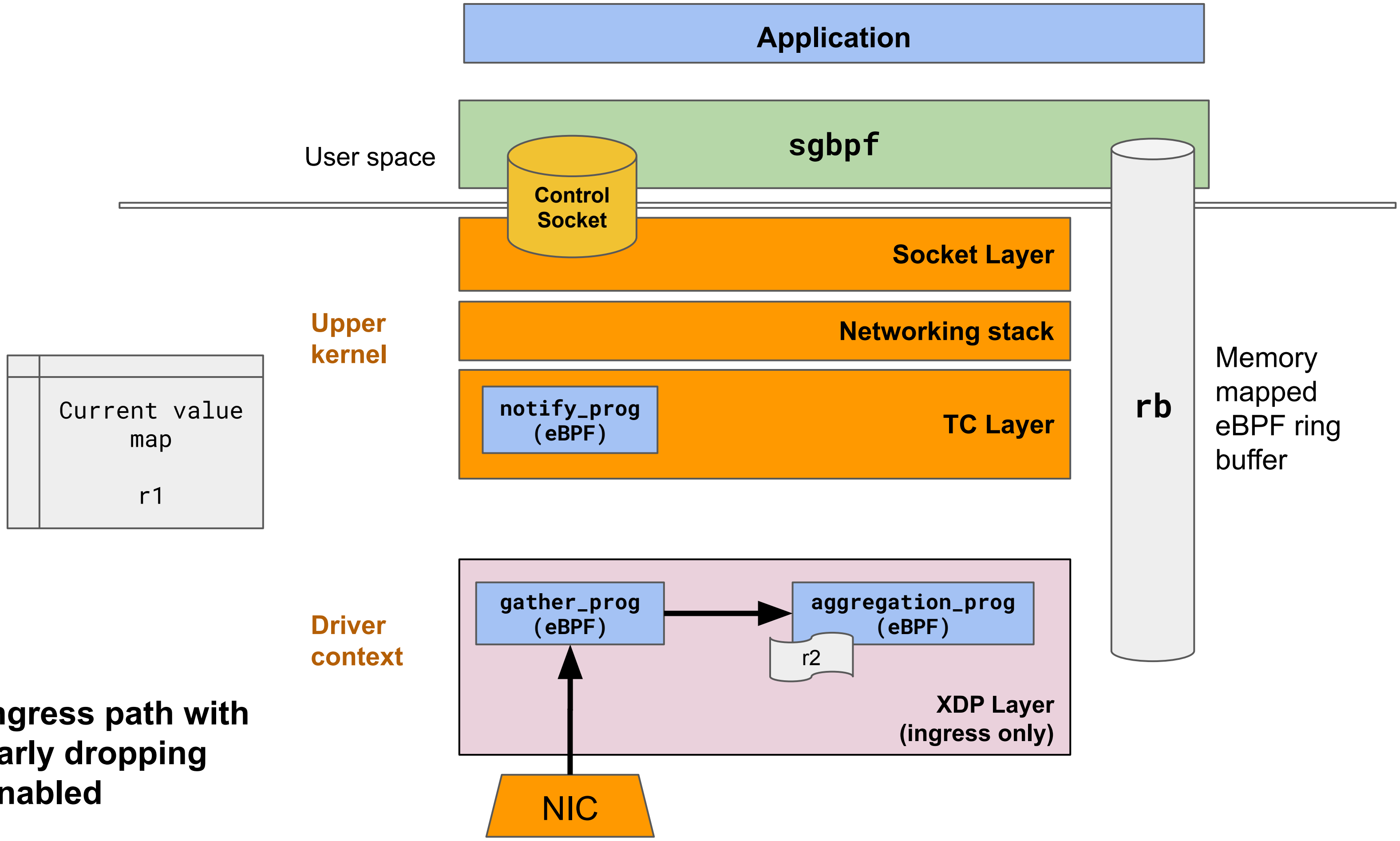

Application
sgbpf
User space
Control Socket
Socket Layer
Upper kernel
Networking stack
Current value map
r1
notify_prog (eBPF)
TC Layer
rb
Memory mapped eBPF ring buffer
Driver context
gather_prog (eBPF)
aggregation_prog (eBPF)
XDP Layer (ingress only)
r2
NIC
Ingress path with early dropping enabled



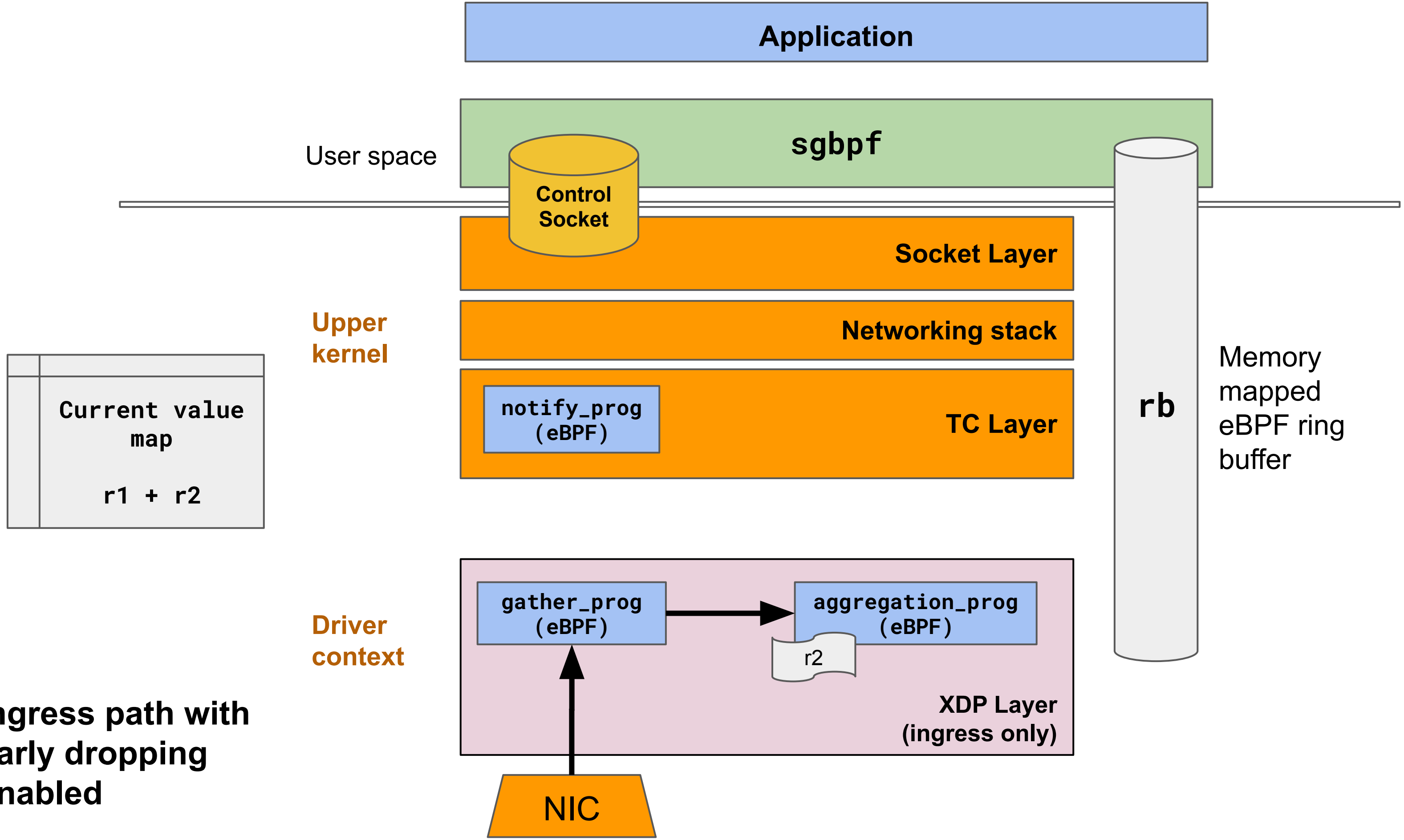
Ingress path with early dropping enabled



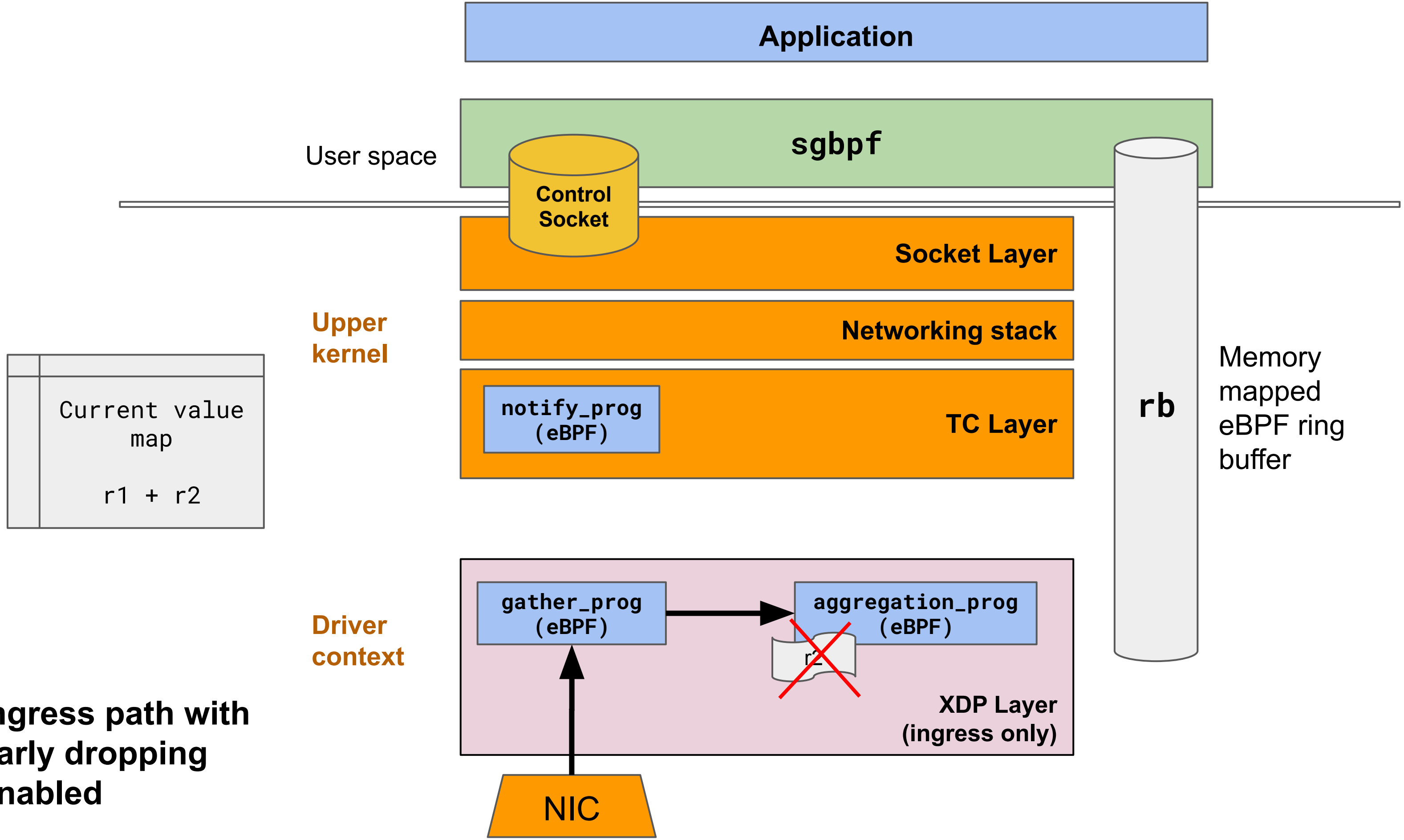
Application
sgbpf
User space
Control Socket
Socket Layer
Upper kernel
Networking stack
Current value map
r1
notify_prog (eBPF)
TC Layer
rb
Memory mapped eBPF ring buffer
Driver context
gather_prog (eBPF)
r2
aggregation_prog (eBPF)
XDP Layer (ingress only)
NIC
Ingress path with early dropping enabled



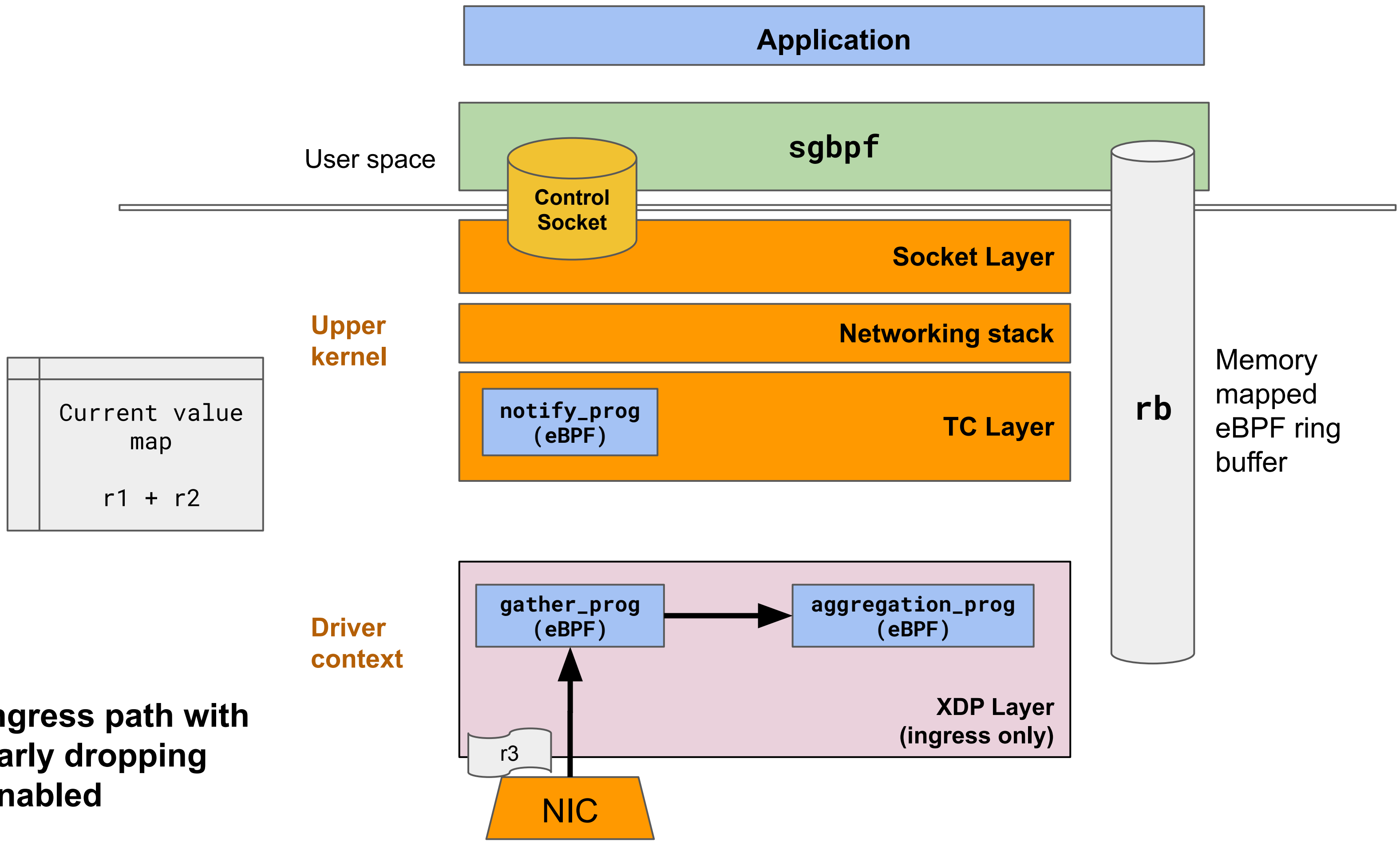
Ingress path with early dropping enabled



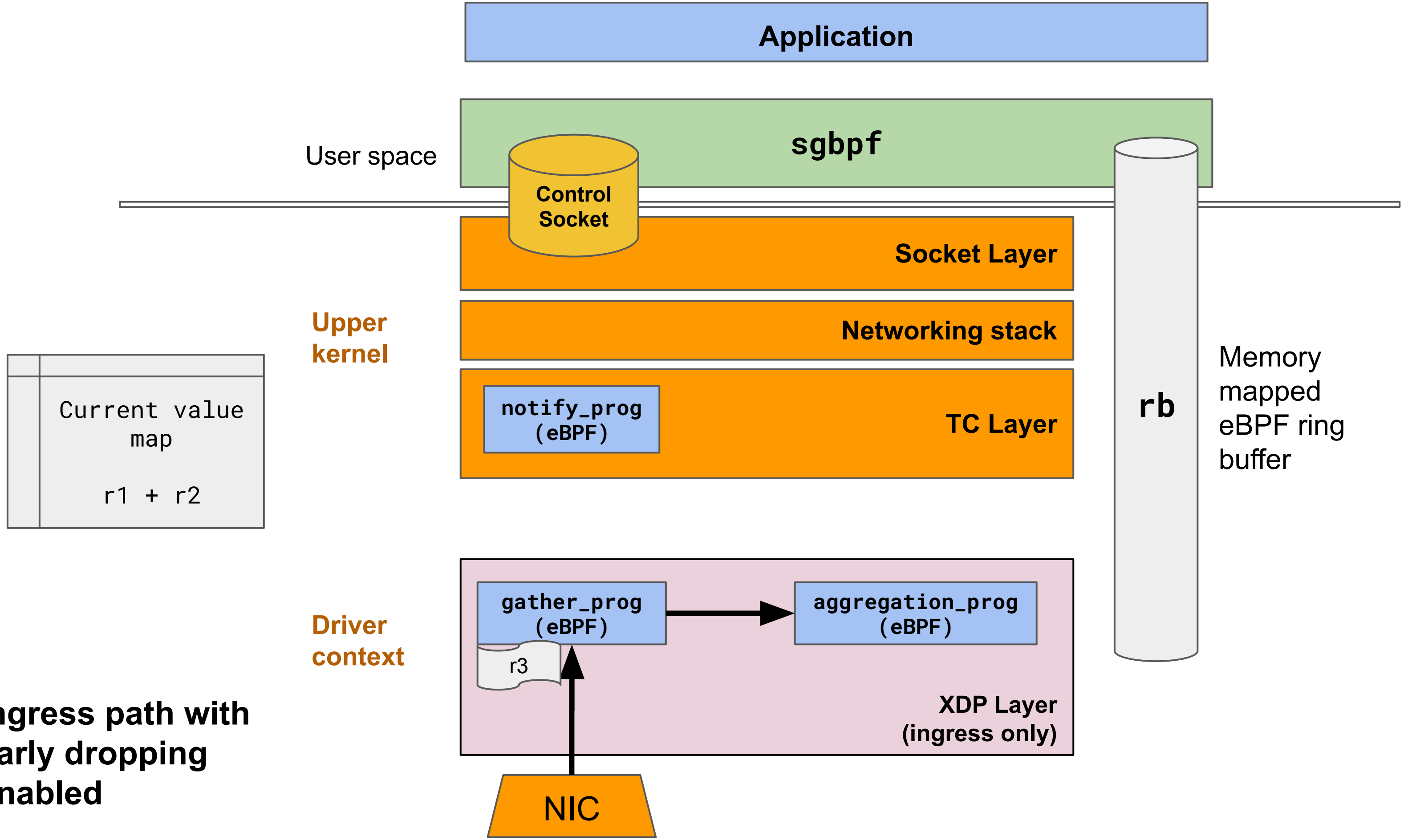
Ingress path with early dropping enabled



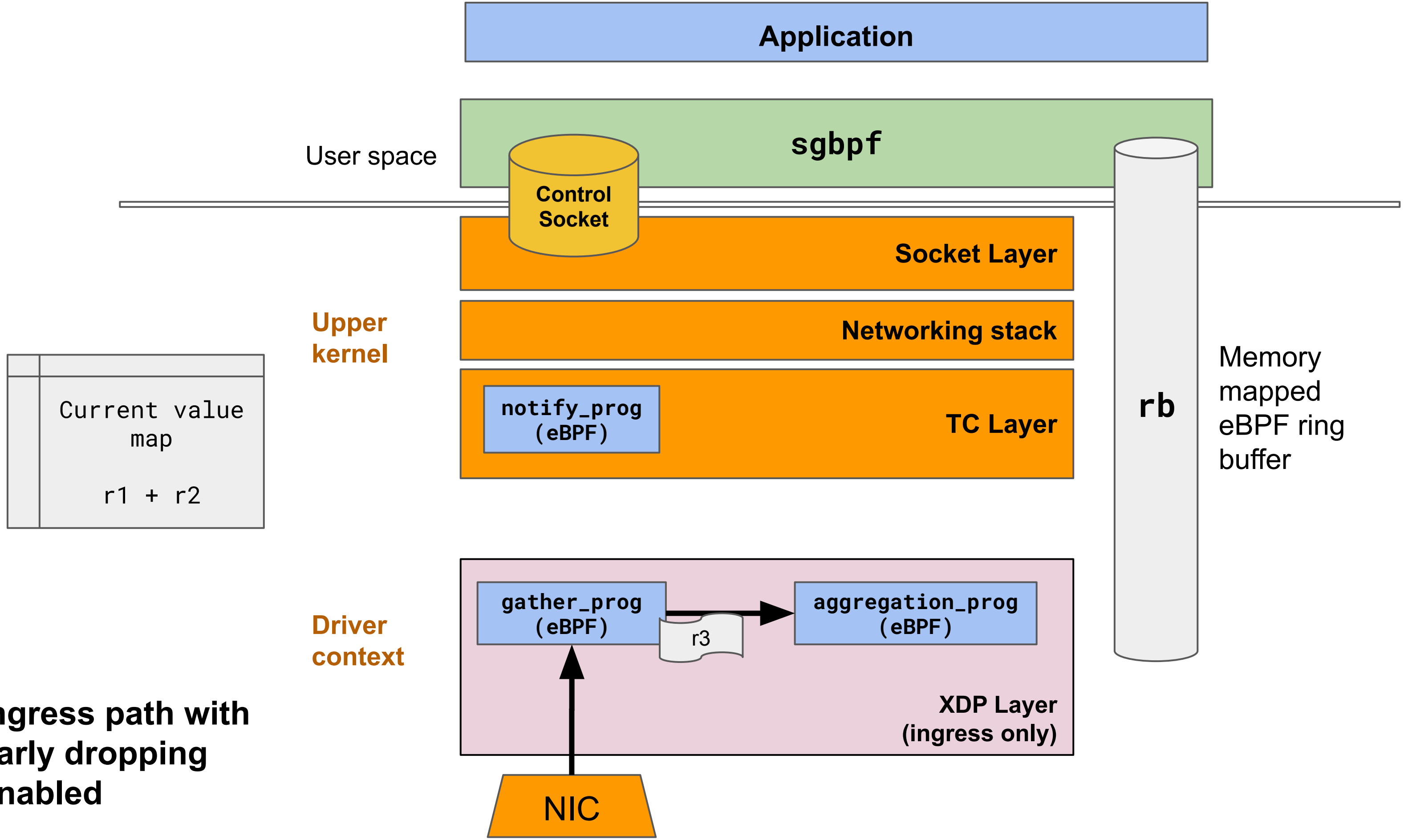
Ingress path with early dropping enabled



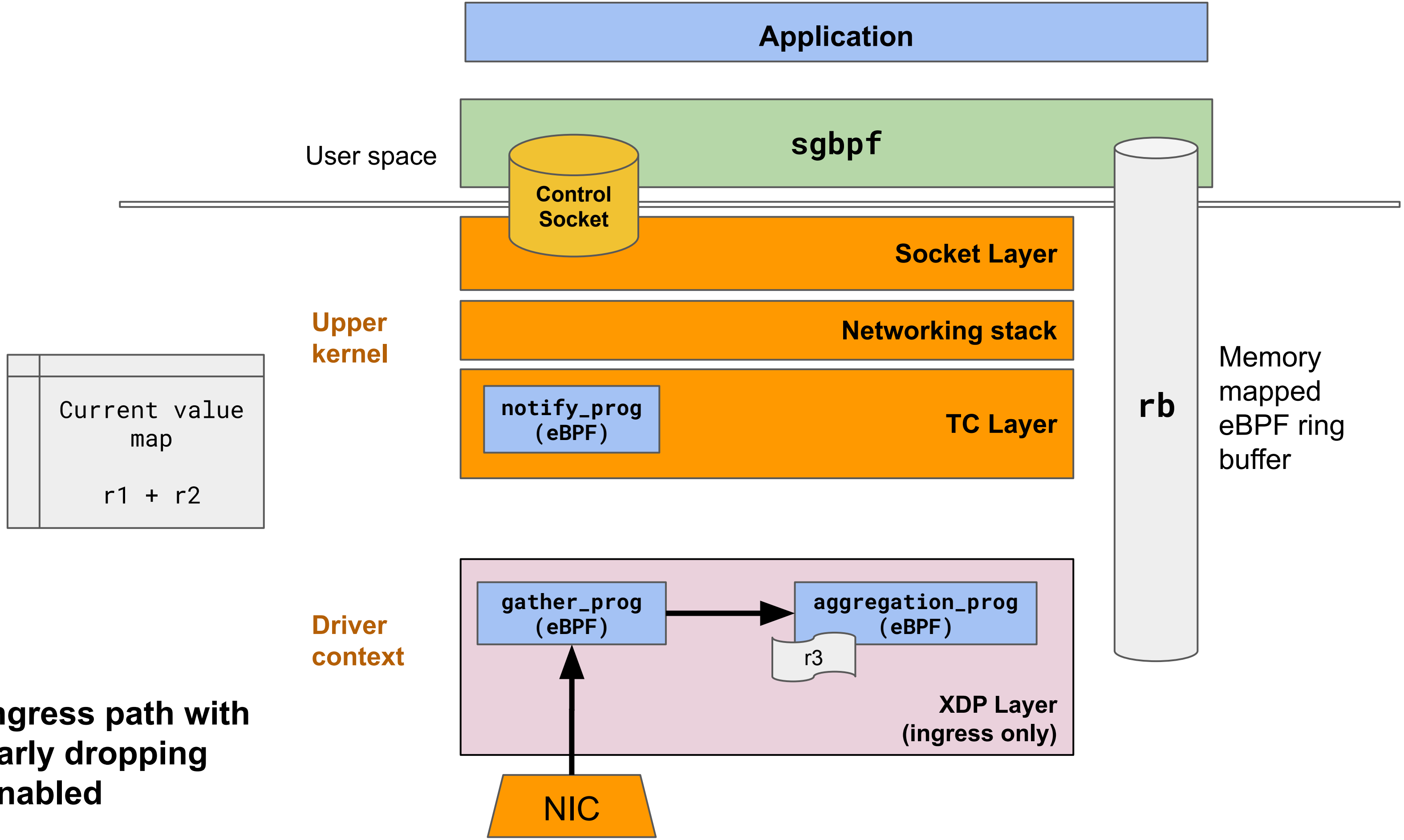
Ingress path with early dropping enabled



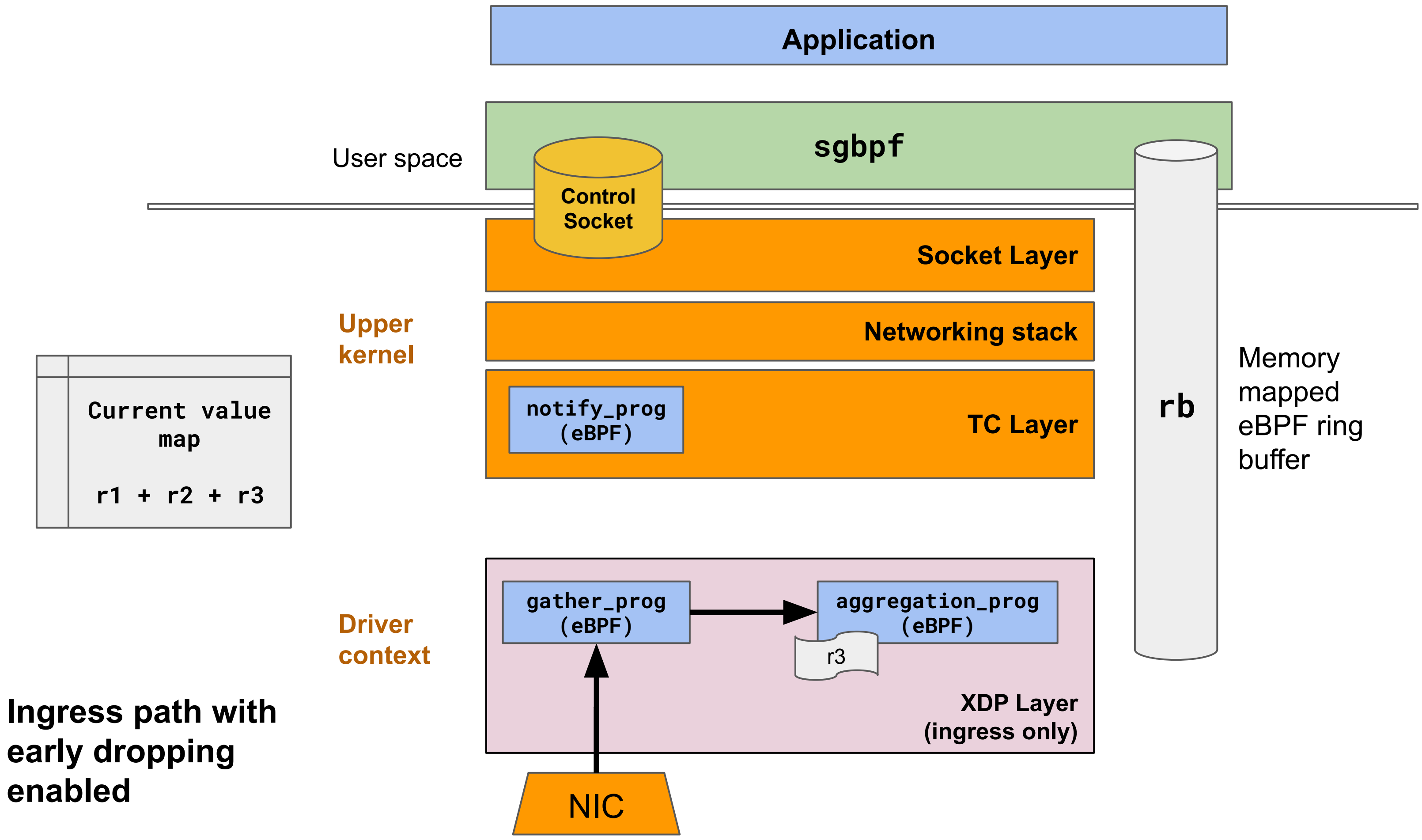
Ingress path with early dropping enabled

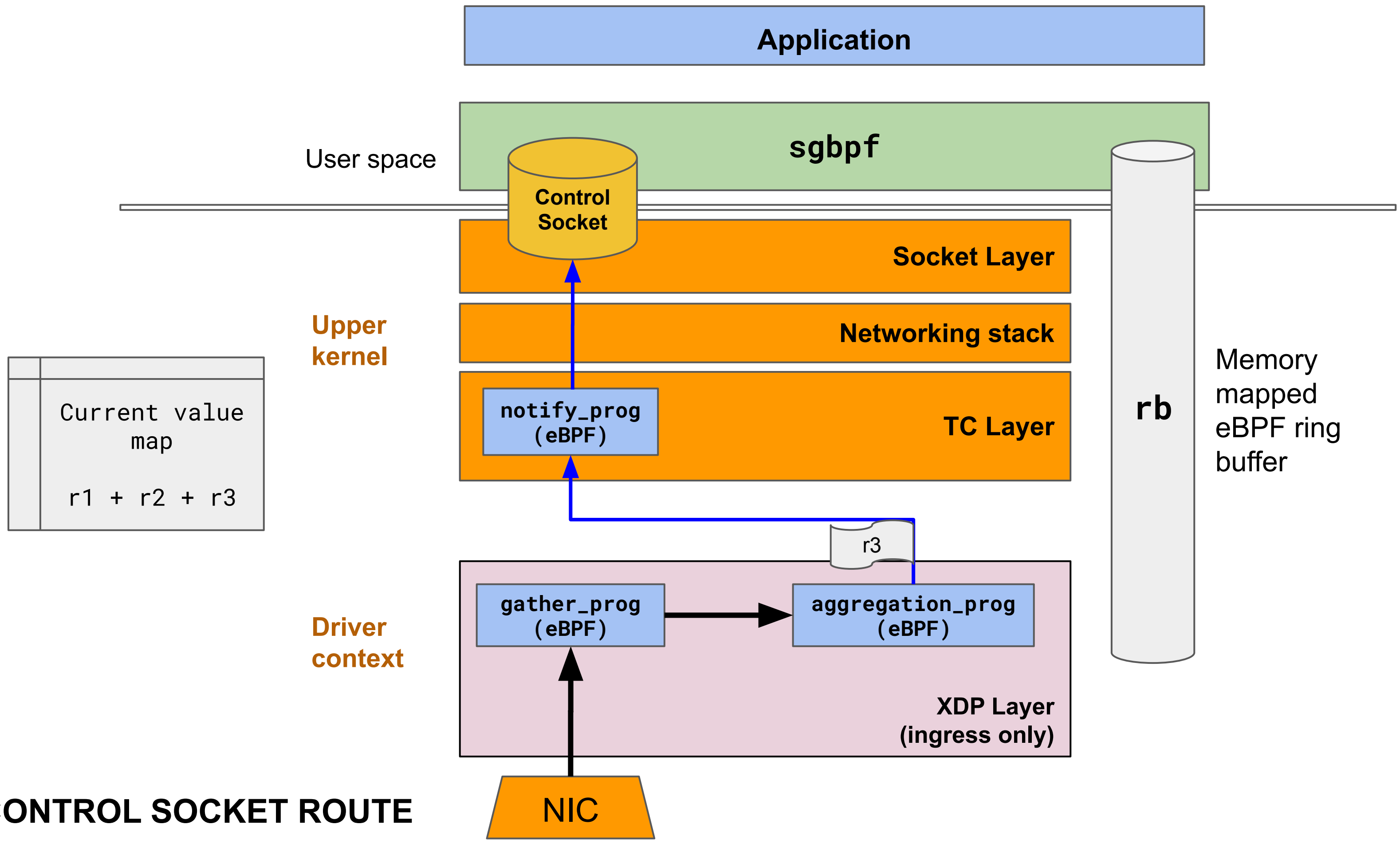


Ingress path with early dropping enabled

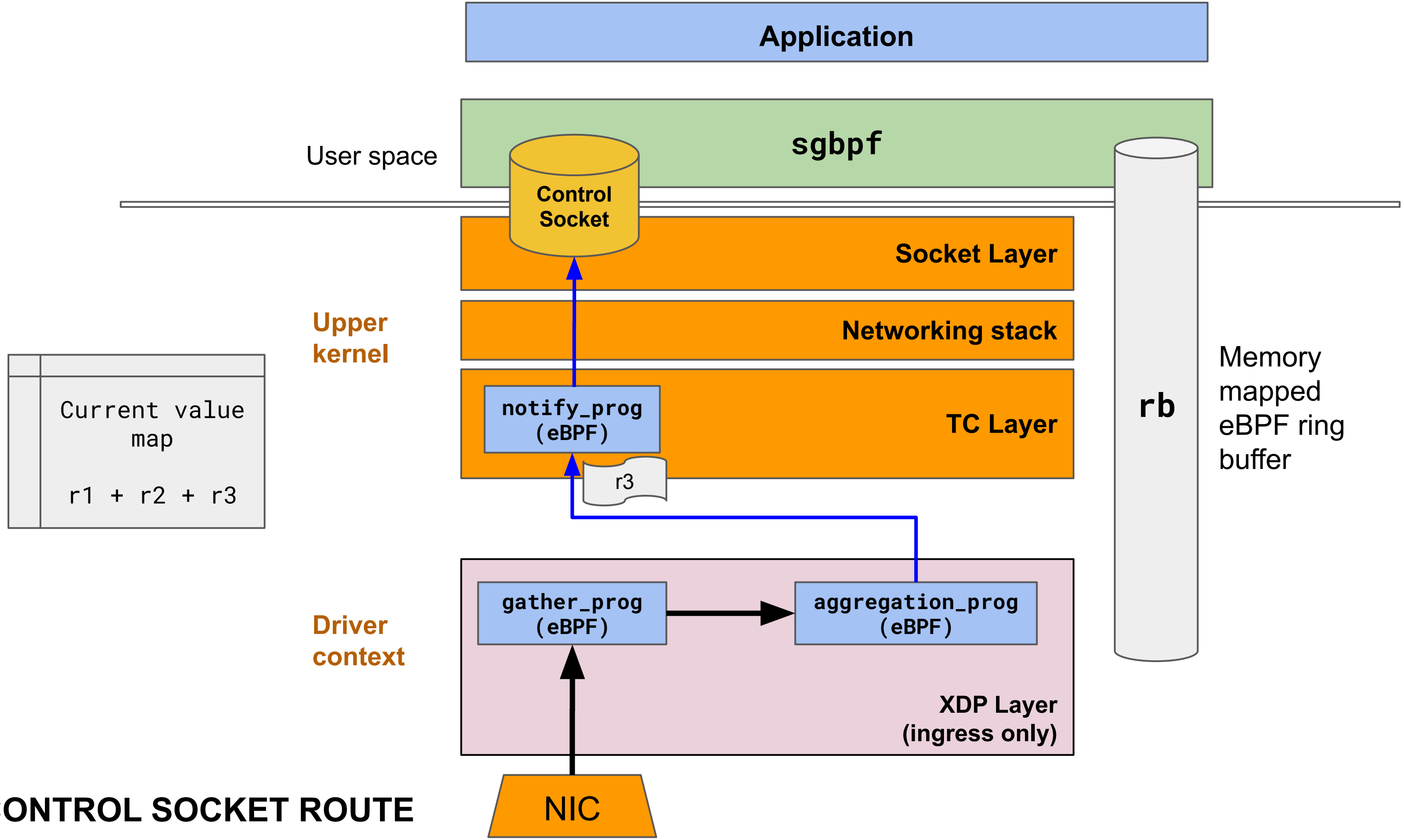


Ingress path with early dropping enabled

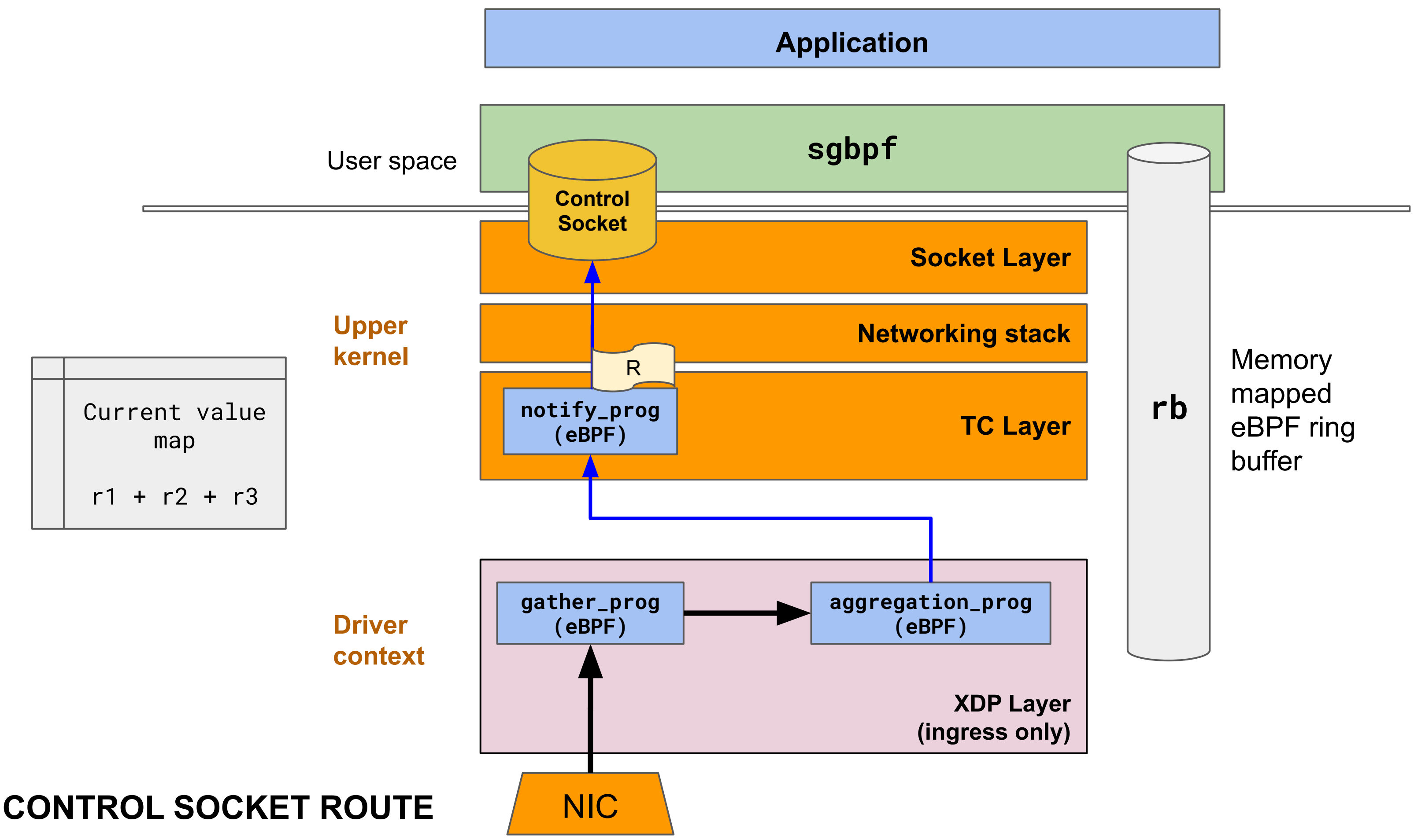

Application
sgbpf
User space
Control Socket
Socket Layer
Upper kernel
Networking stack
notify_prog (eBPF)
TC Layer
rb
Memory mapped eBPF ring buffer
Current value map
r1 + r2 + r3
Driver context
gather_prog (eBPF)
aggregation_prog (eBPF)
r3
XDP Layer (ingress only)
NIC
Ingress path with early dropping enabled

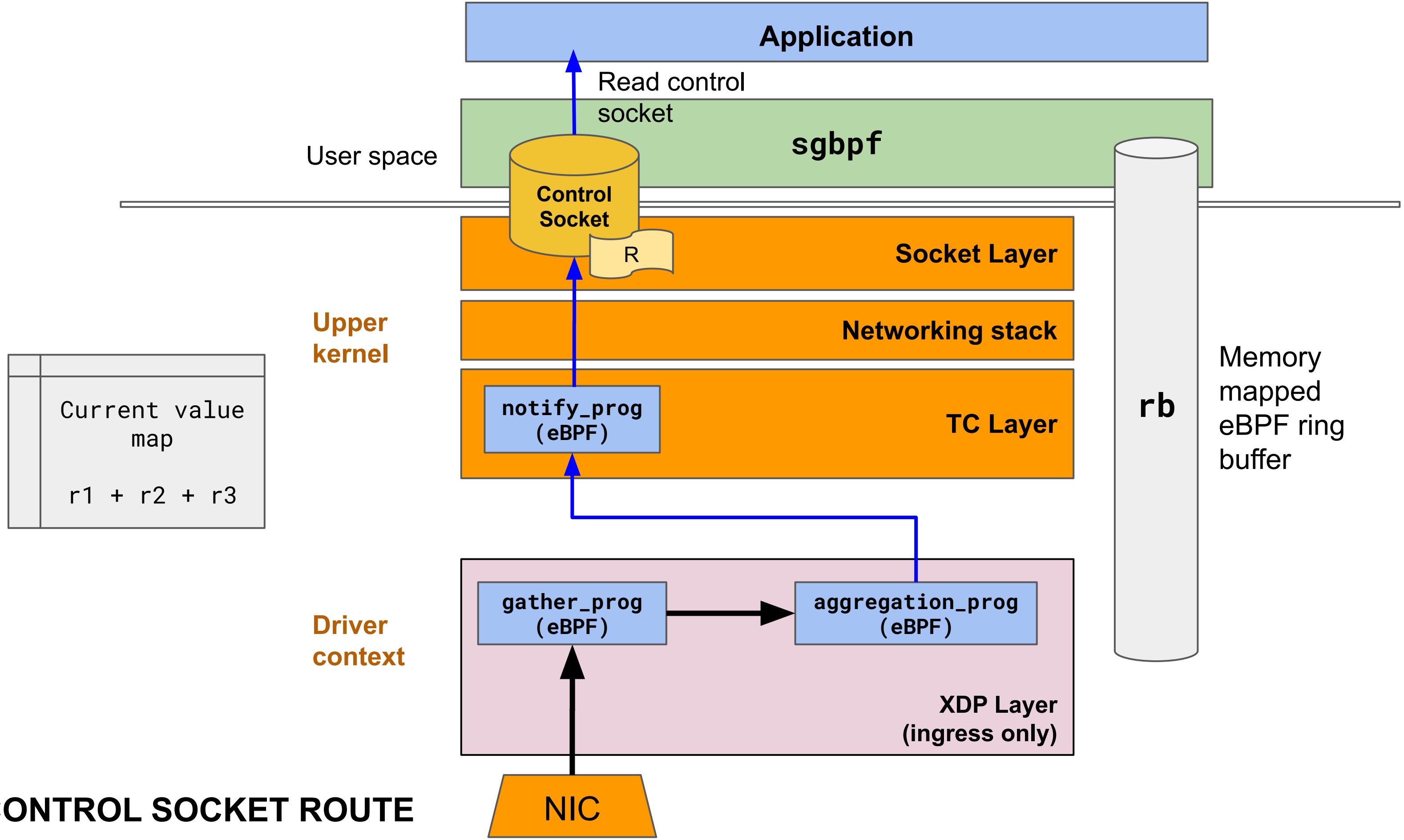


CONTROL SOCKET ROUTE

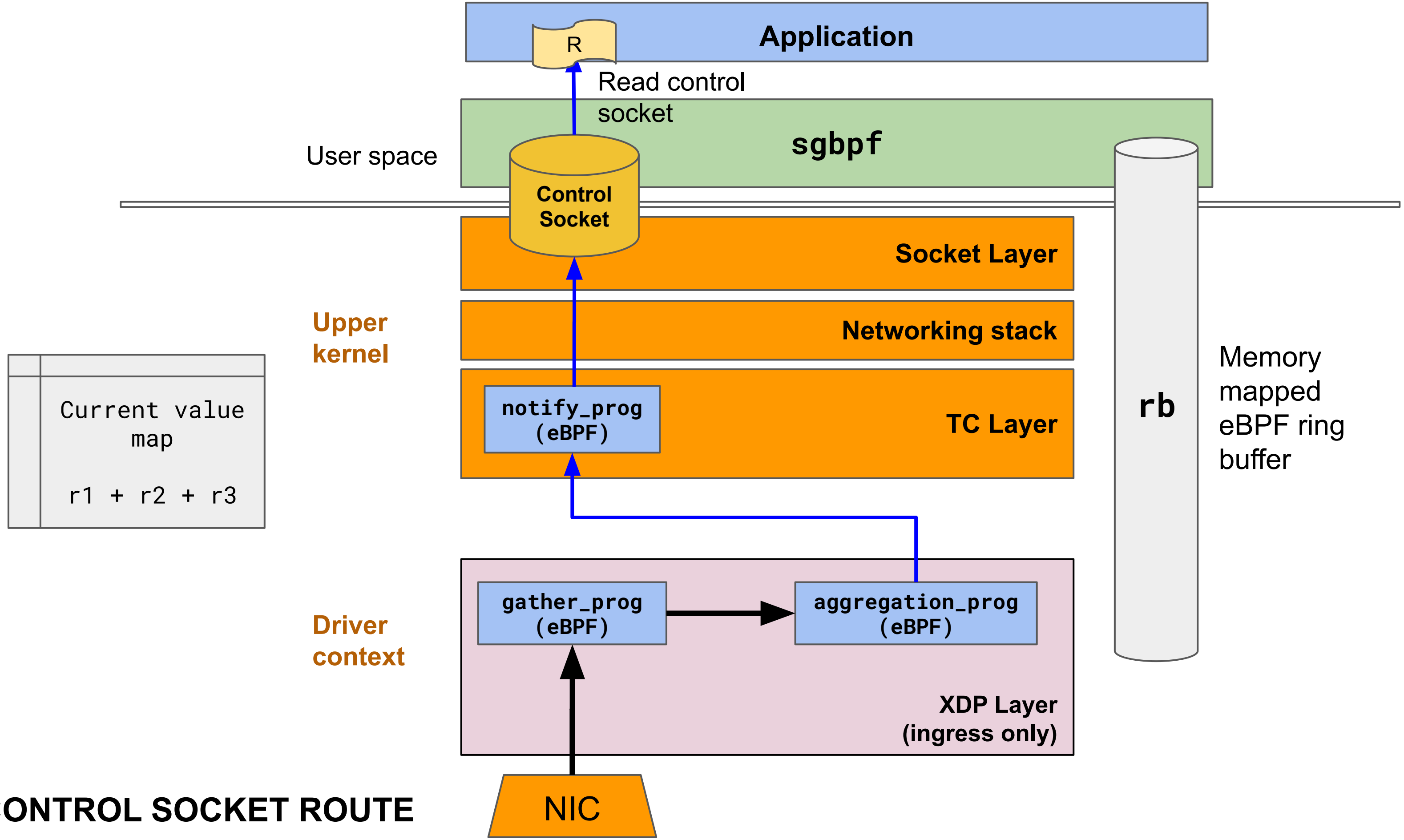


CONTROL SOCKET ROUTE

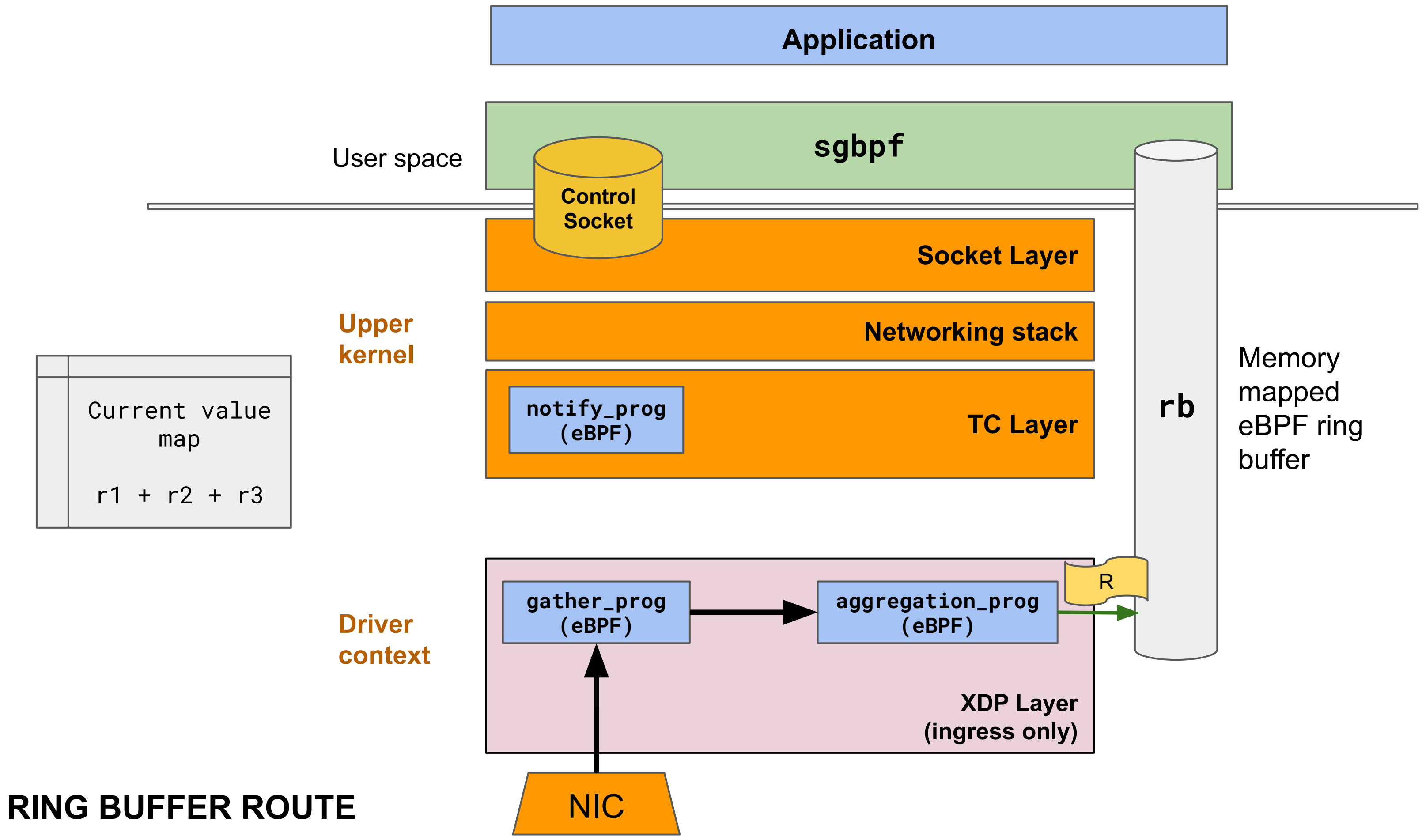

Application
sgbpf
User space
Control Socket
Socket Layer
Upper kernel
Networking stack
R
notify_prog (eBPF)
TC Layer
Current value map
r1 + r2 + r3
rb
Memory mapped eBPF ring buffer
Driver context
gather_prog (eBPF)
aggregation_prog (eBPF)
XDP Layer (ingress only)
CONTROL SOCKET ROUTE
NIC



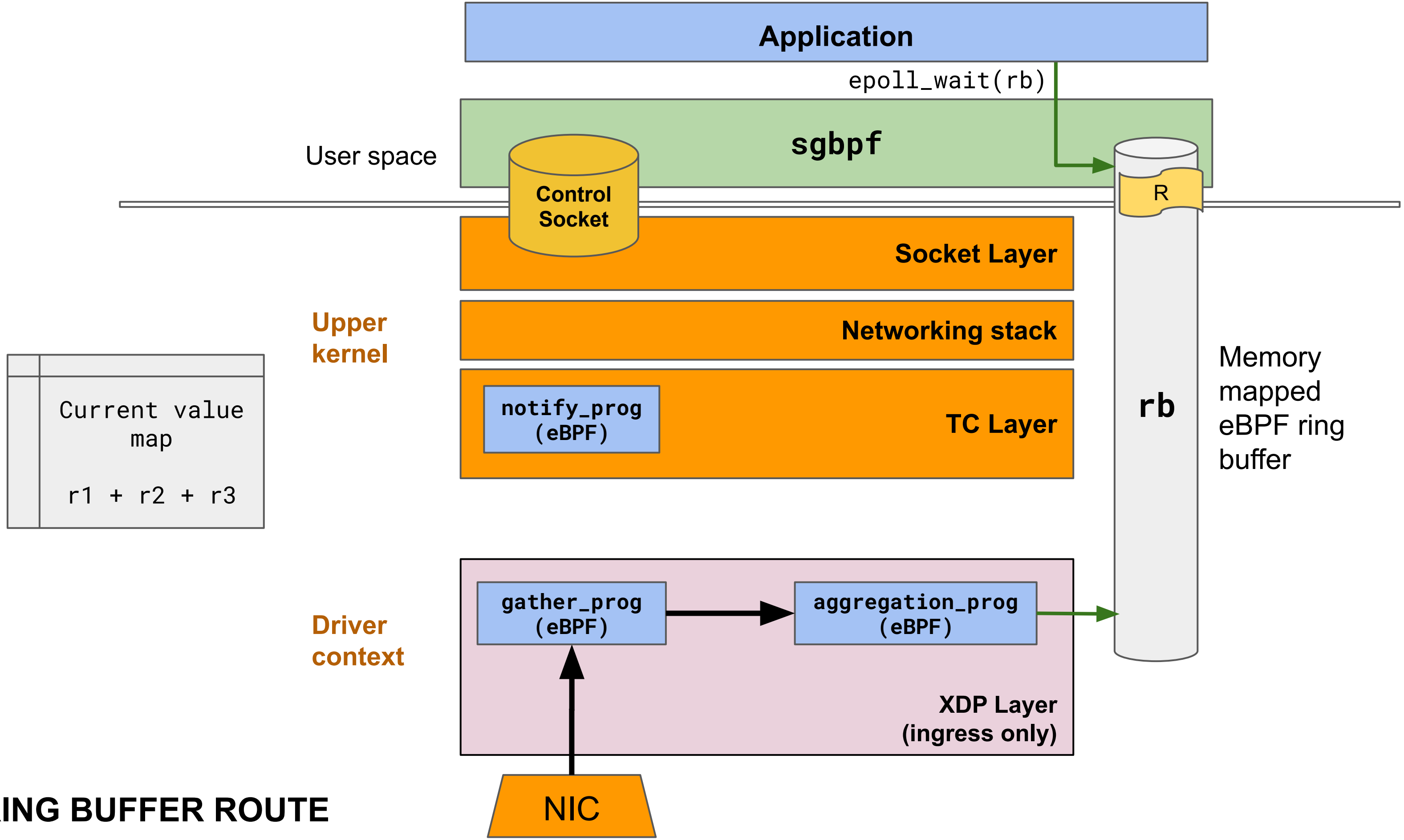
CONTROL SOCKET ROUTE



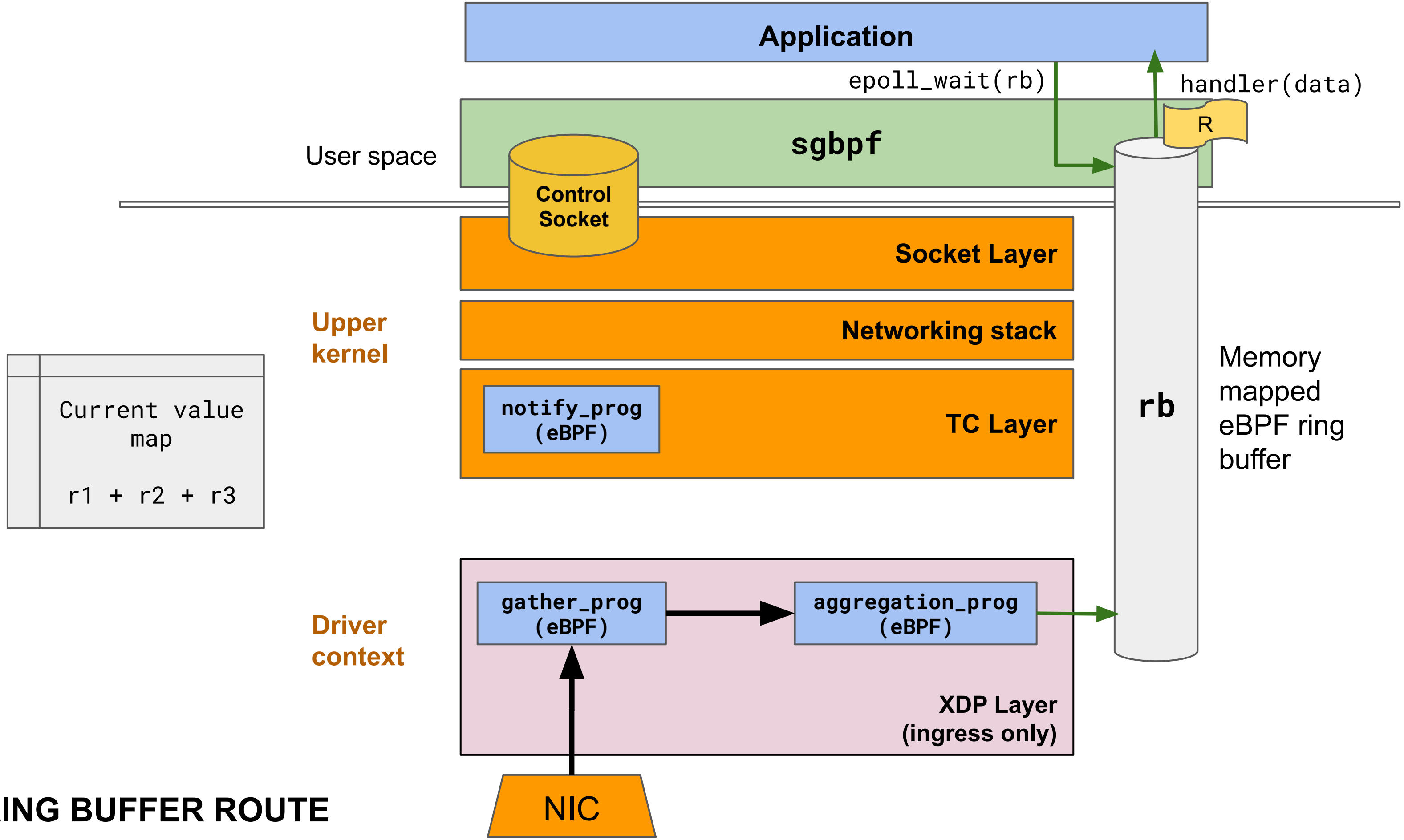
CONTROL SOCKET ROUTE



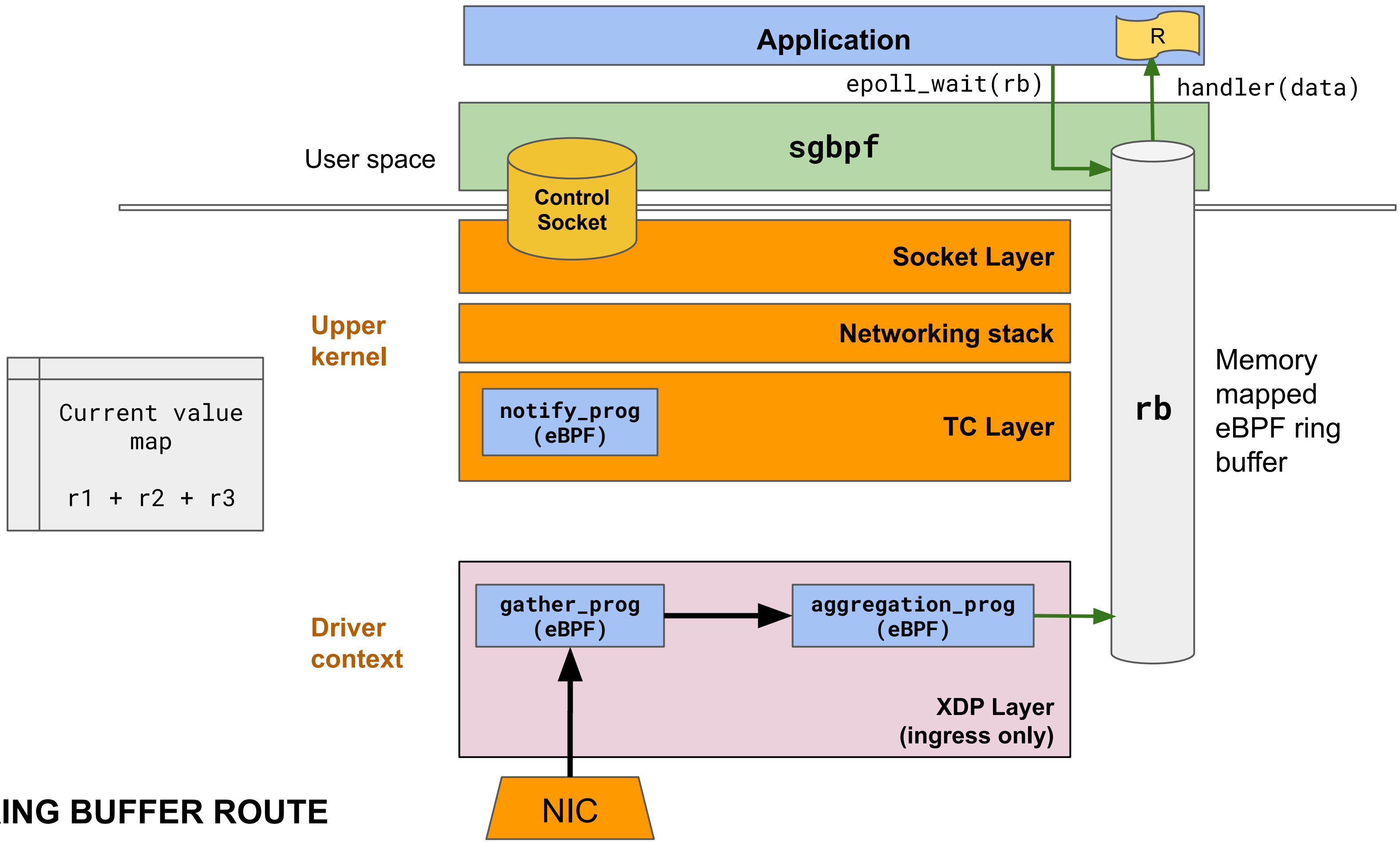
Application
sgbpf
User space
Control
Socket
Socket Layer
Networking stack
Upper
kernel
notify_prog
(eBPF)
TC Layer
rb
Memory
mapped
eBPF ring
buffer
Current value
map
r1 + r2 + r3
R
gather_prog
(eBPF)
aggregation_prog
(eBPF)
Driver
context
XDP Layer
(ingress only)
NIC
RING BUFFER ROUTE



RING BUFFER ROUTE



RING BUFFER ROUTE



RING BUFFER ROUTE

Example - custom aggregation program

```c
#include "bpf_h/helpers.bpf.h"

SEC("xdp")
int aggregation_prog(struct xdp_md* xdp_ctx) {
    struct aggregation_prog_ctx ctx;
    AGGREGATION_PROG_INTRO(ctx, xdp_ctx);

    AGGREGATION_PROG_ACQUIRE_LOCK(ctx);
    for (__u32 i = 0; i < RESP_MAX_VECTOR_SIZE; ++i) {
        ctx.current_value[i] += ((RESP_VECTOR_TYPE*) ctx.pk_msg->body)[i];
    }
    AGGREGATION_PROG_RELEASE_LOCK(ctx);

    AGGREGATION_PROG_OUTRO(ctx, DISCARD_PK);
}
```

Example - custom aggregation program

```c
#include "bpf_h/helpers.bpf.h"

SEC("xdp")
int aggregation_prog(struct xdp_md* xdp_ctx) {
    struct aggregation_prog_ctx ctx;
    AGGREGATION_PROG_INTRO(ctx, xdp_ctx);

    AGGREGATION_PROG_ACQUIRE_LOCK(ctx);
    for (__u32 i = 0; i < RESP_MAX_VECTOR_SIZE; ++i) {
        ctx.current_value[i] += ((RESP_VECTOR_TYPE*) ctx.pk_msg->body)[i];
    }
    AGGREGATION_PROG_RELEASE_LOCK(ctx);

    AGGREGATION_PROG_OUTRO(ctx, DISCARD_PK);
}
```

Helper macros to reduce boilerplate

Example - custom aggregation program

```c
#include "bpf_h/helpers.bpf.h"

SEC("xdp")
int aggregation_prog(struct xdp_md* xdp_ctx) {
    struct aggregation_prog_ctx ctx;
    AGGREGATION_PROG_INTRO(ctx, xdp_ctx);

    AGGREGATION_PROG_ACQUIRE_LOCK(ctx);
    for (__u32 i = 0; i < RESP_MAX_VECTOR_SIZE; ++i) {
        ctx.current_value[i] += ((RESP_VECTOR_TYPE*) ctx.pk_msg->body)[i];
    }
    AGGREGATION_PROG_RELEASE_LOCK(ctx);

    AGGREGATION_PROG_OUTRO(ctx, DISCARD_PK);
}
```

Aggregation takes place holding a per-request spinlock

Example - custom aggregation program

```
#include "bpf_h/helpers.bpf.h"

SEC("xdp")
int aggregation_prog(struct xdp_md* xdp_ctx) {
    struct aggregation_prog_ctx ctx;
    AGGREGATION_PROG_INTRO(ctx, xdp_ctx);

    AGGREGATION_PROG_ACQUIRE_LOCK(ctx);
    for (__u32 i = 0; i < RESP_MAX_VECTOR_SIZE; ++i) {
        ctx.current_value[i] += ((RESP_VECTOR_TYPE*) ctx.pk_msg->body)[i];
    }
    AGGREGATION_PROG_RELEASE_LOCK(ctx);

    AGGREGATION_PROG_OUTRO(ctx, DISCARD_PK);
}
```

Early dropping specified

Example - custom aggregation program

```c
#include "bpf_h/helpers.bpf.h"

SEC("xdp")
int aggregation_prog(struct xdp_md* xdp_ctx) {
    struct aggregation_prog_ctx ctx;
    AGGREGATION_PROG_INTRO(ctx, xdp_ctx);

    AGGREGATION_PROG_ACQUIRE_LOCK(ctx);
    for (__u32 i = 0; i < RESP_MAX_VECTOR_SIZE; ++i) {
        ctx.current_value[i] += ((RESP_VECTOR_TYPE*) ctx.pk_msg->body)[i];
    }
    AGGREGATION_PROG_RELEASE_LOCK(ctx);

    AGGREGATION_PROG_OUTRO(ctx, DISCARD_PK);
}
```

Provided by the developer as a C file

Compiled into a BPF object file which is loaded into the kernel at runtime

Example - user-space API to read the control socket

```
sgbpf::Service service{
    "path/to/bpjobjs",
    "lo",
    sgbpf::Worker::fromFile("workers.cfg"),
    sgbpf::PacketAction::Discard,          // packet fate post aggregation
    sgbpf::CtrlSockMode::DefaultUnix       // data delivery API
};

sgbpf::ReqParams params {
    .completionPolicy = sgbpf::GatherCompletionPolicy::WaitN,
    .numWorkersToWait = 5,
    .timeout          = std::chrono::microseconds{500}
}

sgbpf::Request* req = service.scatter(msg, msgLen, params);
```

Configure *sgbpf*

Example - user-space API to read the control socket

```
sgbpf::Service service{
    "path/to/bpjobjs",
    "lo",
    sgbpf::Worker::fromFile("workers.cfg"),
    sgbpf::PacketAction::Discard,          // packet fate post aggregation
    sgbpf::CtrlSockMode::DefaultUnix       // data delivery API
};

sgbpf::ReqParams params {
    .completionPolicy = sgbpf::GatherCompletionPolicy::WaitN,
    .numWorkersToWait = 5,
    .timeout          = std::chrono::microseconds{500}
}

sgbpf::Request* req = service.scatter(msg, msgLen, params);
```

Invoke a scatter-gather request

Example - user-space API to read the control socket

```
// Using sgbpf::CtrlSockMode::DefaultUnix
sg_msg_t buf;
read(service.ctrlSkFd(), &buf, sizeof(sg_msg_t));
handle(buf.body);
```

The application can use the raw Unix file descriptor of the control socket as it wishes (including configuring it as a non-blocking file)

Example - user-space API to read the control socket

```
// Using sgbpf::CtrlSockMode::Ringbuf
service.setRingbufCallback([&](char* data, int reqID) -> void {
    handle(data);
});

while (1) {
    int completions = service.epollRingbuf(EPOLL_TIMEOUT_MS);
    if (completions > 0)
        break;
}
```

The application can register a callback function which executes whenever aggregated data is written to the ring buffer (monitored by epoll)

Example - user-space API to read the control socket

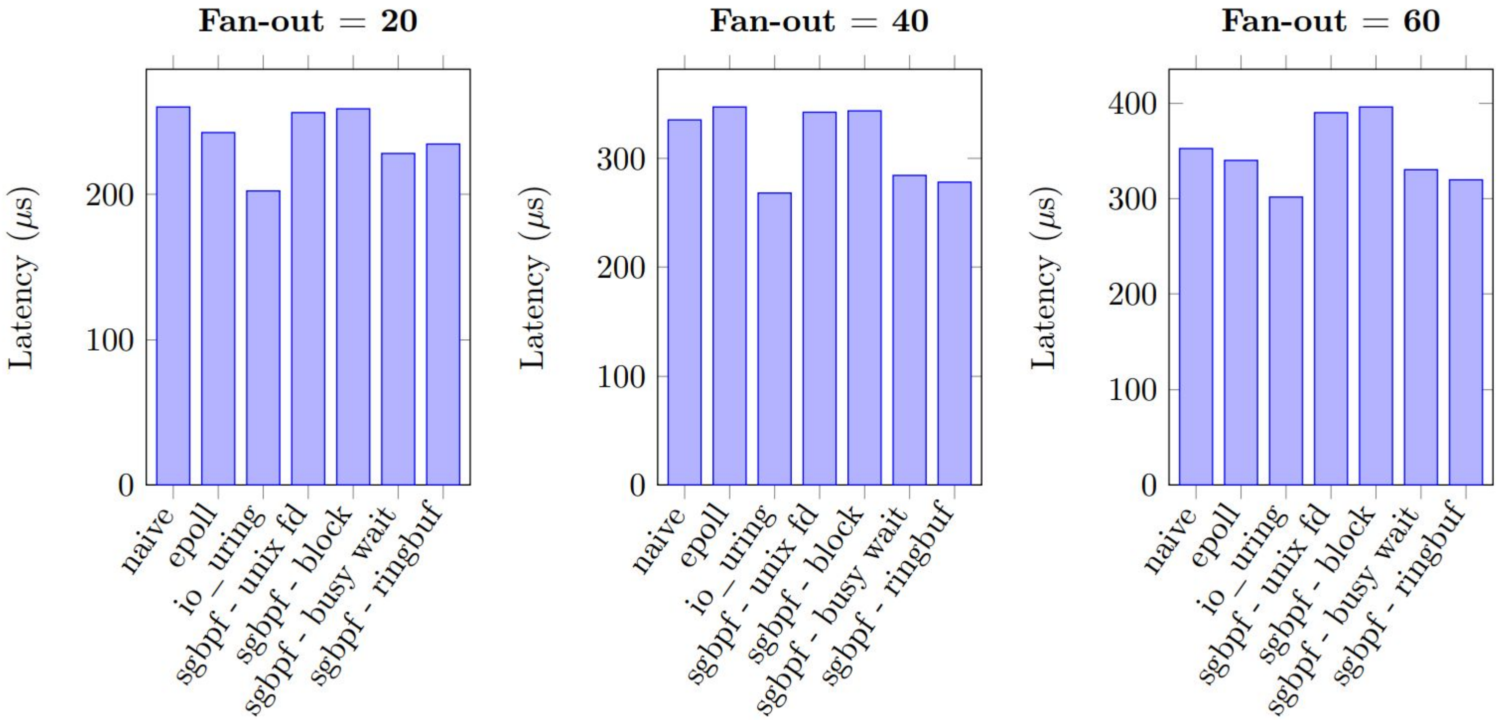
```
// Using sgbpf::CtrlSockMode::BusyWait
while (!req->isReady()) {
    service.processEvents(req->id());
}

handle(req->ctrlSockData());
```

The application can busy-wait on the control socket using the methods provided in *sgbpf* without incurring any extra system calls

Performance evaluation

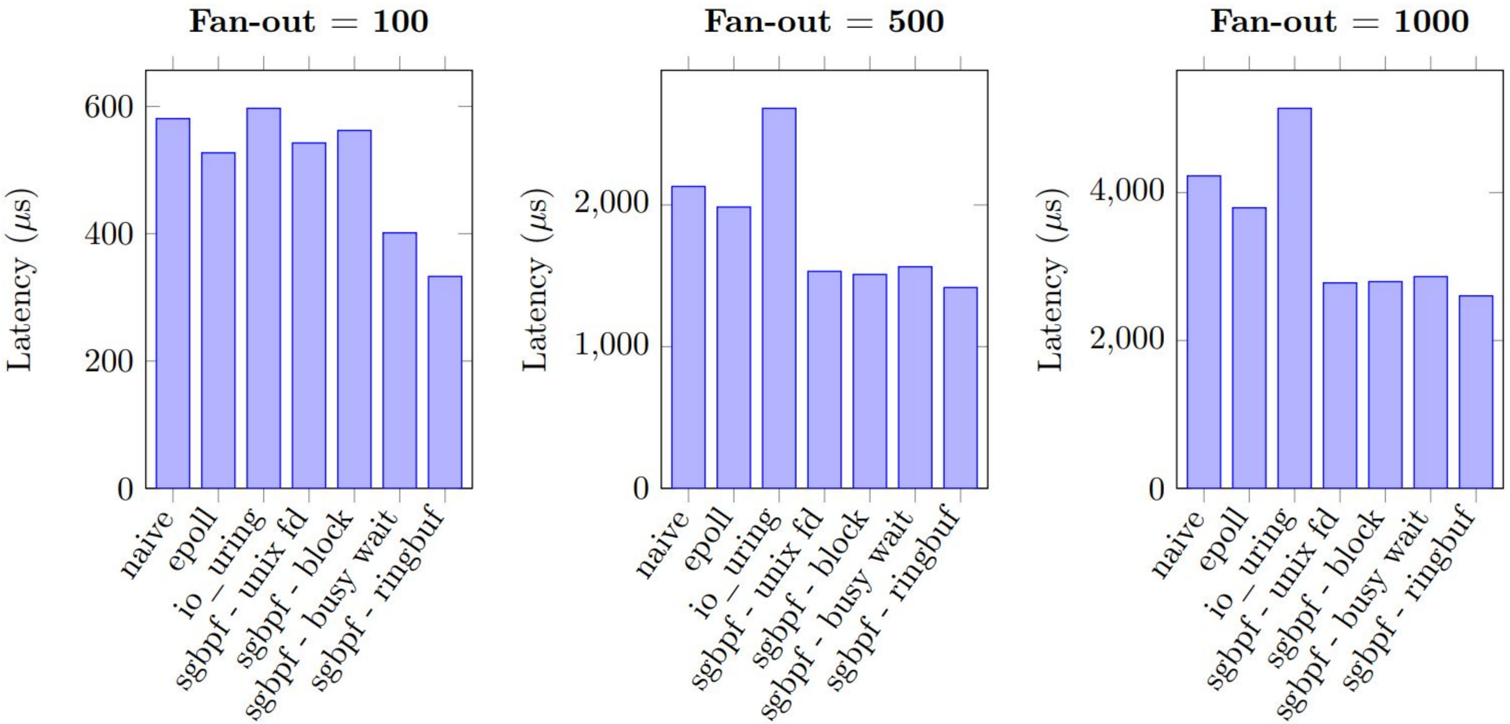
- Metrics of interest:
  - Unloaded latency
  - Throughput under load
- Compare performance with baseline implementations using standard Linux I/O APIs
  - Naive implementation
  - `epoll`-based implementation
  - `io_uring`-based implementation (with provided buffers)
- Results shown are obtained from local benchmarks
  - Workers are executed as local processes on a single machine

Performance evaluation - unloaded latency



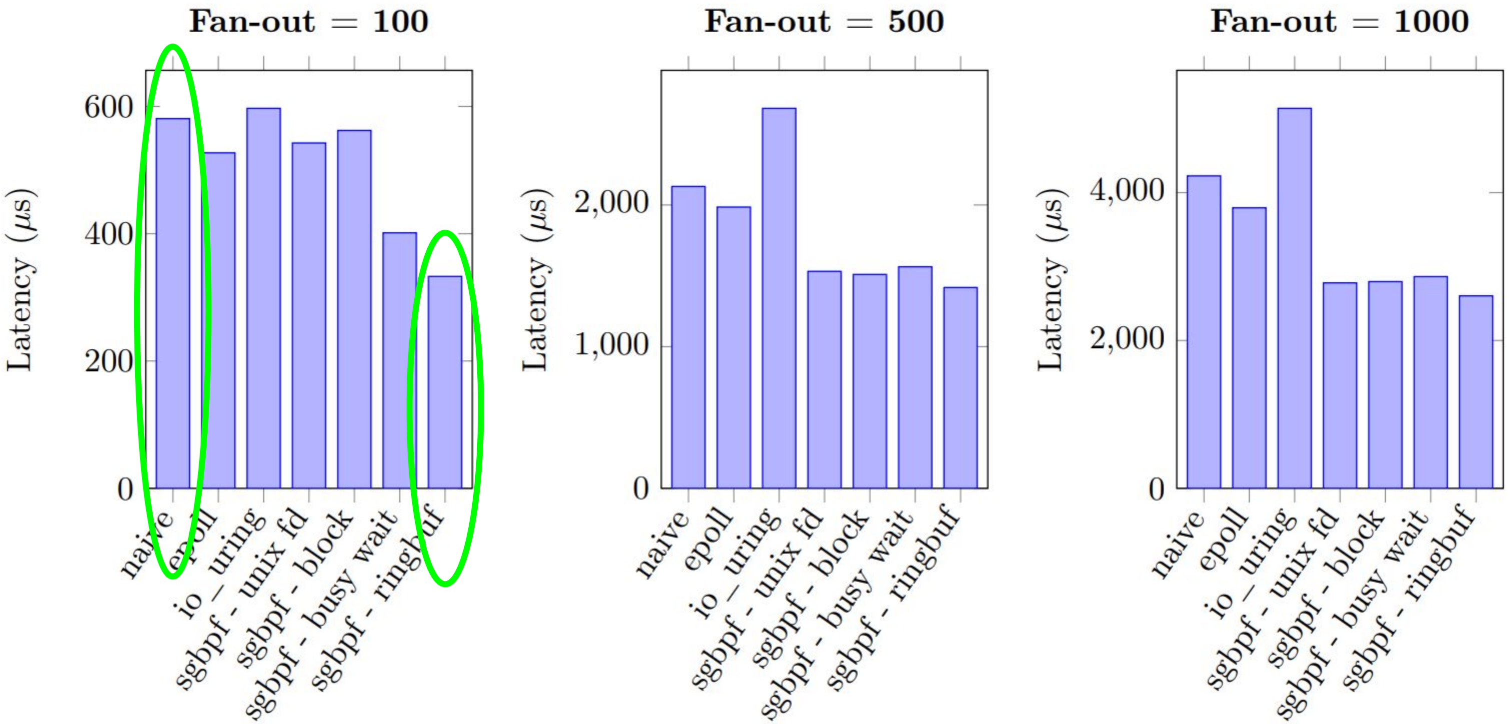
sgbpf achieves comparable latency results for small fan-outs

Performance evaluation - unloaded latency



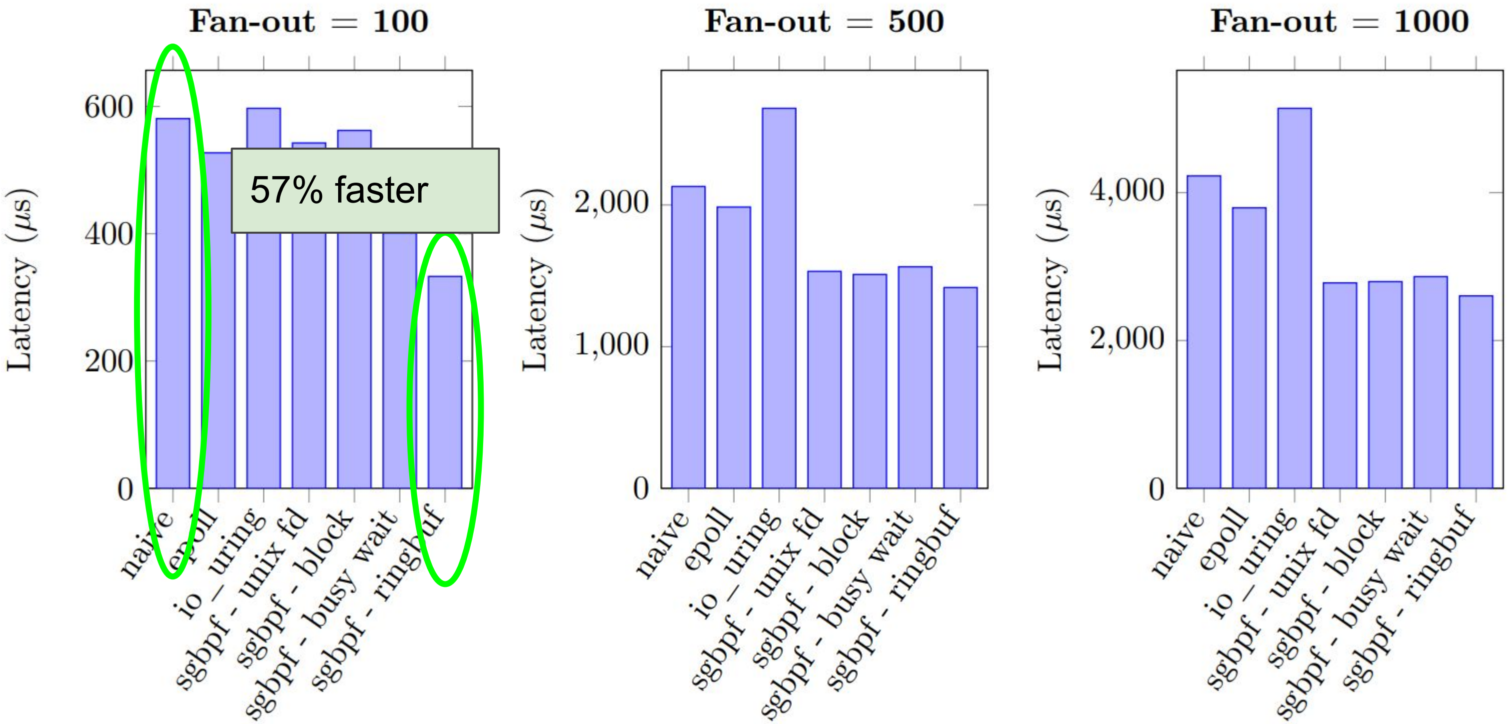
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency



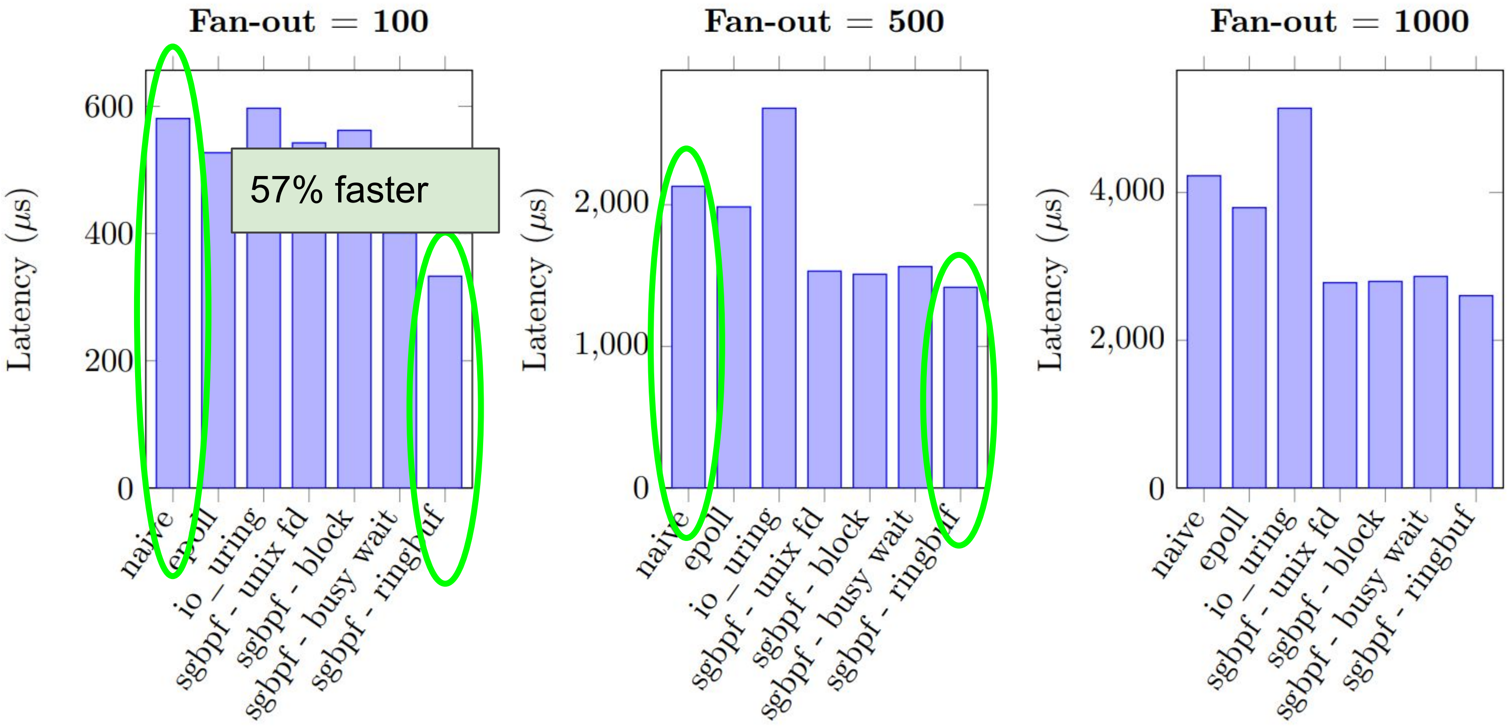
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency



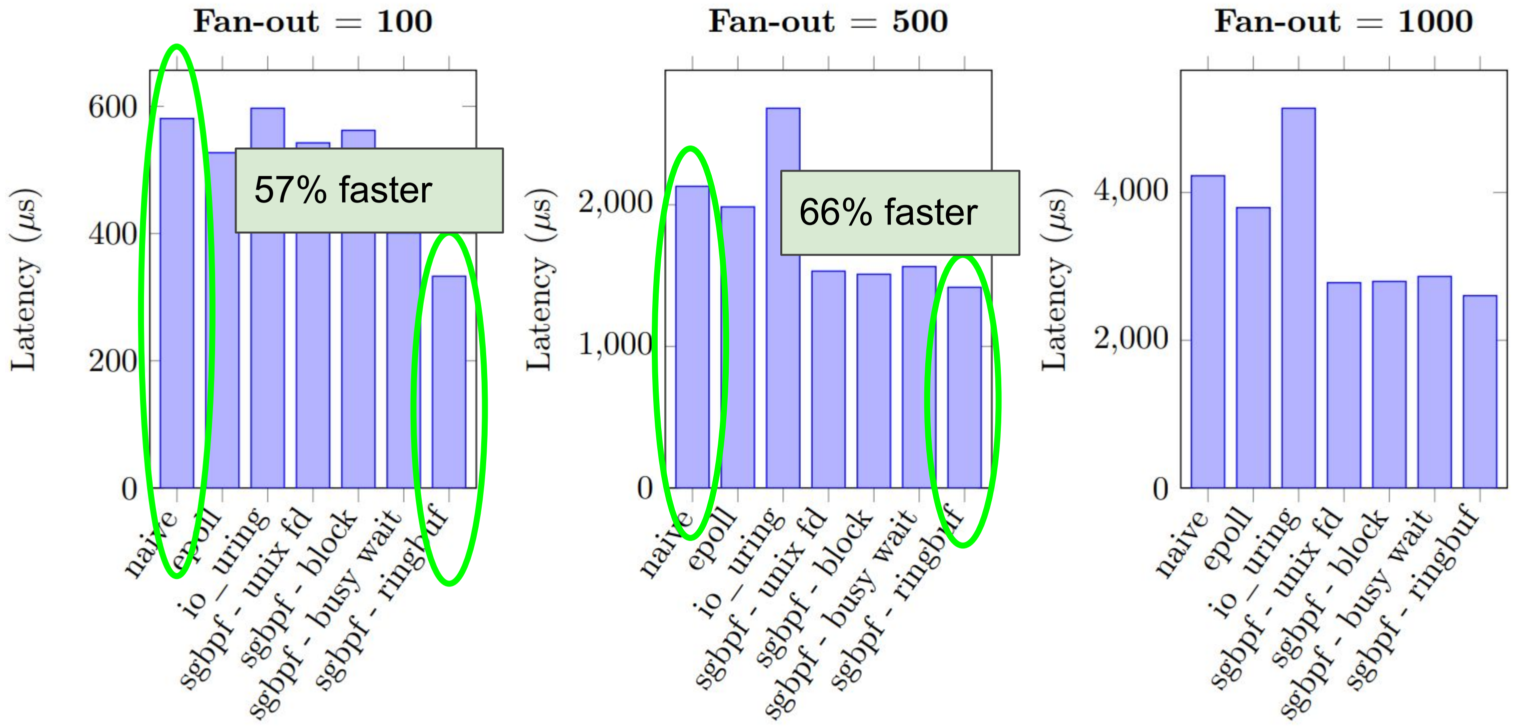
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency



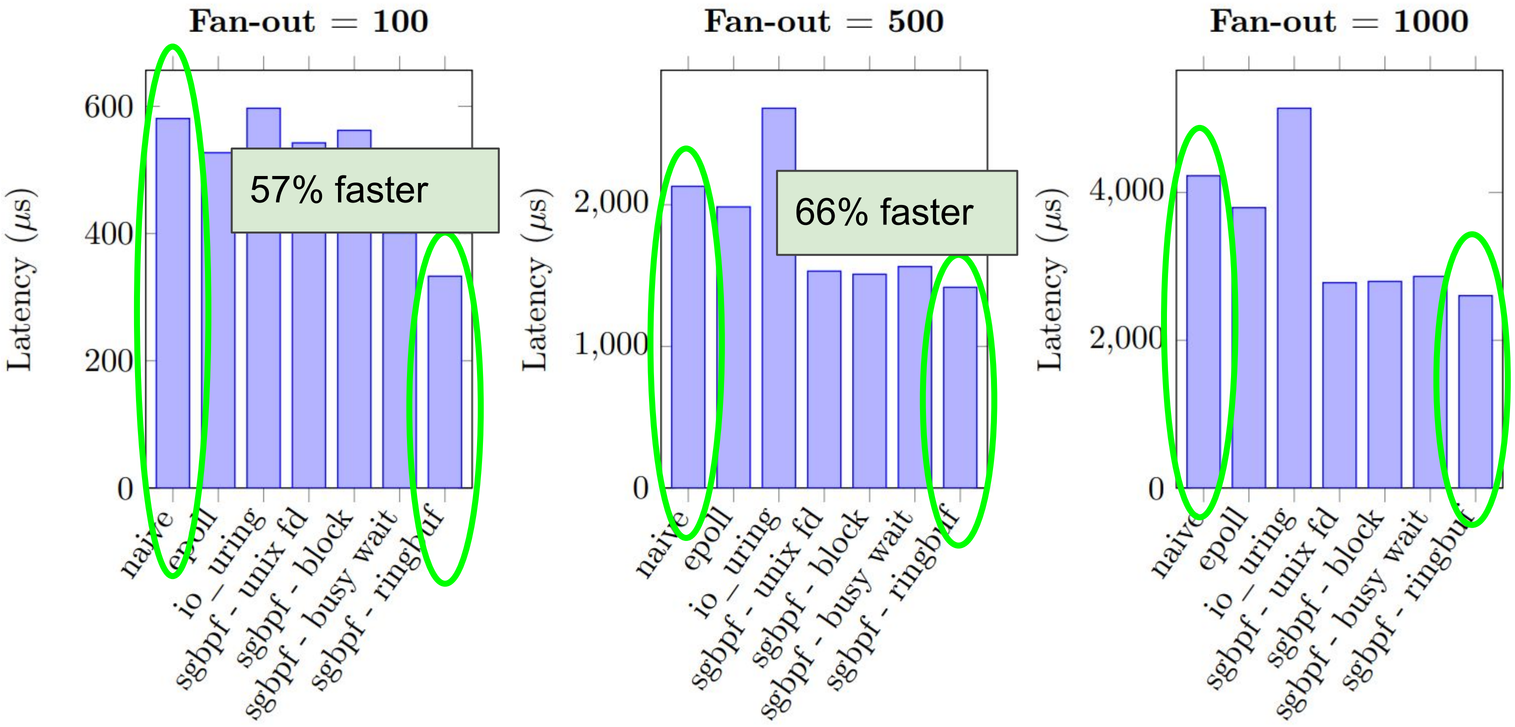
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency



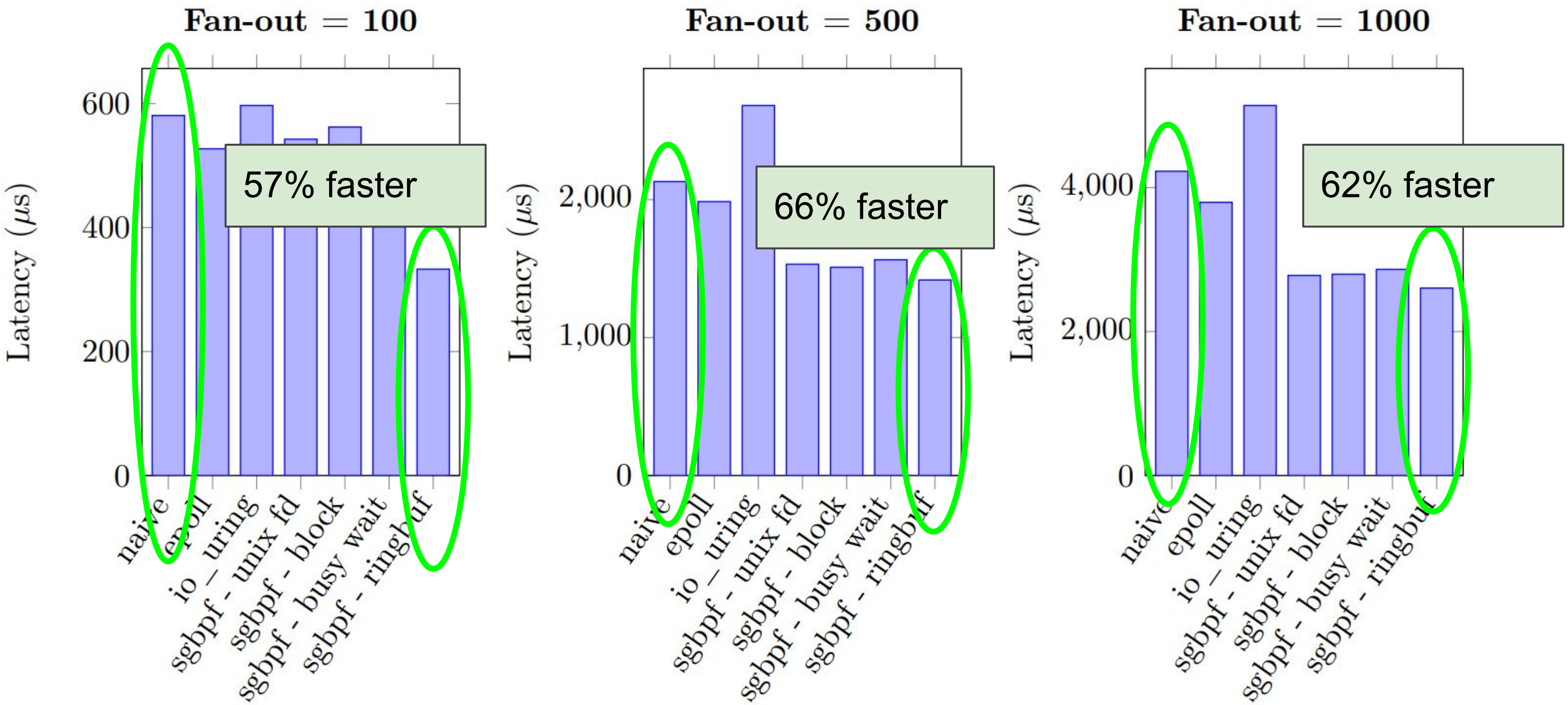
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency



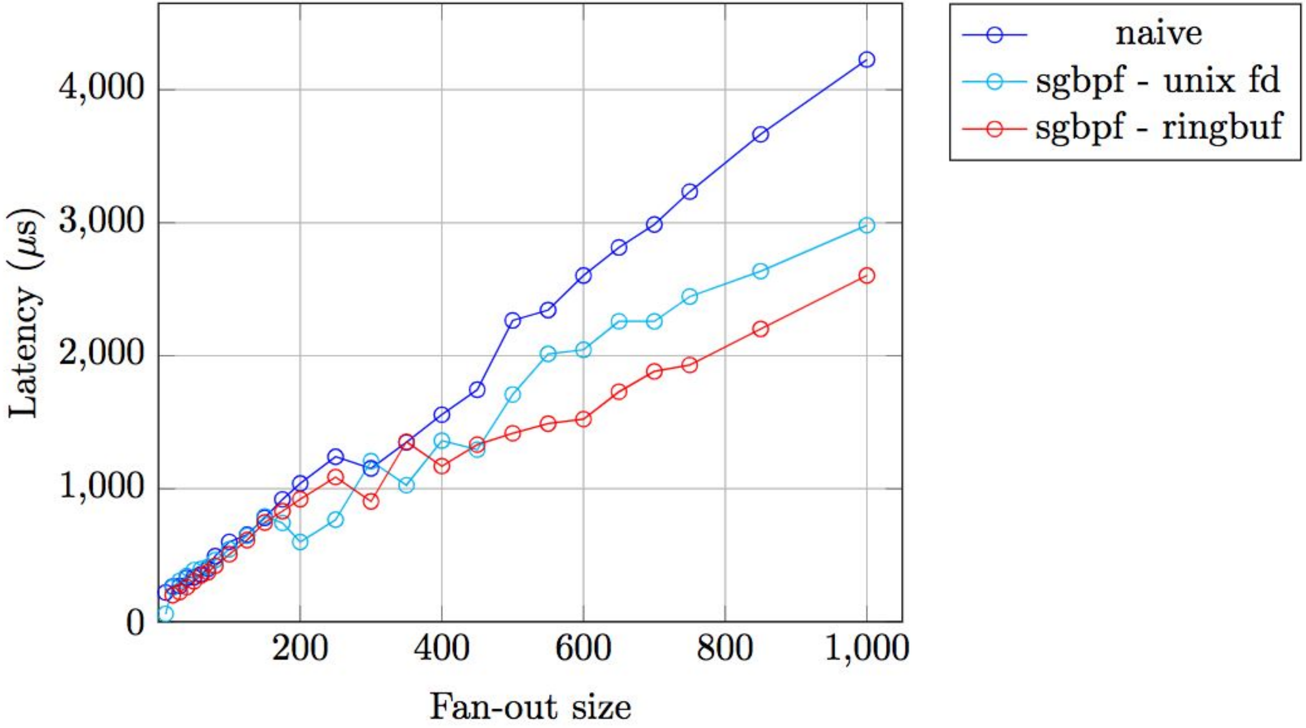
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency



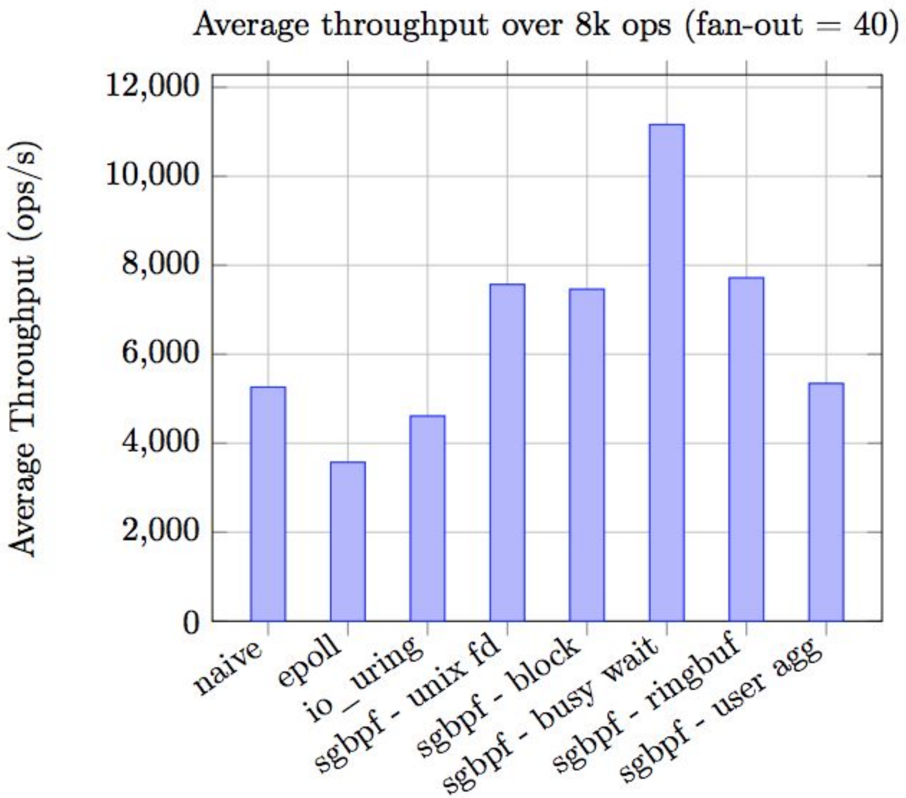
For larger fan-outs, *sgbpf* achieves substantial latency improvements

Performance evaluation - unloaded latency

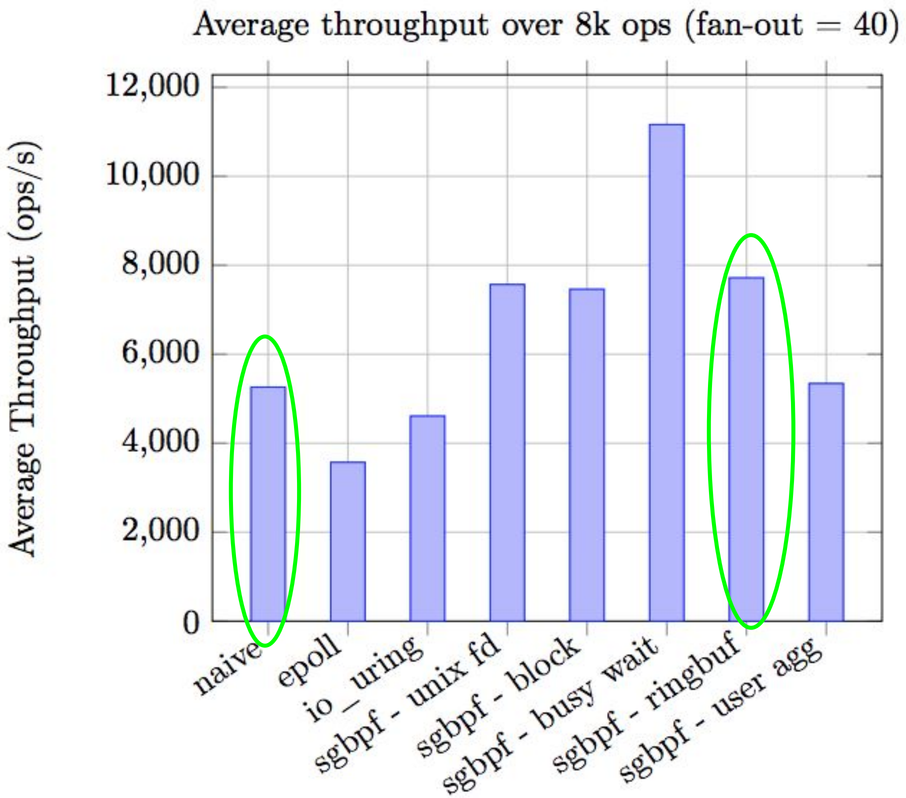


sgbpf scales better than standard I/O APIs due to total kernel overhead

Performance evaluation - throughput under load

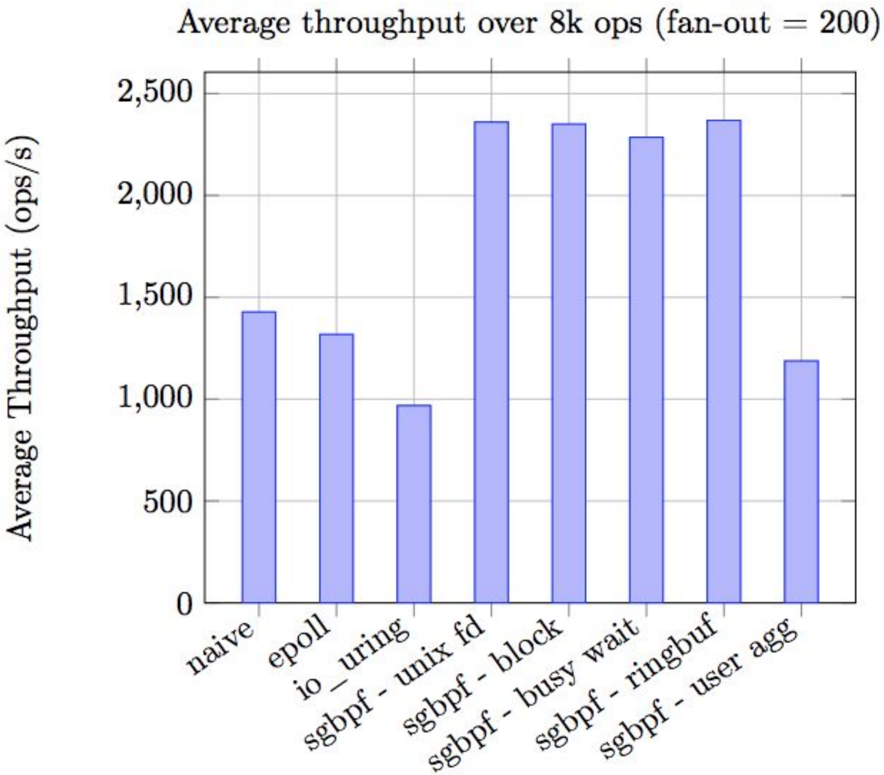


Performance evaluation - throughput under load

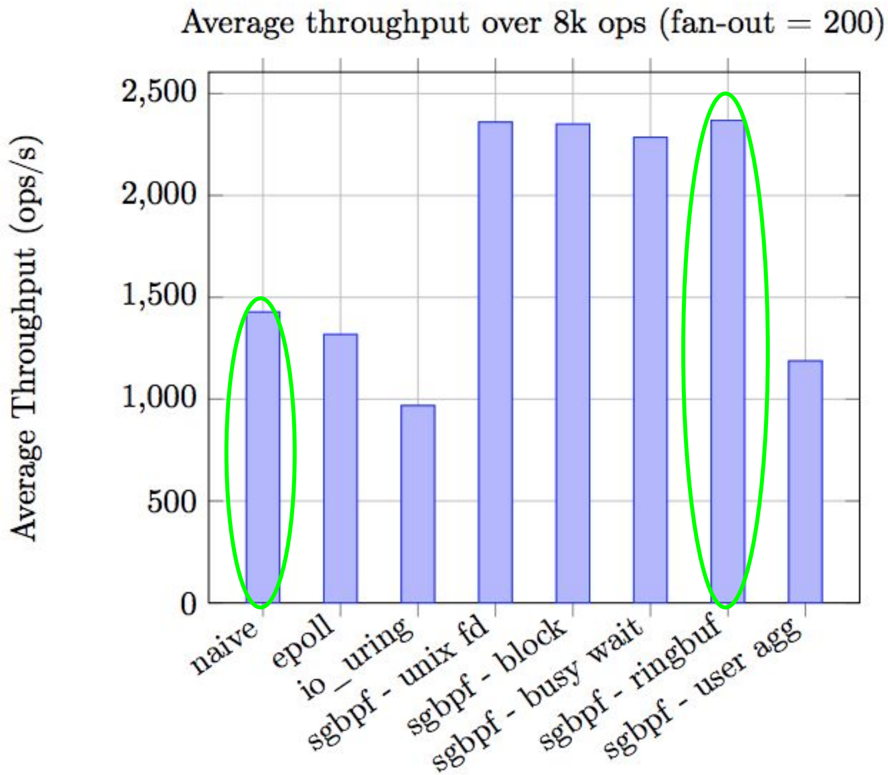


Even at small fan-outs, sgbpf achieves at least 46% better throughput

Performance evaluation - throughput under load



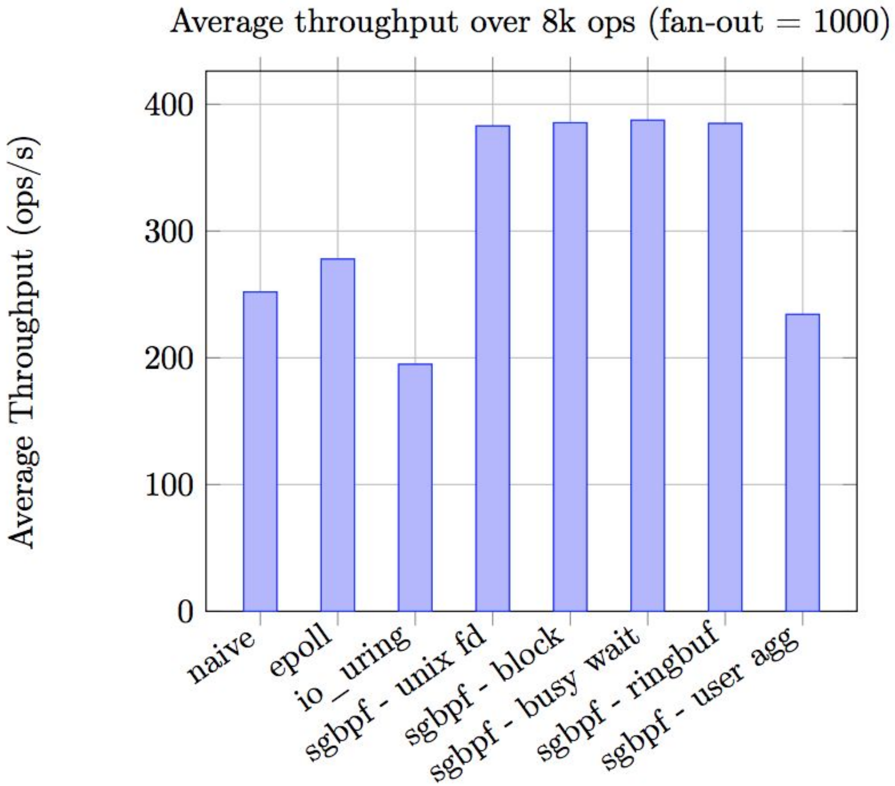
Performance evaluation - throughput under load



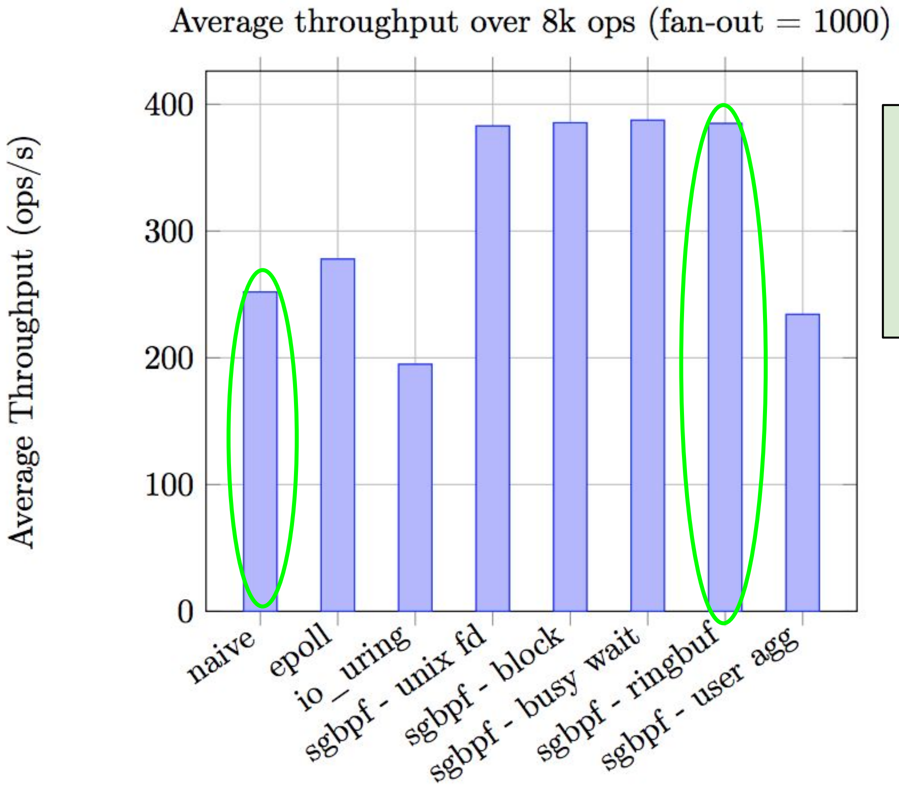
66% faster on average than naive version

As the fan-out increases, throughput difference starts to increase

Performance evaluation - throughput under load



Performance evaluation - throughput under load



53% faster on average than naive version

42% faster than epoll version (best standard baseline)

At extremely large fan-outs, the relative difference decreases slightly but still comfortably beats the standard APIs

Conclusions

- *sgbpf* reduces the kernel bottleneck in scatter-gather communications using eBPF
    - Minimises user-kernel crossings and kernel network stack traversals
    - Requires at most 2 syscalls (depends on API used)
    - Mainly achieved by performing aggregation as soon as possible inside the kernel

- While previous work shows eBPF can be used to accelerate specific applications, this project shows that generic high-level communication patterns can also incur performance benefits without resorting to extreme solutions.

- The code is open-sourced on Github

Future work

- Supporting in-kernel aggregation for multi-packet responses
- TCP implementation for responses
- Other optimisations and techniques for higher performance (see report)

Thank you for listening

sgbpf source code is available here on Github

References

[1] Dean J, Barroso LA. The tail at scale. Communications of the ACM. 2013 Feb 1;56(2):74-80.

[2] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), volume 3, page 4, 2021.

[3] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1391–1407, 2023.