Selective Packet Capture at High Speed Rates

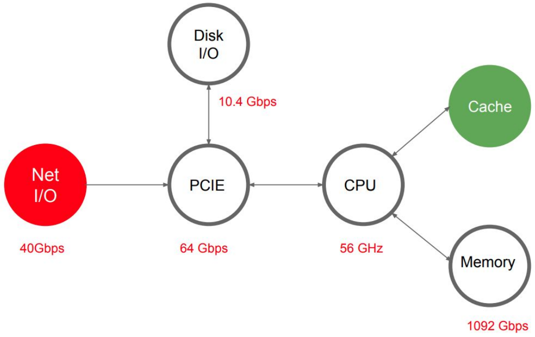
Reservoir Labs

Peter Cullen, James Ezick, Kelly Fox, Troy Hanson, Richard Lethin, Erik Mogus, Jordi Ros-Giralt, Alison Ryan,
{cullen, ezick, fox, hanson, lethin, mogus, giralt, ryan}@reservoir.com

Presenter: Jordi Ros-Giralt | giralt@reservoir.com

2nd European Zeek (Bro) Workshop

April 10, 2019



Reservoir Labs

632 Broadway
Suite 803
New York, NY 10012

Roadmap

- Selective Packet Capture: Problem definition
- Optimizations
  - Long queue emulation
  - Lockless bimodal queues
  - Tail early dropping
  - LFN tables
  - Multiresolution priority queues
- Zeek script

Packet Capturing at Very High Speed Rates

- Scalability issue: performing packet capture is either intractable or requires highly expensive hardware both in processing and storage.
- Liability issue: indiscriminate packet capture poses a liability issue.
- Selective Packet Capture (SPC) provides a sweet-spot solution to both of these problems.
- SPC gets a "free lunch" by leveraging all the heavy lifting work done by Zeek

Problem Definition

Capturing packets at very high speed rate is an HPC problem…

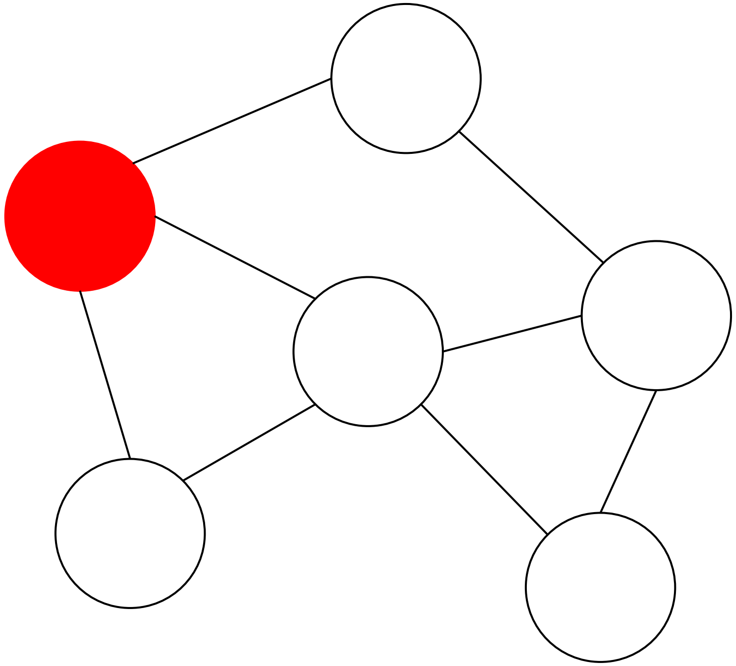
So let's talk first about performance optimization...

Problem Definition

- System wide performance optimization of network components like routers, firewalls, or network analyzers such as a Zeek sensor is complex.
- Hundreds of different SW algorithms and data structures interrelated in subtle ways.
- Two interdependent problems:
    - Shifting micro-bottlenecks
    - Nonlinear performance collapse
- Special focus on the problem of packet capturing at very high speed rates

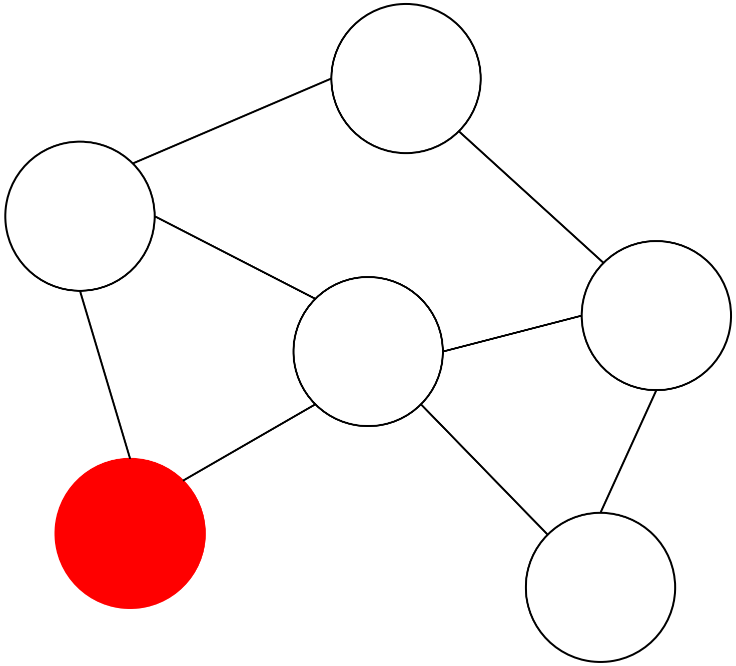
Shifting Micro-Bottlenecks

It's difficult...



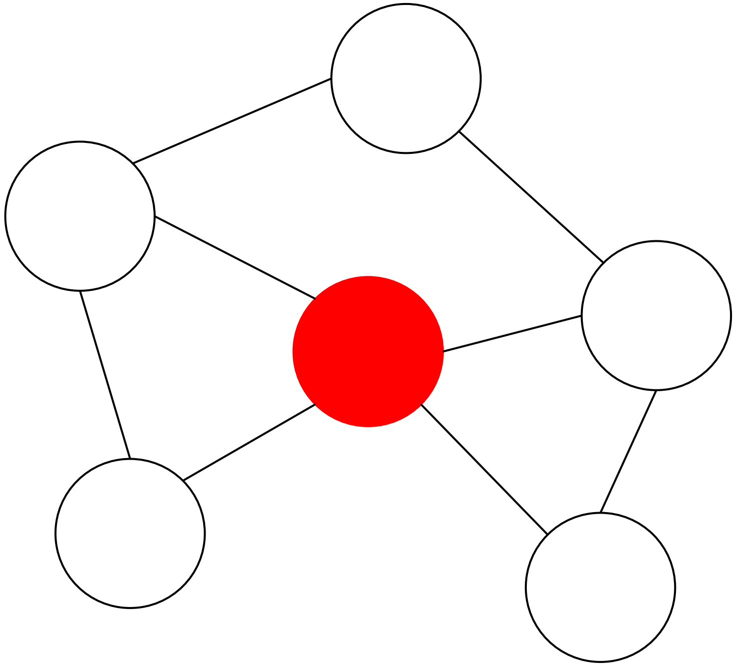
Shifting Micro-Bottlenecks

...to optimize...



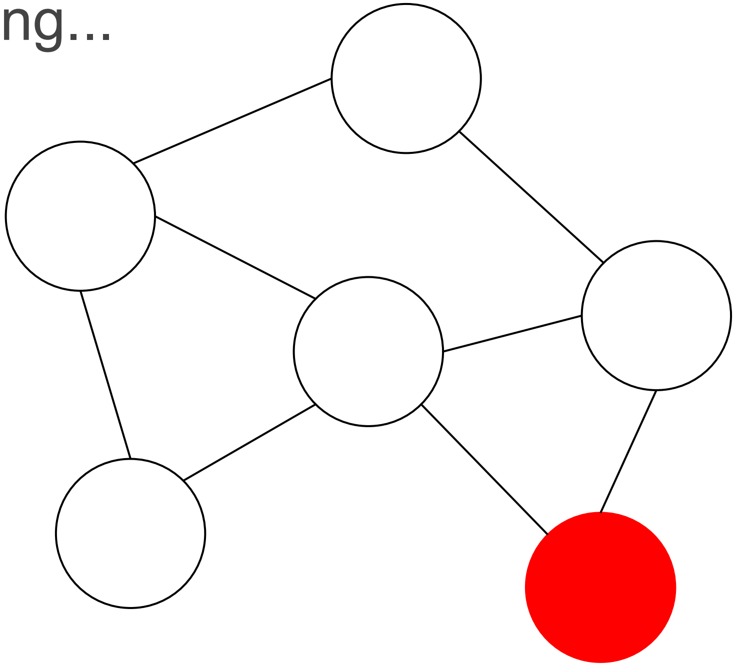
Shifting Micro-Bottlenecks

...bottlenecks...



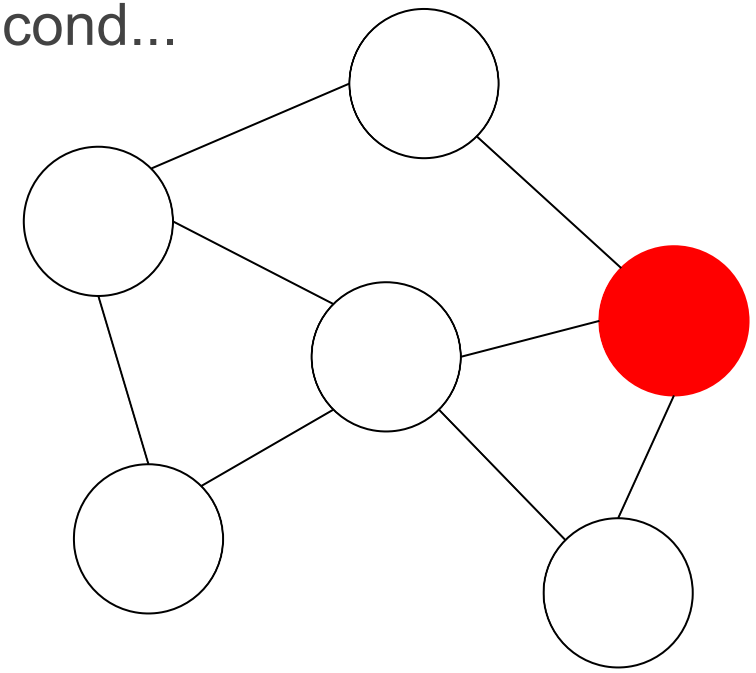
Shifting Micro-Bottlenecks

...that keep moving...



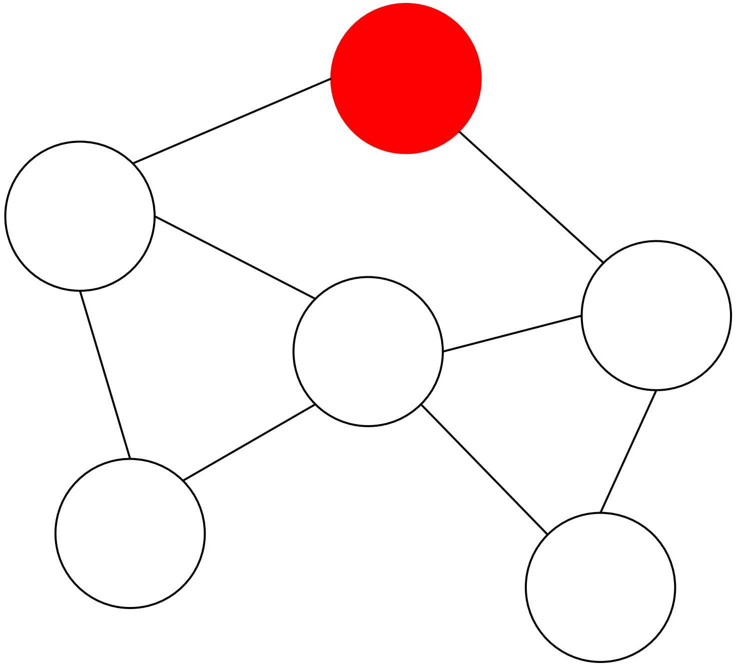
Shifting Micro-Bottlenecks

...every microsecond...

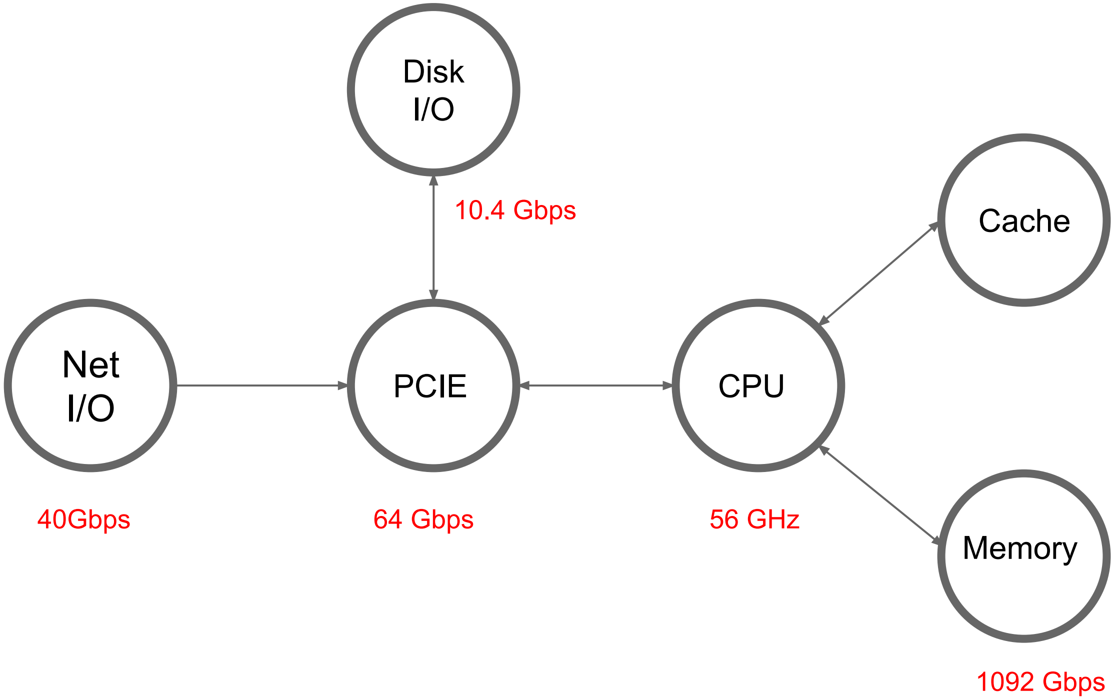


Shifting Micro-Bottlenecks

...or so.



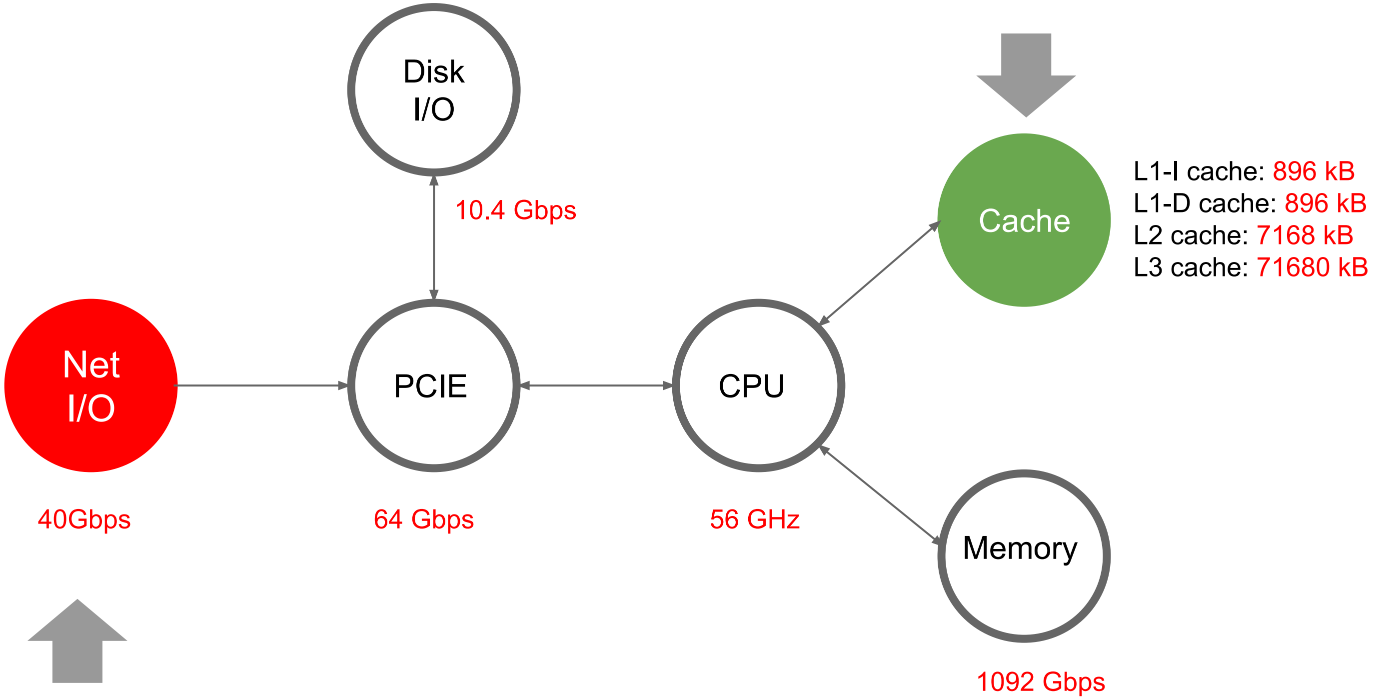
Non-linear Performance Collapse



L1-I cache: 896 kB
L1-D cache: 896 kB
L2 cache: 7168 kB
L3 cache: 71680 kB

Non-linear Performance Collapse

Healthy cache regime:
- CPU operates out of cache
- High cache hit ratios

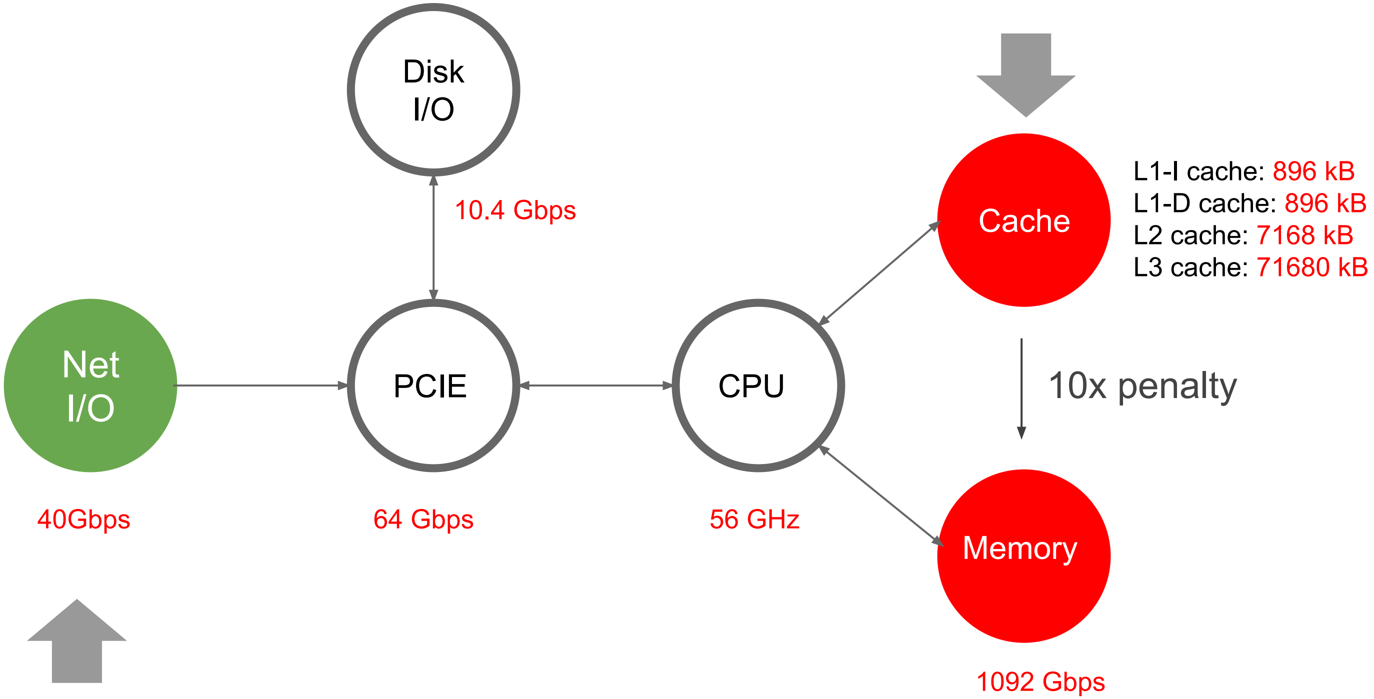


L1-I cache: 896 kB
L1-D cache: 896 kB
L2 cache: 7168 kB
L3 cache: 71680 kB

State 1: network is the bottleneck

Non-linear Performance Collapse

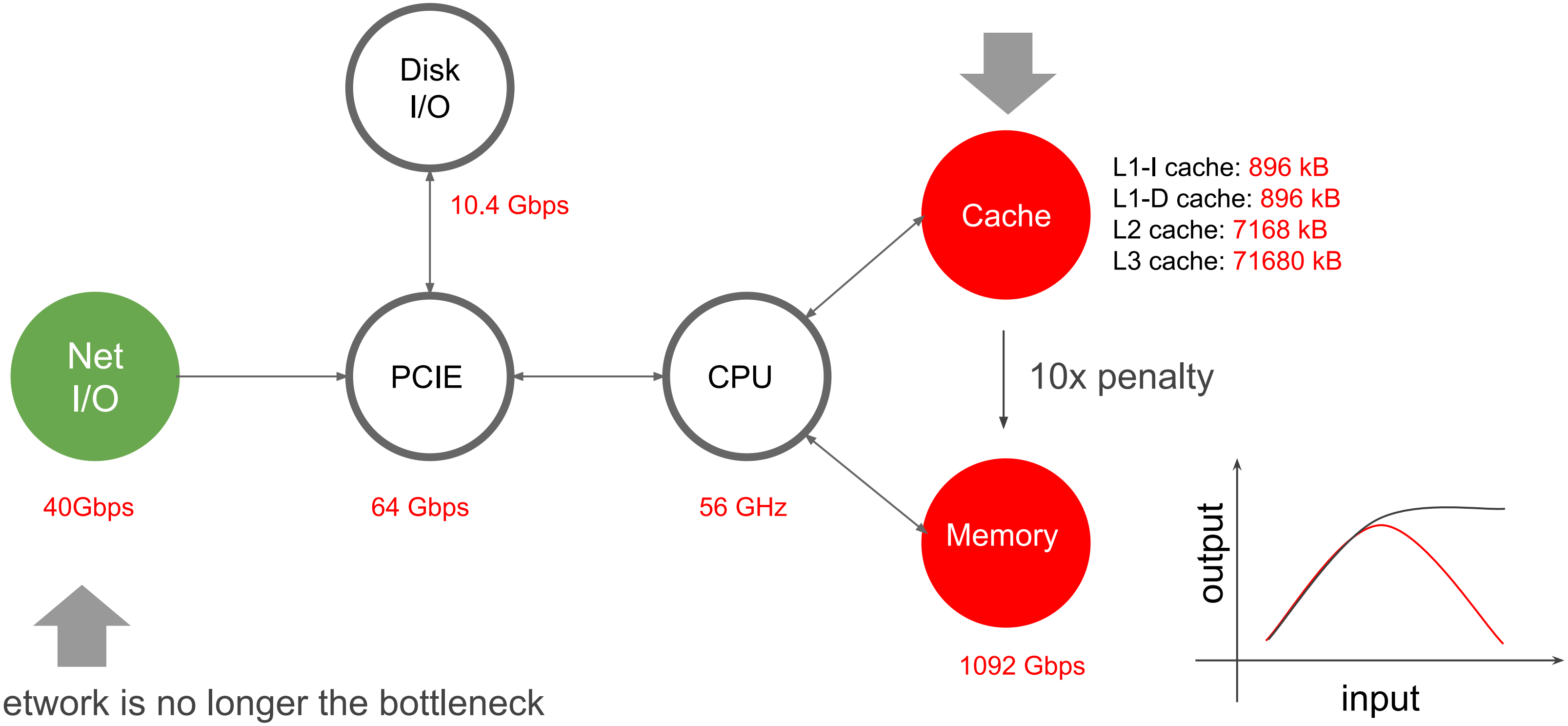
Highly inefficient memory regime:
- CPU operates out of RAM
- High cache miss ratios



State 2: network is no longer the bottleneck

Non-linear Performance Collapse

Highly inefficient memory regime:
- CPU operates out of RAM
- High cache miss ratios


Disk I/O

10.4 Gbps

Net I/O

40Gbps

PCIE

64 Gbps

CPU

56 GHz

Cache

L1-I cache: 896 kB
L1-D cache: 896 kB
L2 cache: 7168 kB
L3 cache: 71680 kB

10x penalty

Memory

1092 Gbps

State 2: network is no longer the bottleneck

By removing the network bottleneck, system spends more time processing packets that will need to be dropped anyway → net performance degradation (performance collapse)

Performance Optimization: Algorithms and Data Structures

Long queue emulation	Reduces packet drops due to fixed-size hardware rings
Lockless bimodal queues	Improves packet capturing performance
Tail early dropping (TED)	Increases information entropy and extracted metadata
LFN tables	Reduces state sharing overhead
Multiresolution priority queues	Reduces cost of processing timers

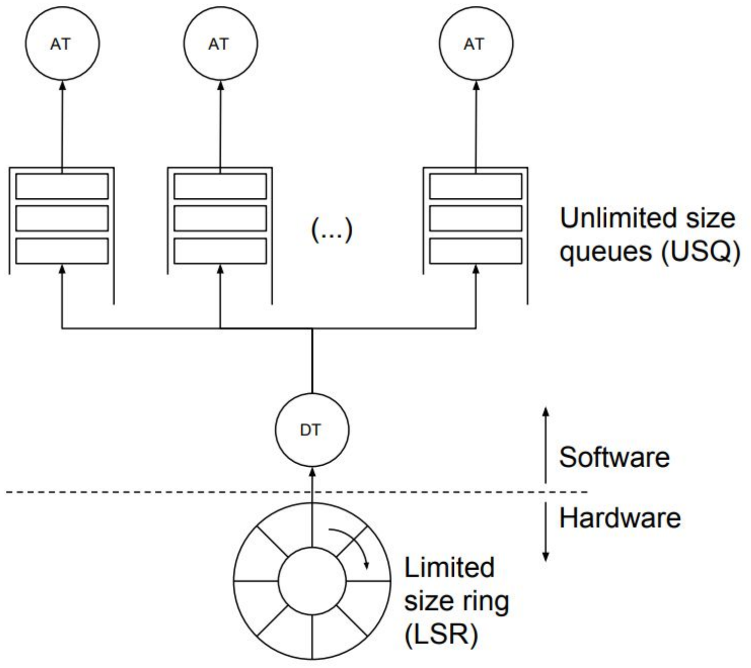
Patented technologies

Performance Optimization: Algorithms and Data Structures

Long queue emulation	Reduces packet drops due to fixed-size hardware rings
Lockless bimodal queues	Improves packet capturing performance
Tail early dropping (TED)	Increases information entropy and extracted metadata
LFN tables	Reduces state sharing overhead
Multiresolution priority queues	Reduces cost of processing timers

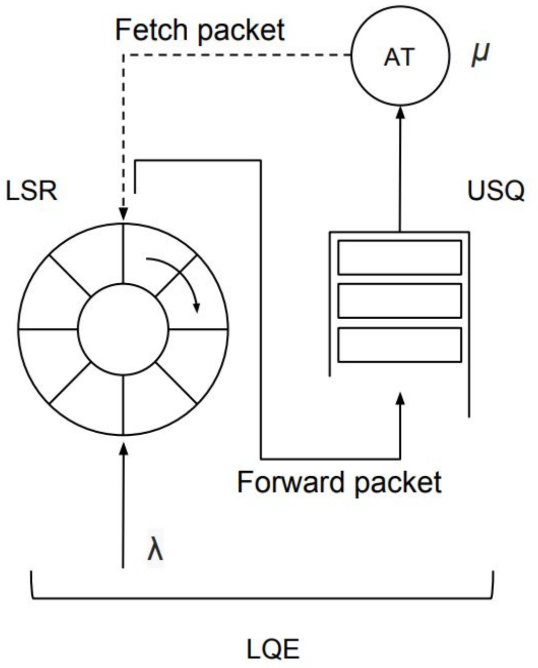
Long Queue Emulation

Dispatcher Model:



- Packet read cache penalty.
- Descriptor read cache penalty

Long queue emulation Model:



- Packet drop penalty under certain conditions

Long Queue Emulation: Operational Lemma

Lemma 1. Long queue emulation performance.

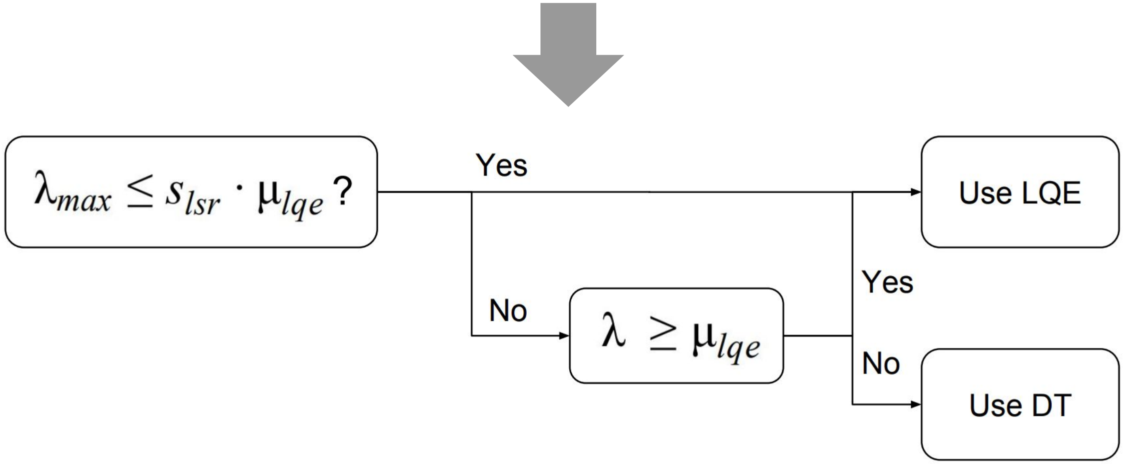
λ : average packet arrival rate

λ_{max} : maximum packet arrival rate

μ_{dt} : packet processing rate of the DT model

μ_{lqe} : packet processing rate of the LQE model

s_{lsr} : size of the LSR ring



$\lambda_{max} \leq s_{lsr} \cdot \mu_{lqe}$?

Yes

No

$\lambda \geq \mu_{lqe}$

Yes

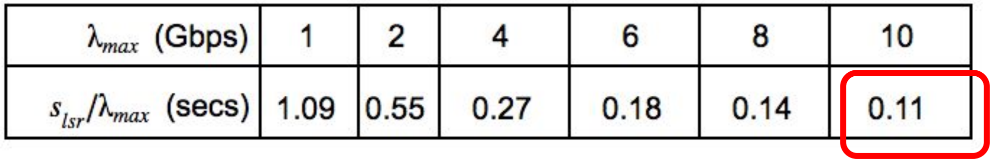
No

Use LQE

Use DT

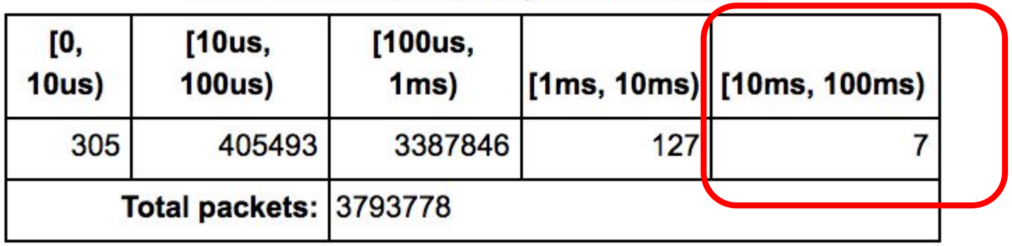
Long Queue Emulation in Practice

Table 1. Maximum packet processing time for a Solarflare SFN7122F NIC



λ_{max} (Gbps)	1	2	4	6	8	10
s_{lsr}/λ_{max} (secs)	1.09	0.55	0.27	0.18	0.14	0.11

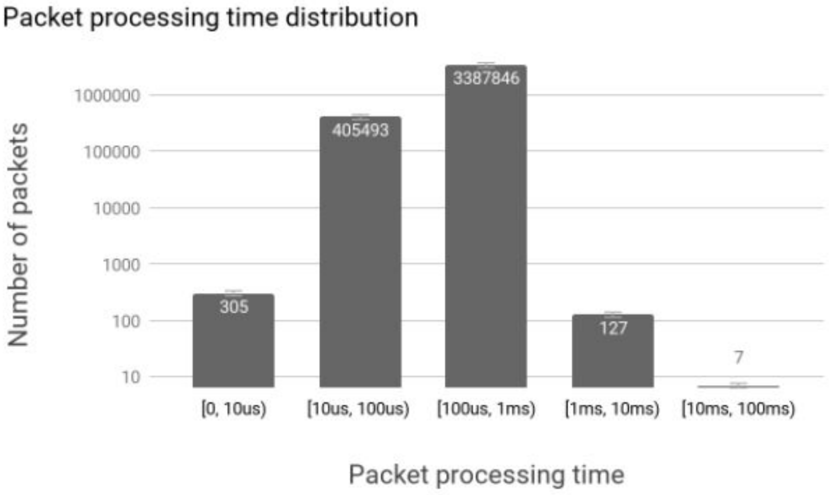
Table 2. *Packet processing time distribution.*



[0, 10us)	**[10us, 100us)**	**[100us, 1ms)**	**[1ms, 10ms)**	**[10ms, 100ms)**
305	405493	3387846	127	7
Total packets:		3793778		

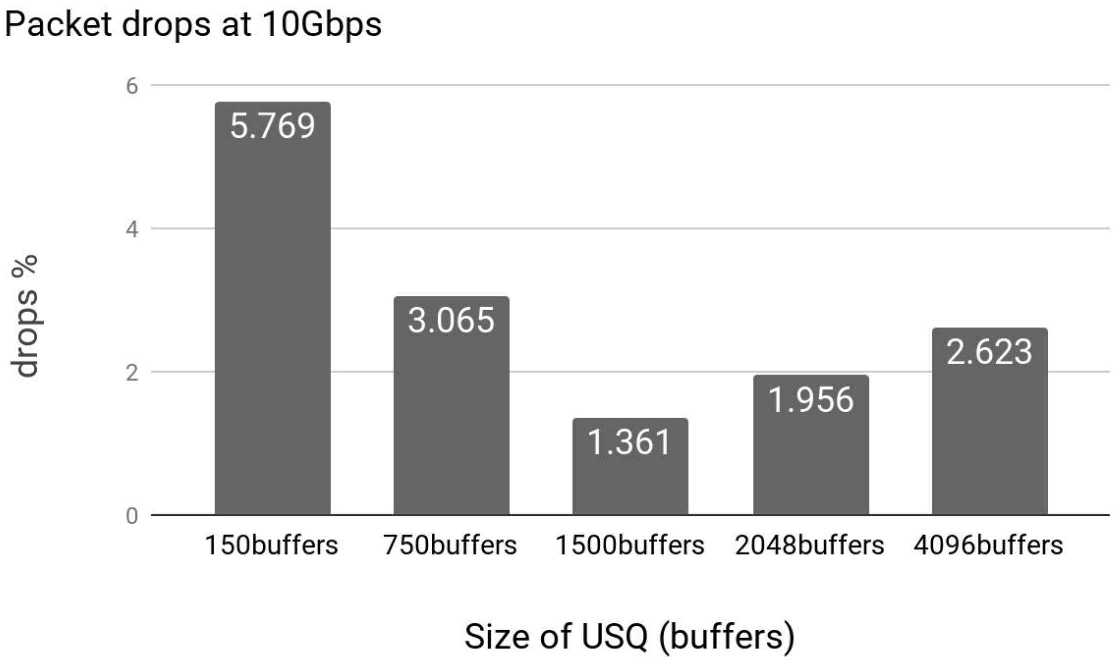
$$\lambda_{max} \leq s_{lsr} \cdot \mu_{lqe} \,?$$

Use LQE



Long Queue Emulation

- Optimal LQE size

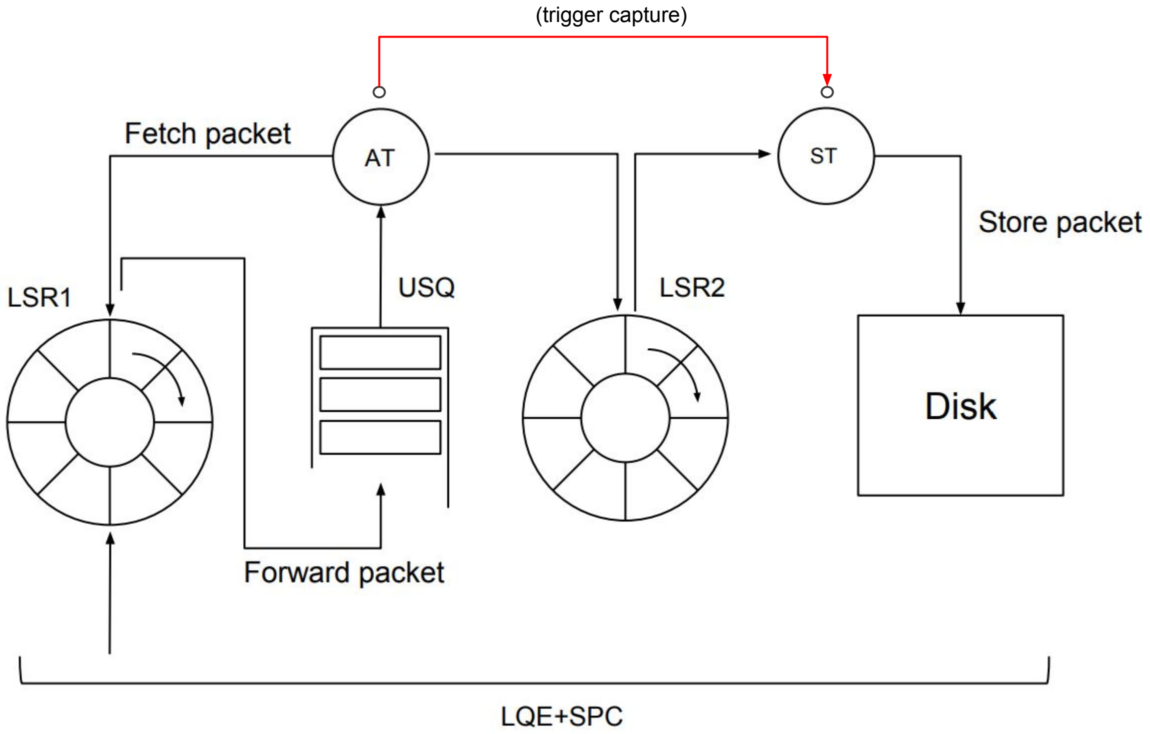


Performance Optimization: Algorithms and Data Structures

Long queue emulation	Reduces packet drops due to fixed-size hardware rings
Lockless bimodal queues	Improves packet capturing performance
Tail early dropping	Increases information entropy and extracted metadata
LFN tables	Reduces state sharing overhead
Multiresolution priority queues	Reduces cost of processing timers

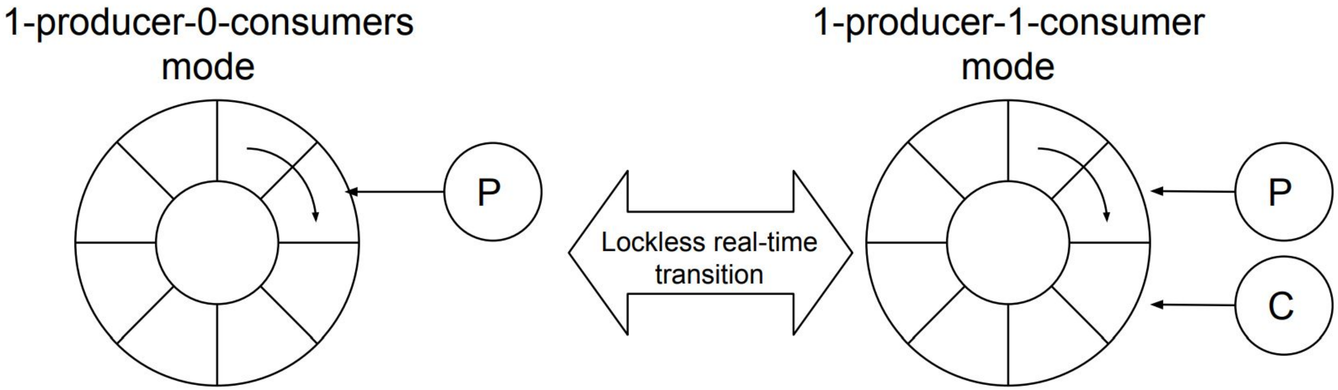
Lockless Bimodal Queues

- Goal: move packets from the memory ring to disk without using locks

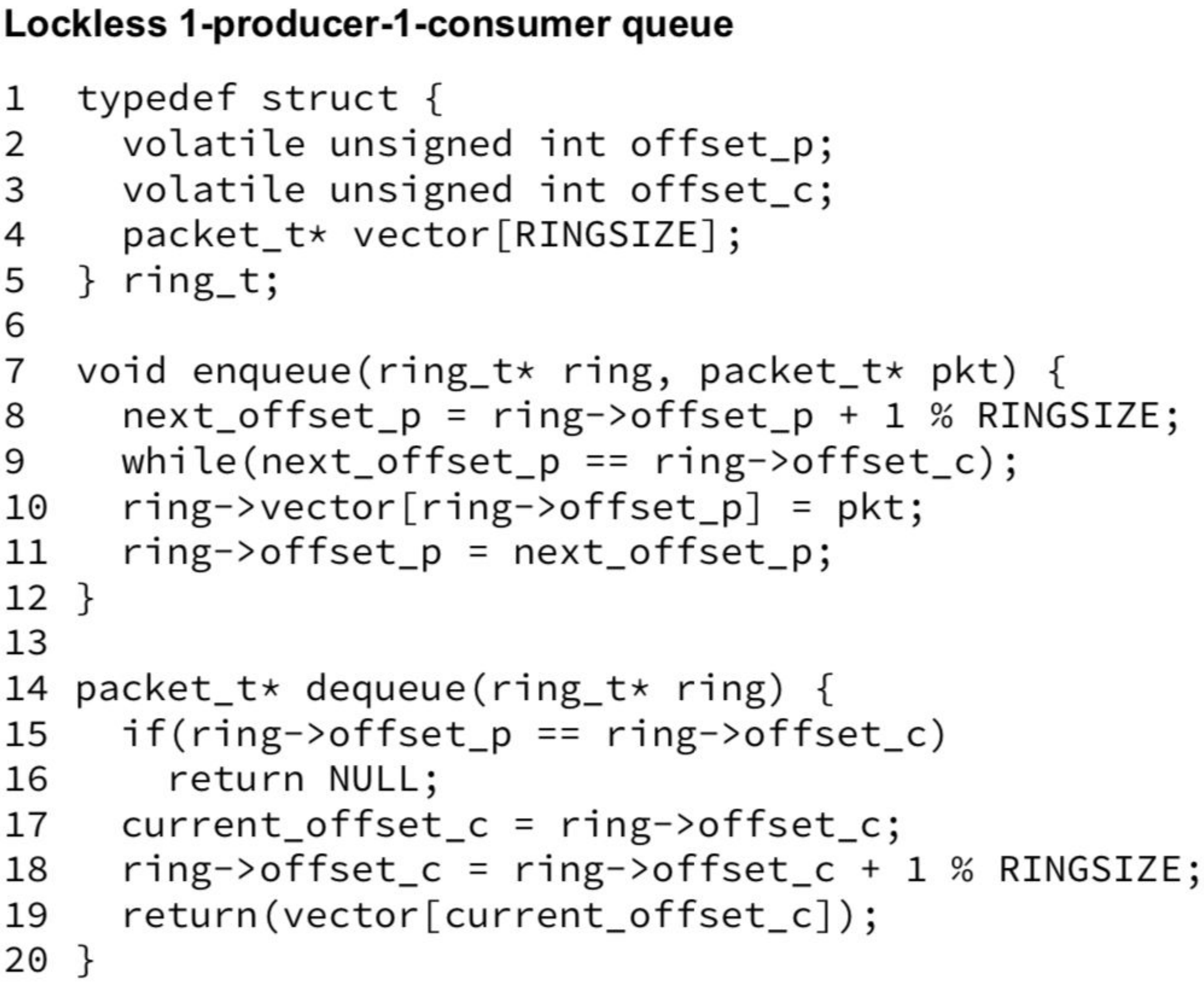


Lockless Bimodal Queues

- Goal: move packets from the memory ring to disk without using locks



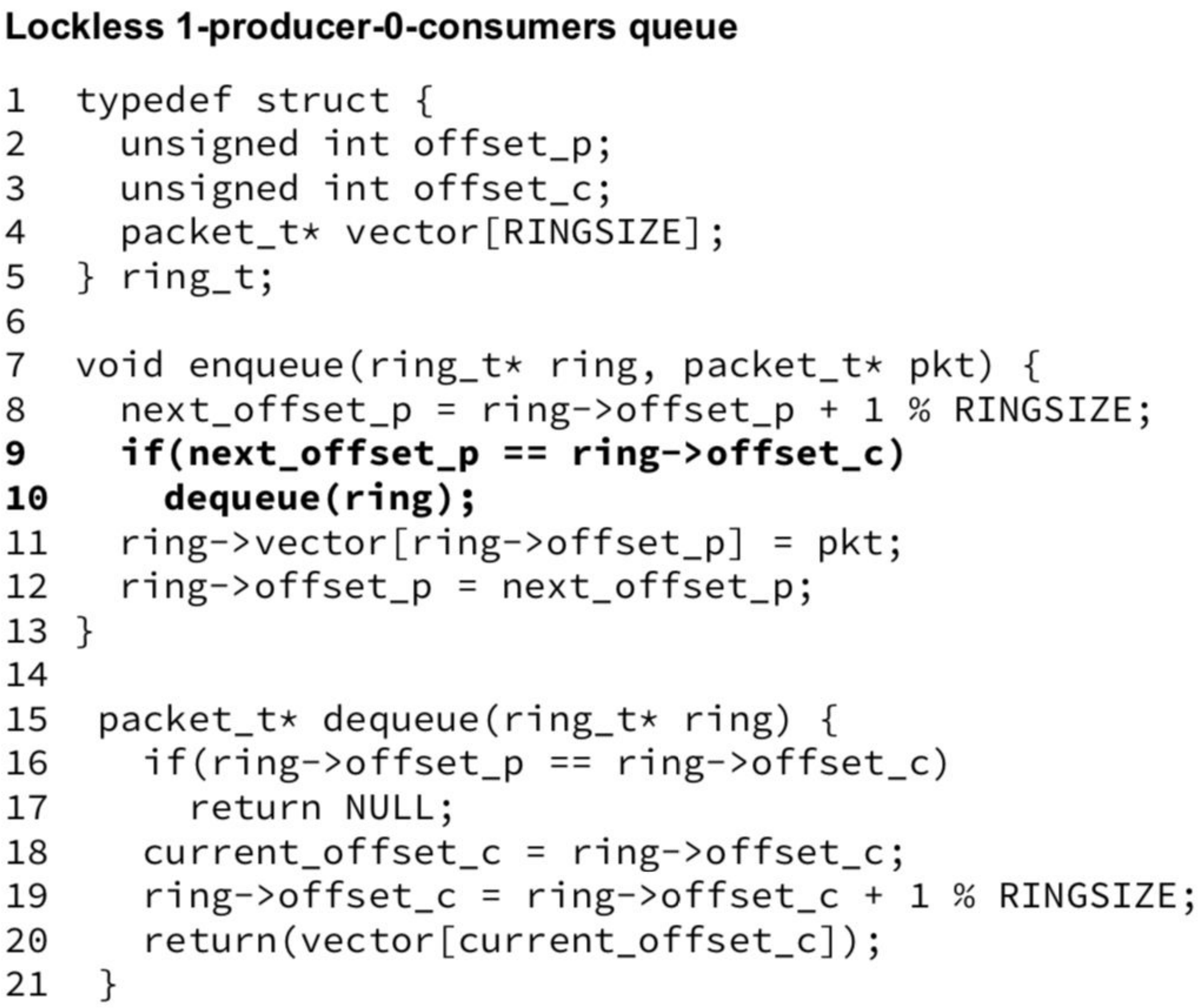
Lockless Bimodal Queues



Lockless 1-producer-1-consumer queue

```
typedef struct {
  volatile unsigned int offset_p;
  volatile unsigned int offset_c;
  packet_t* vector[RINGSIZE];
} ring_t;

void enqueue(ring_t* ring, packet_t* pkt) {
  next_offset_p = ring->offset_p + 1 % RINGSIZE;
  while(next_offset_p == ring->offset_c);
  ring->vector[ring->offset_p] = pkt;
  ring->offset_p = next_offset_p;
}

packet_t* dequeue(ring_t* ring) {
  if(ring->offset_p == ring->offset_c)
    return NULL;
  current_offset_c = ring->offset_c;
  ring->offset_c = ring->offset_c + 1 % RINGSIZE;
  return(vector[current_offset_c]);
}
```

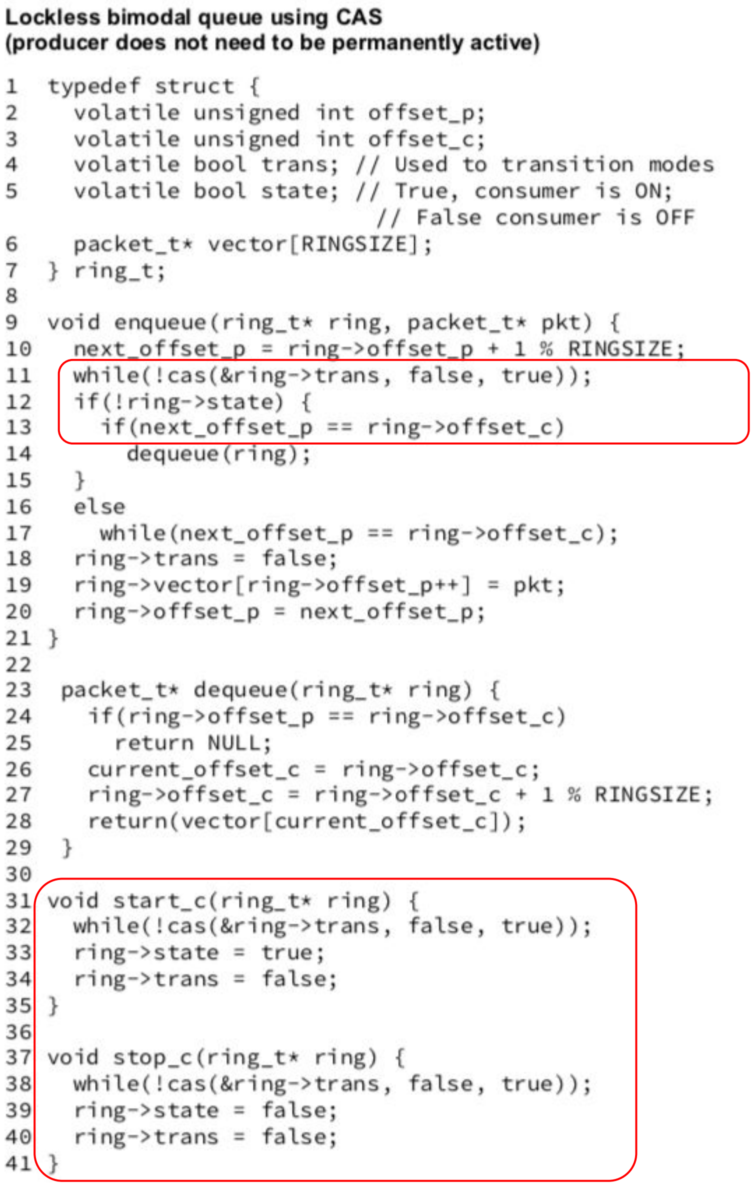
Lockless Bimodal Queues



Lockless 1-producer-0-consumers queue

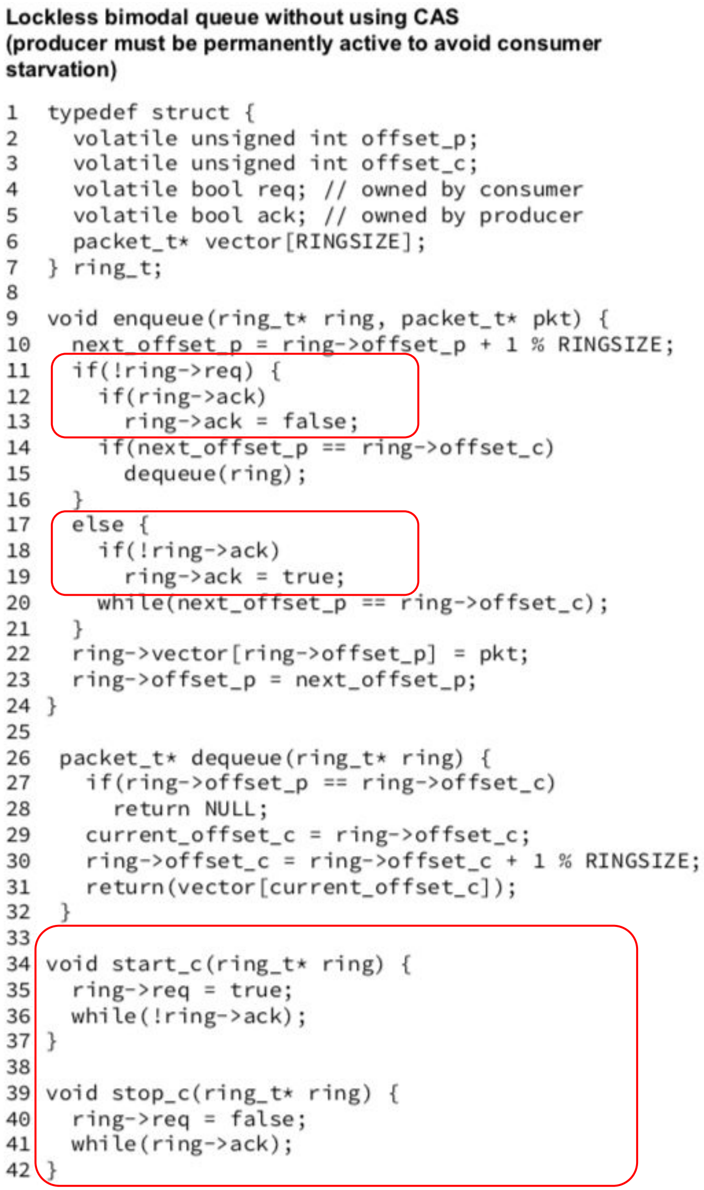
```
typedef struct {
  unsigned int offset_p;
  unsigned int offset_c;
  packet_t* vector[RINGSIZE];
} ring_t;

void enqueue(ring_t* ring, packet_t* pkt) {
  next_offset_p = ring->offset_p + 1 % RINGSIZE;
  if(next_offset_p == ring->offset_c)
    dequeue(ring);
  ring->vector[ring->offset_p] = pkt;
  ring->offset_p = next_offset_p;
}

 packet_t* dequeue(ring_t* ring) {
   if(ring->offset_p == ring->offset_c)
     return NULL;
   current_offset_c = ring->offset_c;
   ring->offset_c = ring->offset_c + 1 % RINGSIZE;
   return(vector[current_offset_c]);
 }
```

Lockless Bimodal Queues



Lockless bimodal queue using CAS
(producer does not need to be permanently active)

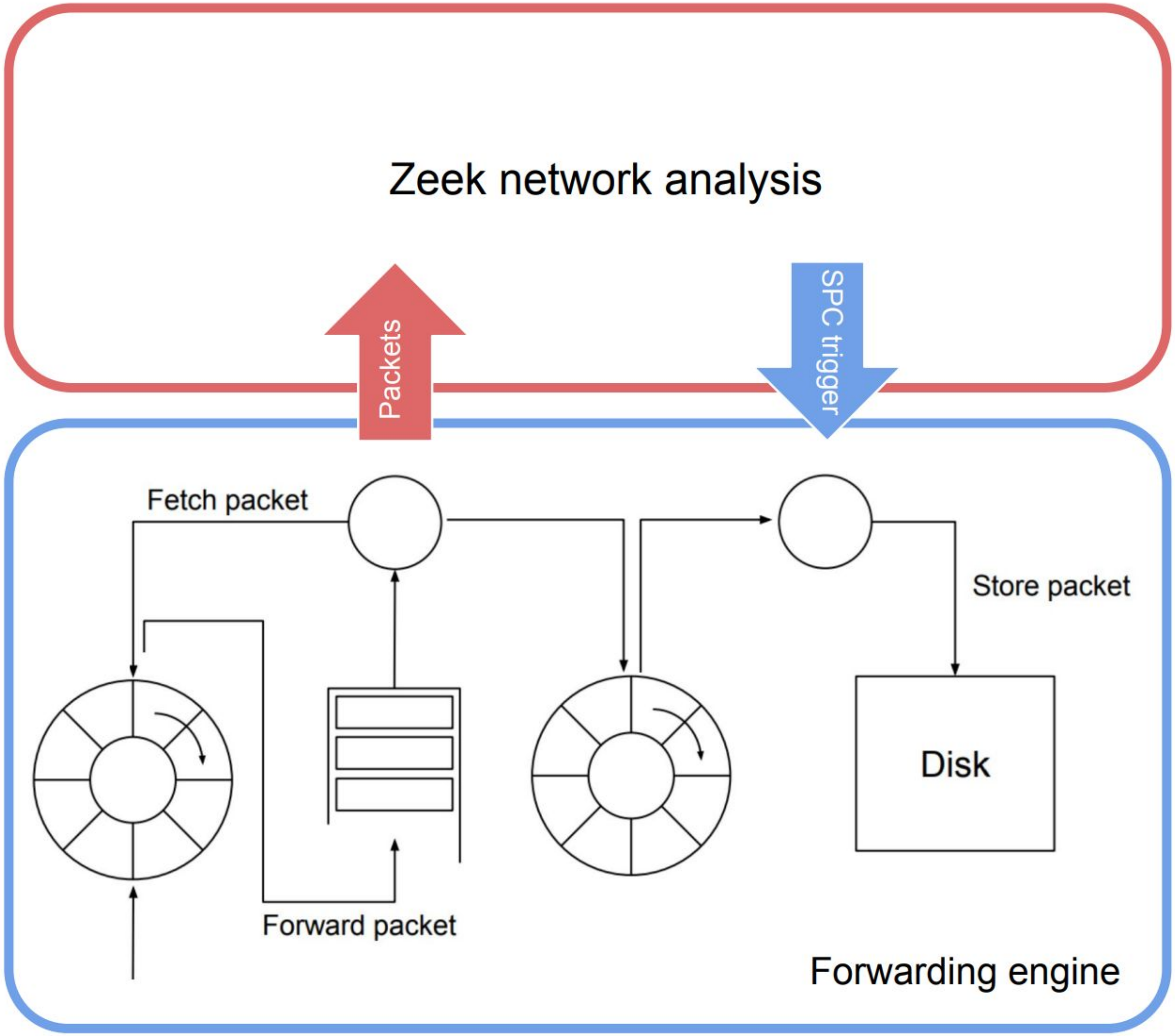
```
typedef struct {
  volatile unsigned int offset_p;
  volatile unsigned int offset_c;
  volatile bool trans; // Used to transition modes
  volatile bool state; // True, consumer is ON;
                        // False consumer is OFF
  packet_t* vector[RINGSIZE];
} ring_t;

void enqueue(ring_t* ring, packet_t* pkt) {
  next_offset_p = ring->offset_p + 1 % RINGSIZE;
  while(!cas(&ring->trans, false, true));
  if(!ring->state) {
    if(next_offset_p == ring->offset_c)
      dequeue(ring);
  }
  else
    while(next_offset_p == ring->offset_c);
  ring->trans = false;
  ring->vector[ring->offset_p++] = pkt;
  ring->offset_p = next_offset_p;
}

 packet_t* dequeue(ring_t* ring) {
   if(ring->offset_p == ring->offset_c)
     return NULL;
   current_offset_c = ring->offset_c;
   ring->offset_c = ring->offset_c + 1 % RINGSIZE;
   return(vector[current_offset_c]);
 }

void start_c(ring_t* ring) {
  while(!cas(&ring->trans, false, true));
  ring->state = true;
  ring->trans = false;
}

void stop_c(ring_t* ring) {
  while(!cas(&ring->trans, false, true));
  ring->state = false;
  ring->trans = false;
}
```



Lockless bimodal queue without using CAS
(producer must be permanently active to avoid consumer starvation)

```
typedef struct {
  volatile unsigned int offset_p;
  volatile unsigned int offset_c;
  volatile bool req; // owned by consumer
  volatile bool ack; // owned by producer
  packet_t* vector[RINGSIZE];
} ring_t;

void enqueue(ring_t* ring, packet_t* pkt) {
  next_offset_p = ring->offset_p + 1 % RINGSIZE;
  if(!ring->req) {
    if(ring->ack)
      ring->ack = false;
    if(next_offset_p == ring->offset_c)
      dequeue(ring);
  }
  else {
    if(!ring->ack)
      ring->ack = true;
    while(next_offset_p == ring->offset_c);
  }
  ring->vector[ring->offset_p] = pkt;
  ring->offset_p = next_offset_p;
}

 packet_t* dequeue(ring_t* ring) {
   if(ring->offset_p == ring->offset_c)
     return NULL;
   current_offset_c = ring->offset_c;
   ring->offset_c = ring->offset_c + 1 % RINGSIZE;
   return(vector[current_offset_c]);
 }

void start_c(ring_t* ring) {
  ring->req = true;
  while(!ring->ack);
}

void stop_c(ring_t* ring) {
  ring->req = false;
  while(ring->ack);
}
```

SPC Workflow



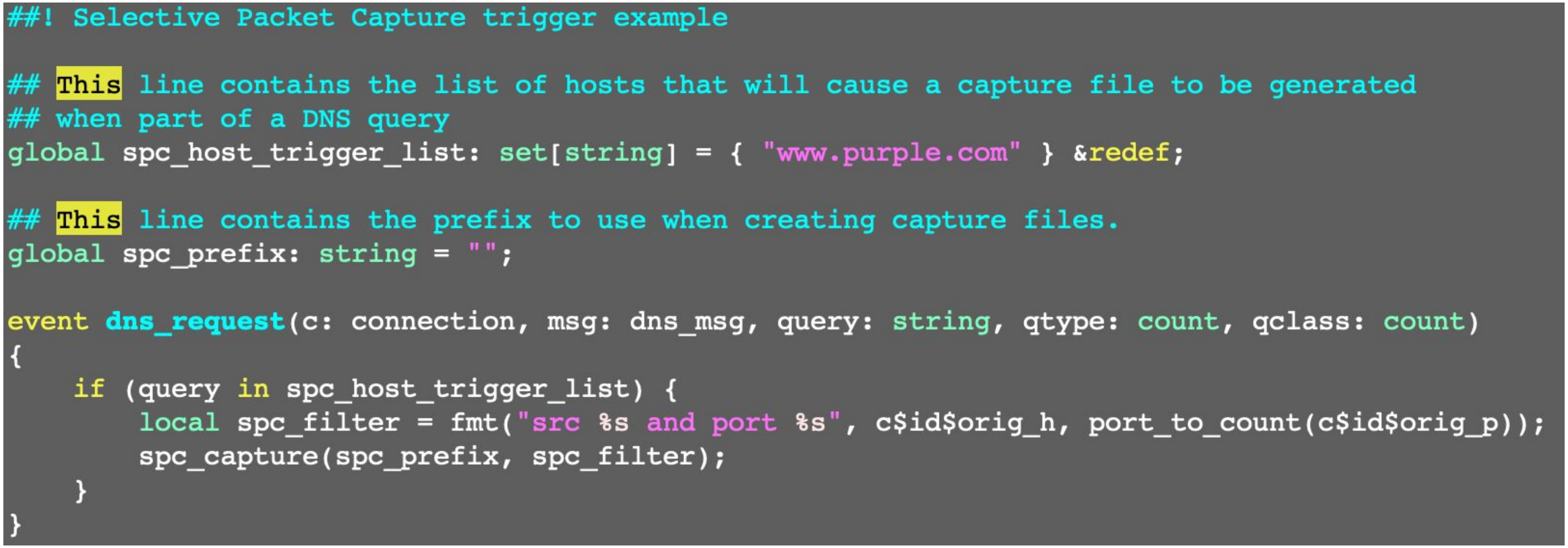
Selective Packet Capture API

- The function spc_capture() takes two arguments as shown by its function prototype:

```
## API for capturing a Pcap
function spc_capture(prefix: string, filter: string);
```

- The prefix argument allows users to specify a prefix for the generated Pcap file name. The filter argument can be used to specify a BPF filter applied to the captured packets as they are written to the pcap file. See https://www.tcpdump.org/manpages/pcap-filter.7.html for the expression syntax of the BPF filter. If set to the empty string “”, all packets (without any filtering) are written to the Pcap file.

Selective Packet Capture by Example



```
##! Selective Packet Capture trigger example

## This line contains the list of hosts that will cause a capture file to be generated
## when part of a DNS query
global spc_host_trigger_list: set[string] = { "www.purple.com" } &redef;

## This line contains the prefix to use when creating capture files.
global spc_prefix: string = "";

event dns_request(c: connection, msg: dns_msg, query: string, qtype: count, qclass: count)
{
    if (query in spc_host_trigger_list) {
        local spc_filter = fmt("src %s and port %s", c$id$orig_h, port_to_count(c$id$orig_p));
        spc_capture(spc_prefix, spc_filter);
    }
}
```

Thank You

Reservoir Labs

632 Broadway
Suite 803
New York, NY 10012

812 SW Washington St.
Suite 1200
Portland, OR 97205