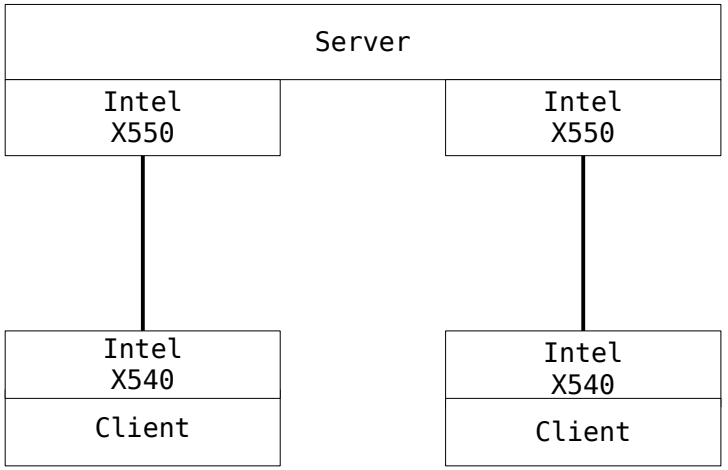
Improving the DragonFlyBSD Network Stack

Yanmin Qiao
sephe@dragonflybsd.org

DragonFlyBSD project

Nginx performance and latency evaluation: HTTP/1.1, 1KB web object, 1 request/connect



Server:
2x E5-2620v2, 32GB DDR3-1600, Hyperthreading enabled.

nginx:
Installed from dports.

nginx.conf:
Access log is disabled. 16K connections/worker.

15K concurrent connections on each client.

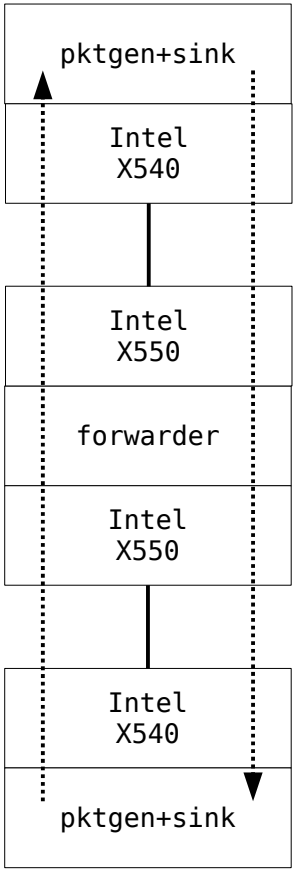
Client:
i7-3770, 16GB DDR3-1600, Hyperthreading enabled.

MSL on clients and server are changed to 20ms by:
route change -net net -msl 20

/boot/loader.conf:
kern.ipc.nmbclusters=524288

/etc/sysctl.conf:
machdep.mwait.CX.idle=AUTODEEP
kern.ipc.somaxconn=256
net.inet.ip.portrange.last=40000

IPv4 forwarding performance evalution: 64B packets



pktgen+sink:
i7-3770, 16GB DDR3-1600,
Hyperthreading enabled.

forwarder:
2x E5-2620v2, 32GB DDR3-1600,
Hyperthreading enabled.

/boot/loader.conf:
kern.ipc.nmbclusters=5242880

/etc/sysctl.conf:
machdep.mwait.CX.idle=AUTODEEP

Traffic generator:
DragonFlyBSD's in kernel packet generator, which has no issue to generate 14.8Mpps.

Traffic sink:
ifconfig ix0 monitor

Traffic:
Each pktgen targets 208 IPv4 addresses, which are mapped to one link layer address on 'forwarder'.

Performance counting:
Only packets sent by the forwarder are counted.

TOC

1. Use all available CPUs for network processing.
2. Direct input with polling(4).
3. Kqueue(2) accept queue length report.
4. Sending aggregation from the network stack.
5. Per CPU IPFW states.

Use all available CPUs for network processing

Issues of using only power-of-2 count of CPUs.

- On a system with 24 CPUs, only 16 CPUs will be used.

  Bad for kernel-only applications: forwarding and bridging.

- Performance and latency of network heavy userland applications can not be fully optimized.

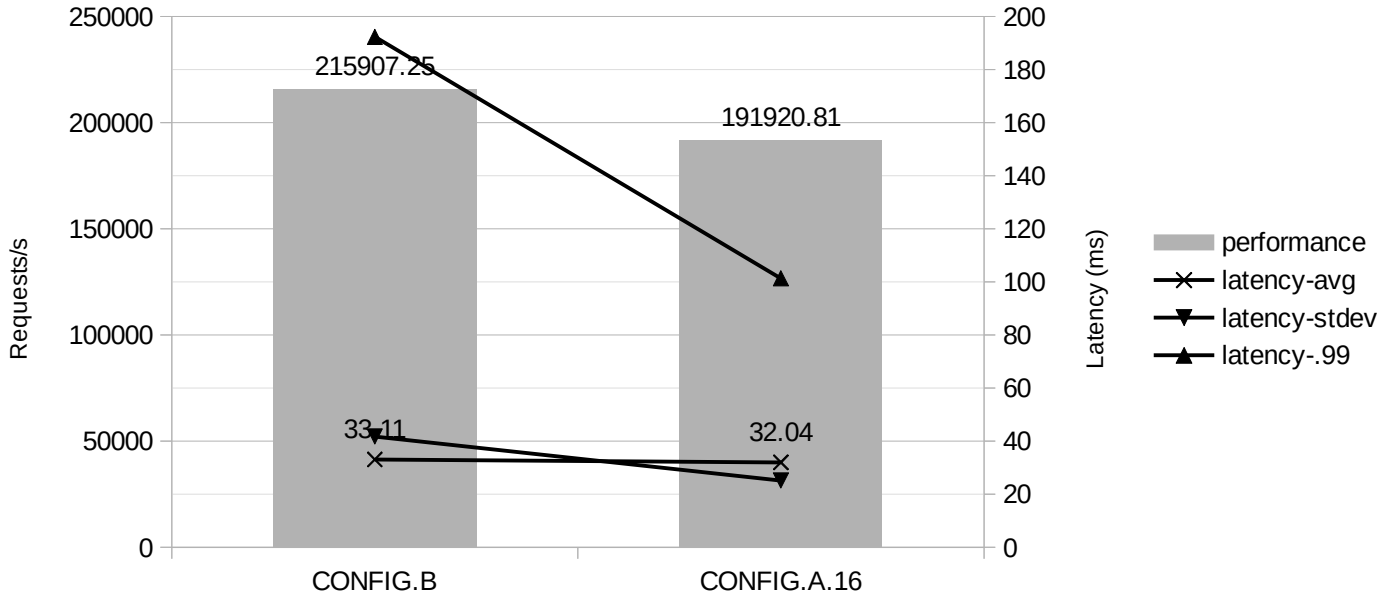
- 6, 10, 12 and 14 cores per CPU package are quite common these days.

Use all available CPUs for network processing (24 CPUs, 16 netisrs)

CONFIG.B: 32 nginx workers, no affinity.

CONFIG.A.16: 16 nginx workers, affinity.

<u>None of them</u> are optimal.



Use all available CPUs for network processing

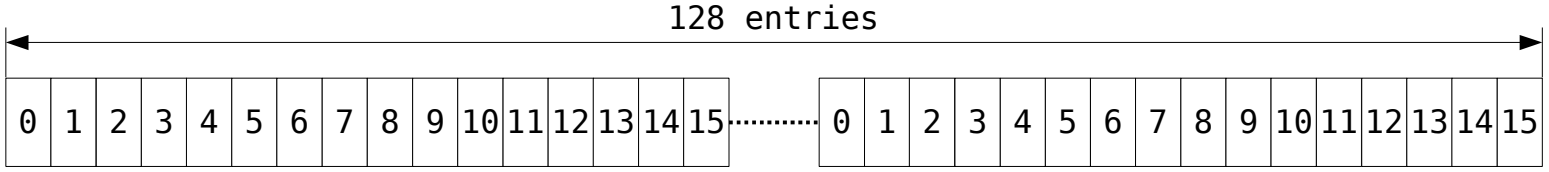
Host side is easy

(RSS_hash & 127) % ncpus

How NIC RSS redirect table should be setup?

Simply do this?

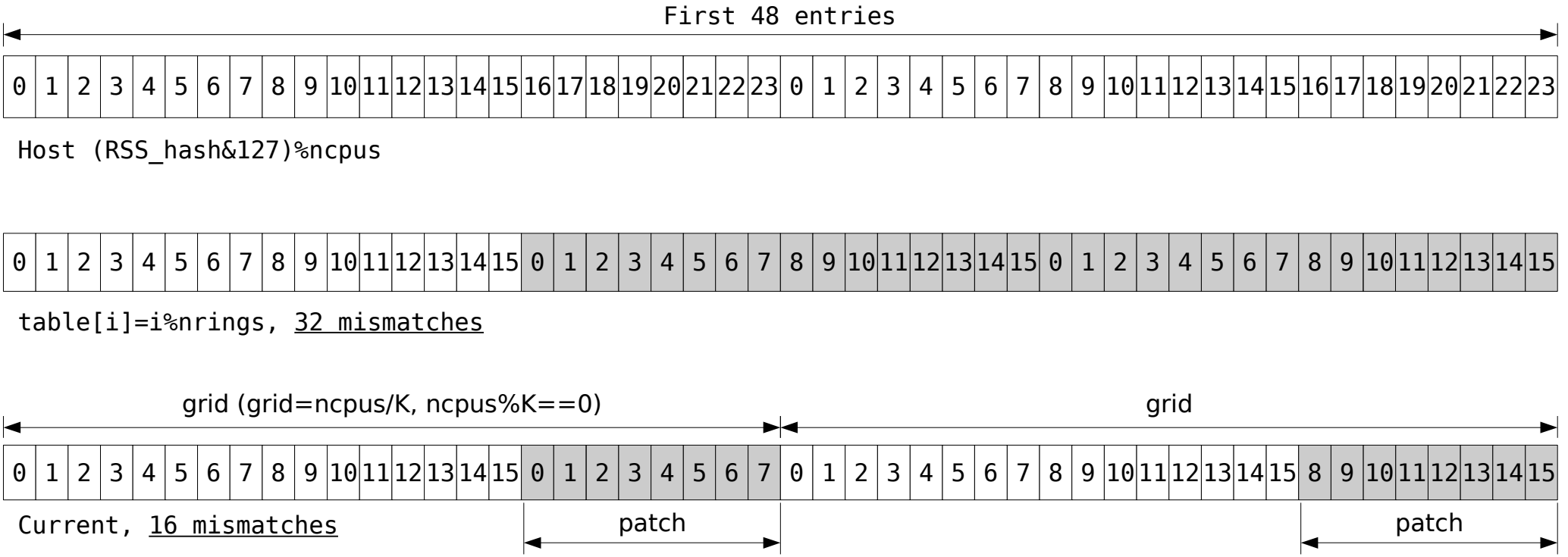
table[i] = i % nrings, i.e.



Use all available CPUs for network processing (NIC RSS redirect table)

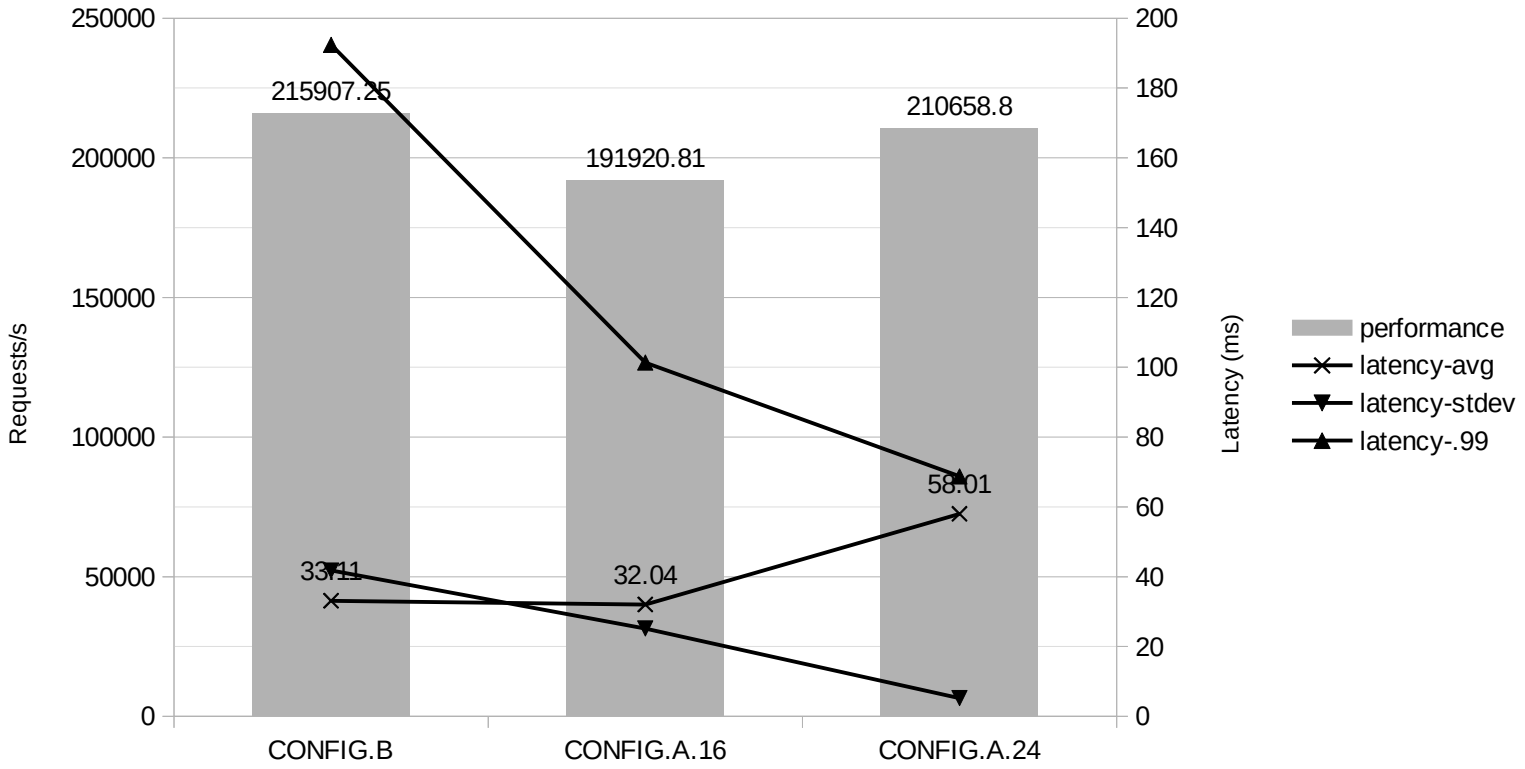
Issue with table[i]=i%nrings

e.g. 24 CPUs, 16 RX rings.

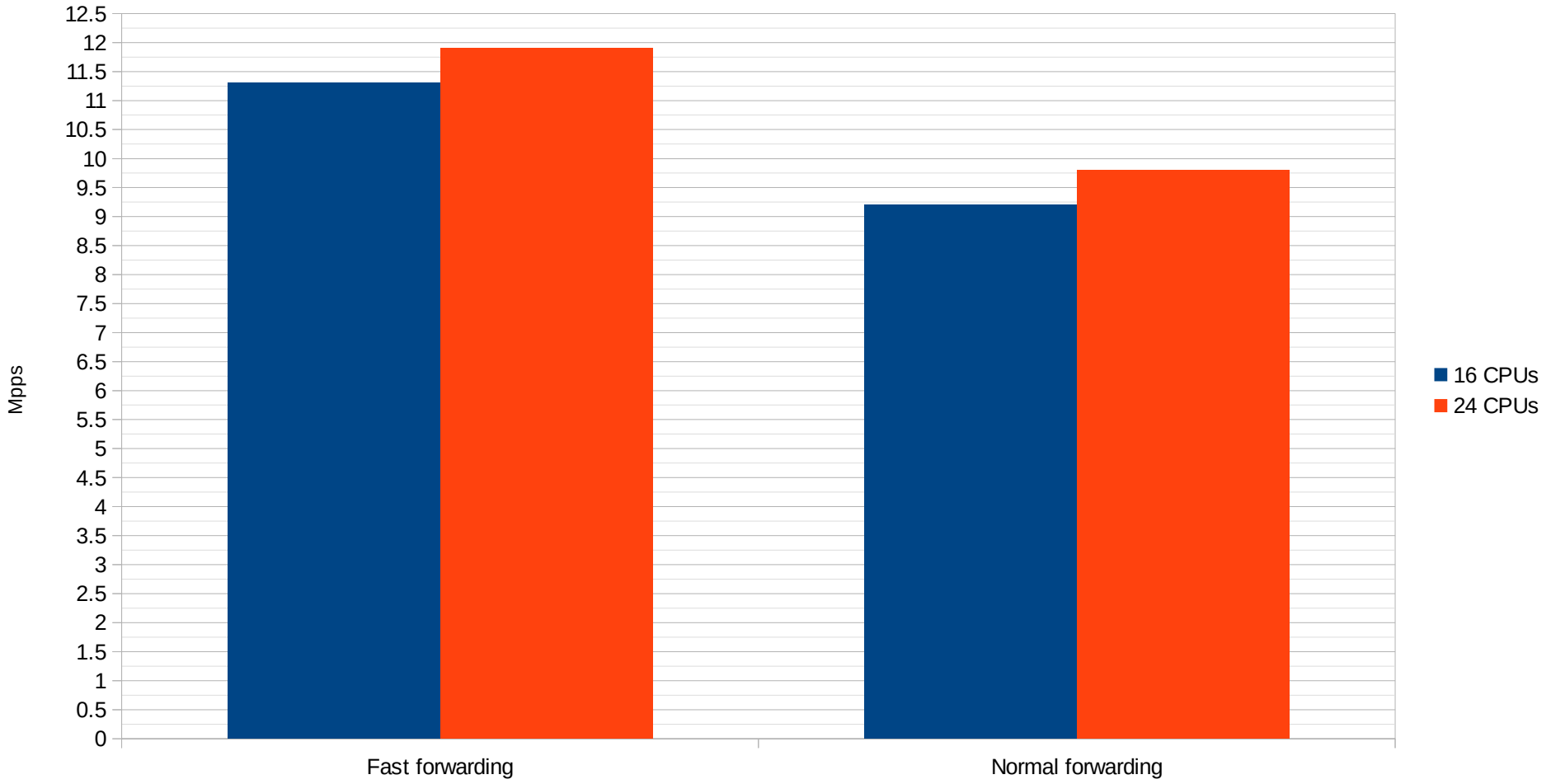


Use all available CPUs for network processing (result: nginx, 24 CPUs)

CONFIG.A.24: 24 nginx workers, affinity



Use all available CPUs for network processing (result: forwarding)



Use all available CPUs for network processing

if_ringmap APIs for drivers

if_ringmap_alloc(), if_ringmap_alloc2(), if_ringmap_free()

Allocate/free a ringmap. **if_ringmap_alloc2()** for NICs requiring power-of-2 count of rings, e.g. *jme(4)*.

if_ringmap_align()

If N_txrings!=N_rxrings, it makes sure that TX ring and the respective RX ring are assigned to the same CPU. Required by *bce(4)* and *bnx(4)*.

if_ringmap_match()

If N_txrings!=N_rxrings, it makes sure that distribution of TX rings and RX rings is more even. Used by *igb(4)* and *ix(4)*.

if_ringmap_cpumap(), if_ringmap_rdrtable()

Get ring CPU binding information, and RSS redirect table.

TOC

1. Use all available CPUs for network processing.
2. Direct input with polling(4).
3. Kqueue(2) accept queue length report.
4. Sending aggregation from the network stack.
5. Per CPU IPFW states.

Direct input with polling(4)

- Hold RX ring serializer.

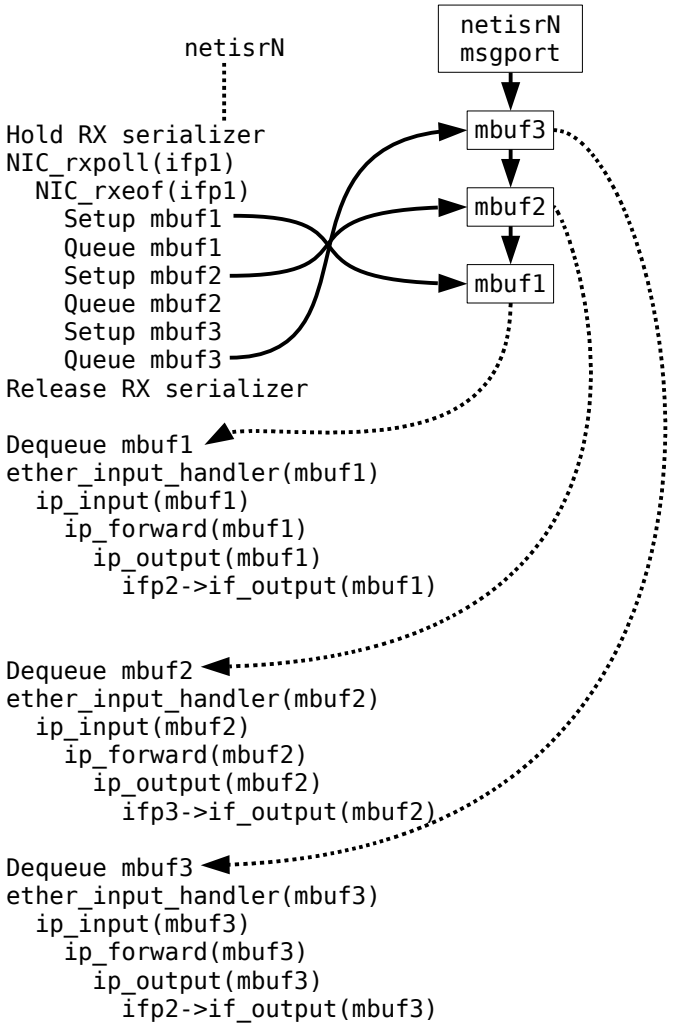
Mainly for stability,

i.e. bring down NIC when there are heavy RX traffic.

- Have to requeue the mbufs.

To avoid dangerous deadlock against RX ring serializer.

Excessive L1 cache eviction and refetching.

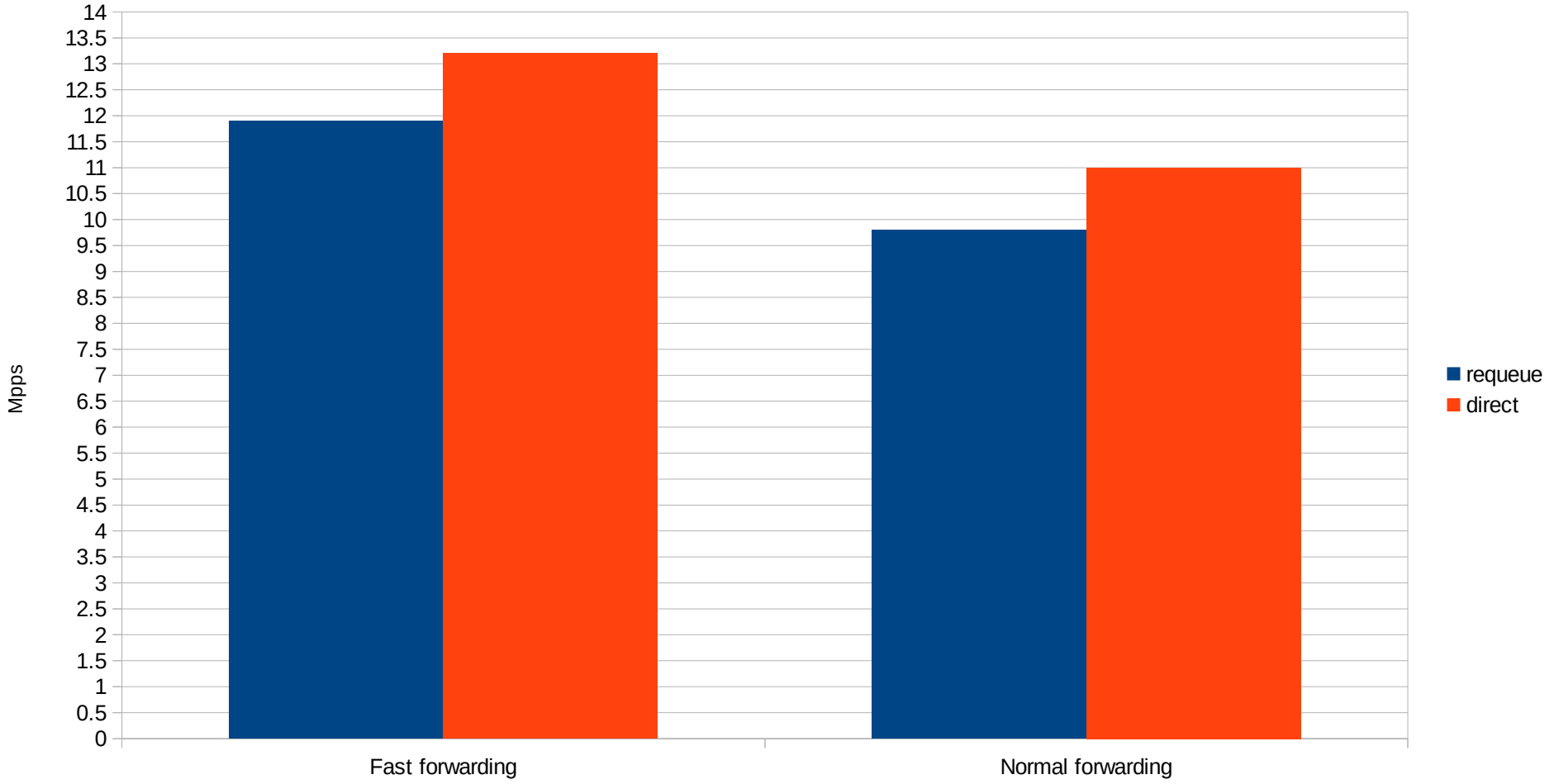


Direct input with polling(4)

- Driver synchronizes netisrs at stop time.
- Then holding RX ring serializer is no longer needed.
- Mbuf requeuing is not necessary.

```
                netisrN
                  ⋮
NIC_rxpoll(ifp1)
  NIC_rxeof(ifp1)
    Setup mbuf1
    ether_input_handler(mbuf1)
      ip_input(mbuf1)
        ip_forward(mbuf1)
          ip_output(mbuf1)
            ifp2->if_output(mbuf1)

    Setup mbuf2
    ether_input_handler(mbuf2)
      ip_input(mbuf2)
        ip_forward(mbuf2)
          ip_output(mbuf2)
            ifp3->if_output(mbuf2)

    Setup mbuf3
    ether_input_handler(mbuf3)
      ip_input(mbuf3)
        ip_forward(mbuf3)
          ip_output(mbuf3)
            ifp2->if_output(mbuf3)
```

Direct input with polling(4) (result: forwarding)



TOC

1. Use all available CPUs for network processing.
2. Direct input with polling(4).
3. Kqueue(2) accept queue length report.
4. Sending aggregation from the network stack.
5. Per CPU IPFW states.

Kqueue(2) accept queue length report

Applications, e.g. nginx, utilize kqueue(2) reported accept queue length like this:

```
do {
    s = accept(ls, &addr, &addrlen);
    /* Setup accepted socket. */
} while (--accept_queue_length);
```

The Setup accepted socket part:

- Time consuming.
- Destabilize and increase request handling latency.

Kqueue(2) accept queue length report

Backlog of listen(2) socket.

Major factor affects kqueue(2) accept queue length report.

Low listen(2) socket backlog.

e.g. 32, excessive connection drops.

High listen(2) socket backlog.

e.g. 256, destabilize and increase request handling latency.

Kqueue(2) accept queue length report

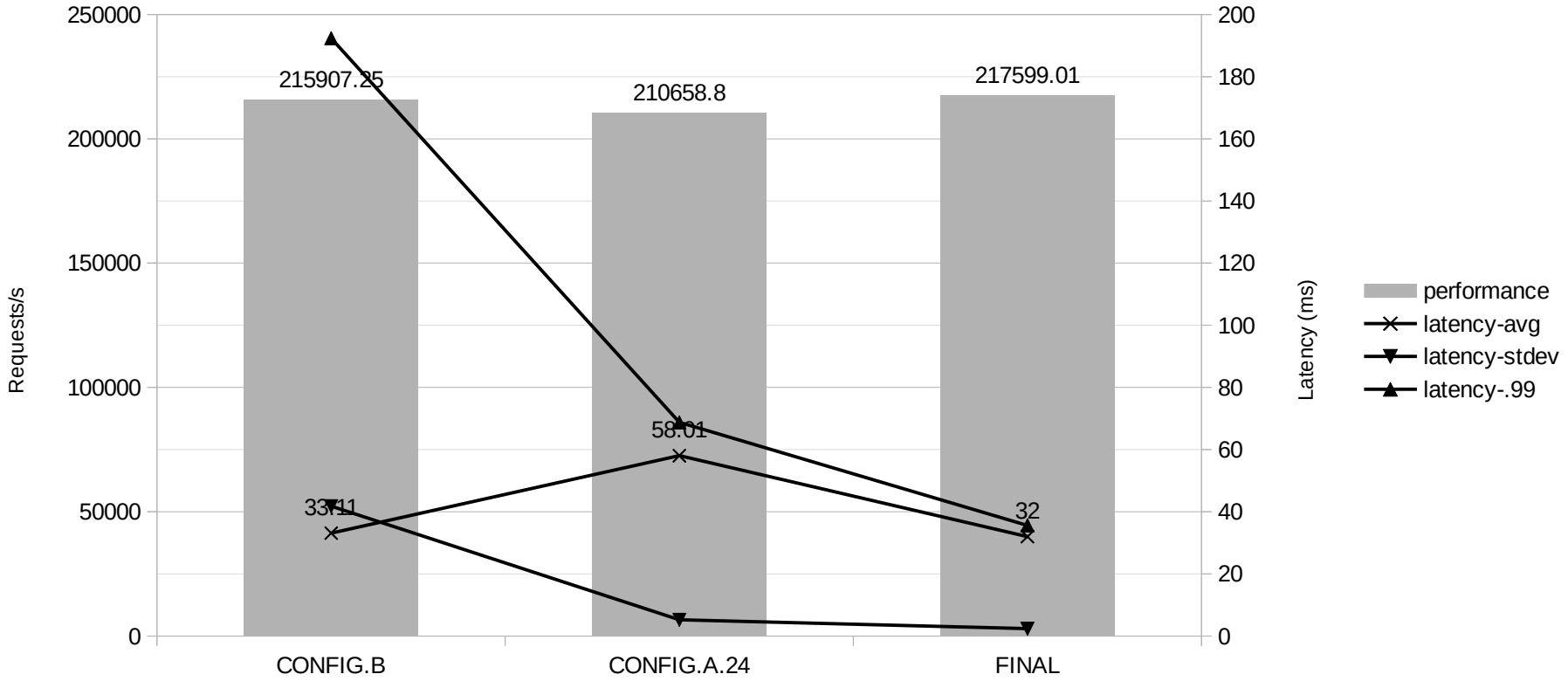
The solution:

- Leave listen(2) backlog high, e.g. 256.

- Limit the accept queue length from kqueue(2), e.g. 32. Simple one liner change.

Kqueue(2) accept queue length report (result: nginx, 24 CPUs)

FINAL: 24 nginx workers, affinity, accept queue length reported by kqueue(2) is limited to 32.



TOC

1. Use all available CPUs for network processing.
2. Direct input with polling(4).
3. Kqueue(2) accept queue length report.
4. Sending aggregation from the network stack.
5. Per CPU IPFW states.

Sending aggregation from the network stack

- High performance drivers have already deployed mechanism to reduce TX ring doorbell writes.

- The effectiveness of the drivers' optimization depends on how network stack feeds the packets.

Sending aggregation from the network stack

Preconditions:

- All packet sending runs in netisrs.

- Ordered ‘rollup’:

Protocols register ‘rollup’ function.

TCP ACK aggregation ‘rollup’ with higher priority.

Sending aggregation ‘rollup’ with lower priority.

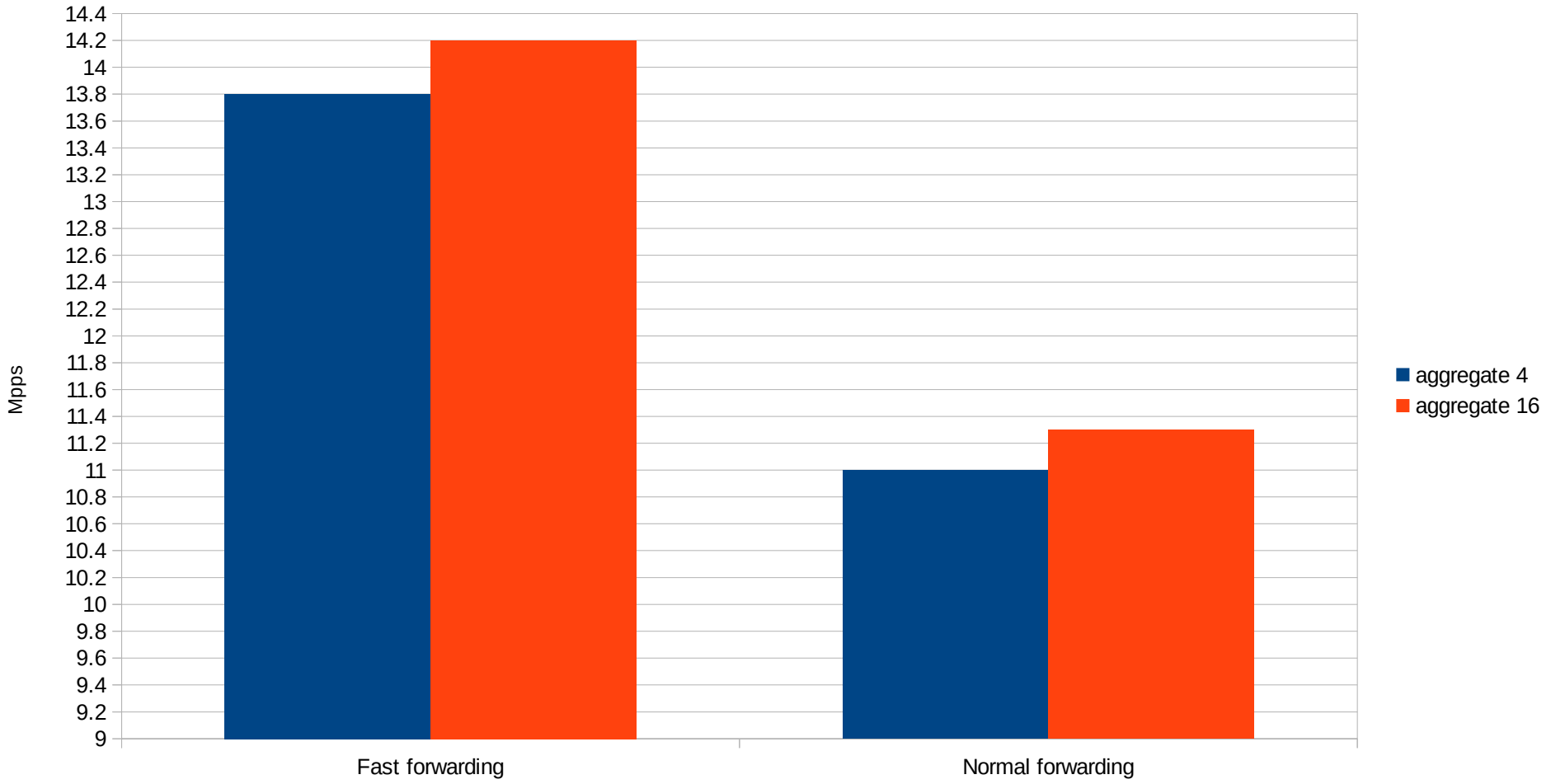
Rollups are called when:

No more netmsg.

32 netmsgs are handled.

```
                    netisrN
                       ⋮
Dequeue ifp1 RX ring[N] polling msg
NIC_rxpoll(ifp1, RX ring[N])
  NIC_rxeof(ifp1, RX ring[N])
    Setup mbuf1
    ether_input_handler(mbuf1)
      ip_input(mbuf1)
        ip_forward(mbuf1)
          ip_output(mbuf1)
            ifp2->if_output(mbuf1)
              Queue mbuf1 to ifp2.IFQ[N]
    Setup mbuf2
    ether_input_handler(mbuf2)
      ip_input(mbuf2)
        ip_forward(mbuf2)
          ip_output(mbuf2)
            ifp3->if_output(mbuf2)
              Queue mbuf2 to ifp3.IFQ[N]

netisrN.msgport has no more msgs

rollup1()
  // Flush pending aggregated TCP ACKs
  tcp_output(tcpcb1)
    ip_output(mbuf3)
      ifp2->if_output(mbuf3)
        Queue mbuf3 to ifp2.IFQ[N]
  tcp_output(tcpcb2)
    ip_output(mbuf4)
      ifp3->if_output(mbu4)
        Queue mbuf4 to ifp3.IFQ[N]

rollup2()
  // Flush pending mbufs on IFQs
  ifp2->if_start(ifp2.IFQ[N])
    Put mbuf1 to ifp2 TX ring[N]
    Put mbuf3 to ifp2 TX ring[N]
    Write ifp2 TX ring[N] doorbell reg
  ifp3->if_start(ifp3.IFQ[N])
    Put mbuf2 to ifp3 TX ring[N]
    Put mbuf4 to ifp3 TX ring[N]
    Write ifp3 TX ring[N] doorbell reg

Wait for more msgs on netisrN.msgport
```

Sending aggregation from the network stack (result: forwarding)

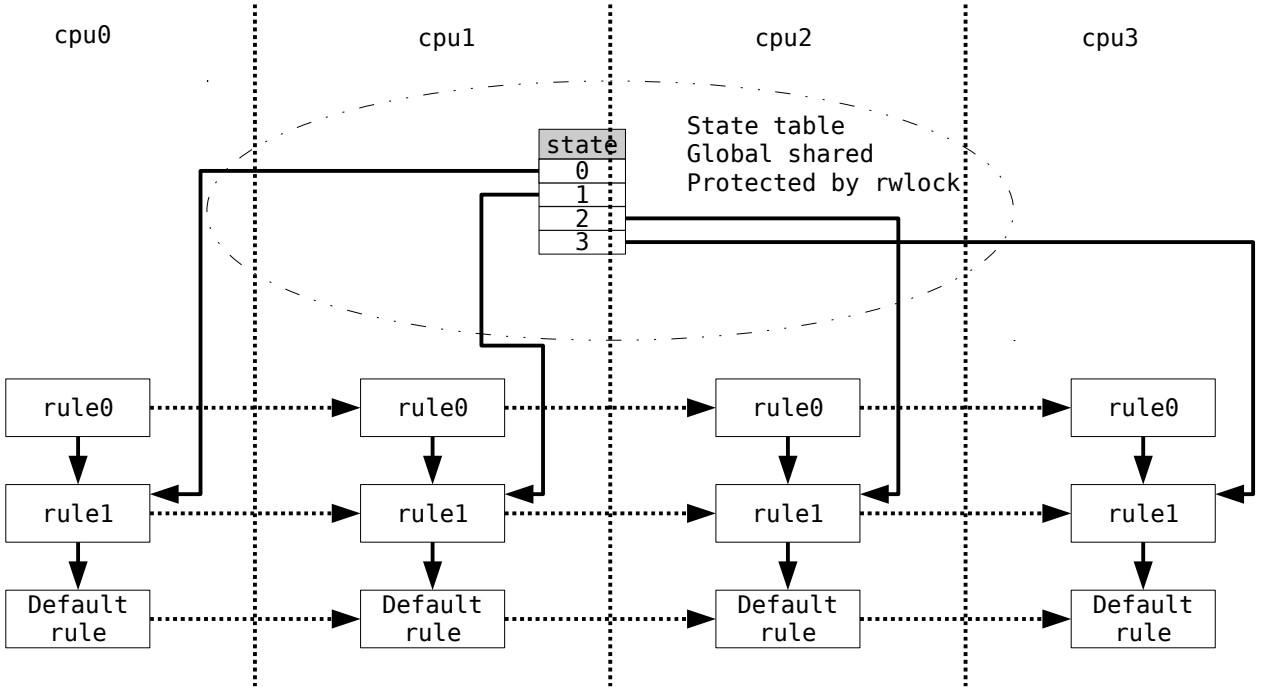


TOC

1. Use all available CPUs for network processing.
2. Direct input with polling(4).
3. Kqueue(2) accept queue length report.
4. Sending aggregation from the network stack.
5. Per CPU IPFW states.

Per CPU IPFW states

IPFW states were rwlocked, even if IPFW rules were per CPU.



Per CPU IPFW states

- **UDP was RSS hash aware in 2014.**

  Prerequisite for per CPU IPFW state.

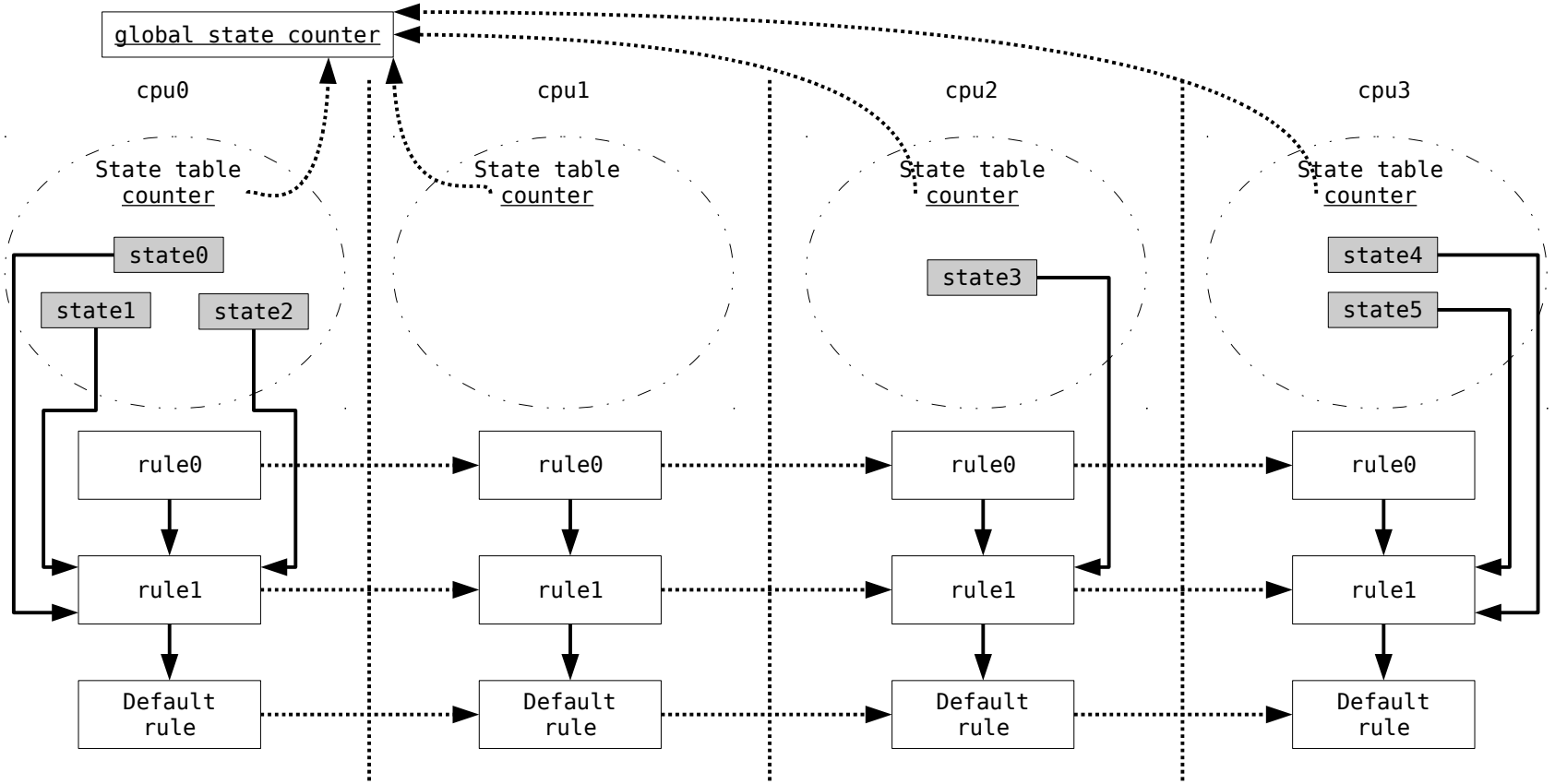
- **Rwlocked IPFW states work pretty well, if the states are relatively stable.**

  e.g. long-lived HTTP/1.1 connections.

- **Rwlocked IPFW states suffer a lot for short lived states; r/w contention.**

  e.g. short-lived HTTP/1.1 connections.

Per CPU IPFW states



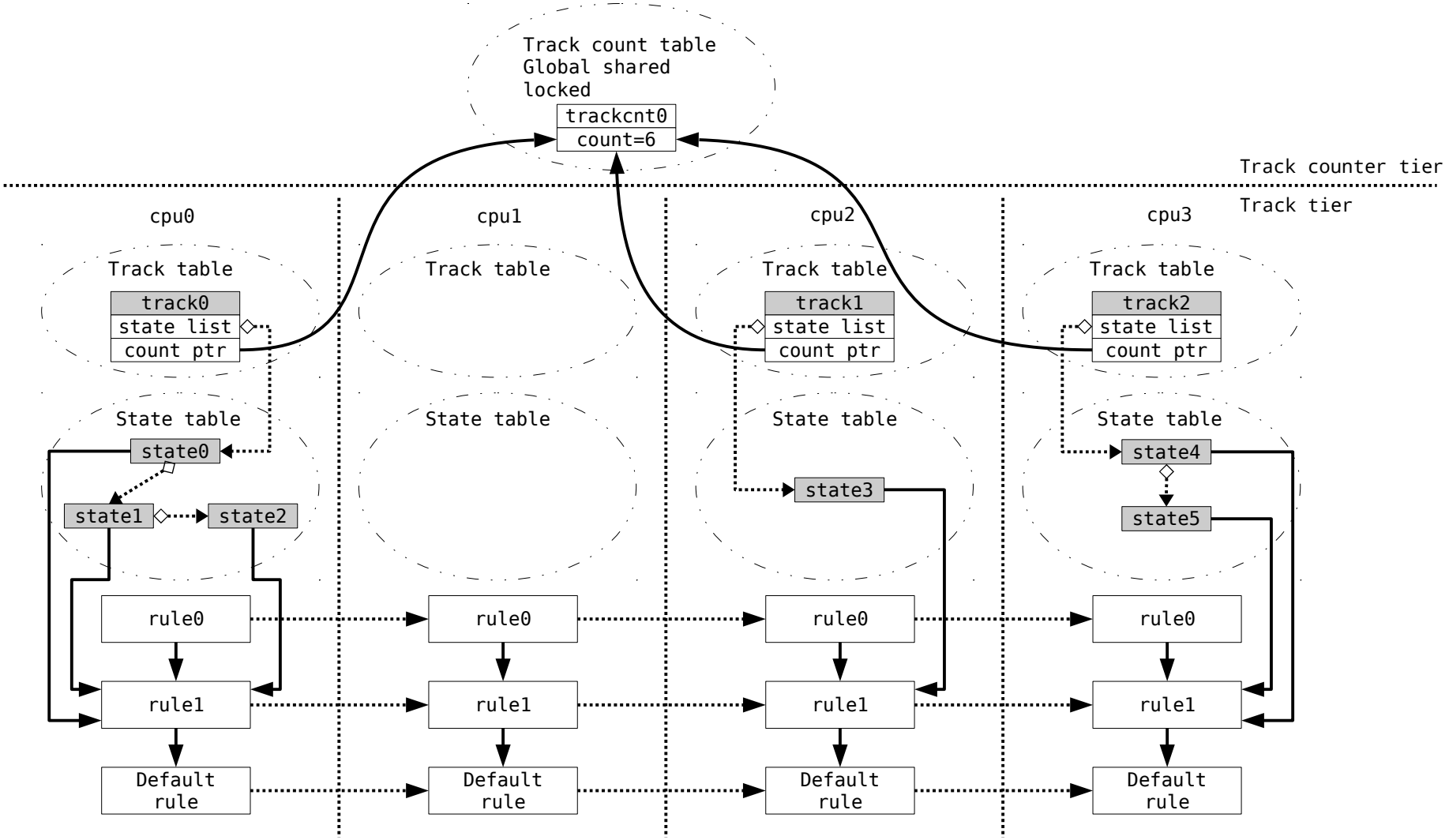
Per CPU IPFW states

Loose state counter update

- Implemented by Matthew Dillon for DragonFlyBSD's slab Allocator. It avoids expensive atomic operations.
- Per CPU state counter.

  Updated when states are installed/removed from the respective CPU.
- Per CPU state counter will be added to global state counter, once it is above a threshold.
- Per CPU state counters will be re-merged into global state counter, if the global state counter goes beyond the limit.
- Allow 5% overflow.

Per CPU IPFW states (limit rule)

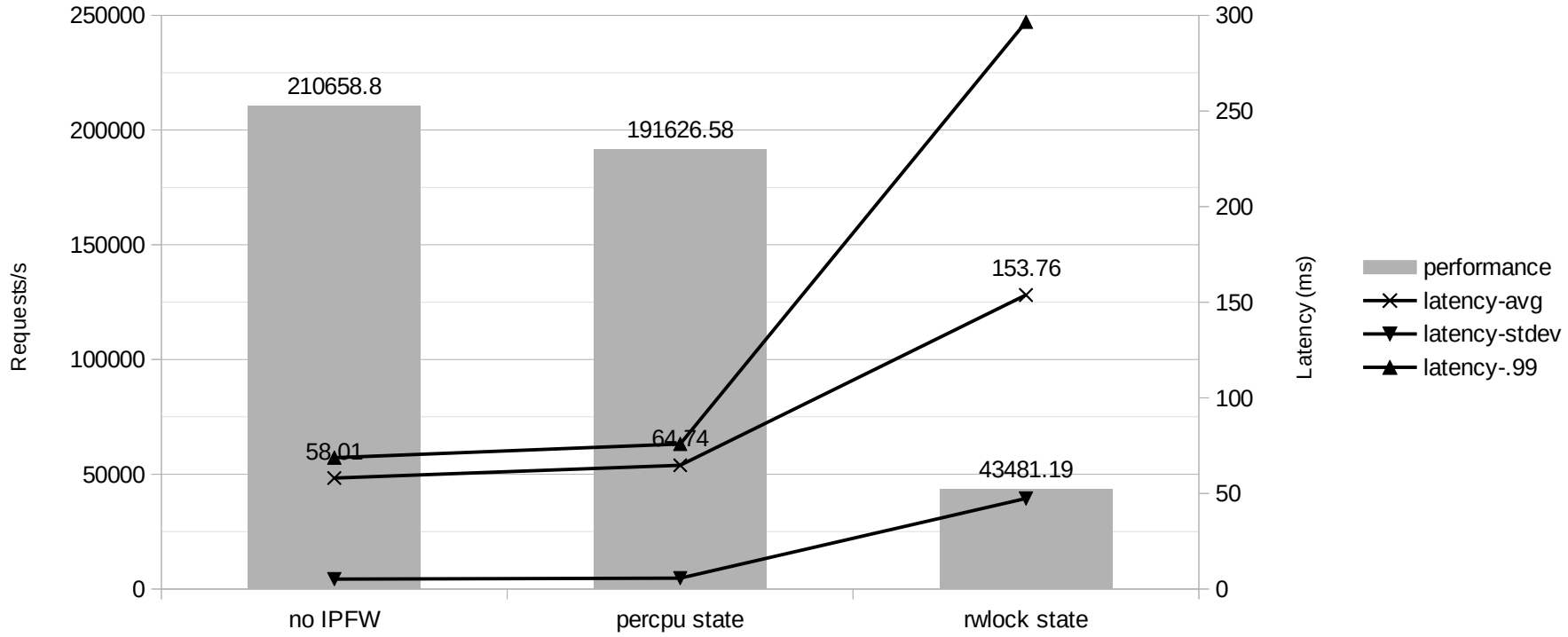


Per CPU IPFW states (result: nginx)

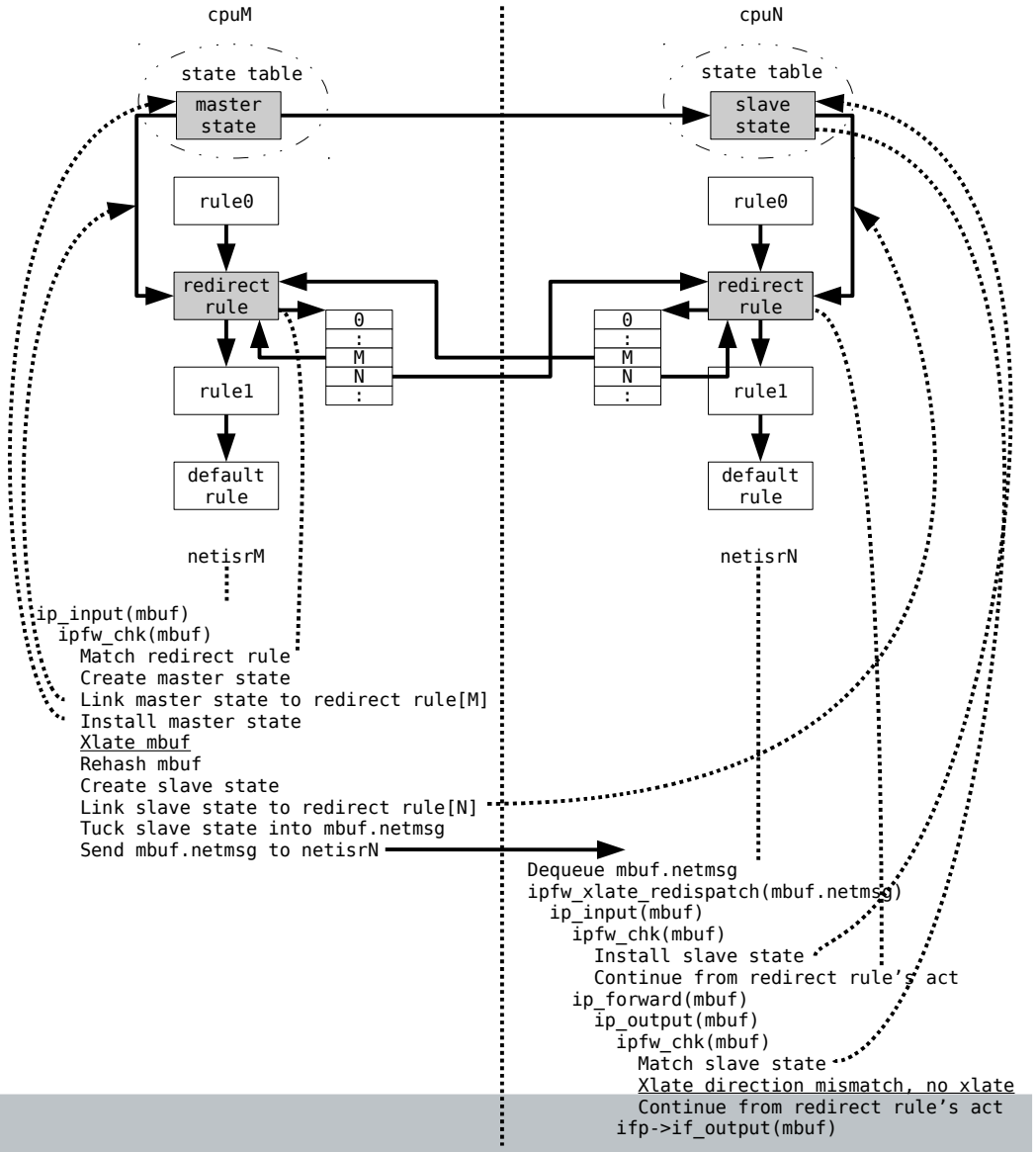
On server running nginx:

ipfw add 1 check-state

ipfw add allow tcp from any to me 80 setup keep-state

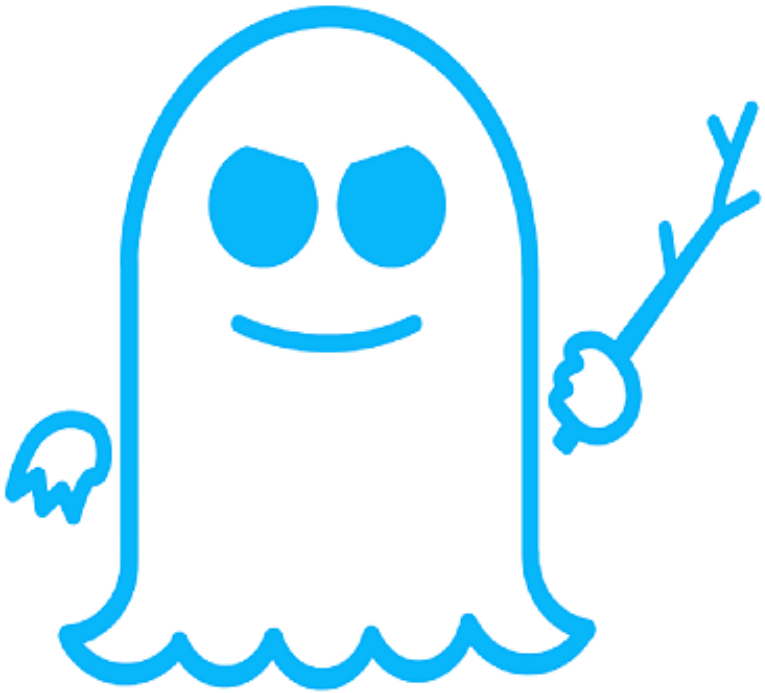


Per CPU IPFW states (redirect)



Performance Impact of

MELTDOWN



SPECTRE

Performance Impact of Meltdown and Spectre