Boost UDP Transaction Performance

Toshiaki Makita
NTT Open Source Software Center

Today's topics

- Background
- Basic technologies for network performance
- How to improve UDP performance

Who is Toshiaki Makita?

- Linux kernel engineer at NTT Open Source Software Center
- Technical support for NTT group companies
- Active patch submitter on kernel networking subsystem

NTT

Background

UDP transactions in the Internet

- **Services using UDP**
  - DNS
  - RADIUS
  - NTP
  - SNMP
  - …
- **Heavily used by network service providers**

Ethernet Bandwidth and Transactions

- **Ethernet bandwidth evolution**
  - 10M -> 100M -> 1G -> 10G -> 40G -> 100G -> ...
  - 10G (or more) NICs are getting common on commodity servers

- **Transactions in 10G network**
  - In the shortest packet case:
    - Maximum 14,880,952 packets/s[*1]
  - Getting hard to handle in a single server...

*1 shortest ethernet frame size 64bytes + preamble+IFG 20bytes = 84 bytes = 672 bits
10,000,000,000 / 672 = 14,880,952

How many transactions to handle?

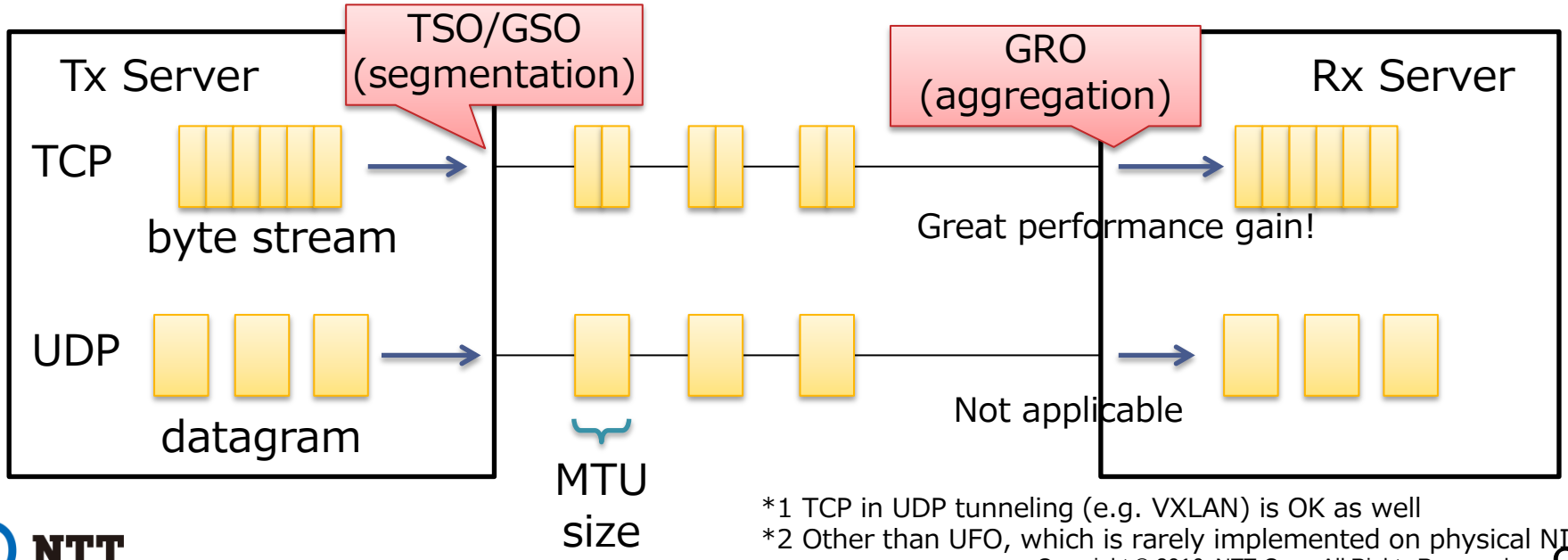
- **UDP payload sizes**
  - DNS
    - A/AAAA query: 40~ bytes
    - A/AAAA response: 100~ bytes
  - RADIUS
    - Access-Request: 70~ bytes
    - Access-Accept: 30~ bytes
    - Typically 100~ bytes with some attributes
  - In many cases 100~ bytes
- **100 bytes transactions in 10G network**
  - Max 7,530,120 transactions/s[*1]
  - Less than shortest packet case, but still challenging

*1 100 bytes + IP/UDP/Ether headers 46bytes + preamble+IFG 20bytes = 166 bytes = 1328 bits
10,000,000,000 / 1328 = 7,530,120

Basic technologies for network performance (not only for UDP)

Basic technologies for network performance

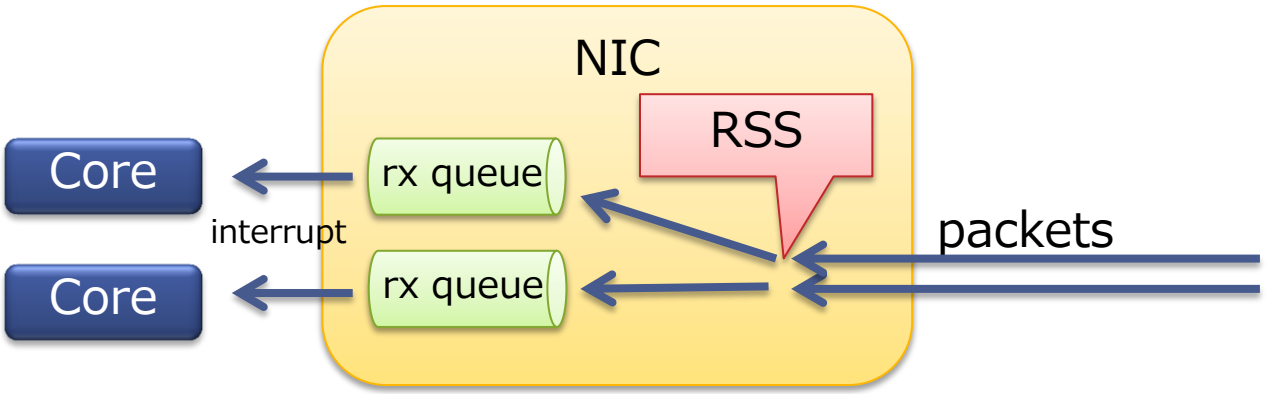
- **TSO/GSO/GRO**
  - Packet segmentation/aggregation
  - Reduce packets to process within server
  - Applicable to TCP*1 (byte stream)
  - Not applicable to UDP*2 (datagram)
    - UDP has explicit boundary between datagrams
    - Cannot segment/aggregate packets



*1 TCP in UDP tunneling (e.g. VXLAN) is OK as well
*2 Other than UFO, which is rarely implemented on physical NICs

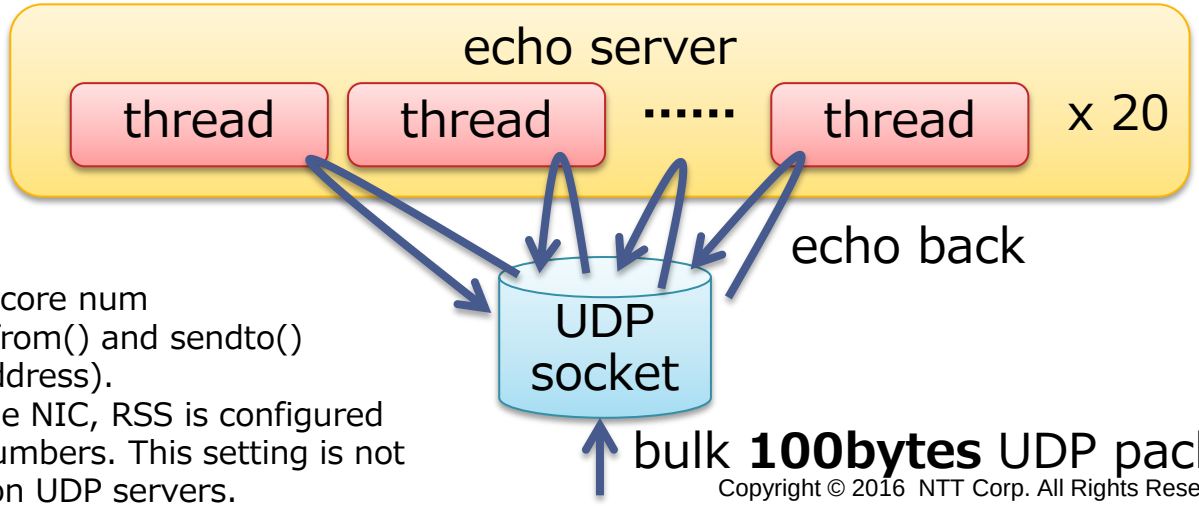
Basic technologies for network performance

- **RSS**
  - Scale network Rx processing in multi-core server
  - RSS itself is a NIC feature
    - Distribute packets to multi-queue in a NIC
    - Each queue has a different interrupt vector
      (Packets on each queue can be processed by different core)
  - Applicable to TCP/UDP
  - Common 10G NICs have RSS



Performance with RSS enabled NIC

- **100 bytes UDP transaction performance**
  - Measured by simple*1 (multi-threaded) echo server
  - OS: **kernel 4.6.3** (in RHEL 7.2 environment)
  - Mid-range commodity server with **20 cores** and **10G** NIC:
    - NIC: Intel 82599ES (has RSS, **max 64 queues**)
    - CPU: Xeon E5-2650 v3 (2.3 GHz 10 cores) * 2 sockets
      Hyper-threading off (make analysis easy, enabled later)
  - Results: **270,000** transactions/s (tps) (approx. **360Mbps**)
    - **3.6%** utilization of 10G bandwidth



*1 create as many threads as core num
each thread just calls recvfrom() and sendto()
*2 There is only 1 client (IP address).
To spread UDP traffic on the NIC, RSS is configured to see UDP port numbers. This setting is not needed for common UDP servers.

How to improve this?

Identify bottleneck

- **sar -u ALL -P ALL 1**

	CPU	%usr	%nice	%sys	%iowait	%steal	%irq	%soft	%guest	%gnice	%idle
19:57:54	all	0.37	0.00	42.58	0.00	0.00	0.00	50.00	0.00	0.00	7.05
19:57:54	0	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	2	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	3	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	4	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	5	1.82	0.00	83.64	0.00	0.00	0.00	0.00	0.00	0.00	14.55
[illegible]	6	0.00	0.00	87.04	0.00	0.00	0.00	0.00	0.00	0.00	12.96
[illegible]	7	0.00	0.00	85.19	0.00	0.00	0.00	0.00	0.00	0.00	14.81
[illegible]	8	0.00	0.00	85.45	0.00	0.00	0.00	0.00	0.00	0.00	14.55
19:57:54	9	0.00	0.00	85.19	0.00	0.00	0.00	0.00	0.00	0.00	14.81
19:57:54	10	[illegible]	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	11	[illegible]	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	12	[illegible]	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	13	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	14	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
19:57:54	15	1.82	0.00	83.64	0.00	0.00	0.00	0.00	0.00	0.00	14.55
19:57:54	16	0.00	0.00	87.04	0.00	0.00	0.00	0.00	0.00	0.00	12.96
19:57:54	17	1.82	0.00	83.64	0.00	0.00	0.00	0.00	0.00	0.00	14.55
19:57:54	18	0.00	0.00	85.45	0.00	0.00	0.00	0.00	0.00	0.00	14.55
19:57:54	19	0.00	0.00	85.45	0.00	0.00	0.00	0.00	0.00	0.00	14.55

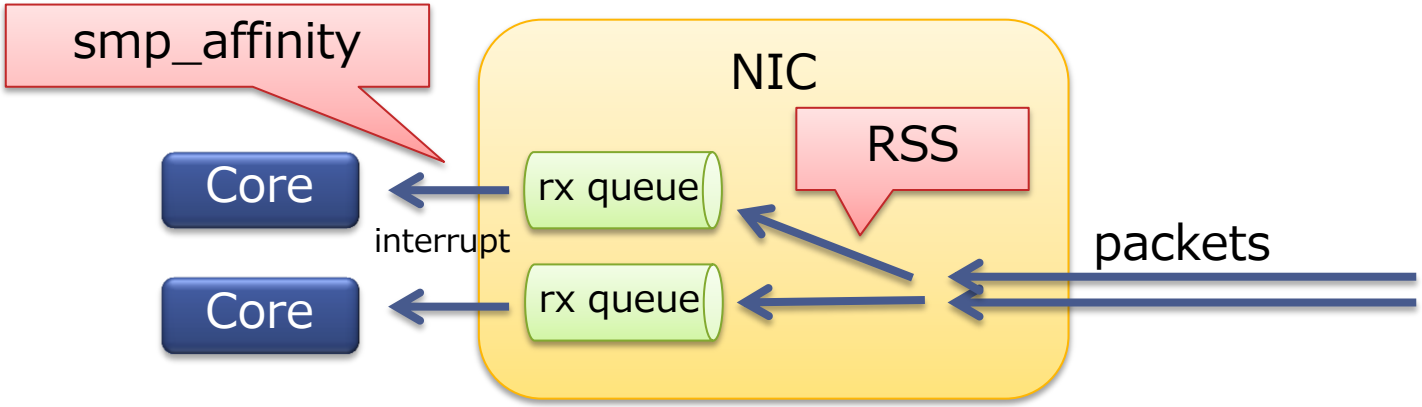
Node 0

Node 1

- **softirq (interrupt processing) is performed only on NUMA Node 0, why?**
  - although we have enough (64) queues for 20 cores...

softirq (interrupt processing) with RSS

- RSS distributes packets to rx-queues
- Interrupt destination of each queue is determined by /proc/irq/<irq>/smp_affinity



- smp_affinity is usually set by irqbalance daemon

Check smp_affinity

- **smp_affinity*1**

```
$ for ((irq=105; irq<=124; irq++)); do
>   cat /proc/irq/$irq/smp_affinity
> done
01000    -> 12    -> Node 0
00800    -> 11    -> Node 0
00400    -> 10    -> Node 0
00400    -> 10    -> Node 0
01000    -> 12    -> Node 0
04000    -> 14    -> Node 0
00400    -> 10    -> Node 0
00010    ->  4    -> Node 0
00004    ->  2    -> Node 0
02000    -> 13    -> Node 0
```

```
04000    -> 14    -> Node 0
00001    ->  0    -> Node 0
02000    -> 13    -> Node 0
01000    -> 12    -> Node 0
00008    ->  3    -> Node 0
00800    -> 11    -> Node 0
00800    -> 11    -> Node 0
04000    -> 14    -> Node 0
00800    -> 11    -> Node 0
02000    -> 13    -> Node 0
```

- **irqbalance is using only Node 0 (cores 0-4, 10-14)**
  - Can we change this?

*1 irq number can be obtained from /proc/interrupts

Check affinity_hint

- **Some NIC drivers provide affinity_hint**

```
$ for ((irq=105; irq<=124; irq++)); do
>   cat /proc/irq/$irq/affinity_hint
> done
00001    -> 0
00002    -> 1
00004    -> 2
00008    -> 3
00010    -> 4
00020    -> 5
00040    -> 6
00080    -> 7
00100    -> 8
00200    -> 9
```

```
00400    -> 10
00800    -> 11
01000    -> 12
02000    -> 13
04000    -> 14
08000    -> 15
10000    -> 16
20000    -> 17
40000    -> 18
80000    -> 19
```

- **affinity_hint is evenly distributed**
- **To honor the hint, add "-h exact" option to irqbalance (via /etc/sysconfig/irqbalance, etc.)*1**

*1 If your NIC doesn't provide hint, you can use "-i" option or stop irqbalance to set their affinity manually

Change irqbalance option

- **Added "-h exact" and restarted irqbalance**

```
$ for ((irq=105; irq<=124; irq++)); do
>   cat /proc/irq/$irq/smp_affinity
> done
00001    -> 0
00002    -> 1
00004    -> 2
00008    -> 3
00010    -> 4
00020    -> 5
00040    -> 6
00080    -> 7
00100    -> 8
00200    -> 9
```

```
00400    -> 10
00800    -> 11
01000    -> 12
02000    -> 13
04000    -> 14
08000    -> 15
10000    -> 16
20000    -> 17
40000    -> 18
80000    -> 19
```

- **With hint honored, irqs are distributed to all cores**

Change irqbalance option

- sar -u ALL -P ALL 1

20:06:07	CPU	%usr	%nice	%sys	%iowait	%steal	%irq	%soft	%guest	%gnice	%idle
20:06:07	all	0.00	0.00	19.18	0.00	0.00	0.00	80.82	0.00	0.00	0.00
20:06:07	0	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	2	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	3	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	4	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	5	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
[illegible]	6	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
[illegible]	7	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
[illegible]	8	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	9	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	10	[illegible]	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	11	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	12	[illegible]	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	13	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	14	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	15	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:06:07	16	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20:06:07	17	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20:06:07	18	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20:06:07	19	0.00	0.00	93.33	0.00	0.00	0.00	6.67	0.00	0.00	0.00

Node 0

Node 1

- Though irqs looks distributed evenly, core 16-19 are not used for softirq...
- Nodes look irrelevant this time

Check rx-queue stats

Innovative R&D by NTT

- **ethtool -S*1**

```
$ ethtool -S ens1f0 | grep 'rx_queue_.*_packets'
     rx_queue_0_packets: 198005155
     rx_queue_1_packets: 153339750
     rx_queue_2_packets: 162870095
     rx_queue_3_packets: 172303801
     rx_queue_4_packets: 153728776
     rx_queue_5_packets: 158138563
     rx_queue_6_packets: 164411653
     rx_queue_7_packets: 165924489
     rx_queue_8_packets: 176545406
     rx_queue_9_packets: 165340188
     rx_queue_10_packets: 150279834
     rx_queue_11_packets: 150983782
     rx_queue_12_packets: 157623687
     rx_queue_13_packets: 150743910
     rx_queue_14_packets: 158634344
     rx_queue_15_packets: 158497890
     rx_queue_16_packets: 4
     rx_queue_17_packets: 3
     rx_queue_18_packets: 0
     rx_queue_19_packets: 8
```

- **Revealed RSS has not distributed packets to queues 16-19**

NTT

*1 Output format depends on drivers

RSS Indirection Table

- **RSS has indirection table which determines to which queue it spreads packets**
- **Can be shown by ethtool -x**

```
$ ethtool -x ens1f0
RX flow hash indirection table for ens1f0 with 20 RX ring(s):
    0:      0     1     2     3     4     5     6     7
    8:      8     9    10    11    12    13    14    15
   16:      0     1     2     3     4     5     6     7
   24:      8     9    10    11    12    13    14    15
   32:      0     1     2     3     4     5     6     7
   40:      8     9    10    11    12    13    14    15
   48:      0     1     2     3     4     5     6     7
   56:      8     9    10    11    12    13    14    15
   64:      0     1     2     3     4     5     6     7
   72:      8     9    10    11    12    13    14    15
   80:      0     1     2     3     4     5     6     7
   88:      8     9    10    11    12    13    14    15
   96:      0     1     2     3     4     5     6     7
  104:      8     9    10    11    12    13    14    15
  112:      0     1     2     3     4     5     6     7
  120:      8     9    10    11    12    13    14    15
```

flow hash (hash value from packet header)

rx-queue number

- **Only rx-queue 0-15 are used, 16-19 not used**

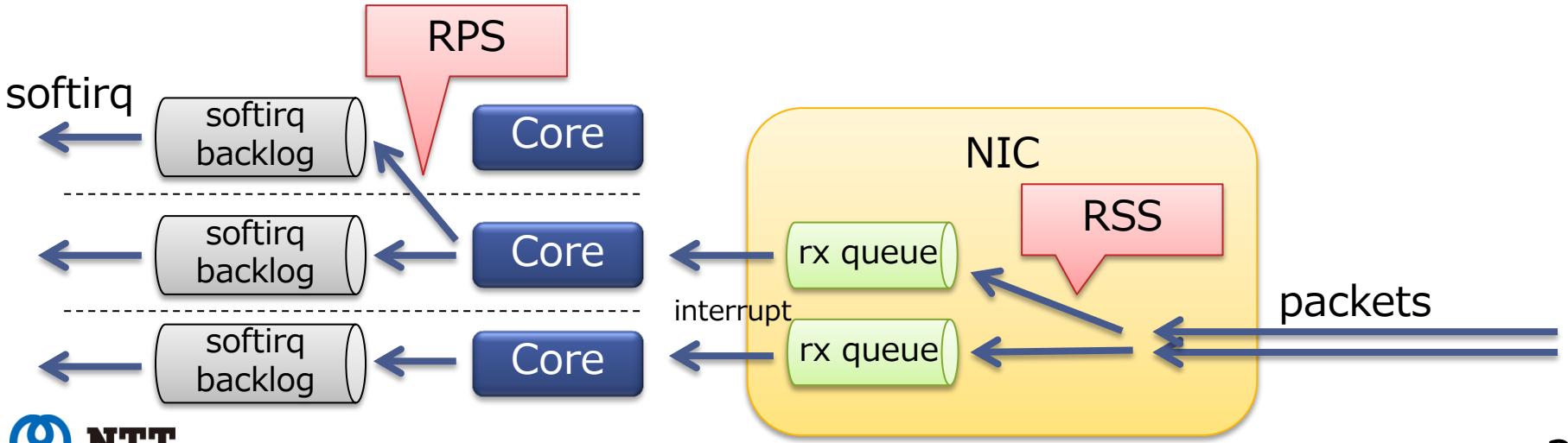
RSS Indirection Table

- **Change to use all 0-19?**

```
# ethtool -X ens1f0 equal 20
Cannot set RX flow hash configuration: Invalid argument
```

- **This NIC's max rx-queues in the indirection table is actually 16 so we cannot use 20 queues**
  - although we have 64 rx-queues...

- **Use RPS instead**
  - Software emulation of RSS



Use RPS

- **This time I spread flows from rx-queue 6-9 to core 6-9 and 16-19**
  - Because they are all in Node 1
  - rx-queue 6 -> core 6, 16
  - rx-queue 7 -> core 7, 17
  - rx-queue 8 -> core 8, 18
  - rx-queue 9 -> core 9, 19

```
# echo 10040 > /sys/class/net/ens1f0/queues/rx-6/rps_cpus
# echo 20080 > /sys/class/net/ens1f0/queues/rx-7/rps_cpus
# echo 40100 > /sys/class/net/ens1f0/queues/rx-8/rps_cpus
# echo 80200 > /sys/class/net/ens1f0/queues/rx-9/rps_cpus
```

Use RPS

- sar -u ALL -P ALL 1

20:18:53	CPU	%usr	%nice	%sys	%iowait	%steal	%irq	%soft	%guest	%gnice	%idle
20:18:54	all	0.00	0.00	2.38	0.00	0.00	0.00	97.62	0.00	0.00	0.00
20:18:54	0	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	2	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	3	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	4	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	5	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	6	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	7	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	8	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	9	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	10	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	11	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	12	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	13	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	14	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	15	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
20:18:54	16	0.00	0.00	15.56	0.00	0.00	0.00	84.44	0.00	0.00	0.00
20:18:54	17	0.00	0.00	6.98	0.00	0.00	0.00	93.02	0.00	0.00	0.00
20:18:54	18	0.00	0.00	18.18	0.00	0.00	0.00	81.82	0.00	0.00	0.00
20:18:54	19	2.27	0.00	6.82	0.00	0.00	0.00	90.91	0.00	0.00	0.00

- softirq is almost evenly distributed

RSS & affinity_hint & RPS

Innovative R&D by NTT

- **Now thanks to affinity_hint and RPS, we succeeded to spread flows almost evenly**

- **Performance change**
  - Before: **270,000** tps (approx. **360Mbps**)
  - After: **17,000** tps (approx. **23Mbps**)
    - Got worse...

- **Probably the reason is too heavy softirq**
  - softirq is almost 100% in total
  - Need finer-grained profiling than sar

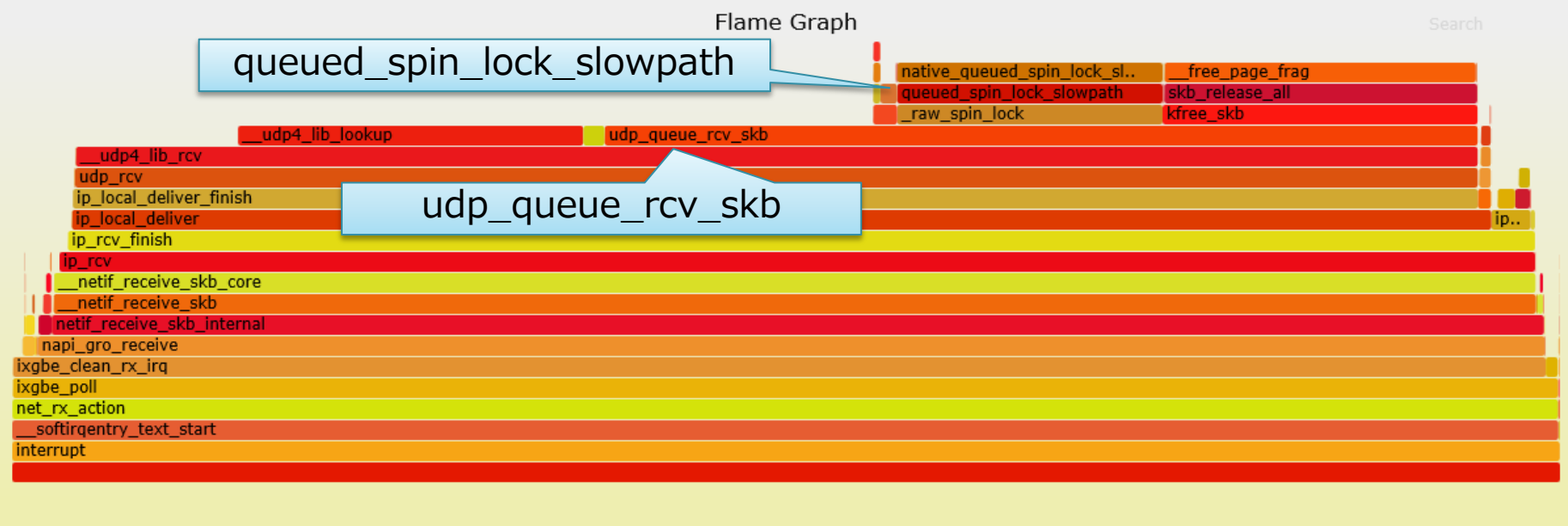
NTT

Profile softirq

- **perf**
  - Profiling tool developed in kernel tree
  - Identify hot spots by sampling CPU cycles
- **Example usage of perf**
  - perf record -a -g -- sleep 5
    - Save sampling results for 5 seconds to perf.data file
- **FlameGraph**
  - Visualize perf.data in svg format
  - https://github.com/brendangregg/FlameGraph

Profile softirq

- **FlameGraph of CPU0*1**
  - x-axis (width): CPU consumption
  - y-axis (height): Depth of call stack

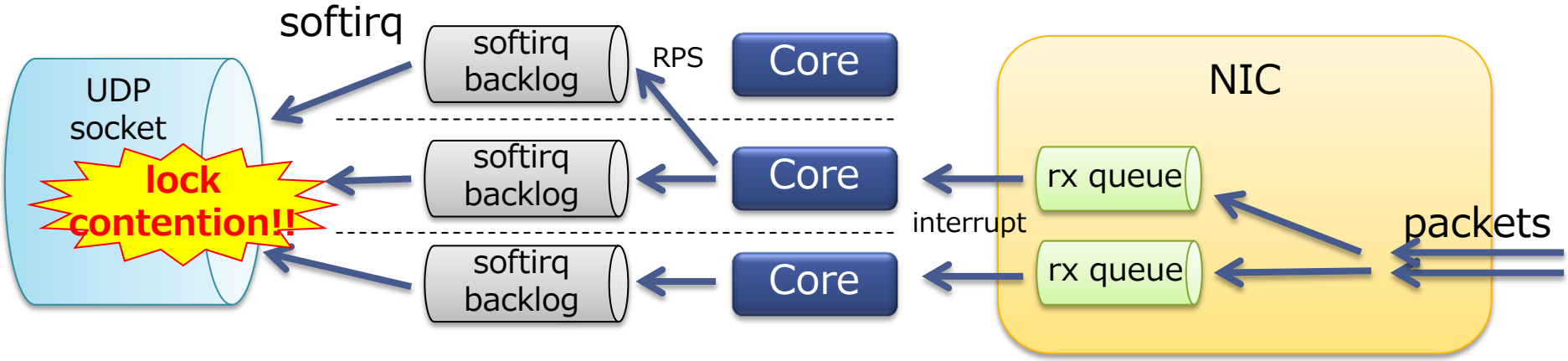


- **queued_spin_lock_slowpath: lock is contended**
- **udp_queue_rcv_skb: aquires socket lock**

*1 I filtered call stack under irq context from output of perf script to make the chart easier to see irq context is shown as "interrupt" here

Socket lock contention

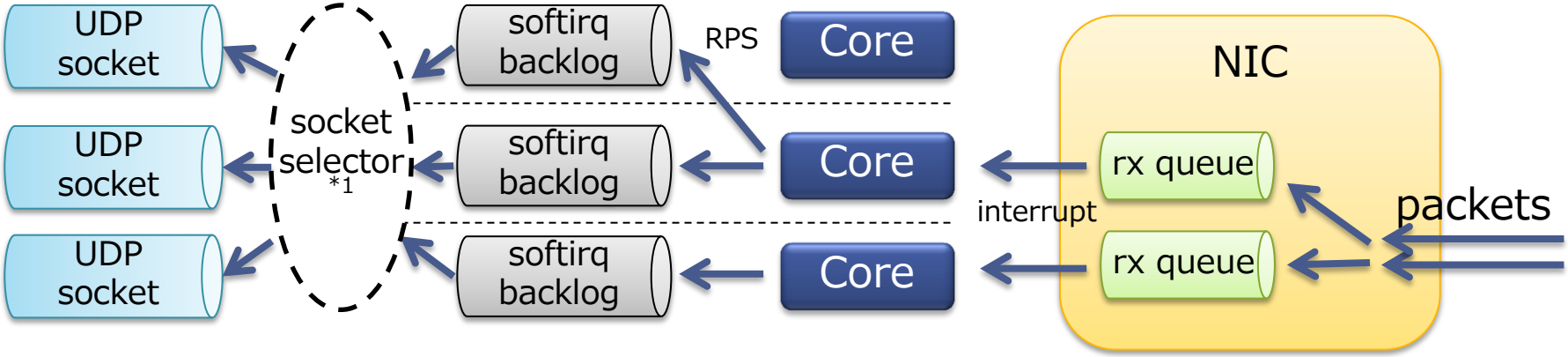
- Echo server has only one socket bound to a certain port
- softirq of each core pushes packets into socket queue concurrently



- socket lock gets contended

Avoid lock contention

- **Split sockets by SO_REUSEPORT**
  - Introduced by kernel 3.9



- **SO_REUSEPORT allows multiple UDP sockets to bind the same port**
  - One of the sockets is chosen on queueing each packet

```
int on = 1;
int sock = socket(AF_INET, SOCK_DGRAM, 0);
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &on, sizeof(on));
bind(sock, ...);
```

*1 select a socket by flow (packet header) hash by default

Use SO_REUSEPORT

Innovative R&D by NTT

- sar -u ALL -P ALL 1

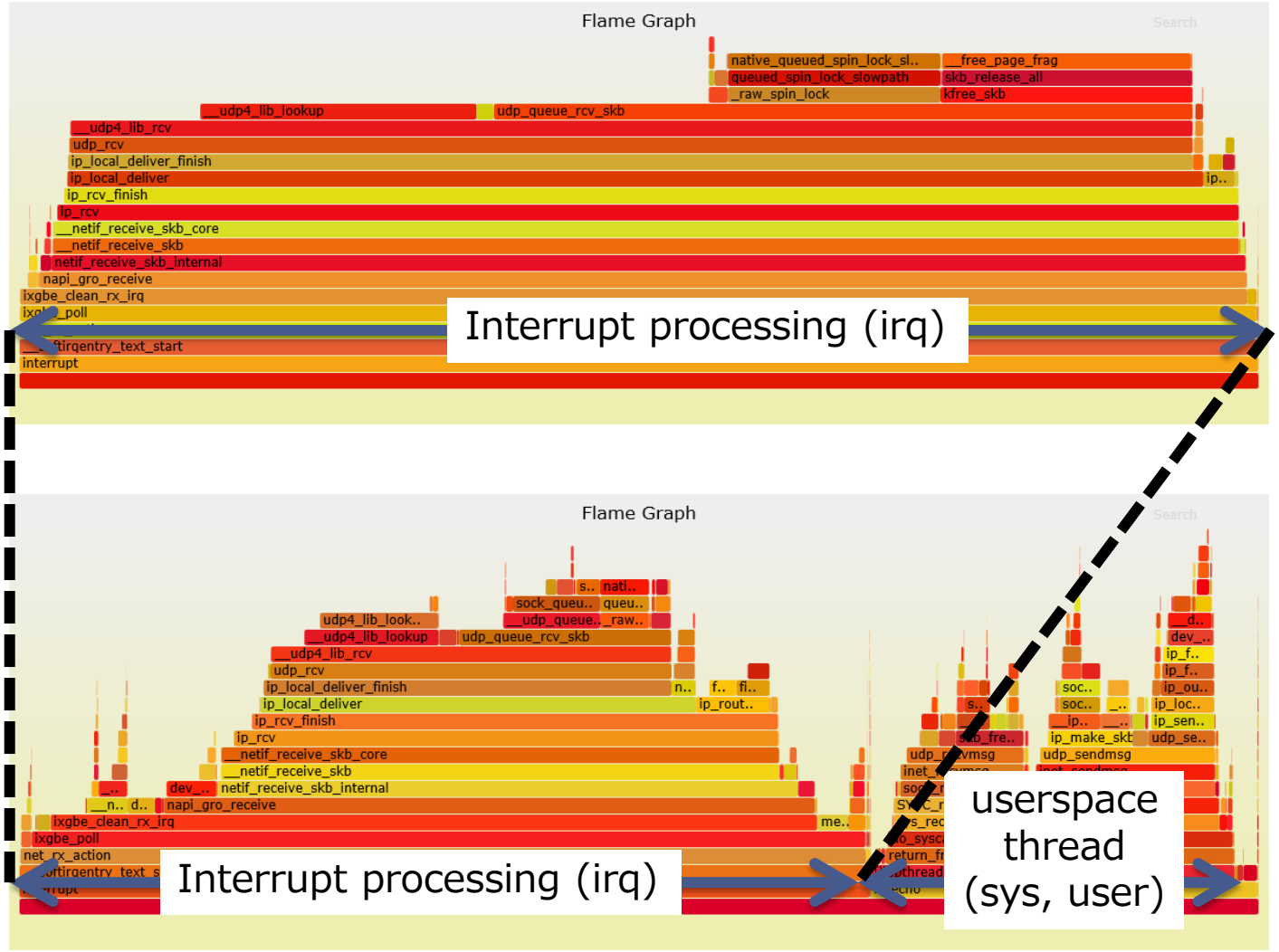
20:44:33	CPU	%usr	%nice	%sys	%iowait	%steal	%irq	%soft	%guest	%gnice	%idle
20:44:34	all	3.26	0.00	37.23	0.00	0.00	0.00	59.52	0.00	0.00	0.00
20:44:34	0	3.33	0.00	28.33	0.00	0.00	0.00	68.33	0.00	0.00	0.00
20:44:34	1	3.33	0.00	25.00	0.00	0.00	0.00	71.67	0.00	0.00	0.00
20:44:34	2	1.67	0.00	23.33	0.00	0.00	0.00	75.00	0.00	0.00	0.00
20:44:34	3	3.28	0.00	32.79	0.00	0.00	0.00	63.93	0.00	0.00	0.00
20:44:34	4	3.33	0.00	33.33	0.00	0.00	0.00	63.33	0.00	0.00	0.00
20:44:34	5	1.69	0.00	23.73	0.00	0.00	0.00	74.58	0.00	0.00	0.00
20:44:34	6	3.28	0.00	50.82	0.00	0.00	0.00	45.90	0.00	0.00	0.00
20:44:34	7	3.45	0.00	50.00	0.00	0.00	0.00	46.55	0.00	0.00	0.00
20:44:34	8	1.69	0.00	37.29	0.00	0.00	0.00	61.02	0.00	0.00	0.00
20:44:34	9	1.67	0.00	33.33	0.00	0.00	0.00	65.00	0.00	0.00	0.00
20:44:34	10	1.69	0.00	18.64	0.00	0.00	0.00	79.66	0.00	0.00	0.00
20:44:34	11	3.23	0.00	35.48	0.00	0.00	0.00	61.29	0.00	0.00	0.00
20:44:34	12	1.69	0.00	27.12	0.00	0.00	0.00	71.19	0.00	0.00	0.00
20:44:34	13	1.67	0.00	21.67	0.00	0.00	0.00	76.67	0.00	0.00	0.00
20:44:34	14	1.67	0.00	21.67	0.00	0.00	0.00	76.67	0.00	0.00	0.00
20:44:34	15	3.33	0.00	35.00	0.00	0.00	0.00	61.67	0.00	0.00	0.00
20:44:34	16	6.67	0.00	68.33	0.00	0.00	0.00	25.00	0.00	0.00	0.00
20:44:34	17	5.00	0.00	65.00	0.00	0.00	0.00	30.00	0.00	0.00	0.00
20:44:34	18	6.78	0.00	54.24	0.00	0.00	0.00	38.98	0.00	0.00	0.00
20:44:34	19	4.92	0.00	63.93	0.00	0.00	0.00	31.15	0.00	0.00	0.00

- CPU consumption in softirq became some more reasonable

NTT

Use SO_REUSEPORT

- before
- after



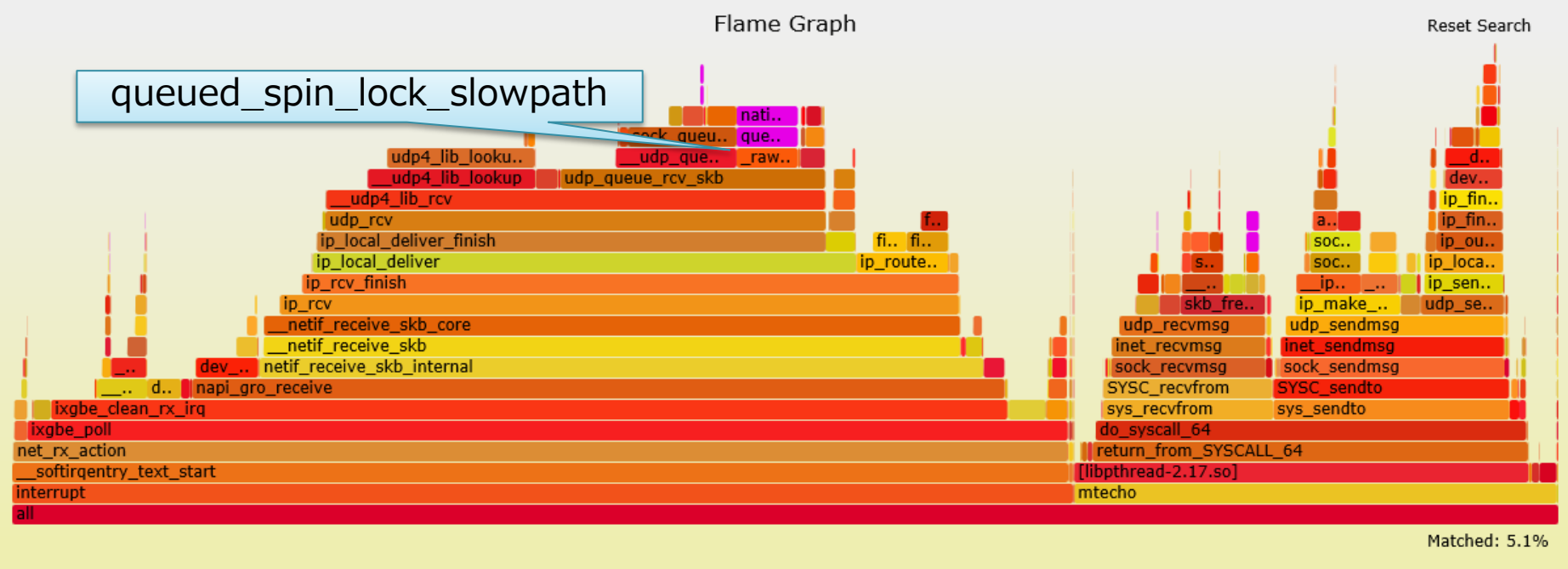
Userspace starts to work

Use SO_REUSEPORT

- **Perfomance change**
  - RSS: 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: **2,540,000** tps (approx. **3370Mbps**)
    - Great improvement!
    - but...

Use SO_REUSEPORT

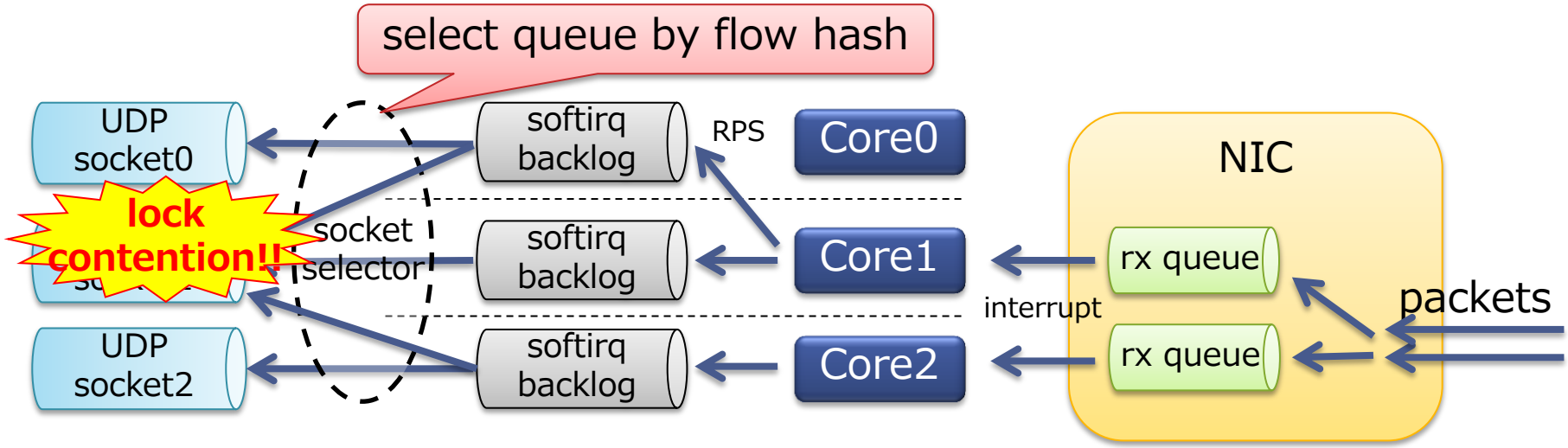
- More analysis



- Socket lock is still contended

Socket lock contention again

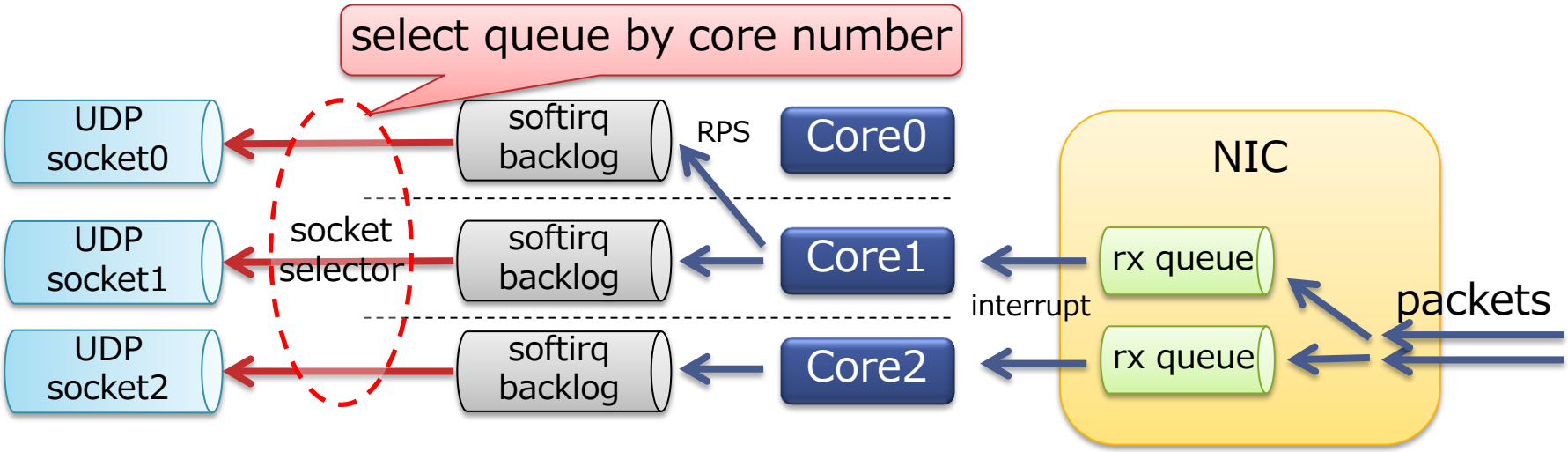
- SO_REUSEPORT uses flow hash to select queue by default
- Same sockets can be selected by different cores



- Socket lock still gets contended

Avoid socket lock contention

- **Select socket by core number**
  - Realized by SO_ATTACH_REUSEPORT_CBPF/EBPF*1
  - Introduced by kernel 4.5

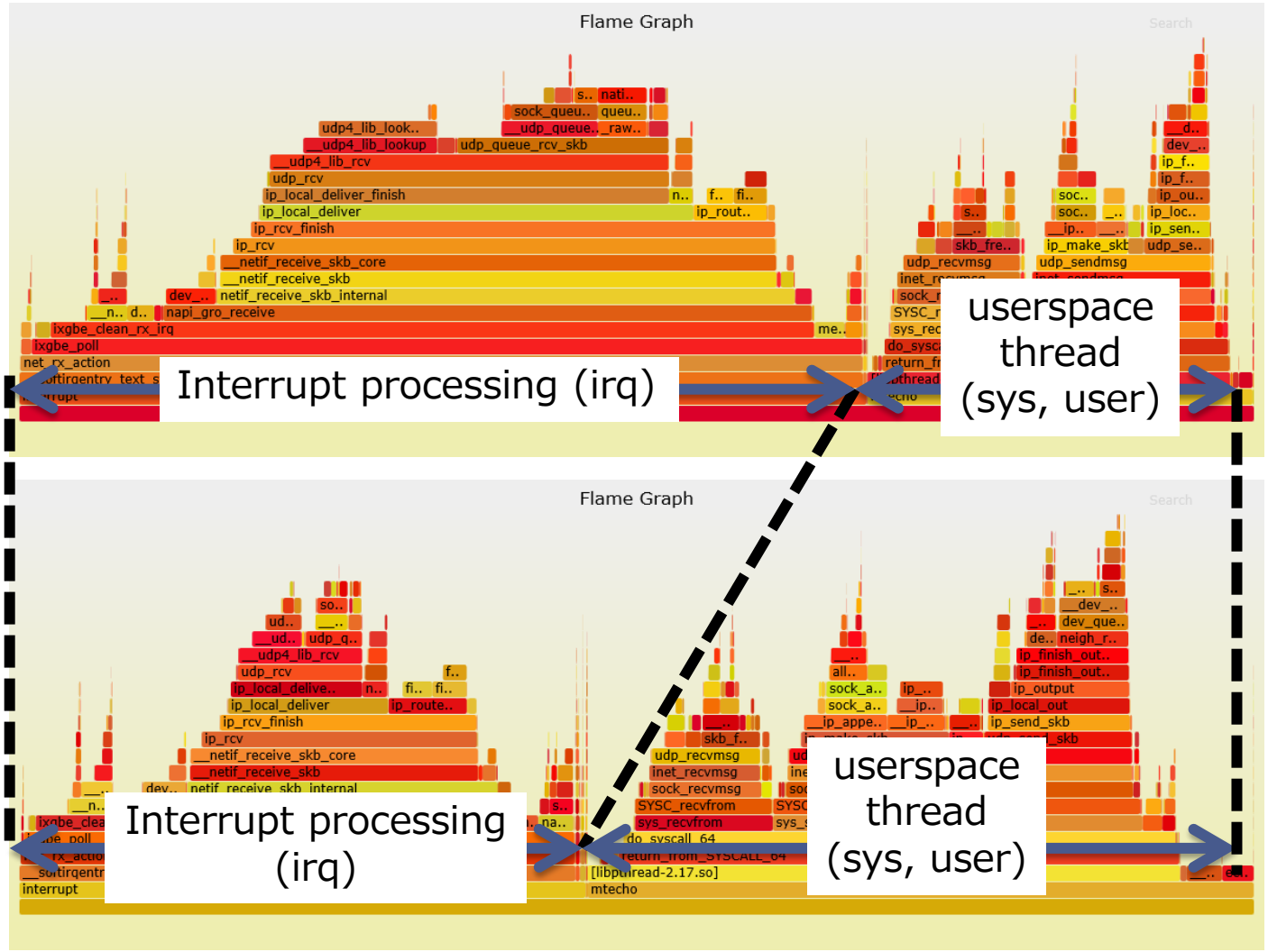


- **No lock contention between softirq**
- **Usage**
  - See example in kernel source tree
    - tools/testing/selftests/net/reuseport_bpf_cpu.c

*1 BPF allows much more flexible logic but this time only cpu number is used

Use SO_ATTACH_REUSEPORT_EPBF

- before
- after



irq overhead gets less

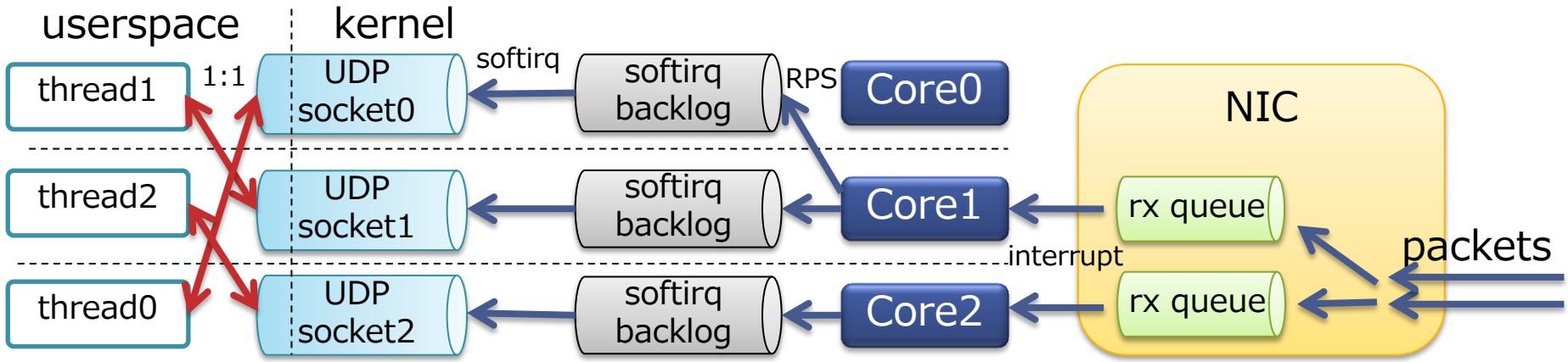
Use SO_ATTACH_REUSEPORT_EBPF

- **Perfomance change**
  - RSS: 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: **4,250,000** tps (approx. **5640Mbps**)

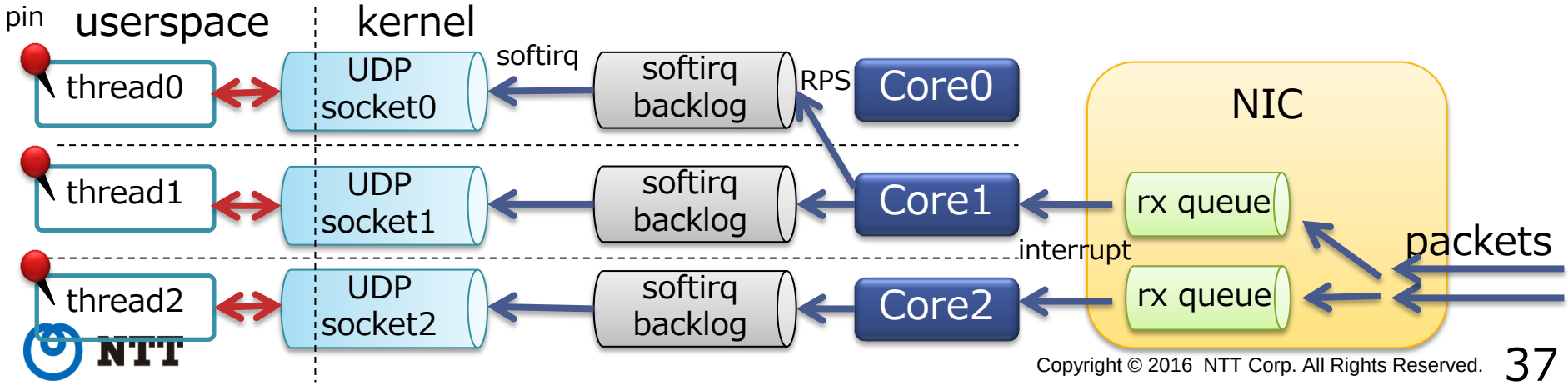
NTT

Pin userspace threads

- **Userspace threads : sockets == 1 : 1**
  - No lock contention
- **But not necessarily on the same core as softirq**



- **Pin userspace thread on the same core for better cache affinity**
  - cgroup, taskset, pthread_setaffinity_np(), ... any way you like



Pin userspace threads

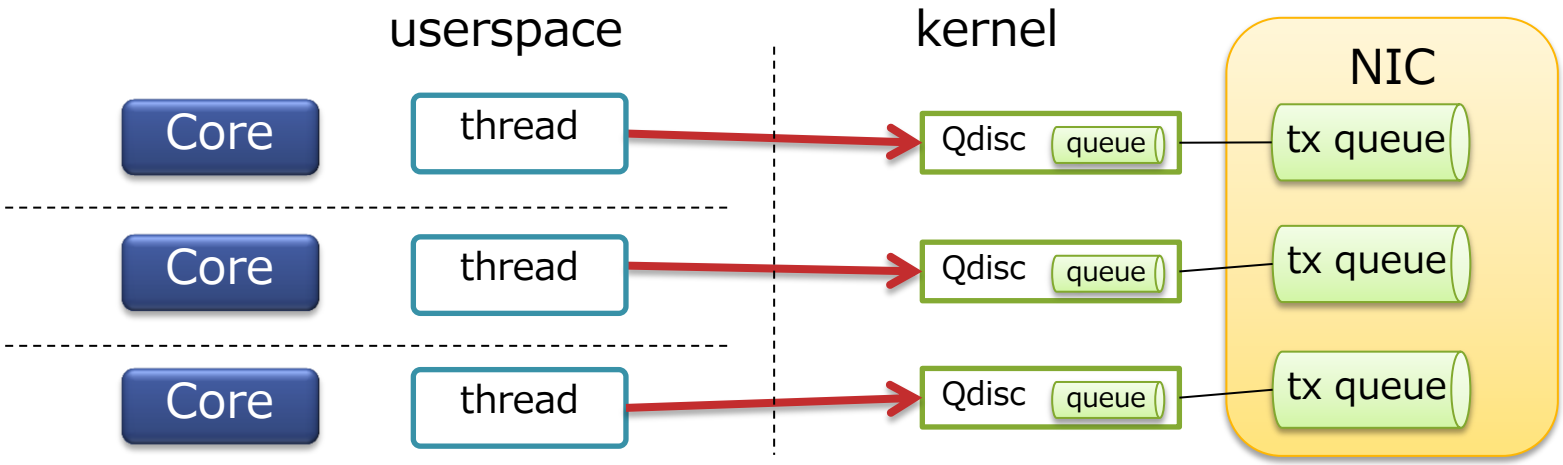
- **Perfomance change**
  - RSS: 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: **5,050,000** tps (approx. **6710Mbps**)

Tx lock contention?

- So far everything has been about Rx
- No lock contention on Tx?

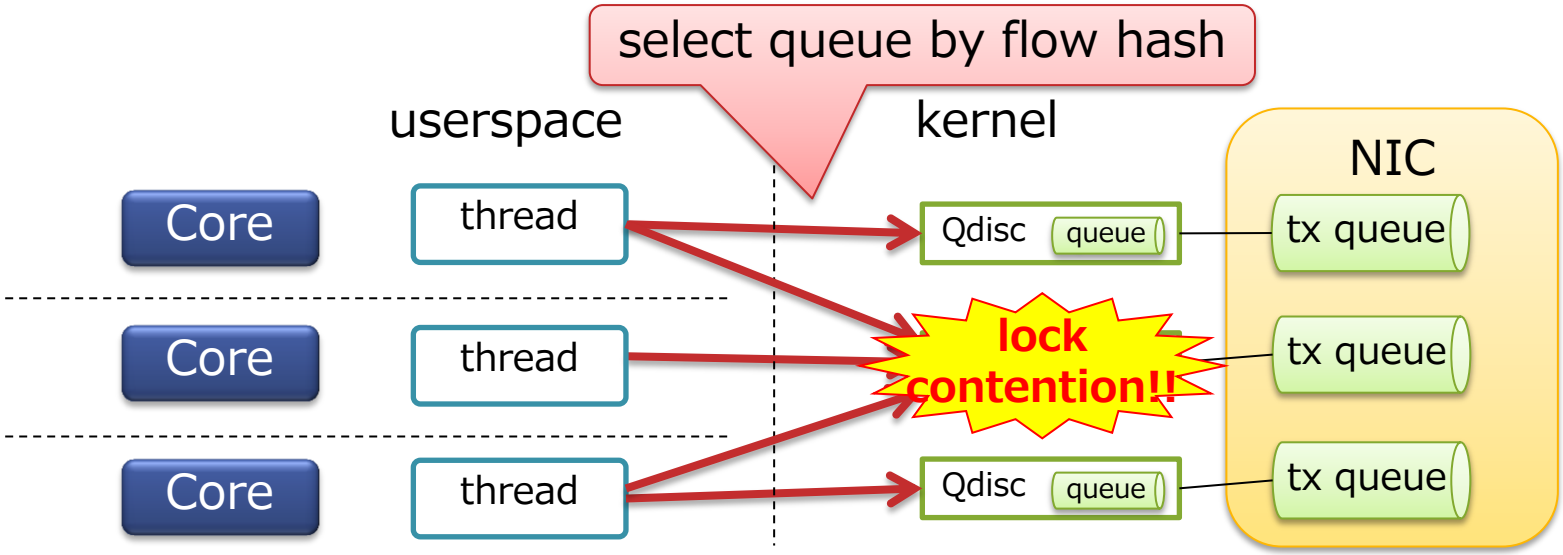
Tx queue

- kernel has Qdisc (Queueing discipline)
- Each Qdisc is linked to NIC tx-queue
- Each Qdisc has its lock



Tx queue lock contention

- By default Qdisc is selected by flow hash
- Thus lock contention can happen



- We haven't seen contention on Tx, why?

Avoid Tx queue lock contention

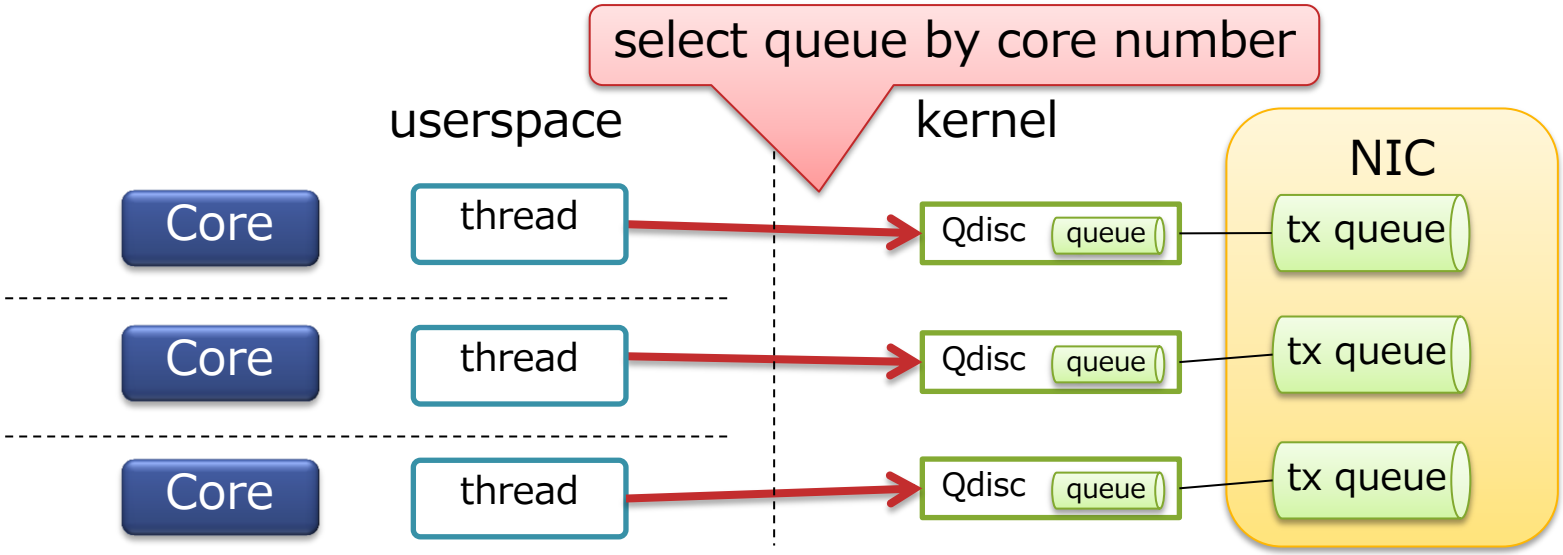
- **Because ixgbe (Intel 10GbE NIC driver) has an ability to set XPS automatically**

```
$ for ((txq=0; txq<20; txq++)); do
>   cat /sys/class/net/ens1f0/queues/tx-$txq/xps_cpus
> done
00001     -> core 0
00002     -> core 1
00004     -> core 2
00008     -> core 3
00010     -> core 4
00020     -> core 5
00040     -> core 6
00080     -> core 7
00100     -> core 8
00200     -> core 9
```

```
00400     -> core 10
00800     -> core 11
01000     -> core 12
02000     -> core 13
04000     -> core 14
08000     -> core 15
10000     -> core 16
20000     -> core 17
40000     -> core 18
80000     -> core 19
```

XPS

- **XPS allows kernel to select Tx queue (Qdisc) by core number**



- **Tx has no lock contention**

How effective is XPS?

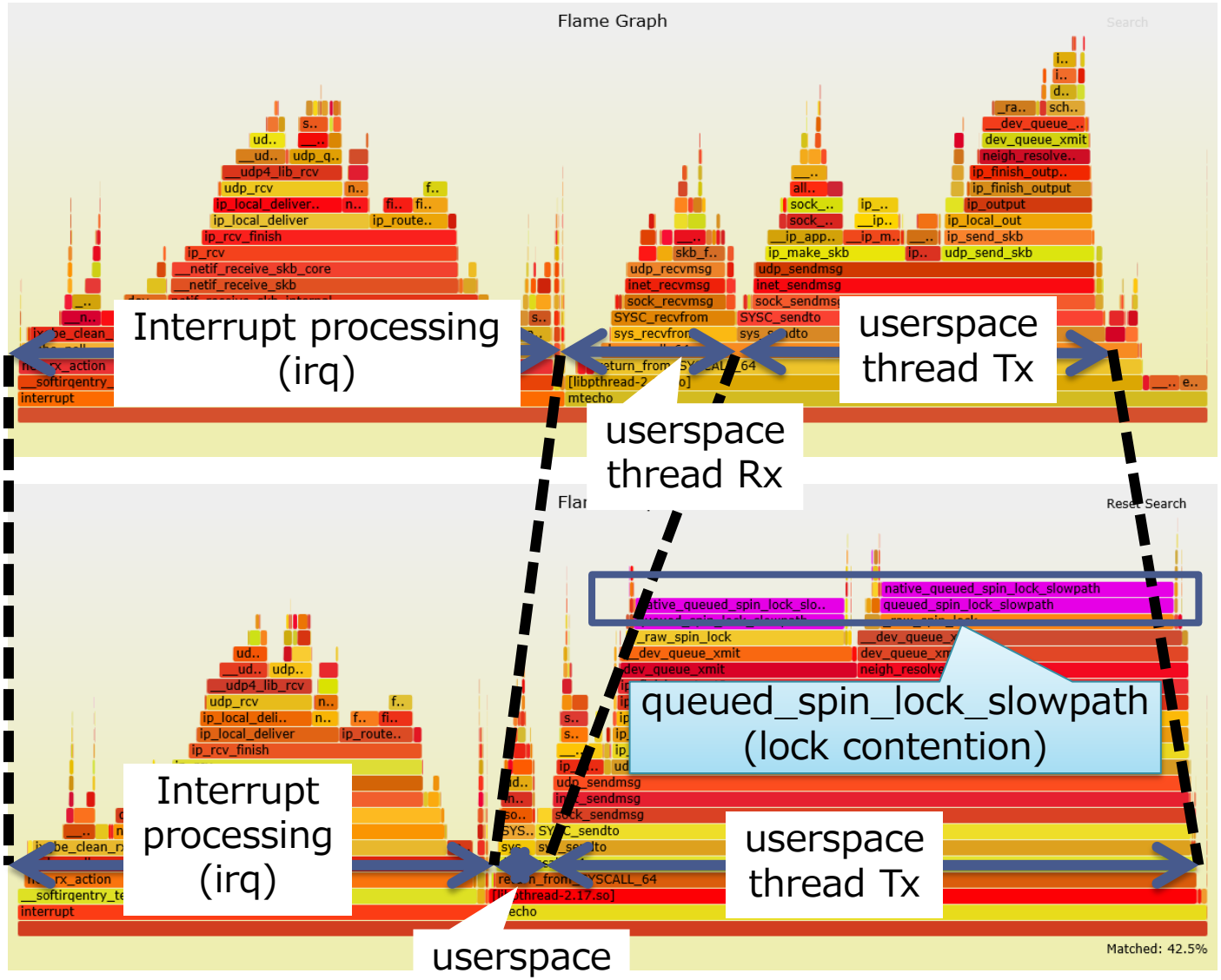
- **Try disabling it**

```
# for ((txq=0; txq<20; txq++)); do
>   echo 0 > /sys/class/net/ens1f0/queues/tx-$txq/xps_cpus
> done
```

  - Before: **5,050,000** tps (approx. **6710Mbps**)
  - After: **1,086,000** tps (approx. **1440Mbps**)

Disabling XPS

- **XPS enabled**
- **XPS disabled**



Enable XPS

- **Enable XPS again**

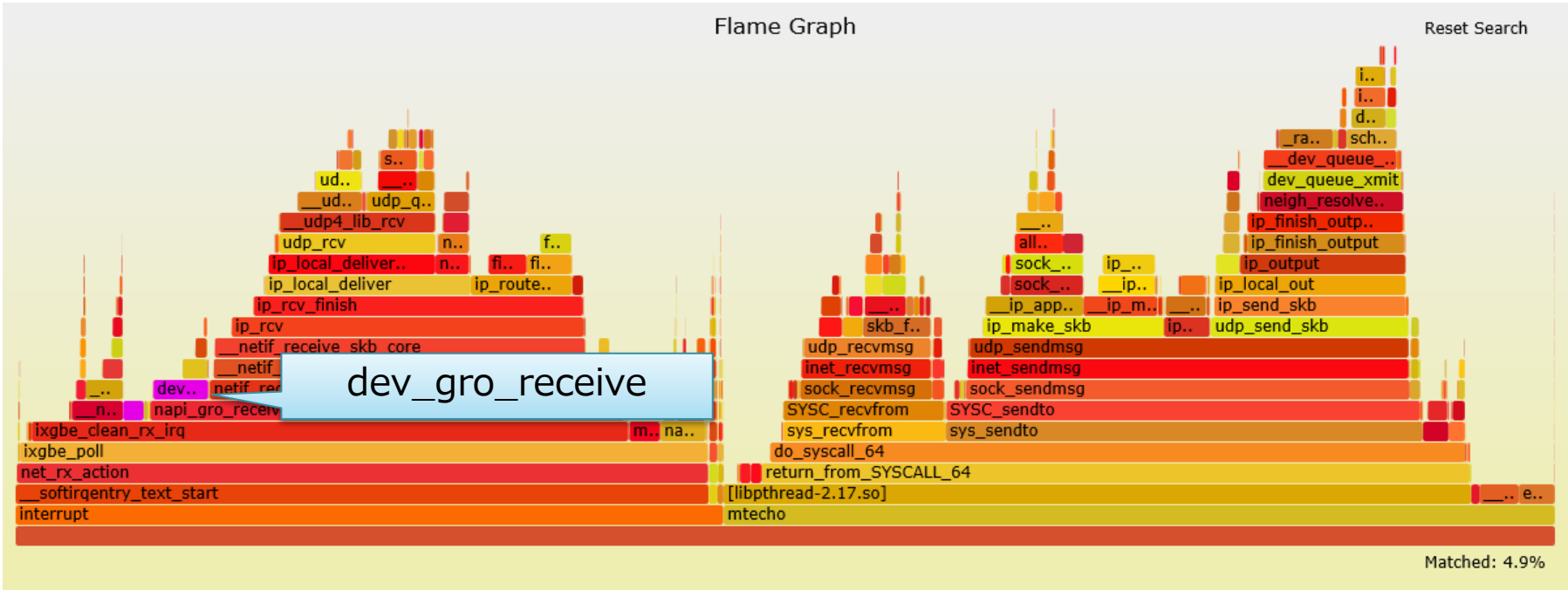
```
# echo 00001 > /sys/class/net/<NIC>/queues/tx-0/xps_cpus
# echo 00002 > /sys/class/net/<NIC>/queues/tx-1/xps_cpus
# echo 00004 > /sys/class/net/<NIC>/queues/tx-2/xps_cpus
# echo 00008 > /sys/class/net/<NIC>/queues/tx-3/xps_cpus
...
```

- **Although ixgbe can automatically set XPS, not all drivers can do that**
- **Make sure to check xps_cpus is configured**

Optimization per core

- **By making full use of multi-core with avoiding contention, we achieved**
  - **5,050,000** tps (approx. **6710Mbps**)
- **To get more performance, reduce overhead per core**

Optimization per core



- GRO is enabled by default
- Consuming 4.9% of CPU time

GRO

- **GRO is not applicable to UDP*1**
- **Disable it for UDP servers**

```
# ethtool -K <NIC> gro off
```

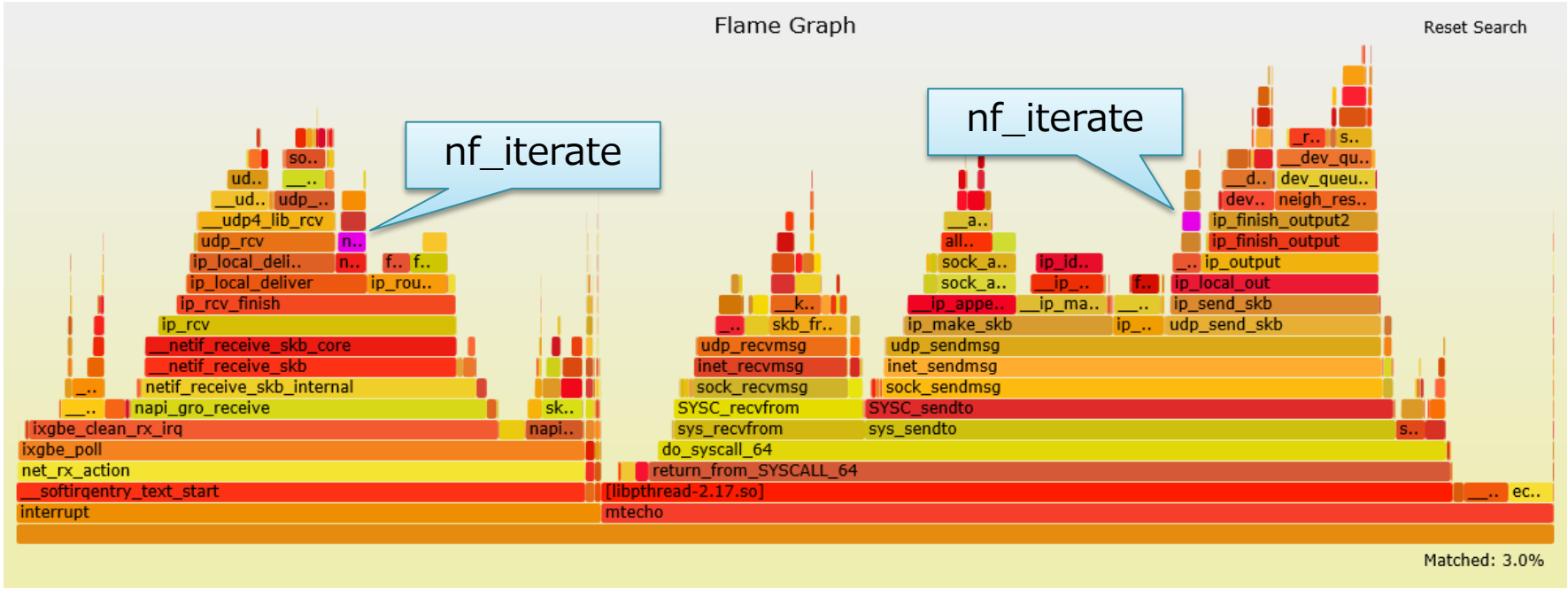
- **WARNING:**
  - Don't disable it if TCP performance matters
    - Disabling GRO makes TCP rx throughput miserably low
  - Don't disable it on KVM hypervisors as well
    - GRO boost throughput of tunneling protocol traffic as well as guest's TCP traffic on hypervisors

*1 Other than UDP tunneling, like VXLAN

Disable GRO

- **Perfomance change**
  - RSS (+XPS): 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: 5,050,000 tps (approx. 6710Mbps)
  - +Disable GRO: **5,180,000** tps (approx. **6880Mbps**)

Optimization per core



- **iptables-related processing (nf_iterate) is performed**
  - Although I have not added any rule to iptables
- **Consuming 3.00% of CPU time**

iptables (netfilter)

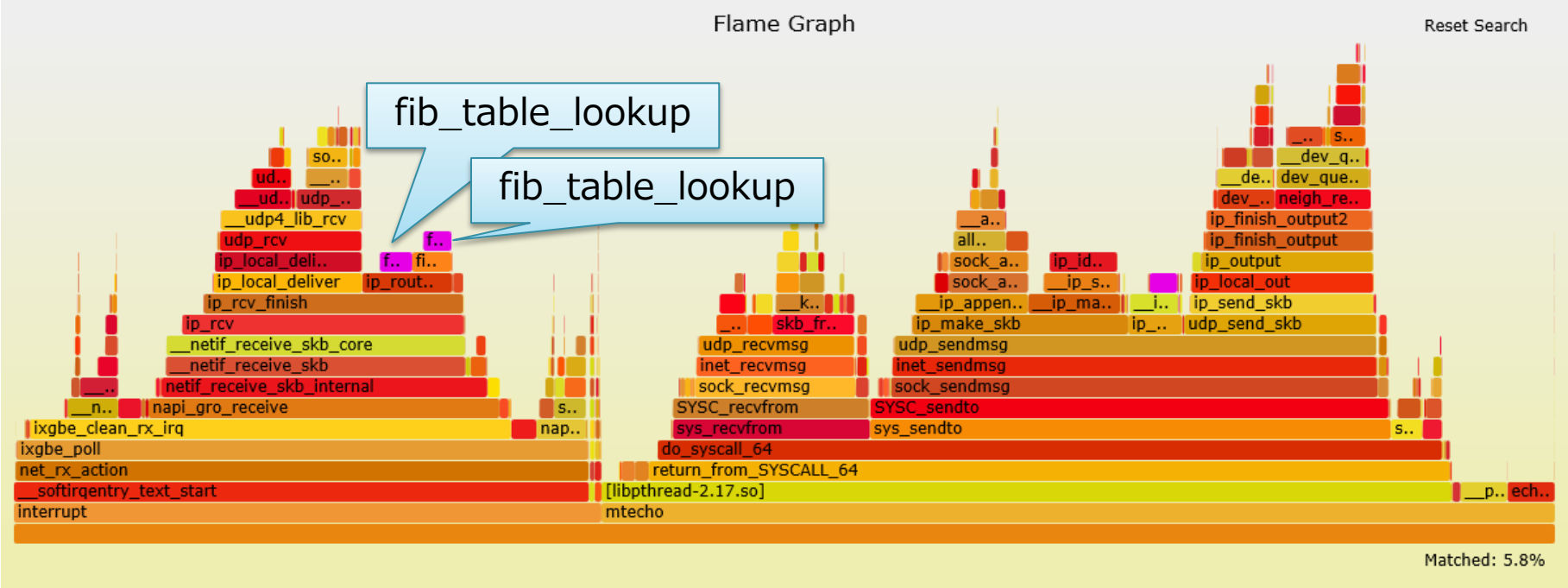
- With iptables kernel module loaded, even if you don't have any rules, it can incur some overhead
- Some distributions load iptables module even when you don't add any rule
- If you are not using iptables, unload the module

```
# modprobe -r iptable_filter
# modprobe -r ip_tables
```

Unload iptables

- **Perfomance change**
  - RSS (+XPS): 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: 5,050,000 tps (approx. 6710Mbps)
  - +Disable GRO: 5,180,000 tps (approx. 6880Mbps)
  - +Unload iptables: **5,380,000** tps (approx. **7140Mbps**)

Optimization per core



- On Rx, FIB (routing table) lookup is done twice
- Each is consuming 1.82%~ of CPU time

FIB lookup on Rx

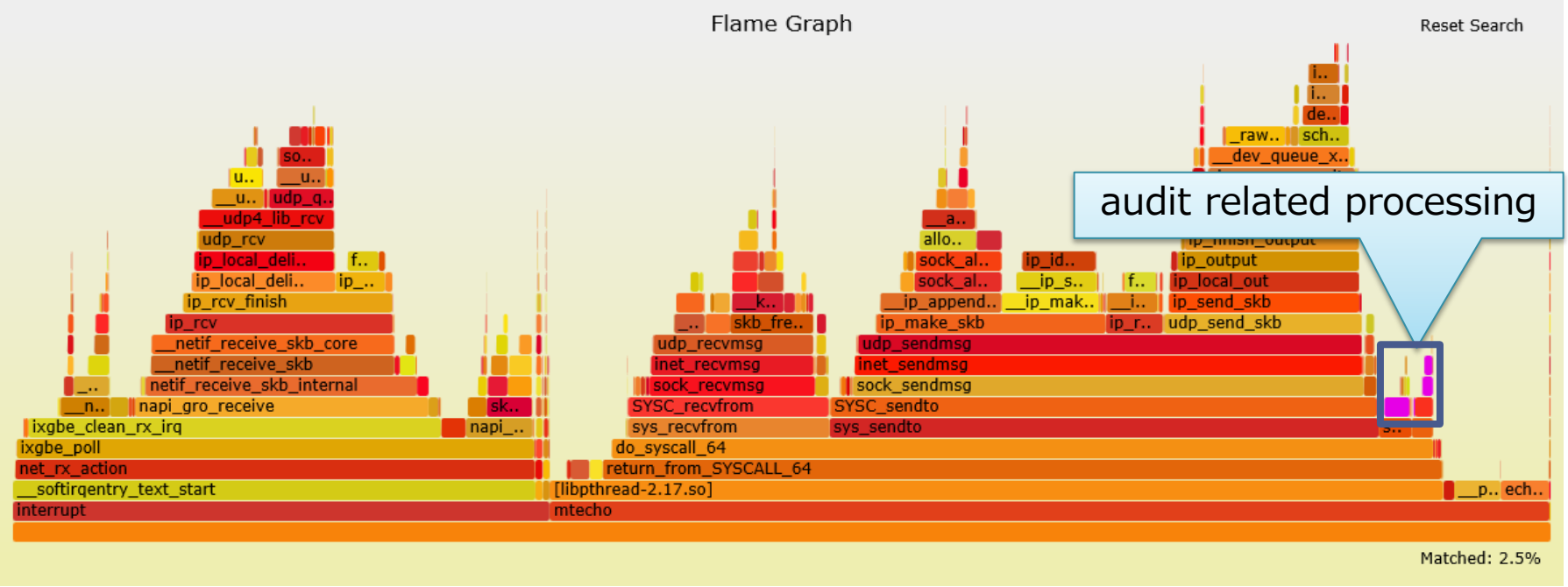
- **One of two times of table lookup is for validating source IP addresses**
  - Reverse path filter
  - Local address check
- **If you really don't need source validation, you can skip it**

```
# sysctl -w net.ipv4.conf.all.rp_filter=0
# sysctl -w net.ipv4.conf.<NIC>.rp_filter=0
# sysctl -w net.ipv4.conf.all.accept_local=1
```

Disable source validation

- **Perfomance change**
  - RSS (+XPS): 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: 5,050,000 tps (approx. 6710Mbps)
  - +Disable GRO: 5,180,000 tps (approx. 6880Mbps)
  - +Unload iptables: 5,380,000 tps (approx. 7140Mbps)
  - +Disable validation: **5,490,000** tps (approx. **7290Mbps**)

Optimization per core



- **Audit is a bit heavy when heavily processing packets**
- **Consuming 2.5% of CPU time**

Audit

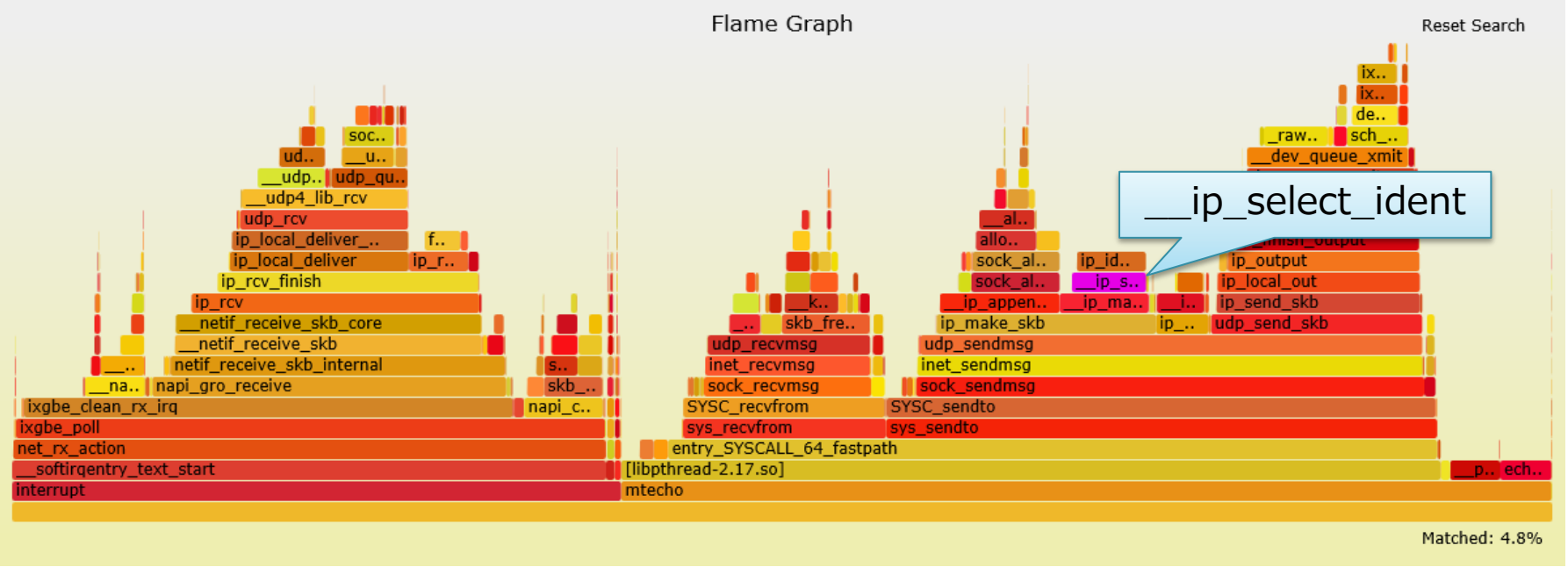
- **If you don't need audit, disable it**

```
# systemctl disable auditd
# reboot
```

Disable audit

- **Perfomance change**
  - RSS (+XPS): 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: 5,050,000 tps (approx. 6710Mbps)
  - +Disable GRO: 5,180,000 tps (approx. 6880Mbps)
  - +Unload iptables: 5,380,000 tps (approx. 7140Mbps)
  - +Disable validation: 5,490,000 tps (approx. 7290Mbps)
  - +Disable audit: **5,860,000** tps (approx. **7780Mbps**)

Optimization per core



- **IP ID field calculation (__ip_select_ident) is heavy**
- **Consuming 4.82% of CPU time**

IP ID field calculation

- **This is an environment-specific issue**
  - This happens if many clients has the same IP address
    - Cache contention by atomic operations
  - It is very likely you don't see this amount of CPU consumption without using tunneling protocol

- **If you really see this problem...**
  - You can skip it only if you never send over-mtu-sized packets
    - Though it is very strict

```
int pmtu = IP_PMTUDISC_DO;
setsockopt(sock, IPPROTO_IP, IP_MTU_DISCOVER, &pmtu, sizeof(pmtu));
```

Skip IP ID calculation

- **Perfomance change**
  - RSS (+XPS): 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: 5,050,000 tps (approx. 6710Mbps)
  - +Disable GRO: 5,180,000 tps (approx. 6880Mbps)
  - +Unload iptables: 5,380,000 tps (approx. 7140Mbps)
  - +Disable validation: 5,490,000 tps (approx. 7290Mbps)
  - +Disable audit: 5,860,000 tps (approx. 7780Mbps)
  - +Skip ID calculation: **6,010,000** tps (approx. **7980Mbps**)

Hyper threading

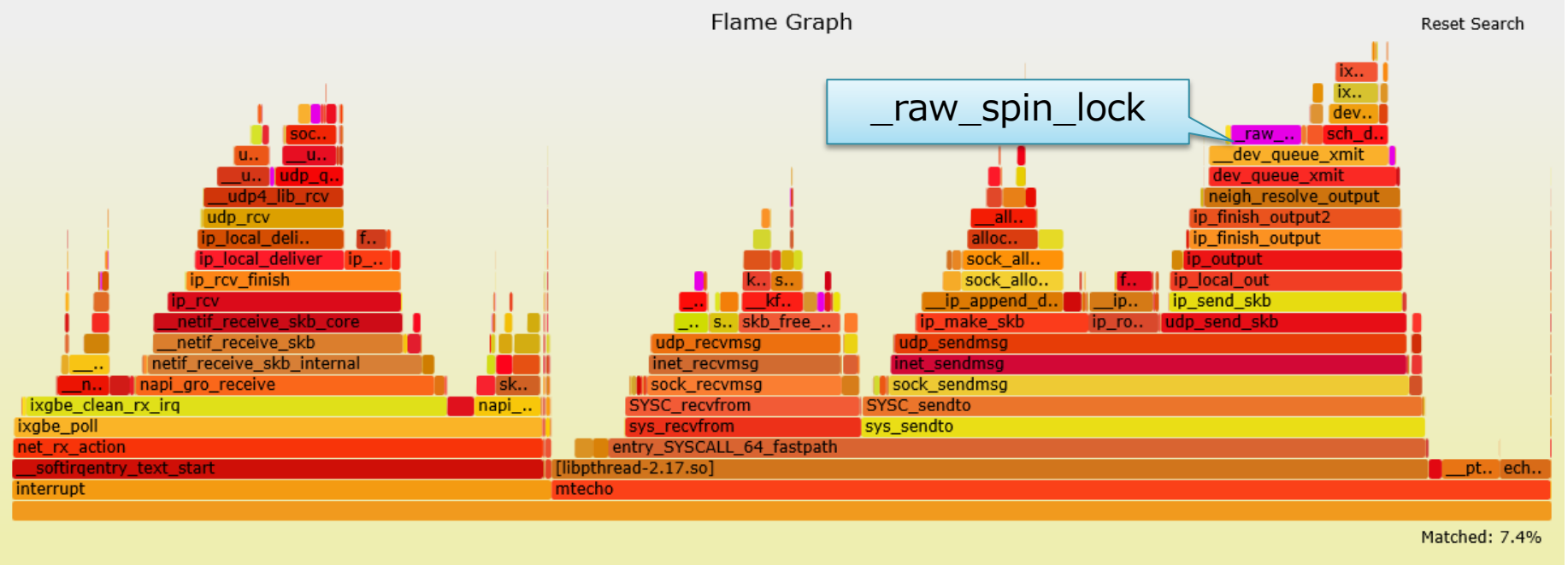
- **So far we have not enabled hyper threading**
- **It makes the number of logical cores 40**
  - Number of physical cores are 20 in this box
- **With 40 cores we need to rely more on RPS**
  - Remind: Max usable rx-queues == 16
- **Enable hyper-threading and set RPS on all rx-queues**
  - queue 0 -> core 0, 20
  - queue 1 -> core 1, 21
  - ...
  - queue 10 -> core 10, 16, 30
  - queue 11 -> core 11, 17, 31
  - ...

Hyper threading

- **Perfomance change**
  - RSS (+XPS): 270,000 tps (approx. 360Mbps)
  - +affinity_hint+RPS: 17,000 tps (approx. 23Mbps)
  - +SO_REUSEPORT: 2,540,000 tps (approx. 3370Mbps)
  - +SO_ATTACH_...: 4,250,000 tps (approx. 5640Mbps)
  - +Pin threads: 5,050,000 tps (approx. 6710Mbps)
  - +Disable GRO: 5,180,000 tps (approx. 6880Mbps)
  - +Unload iptables: 5,380,000 tps (approx. 7140Mbps)
  - +Disable validation: 5,490,000 tps (approx. 7290Mbps)
  - +Disable audit: 5,860,000 tps (approx. 7780Mbps)
  - +Skip ID calculation: 6,010,000 tps (approx. 7980Mbps)
  - +Hyper threading: **7,010,000** tps (approx. **9310Mbps**)

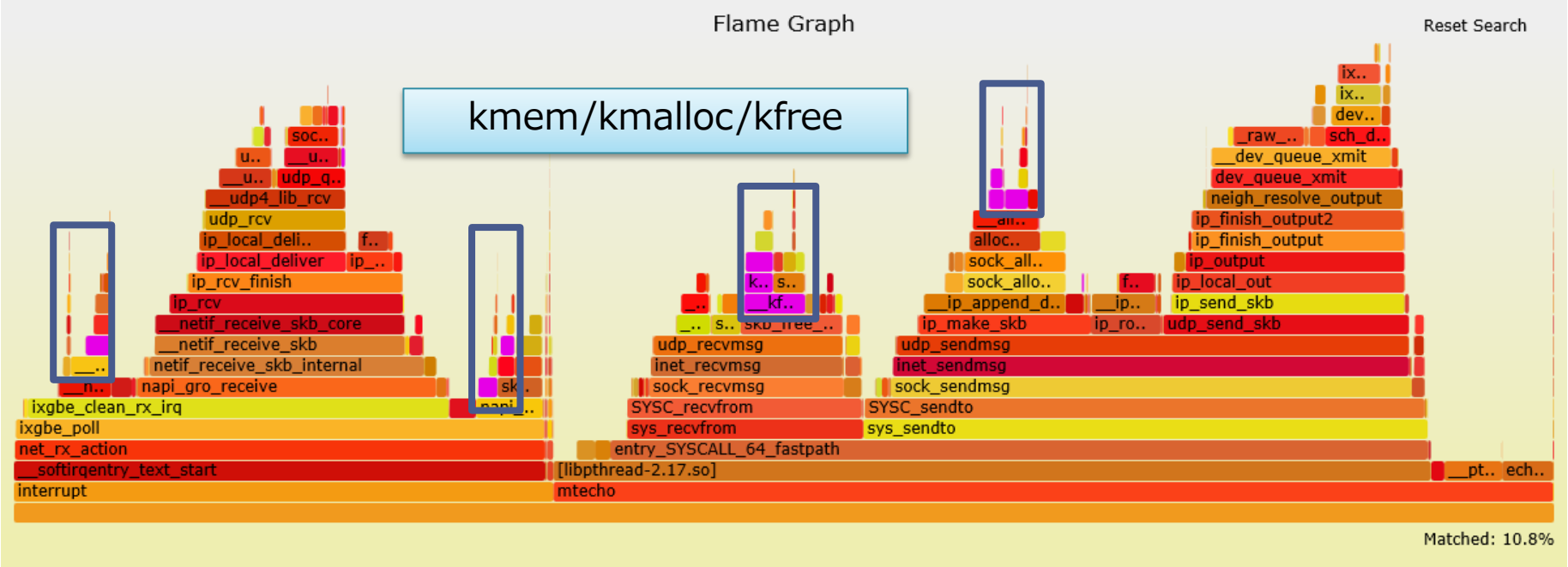
  - I guess more rx-queues would realize even better performance number

More hot spots



- Tx Qdisc lock (_raw_spin_lock) is heavy
- Not contended but involves many atomic operations
- Being optimized in Linux netdev community

More hot spots



- Memory alloc/free (slab)
- Being optimized in netdev community as well

Other challenges

- **Virtualization**
  - UDP servers as guests
  - Hypervisor can saturate CPUs or drop packets
  - We are going to investigate ways to boost performance in virtualized environment as well

Summary

- **For 100bytes, we can achieve almost 10G**
  - From: 270,000 tps (approx. 360Mbps)
  - To: 7,010,000 tps (approx. 9310Mbps)
    - Of course we need to take into account additional userspace work in real applications so this number is not applicable as is

- **To boost UDP performance**
  - Applications (Most important!)
    - implement SO_REUSEPORT
    - implement SO_ATTACH_REUSEPORT_EBPF/CBPF
    - These are useful for TCP listening sockets as well
  - OS settings
    - Check smp_affinity
    - Use RPS if rx-queues are not enough
    - Make sure XPS is configured
    - Consider other tunings to reduce per-core overhead
      - Disable GRO
      - Unload iptables
      - Disable source IP validation
      - Disable auditd
  - Hardware
    - Use NICs which have enough RSS rx-queues if possible (as many queues as core num)

Thank you!