Tuning FreeBSD for routing and firewalling

Olivier Cochard-Labbé, olivier@FreeBSD.org

AsiaBSDCon 2018

ABSTRACT

FreeBSD[1] is often used as a router or a firewall, but the vast majority of tuning guides available for this use case doesn't explain in detail how to calculate each value to be tuned. This study, after describing how to bench a router and the most important basic concepts to understand, demonstrate the benefit of tuning major parameters to obtain the best routing and firewalling performance with FreeBSD 11.1-RELEASE. This study is written by system administrators for system administrators audience: Optimisation will be done by configuration changes and using existing patches only. No kernel coding skills are needed.

I. BASIC CONCEPTS

A) *Benchmarking a router*

The two main functions of a router are:

- Forwarding packets between its interfaces;
- Maintaining routing table using some routing protocols.

This study focuses only on optimising the forwarding rate: Maintaining the routing table belongs to the user-land daemons and is excluded from this study.
The only metric measured for all this study will be the packet forwarding speed using packets-per-second (pps) unit.

B) *Differences with RFC 2544*

RFC 2544[2], Benchmarking Methodology for Network Interconnect Devices, is a well-known reference, but this study will not follow all recommendations given by this RFC for a simplest and faster methodology.
Here are some main divergences:

- Multiple frame size: In this paper, only the worst case matters, which is using the smallest Ethernet frame size. In this document one frame = one packet and unit fps=pps.
- Throughput is defined as the maximum frame rate supported by the DUT (device under test) without any drop: In this document the throughput is the outgoing forwarded frame rate when receiving at the maximum line rate.
- Bidirectional traffic: To simplify methodology, the bench labs described here generates only unidirectional traffic.

C) *Ethernet line rate references*

The first reference to know is the maximum Ethernet line rate [3] (implying smallest frame size):

- Gigabit: 1.48 Mfps (frame-per-second)
- 10 Gigabit: 14.8 Mfps

With these first values and the fact that Ethernet is a full-duplex media, able to receive and transmit at the same time, this means a line-rate router must be able to forward at:

- 3 Mpps = Gigabit line-rate router
- 30 Mpps = 10 Gigabit line-rate router

D) *Throughput to bandwidth*

In real use cases there is no need of these line-rate routers because Internet traffic is not comprise of only small size packets but a mix of multiple sizes. This packet size distribution evolves with time but there is a fixed-in-time reference, called Simple Internet Mix (IMIX)[4], which uses this distribution:

- 1 large (1500 Bytes) packet: 37%
- 4 medium (576 Bytes) packets: 56%
- 7 small (40 Bytes) packets: 7%

Using Simple IMIX distribution it's now possible to convert the reference packets-per-second to a more common value which is the bandwidth in bits per second (bps).

$$\text{bps at the IP layer} = PPS \cdot \left(\frac{7 \cdot 40 + 4 \cdot 576 + 1500}{12}\right) \cdot 8$$

Or the bandwidth at the Ethernet layer (need to add 14 Bytes for Ethernet headers), as seen by switch counters:

$$\text{bps at the Ethernet layer} = PPS \cdot \left(\frac{7 \cdot 54 + 4 \cdot 590 + 1514}{12}\right) \cdot 8$$

For real life use cases, the interesting question is now: "Using a simple IMIX distribution size, what is the corresponding throughput for filling link capacity?"
These are the values that will be used for a new definition of a *Full-duplex IMIX link-speed router* and the minimum objectives to reach are:

- **700K pps = Gigabit IMIX router**
- **7M pps = 10 Gigabit IMIX router**

E) *Benchmarking lab*

A simple benchmarking lab can be set up with only 2 servers like here:

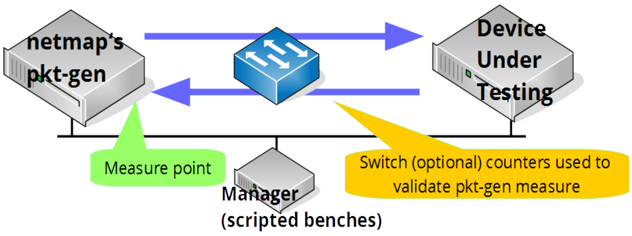


Illustration 1: Simplest benchmarking lab

- The first server with dual port Network Interface Card (NIC) is used as a packet generator and receiver (using netmap's pkt-gen[5]).
- The second server is the Device Under Test (DUT) running FreeBSD that will be tuned.

The purpose is to measure throughput (number of packets per second) forwarded by the DUT under the worst case: Receiving only smallest packet size at media line rate on one interface and forward to the packet receiver using its other interface.

The throughput is measured at the packet receiver side: Using a switch, with advanced monitoring counters for each port, can be useful to double cross-check its counters versus pkt-gen and Ethernet drivers counters.

Full list of hardware setups (CPU and NIC) used for this study is detailed here:

Servers	CPU	cores	GHz	Network card (driver name)
Dell PowerEdge R630	Intel **E5-2650 v4**	2x12x2	2.2	10G Intel 82599ES (**ixgbe**) 10G Chelsio T520-CR (**cxgbe**) 10G Mellanox ConnectX-3 Pro (**mlx4en**) 10-50G Mellanox ConnectX-4 LX (**mlx5en**)
HP ProLiant DL360p Gen8	Intel **E5-2650** v2	8x2	2.6	10G Chelsio T540-CR (**cxgbe**) 10G Emulex OneConnect be3 (**oce**)
SuperMicro 5018A-FTN4	Intel Atom **C2758**	8	2.4	10G Chelsio T540-CR (**cxgbe**)
SuperMicro 5018A-FTN4	Intel Atom **C2758**	8	2.4	10G Intel 82599 (**ixgbe**)
Netgate RCC-VE 4860	Intel Atom **C2558**	4	2.4	Gigabit Intel i350 (**igb**)
PC Engines APU2	AMD **GX-412**TC	4	1	Gigabit Intel i210AT (**igb**)

Illustration 2: Hardware inventory

II. TUNING FORWARDING PERFORMANCE

A) *Multi-queue NIC & RSS*

Current NIC chipset & drivers behaviour:

1. NIC's drivers create one queue per direction (transmit and receive) and per core detected with a maximum number of queues which is drivers dependant: 16 receiving (RX) queues for mlx4en, 8 RX queues for cxgbe and ixgbe as examples.
2. NIC's chipsets use a Toeplitz hash to balance received packets across each RX queues: All 4 tuples of the packets (source IP, destination IP, source port and destination port) are used.

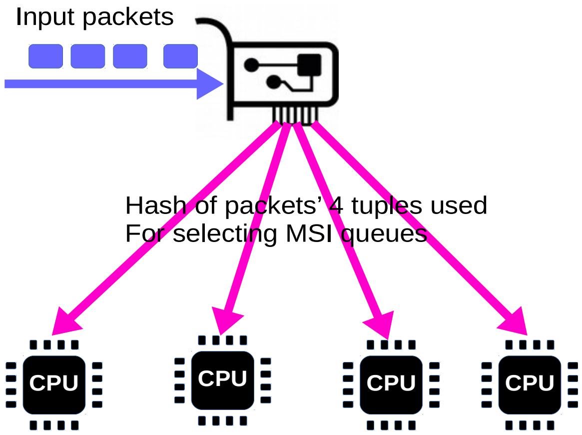


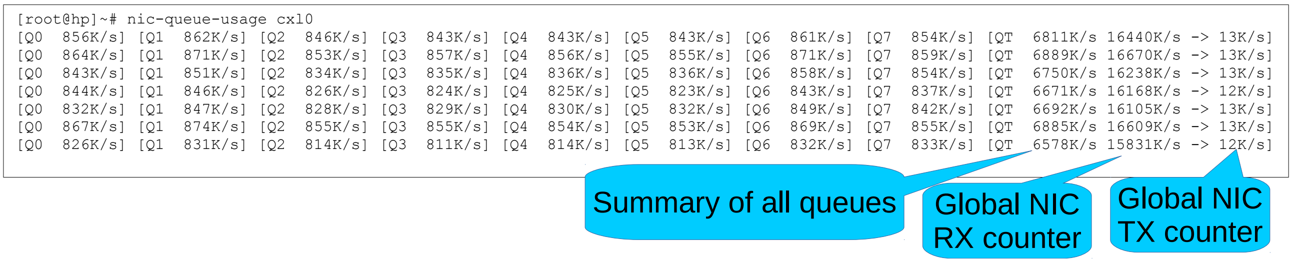
Illustration 3: Toeplitz hash

To being able to load-balance IP flows between cores, IP traffic must include multiple flows for being hashed: Using tunnelling features like IPSec, GRE or PPPoE prevents this distribution.

B) *Checking flow distribution between each queue*

The first step is to check packets are correctly distributed among all NIC's receiving queues. NIC drivers give statistical usage of all their queues but a simple script can be useful for a better view [6], giving output like in Illustration 4: Example of script output displaying each RX queue usage in pps.

This example, by displaying equivalent throughput for all 8 queues, shows a correct distribution between all RX queues.



```
[root@hp]~# nic-queue-usage cxl0
[Q0  856K/s] [Q1  862K/s] [Q2  846K/s] [Q3  843K/s] [Q4  843K/s] [Q5  843K/s] [Q6  861K/s] [Q7  854K/s] [QT  6811K/s 16440K/s -> 13K/s]
[Q0  864K/s] [Q1  871K/s] [Q2  853K/s] [Q3  857K/s] [Q4  856K/s] [Q5  855K/s] [Q6  871K/s] [Q7  859K/s] [QT  6889K/s 16670K/s -> 13K/s]
[Q0  843K/s] [Q1  851K/s] [Q2  834K/s] [Q3  835K/s] [Q4  836K/s] [Q5  836K/s] [Q6  858K/s] [Q7  854K/s] [QT  6750K/s 16238K/s -> 13K/s]
[Q0  844K/s] [Q1  846K/s] [Q2  826K/s] [Q3  824K/s] [Q4  825K/s] [Q5  823K/s] [Q6  843K/s] [Q7  837K/s] [QT  6671K/s 16168K/s -> 12K/s]
[Q0  832K/s] [Q1  847K/s] [Q2  828K/s] [Q3  829K/s] [Q4  830K/s] [Q5  832K/s] [Q6  849K/s] [Q7  842K/s] [QT  6692K/s 16105K/s -> 13K/s]
[Q0  867K/s] [Q1  874K/s] [Q2  855K/s] [Q3  855K/s] [Q4  854K/s] [Q5  853K/s] [Q6  869K/s] [Q7  855K/s] [QT  6885K/s 16609K/s -> 13K/s]
[Q0  826K/s] [Q1  831K/s] [Q2  814K/s] [Q3  811K/s] [Q4  814K/s] [Q5  813K/s] [Q6  832K/s] [Q7  833K/s] [QT  6578K/s 15831K/s -> 12K/s]
```


Illustration 4: Example of script output displaying each RX queue usage in pps

C) Hyper-threading

Load balancing packets between multiple core allows to load-balance IRQ among the cores. But does hyper-threading (HT) technology help regarding IRQ management ?

Testing this impact can be done by benching 3 configurations sets on an 8-core (16 threads) single socket CPU with a Chelsio T540 NIC:

- HT enabled (16 threads) and default cxgbe drivers behaviour creating 8 receiving queue. Notice that cxgbe drivers didn't bind queue to a thread.
- HT enabled (16 threads) and forcing cxgbe drivers to use 16 receiving queues: one for each thread.
- HT disabled (8 cores) and default cxgbe drivers creating 8 receiving queues: one for each core.

Each configuration set is run 5 times (with a reboot between them). Then ministat (statistical tool embedded with FreeBSD) is used on these 3 data sets:

```
x Xeon E5-2650-cxgbe, HT-enabled & 8rxq(default): inet4 packets-per-second
+ Xeon E5-2650-cxgbe, HT-enabled & 16rxq: inet4 packets-per-second
* Xeon E5-2650-cxgbe, HT-disabled & 8rxq: inet4 packets-per-second
+--------------------------------------------------------------------------+
|                                                                        **|
|x      xx    x          +       + + +  +                               ***|
|    |____A_____|                                                         |
|                           |_____AM____|                                 |
|                                                                      |A||
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x   5       4500078       4735822       4648451     4648293.8     94545.404
+   5       4925106       5198632       5104512     5088362.1     102920.87
Difference at 95.0% confidence
        440068 +/- 144126
        9.46731% +/- 3.23827%
        (Student's t, pooled s = 98821.9)
*   5       5765684     5801231.5       5783115     5785004.7     13724.265
Difference at 95.0% confidence
        1.13671e+06 +/- 98524.2
        24.4544% +/- 2.62824%
        (Student's t, pooled s = 67554.4)
```

Illustration 5: Hyperthreading impact on forwarding performance

Between the 2 setups using 8 receiving queues, there is about 24% more PPS forwarded (from 4.65Mpps to 5.85Mpps) with hyper-threading disabled: This confirms that threads didn't help on a forwarding use case, and even decreased the performance because the scheduler didn't make any difference between threads and cores. Hyper-theading will be disabled now for all the rest of this study.

D) Relation between the number of cores and throughput

NIC drivers often allow to configure the number of received (RX) and transmit (TX) queues. Each queue has its own MSI-X IRQ assigned.

A new bench is configured on the same hardware as the previous bench. Multiple configuration sets, forcing the NIC drivers to use from 1 to 8 queues on this 8 core single-socket CPU server give the relation between queue/forwarding performance.

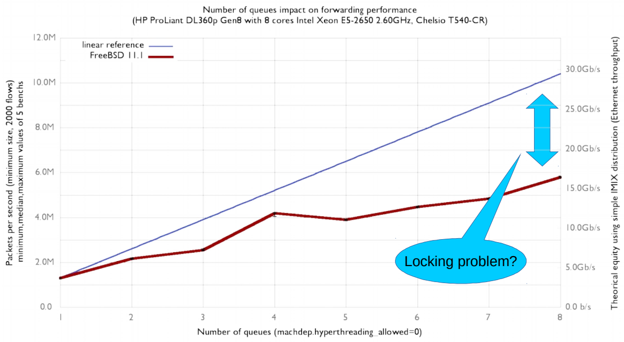


Illustration 6: Number of queues vs forwarding performance

The results show a non-linear performance scale. This kind of problem is often created by lock contention problems on the kernel network path.

Troubleshooting where the kernel spend its time is done in 2 steps:

1. First step is to collect Hardware Performance Monitoring Counter during the bench

```
kldload hwpmc
pmcstat -S CPU_CLK_UNHALTED_CORE -l 20 -O
data.out
stackcollapse-pmc.pl data.out > data.stack
flamegraph.pl data.stack > data.svg
```

2. Second step is to convert this data into Brendan D. Gregg's flamegraph

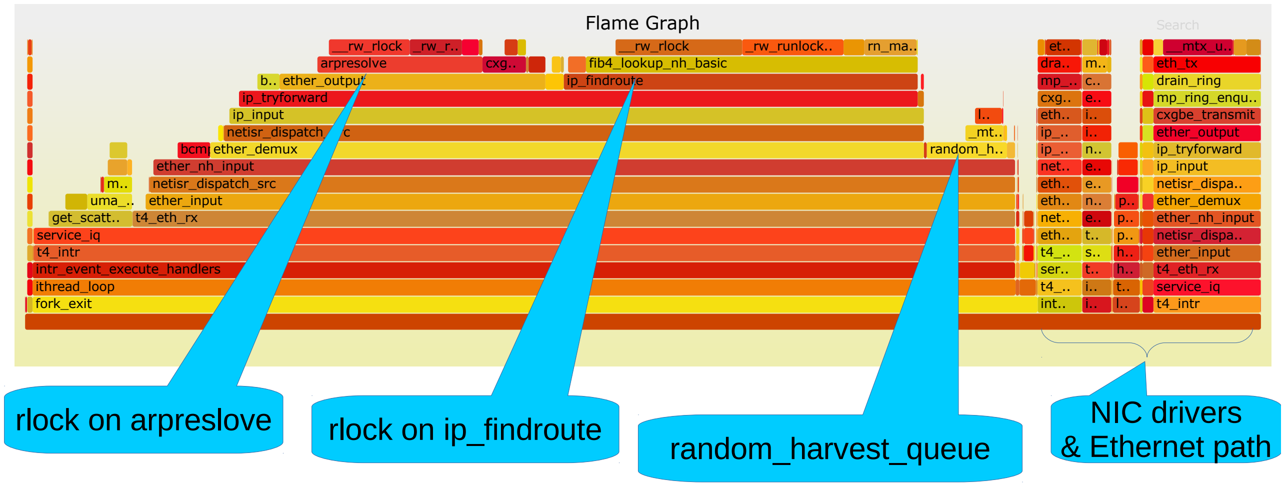


Illustration 7: Forwarding path flamegraph

Flame Graph analysis shows some interesting hot points in 3 functions:

- arpresolve()
- ip_findroute()
- random_harvest_queue()

The first 2 functions are directly related to the kernel network stack. Some simple configuration tunings were tested to limit these lock contentions:

- static arp entries for arpresolve()
- minimal numbers of static routes for ip_findroute()

But none of these mitigates the lock contention. To solve these two problems the network stack needs to be fixed.

E) *Random Harvest Sources*

The third lock contention is due to random_harvest_queue() collecting first 2 bytes of each frame under a single mutex. This problem is easily fixed with a simple configuration change: By excluding Ethernet frames and interrupts to be used as entropy sources we can mitigate this problem.

```
~# sysctl kern.random.harvest
kern.random.harvest.mask_symbolic: [UMA],
[FS_ATIME],SWI,INTERRUPT,NET_NG,NET_ETHER,NET_TUN,MOUSE,KEYBOARD,
ATTACH,CACHED
kern.random.harvest.mask_bin: 00111111111
kern.random.harvest.mask: 511
```

Illustration 8: Default random harverst mask

2 new configuration sets are benched:

1. First one using default random harvest mask value of 511
2. Second with mask value reduced to 351

Setup CPU (cores) & NIC	511 (default) Median of 5	351 Median of 5	ministat
E5-2650v4 (2x12) & ixgbe Xeon & Intel 82599ES	3.74 Mpps	3.78 Mpps	No diff. proven at 95.0% confidence
E5-2650v4 (2x12) & cxgbe Xeon & Chelsio T520	4.82 Mpps	4.87 Mpps	No diff. proven at 95.0% confidence
E5-2650v4 (2x12) & ml4en Xeon & Mellanox ConnectX-3 Pro	3.49 Mpps	3.92 Mpps	11.66% +/- 8.15%
E5-2650v4 (2x12) & ml5en Xeon & Mellanox ConnectX-4 Lx	0 Mpps	0 Mpps	System Overloaded
E5-2650v2 (8) & cxgbe Xeon & Chelsio T540	5.76 Mpps	5.79 Mpps	No diff. proven at 95.0% confidence
E5-2650v2 (8) & oce Xeon & Emulex be3	1.33 Mpps	1.33 Mpps	No diff. proven at 95.0% confidence
C2758 (8) & cxgbe Atom & Chelsio T540	2.83 Mpps	3.17 Mpps	12.52% +/- 1.82%
C2758 (8) & ixgbe Atom & Intel 82599ES	2.3 Mpps	2.43 Mpps	6.14% +/- 1.84%
C2558 (4) & igb Atom & Intel I354	951 Kpps	1 Mpps	4.75% +/- 1.08%
GX412 (4) & igb AMD & Intel I210	726 Kpps	749 Kpps	3.14% +/- 0.70%

Illustration 9: Result of reducing random harvest mask

This first full lab results shows we are far from our objective regarding 10 Gigabit IMIX router:

- Both Gigabit routers (Netgate RCC-VE 4860 and PC Engines APU2) are able to reach the expected throughput with default FreeBSD 11.1 parameters.

- None of the 10 Gigabit routers was able to reach the minimum 7Mpps.
- ml5en driver uses aggressive default parameters that overload the kernel

The 2 network stack lock contention problems (arpresolve and ip_findroute) need to be fixed.

F) arpresolve & ip_findroute

These 2 problems were already analysed and fixed a few years ago by Yandex's team: Alexander V. Chernikov (melifaro@) and Andrey V. Elsukov (ae@) and referenced into FreeBSD's wiki[7]. Their work is stored into the experimental projects/routing[8]. Andrey V. Elsukov has refreshed patches related to arpreslove[9] and ip_findroute[10] to FreeBSD -current. And they were adapted to FreeBSD 11.1 for this study[11]

setup	11.1	11.1-Yandex	ministat
E5-2650v4 (2x12) & ixgbe Xeon & Intel 82599ES	3.78 Mpps	6.46 Mpps	73.58% +/- 7.3%
E5-2650v4 (2x12) & cxgbe Xeon & Chelsio T520	4.87 Mpps	9.60 Mpps	95.36% +/- 3.8%
E5-2650v4 (2x12) & mlx4en Xeon & Mellanox ConnectX-3 Pro	3.92 Mpps	8.01 Mpps	100.5% +/- 15.6%
E5-2650v4 (2x12) & mlx5en Xeon & Mellanox ConnectX-4 Lx	0 Mpps	14.64 Mpps	*NA*
E5-2650v2 (8) & cxgbe Xeon & Chelsio T540	5.75 Mpps	10.9 Mpps	90.56% +/- 1.24
E5-2650v2 (8) & oce Xeon & Emulex be3	1.33 Mpps	1.33 Mpps	No diff. proven at 95.0% confidence
C2758 (8) & cxgbe Atom & Chelsio T540	3.15 Mpps	4.2 Mpps	34.4% +/- 2.9%
C2758 (8) & ixgbe Atom & Intel 82599ES	2.43 Mpps	3.08 Mpps	26% +/- 1.18
C2558 (4) & igb Atom & Intel I354	1 Mpps	1.2 Mpps	20.17% +/- 2.56%
GX412 (4) & igb AMD & Intel I210	747 Kpps	729 Kpps	-2.37% +/- 0.58%

Illustration 11: Result of removing arpresolve & ip_findroude locks

This second bench result shows huge performance improvement allowing almost all 10Gigabit setup to reach the minimal target of 7Mpps with the exception of the Atom based servers and the Intel 82599ES.

Some remarks:

- Notice the very bad performance of Emulex OneConnect (be3): This 10Gigabit NIC is not able to reach the throughput of a simple gigabit NIC (1.44Mpps) and there is no possibility to configure the number of receiving and transmitting queues too (hard-coded at 4)
- Notice the difference on the 24-core server between Mellanox ConnectX-4 versus Chelsio T520 & Intel 82599ES: This will be analysed

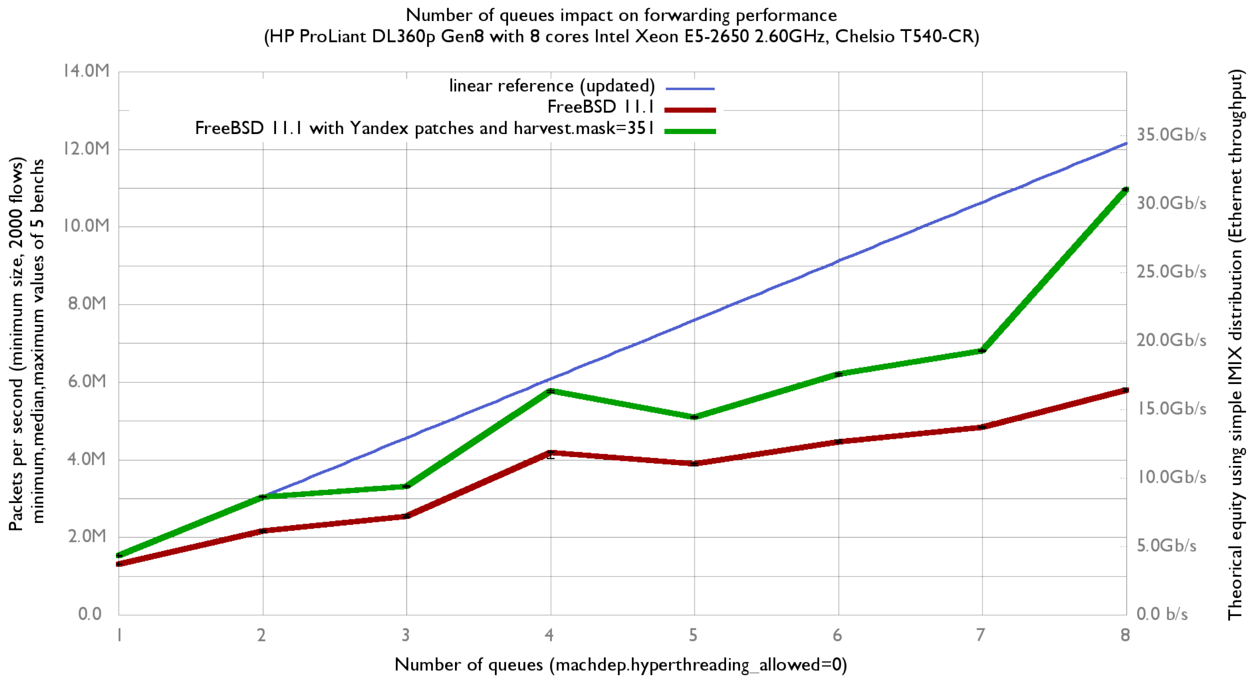


Illustration 10: Number of queues vs forwarding performance on patched FreeBSD 11.1

later in chapter Increasing default number of NIC's queue.

G) Forwarding performance scale on 8 core single socket with AFDATA and RADIX patches

The bench measuring impact of the number of queues vs throughput is run another time but with a Yandex patched 11.1 in Illustration 10: Number of queues vs forwarding performance on patched FreeBSD 11.1.

This graph shows a linear progression, but only if the number of queues is a power-of-two: This can be explained by a Chelsio's RSS hash size optimized for a power of two number of queue. During bootup, cxgbe driver displays this warning if a non-optimum number of queues is detected:

```
cxl0: nrxq (6), hw RSS table size (64); expect
uneven traffic distribution.
cxl1: nrxq (6), hw RSS table size (64); expect
uneven traffic distribution.
```

H) Increasing default number of NIC's queue

Does the performance difference between Mellanox ConnectX-4 versus Chelsio & Intel is related to the default number of queues each driver creates? A new bench forcing all these drivers to use the same number of queues is started.

Bench result shows that increasing number RX queues allows to reduce the difference between cxgbe and mlx5en, and even allows the 10 gigabit Intel setup to reach the minimum expected 7Mpps.

Setup E5-2650v4 (2x12 cores)	8 queues (default for ixgbe & cxgbe)	24 queues (default for mlx5en)	ministat
ixgbe Intel 82599ES	6.72 Mpps	8.07 Mpps	21.34% +/- 4.96%
cxgbe Chelsio T520	9.59 Mpps	12.40 Mpps	29.45% +/- 0.37%
mlx5en Mellanox ConnectX-4 Lx	*7.26 Mpps*	14.64 Mpps	

Illustration 12: Increasing default number of NIC's queues

Notice that Mellanox ConnectX-3 didn't allow user to configure the number of queues.

I) Pining cxgbe queue's interrupt to CPU

Letting the scheduler dynamically move NIC's queue interrupt from one core to another should be avoided. Some NIC drivers (bxe, ixgbe, ixl, e1000, etc.) bind queue interrupts to core but the cxgbe driver didn't do it: Is there a real benefit to pin cxgbe queue to the core?

A new bench using a simple RC shell script [12]that bind cxgbe queue is used. An example of this shell output:

```
~# service chelsio_affinity start
Bind t5nex0:0a IRQ 284 to CPU 0
Bind t5nex0:0a IRQ 285 to CPU 1
Bind t5nex0:0a IRQ 286 to CPU 2
Bind t5nex0:0a IRQ 287 to CPU 3
Bind t5nex0:0a IRQ 288 to CPU 4
Bind t5nex0:0a IRQ 289 to CPU 5
Bind t5nex0:0a IRQ 290 to CPU 6
Bind t5nex0:0a IRQ 291 to CPU 7
```

The bench result shows a very small improvement (about 2%) on the 8-core setup:

```
x Xeon E5-2650v2 & cxgbe, default: inet4 packets-per-second
+ Xeon E5-2650v2 & cxgbe, IRQ pinned to CPU: inet4 packets-per-second
+--------------------------------------------------------------------------+
|                                                                     +    |
|xx  xx    x                                                     + +  +  +|
| |___A___|                                                                |
|                                                                |___A_M_| |
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x   5      10939210      10969716      10952795      10951860     12056.937
+   5      11132364      11161395      11151483      11146670     12273.277
Difference at 95.0% confidence
        194810 +/- 17742.8
        1.77878% +/- 0.163429%
        (Student's t, pooled s = 12165.6)
```

Illustration 13: Pining queue interrupt to CPU

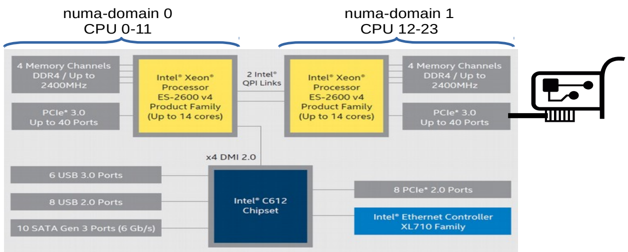
J) NUMA affinity

On the dual-socket server, a dmesg line catches our attention:

```
t5nex0: <Chelsio T520-CR> mem 0xc9200000-0xc927ffff,0xc8000000-
0xc8ffffff,0xc9684000-0xc9685fff irq 50 at device 0.4 numa-domain 1 on pci14
```

Illustration 14: dmesg line about NUMA domain

On this server, the Chelsio card is plugged into a PCIe bus managed by the second socket (numa-domain 1) and not the first (numa-domain 0) one as show in the Intel Xeon architecture diagram:

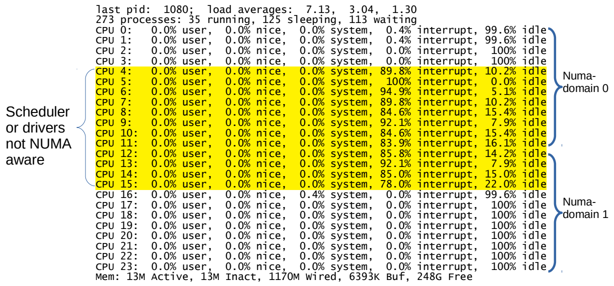


Intel Xeon Processor E5-2600 v4 Product Family: Platform Brief

Illustration 15: Intel Xeon E5-2600 NUMA and PCIe

Does the FreeBSD scheduler or NIC drivers are NUMA aware and avoid the usage of QPI links?

Answering this question is done by configuring cxgbe to use 12 queues and checking which cores are assigned to them during a network performance bench:



```
last pid:  1080;  load averages:  7.13,  3.04,  1.30
273 processes: 35 running, 125 sleeping, 113 waiting
CPU 0:   0.0% user,  0.0% nice,  0.0% system,  0.4% interrupt, 99.6% idle
CPU 1:   0.0% user,  0.0% nice,  0.0% system,  0.4% interrupt, 99.6% idle
CPU 2:   0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 3:   0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 4:   0.0% user,  0.0% nice,  0.0% system, 89.8% interrupt, 10.2% idle
CPU 5:   0.0% user,  0.0% nice,  0.0% system,  100% interrupt,  0.0% idle
CPU 6:   0.0% user,  0.0% nice,  0.0% system, 94.9% interrupt,  5.1% idle
CPU 7:   0.0% user,  0.0% nice,  0.0% system, 89.8% interrupt, 10.2% idle
CPU 8:   0.0% user,  0.0% nice,  0.0% system, 84.6% interrupt, 15.4% idle
CPU 9:   0.0% user,  0.0% nice,  0.0% system, 92.1% interrupt,  7.9% idle
CPU 10:  0.0% user,  0.0% nice,  0.0% system, 84.6% interrupt, 15.4% idle
CPU 11:  0.0% user,  0.0% nice,  0.0% system, 83.9% interrupt, 16.1% idle
CPU 12:  0.0% user,  0.0% nice,  0.0% system, 85.8% interrupt, 14.2% idle
CPU 13:  0.0% user,  0.0% nice,  0.0% system, 92.1% interrupt,  7.9% idle
CPU 14:  0.0% user,  0.0% nice,  0.0% system, 85.0% interrupt, 15.0% idle
CPU 15:  0.0% user,  0.0% nice,  0.0% system, 78.0% interrupt, 22.0% idle
CPU 16:  0.0% user,  0.0% nice,  0.4% system,  0.0% interrupt, 99.6% idle
CPU 17:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 18:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 19:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 20:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 21:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 22:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
CPU 23:  0.0% user,  0.0% nice,  0.0% system,  0.0% interrupt,  100% idle
Mem: 13M Active, 13M Inact, 1170M Wired, 6393K Buf, 248G Free
```


Illustration 16: Default core usage on a NUMA system

FreeBSD 11.1 cxgbe driver is not NUMA aware: The scheduler didn't try to avoid assigning remote numa-domain core to the NIC queue. But does the latency induced by crossing the QPI link have an impact on the forwarding network performance ?

Another bench using cxgbe forced to 12 queues with 3 configurations sets is started:

- Configuration 1: Default (no NUMA affinity)
- Configuration 2: All 12 cxgbe queues pined to core 0 to 11 (remote numa-domain, should give worse performance)
- Configuration 3: All 12 cxgbe queues pined to core 12 to 23 (local numa domain, should give best performance)

The bench result clearly shows an improvement of about 12% with forced NUMA affinity on the same numa-domain as the NIC's PCIe bus:

```
x Xeon 2xE5-2650v4 & cxgbe, default: inet4 packet-per-seconds
+ Xeon 2xE5-2650v4 & cxgbe, affinity-numa0: inet4 packet-per-seconds
* Xeon 2xE5-2650v4 & cxgbe, affinity-numa1: inet4 packet-per-seconds
+--------------------------------------------------------------------------+
|                  +x                                                   *  |
|+     x     x + +x+                                                 ** **|
|         |____A__M_|                                                     |
|     |_______A__M____|                                                   |
|                                                                   |MA_| |
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x   5       9351036       9580847       9571249       9510859     98839.328
+   5       9220385       9603697       9557225     9493098.6      154964.3
No difference proven at 95.0% confidence
*   5      10584085      10670945      10617361      10629374     35170.165
Difference at 95.0% confidence
        1.11851e+06 +/- 108191
        11.7604% +/- 1.25701%
        (Student's t, pooled s = 74182.7)
```

Illustration 18: NUMA affinity impact on forwarding performance

K) Forwarding performance scale on 24-core dual socket

The relation between number of queue on the 2x12 core dual socket is benched with Chelsio, Mellanox and Intel NIC:

This result shows the same benefit of keeping numbers of queue to power of 2 with the cxbge and ixgbe drivers: mlx5en driver didn't have this restriction. There isn't any benefit to use all 24 queues here but only 16 because there is no more linear scale after 8 queues: Theoretically this server should be able to reach the line rate with only 11 queues but it have to use 16 queues (so 16 cores) to reaching it.

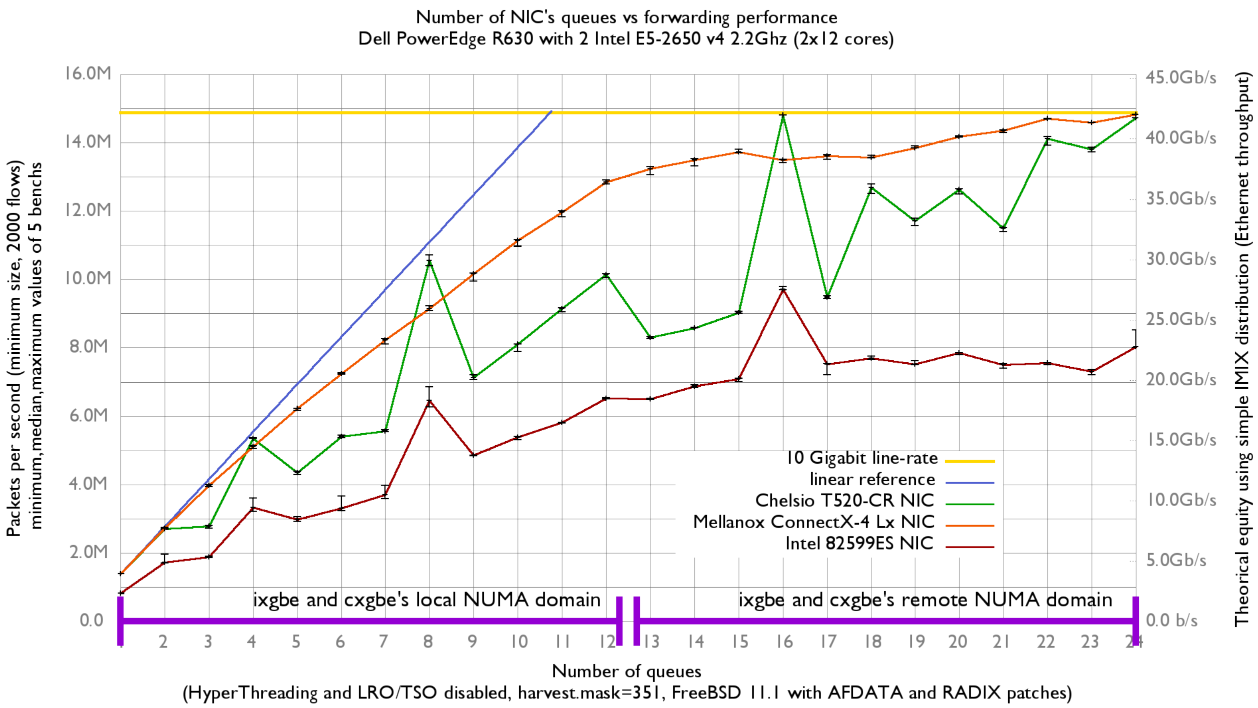


Illustration 17: Number of queues vs forwarding performance on dual-socket

L) NIC drivers tuning

Current NIC's chipsets include lots of hardware acceleration features. But server's NIC are designed for end-host usage and not a router usage, so some tuning are required, here are some examples:

- Checksum offload (rxcsum, txcsum): to be kept enabled.
- VLAN offload (vlanmtu, vlanhwtag, vlanhwfilter, vlanhwcsum,…): to be kept enabled too.
- TSO (TCP Segmentation Offload): split large segments into MTU-sized packets. This feature MUST be disabled on a router (and is incompatible with ipfw NAT engine).
- LRO (Large Received Offload): Breaks the end-to-end principle on a router so MUST be disabled.
- Hardware resources reservation.

Theoretically the TSO and LRO features are useless of a router, so a new bench compares these:

- Configuration set 1: LRO and TSO enabled(default)

```
ifconfig_cxl0="inet 198.18.0.10/24"
ifconfig_cxl1="inet 198.19.0.10/24"
```

- Configuration set 2: LRO and TSO disabled

```
ifconfig_cxl0="inet 198.18.0.10/24 -tso4 -tso6 -
lro -vlanhwtso"
ifconfig_cxl1="inet 198.19.0.10/24 -tso4 -tso6 -
lro -vlanhwtso"
```

Bench result table in Illustration 19: Impact of disabling TSO/LRO on forwarding performance.

Server CPU (cores) & NIC	Enabled (default)	Disabled	ministat
E5-2650v4 (2x12) & ixgbe Xeon & Intel 82599ES	7.97 Mpps	8.07 Mpps	No difference proven at 95.0% confidence
E5-2650v4 (2x12) & cxgbe Xeon & Chelsio T520	12.40 Mpps	12.40 Mpps	No difference proven at 95.0% confidence
E5-2650v4 (2x12) & ml4en Xeon & Mellanox ConnectX-3 Pro	8.05 Mpps	7.85 Mpps	No difference proven at 95.0% confidence
E5-2650v4 (2x12) & ml5en Xeon & Mellanox ConnectX-4 Lx	14.65Mpps	14.83 Mpps	1.3% +/- 0.1%
E5-2650v2 (8) & cxgbe Xeon & Chelsio T540	10.84 Mpps	10.92 Mpps	0.74% +/- 0.26%
C2758 (8) & cxgbe Atom & Chelsio T540	4.20 Mpps	4.18 Mpps	No diff. proven at 95.0% confidence
C2758 (8) & ixgbe Atom & Intel 82599ES	3.06 Mpps	3.06 Mpps	No diff. proven at 95.0% confidence
C2558 (4) & igb Atom & Intel I354	1.2 Mpps	1.2 Mpps	No diff. proven at 95.0% confidence
GX412 (4) & igb AMD & intel I210	729 Kpps	727 Kpps	No diff. proven at 95.0% confidence

Illustration 19: Impact of disabling TSO/LRO on forwarding performance

This result confirms disabling TSO/LRO features do not degrade forwarding performance.

Notice that on 2 identical servers (8core Atom Supermicro 5018A-FTN4), the Chelsio NIC is able to manage 1M pps more than the Intel NIC: 3.06Mpps vs 4.18Mpps.

So some Intel driver parameters were tested to try to increase its performance:

- disabling adaptive interrupt moderation: hw.ix.enable_aim
- Increasing maximum interrupts per second: hw.ix.max_interrupt_rate
- Disabling limit of the maximum number of received packets to process at a time: hw.ix.rx_process_limit

And only the last parameter increases throughput:

Server CPU (cores) & NIC	100(igb), 256(ix), default median	-1 (disabled) median	ministat
E5-2650v4 (2x12) & ixgbe Xeon & Intel 82599ES	8.04 Mpps	8.34 Mpps	3.75% +/- 0.73%
C2758 (8) & ixgbe Atom & Intel 82599ES	3.12 Mpps	3.85 Mpps	22.66% +/- 2.14%
C2558 (4) & igb Atom & Intel I354	1.10 Mpps	1.13 Mpps	1.65% +/- 0.9%
GX412 (4) & igb AMD & Intel I210	730 Kpps	735 Kpps	No diff. proven at 95.0% conf.

Illustration 20: Intel drivers rx_process_limit tuning

Disabling the maximum limit for processing received packets allows to increase the throughput by %22 on the 8-core Atom server. But this Intel NIC has still less 10% throughput (3.85Mpps vs 4.18Mpps) than the Chelsio NIC on the same server.

Regarding the Chelsio driver, the man page mention some sysctl to disallowing (chipset) capabilities preventing the firmware to not reserve hardware resources for some features (TOE, RDMA, ISCSI, FCOE). This is done by adding these line into the /boot/loader.conf file:

```
hw.cxgbe.toecaps_allowed="0"
hw.cxgbe.rdmacaps_allowed="0"
hw.cxgbe.iscsicaps_allowed="0"
hw.cxgbe.fcoecaps_allowed="0"
```

And it gives interesting improvement (almost 20% improvement):

```
x Xeon 2xE5-2650v4 & cxgbe, default caps enabled: inet4 packet-per-seconds
+ Xeon 2xE5-2650v4 & cxgbe, caps disabled: inet4 packet-per-seconds
+--------------------------------------------------------------------------+
|x                                                                        +|
|x                                                                        +|
|x                                                                        +|
|x                                                                        +|
|x                                                                        +|
|A                                                                         |
|                                                                         A|
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x   5      12411366      12413439      12411915      12412289     901.22767
+   5      14796094      14800927      14799082      14798629     2169.6179
Difference at 95.0% confidence
        2.38634e+06 +/- 2422.83
        19.2256% +/- 0.0201158%
        (Student's t, pooled s = 1661.24)
```

Illustration 21: Disabling cxgbe caps

M) Tuning summary for a router

Here are the summary of all information learned to tune a FreeBSD 11.1 router:

- Check for multiples IP flows to being correctly distributed among each NIC's queue
- Disable HyperThreading
- Exclude Ethernet packets & Interrupt as entropy sources
- Apply Yandex's AFDATA and RADIX locks patches
- Use good NIC like Mellanox and Chelsio
- Increase Intel & Chelsio NIC drivers queues if number of core > 8, and with Chelsio use a number of queue = power of 2.
- Intel NIC driver: Remove maximum limit of packets to process
- Chelsio driver: Prevent to reserve resources for unused features
- Disable TSO and LRO

Translated into configuration parameters it gives:

/boot/loader.conf:

```
# Disabling Hyper-threading
machdep.hyperthreading_allowed="0"
# Remove limit of the maximum number of packets
to manage at once (Intel only)
hw.igb.rx_process_limit="-1"
hw.em.rx_process_limit="-1"
hw.ix.rx_process_limit="-1"
# Increase number of cxgbe or Intel queue if
ncpu >8
# This value should be a power of 2 with cxgbe.
# Example of a 24-core server with cxgbe and
ixgbe:
hw.cxgbe.nrxq10g="16"
hw.cxgbe.ntxq10g="16"
hw.ix.num_queues="16"
```

```
# Disabling cxgbe caps
hw.cxgbe.toecaps_allowed="0"
hw.cxgbe.rdmacaps_allowed="0"
hw.cxgbe.iscsicaps_allowed="0"
hw.cxgbe.fcoecaps_allowed="0"
```

/etc/rc.conf:

```
# Exclude Ethernet packets and Interrupt from
entropy source
harvest_mask="351"
# Disable TSO and LRO
ifconfig_X="YYY -tso4 -tso6 -lro -vlanhwtso
```

Applying Yandex patches on FreeBSD 11.1:

```
cd /usr/src
fetch
https://people.freebsd.org/~olivier/fbsd11.1.ae.
afdata-radix.patch
patch < fbsd11.1.ae.afdata-radix.patch
make buildkernel && make installkernel
```

Without these tuning parameters and patches, FreeBSD 11.1-RELEASE is not able to reach the minimum 7Mpps for a 10Gigabit router. But once patches and tuning tips applied, the benefit is resumed here:

Setup CPU (cores) & NIC	Generic 11.1	Yandex patched & tuned 11.1	ministat
E5-2650v4 (2x12) & ixgbe Xeon & Intel 82599ES	3.74 Mpps	8.61 Mpps	127.93% +/- 8.44%
E5-2650v4 (2x12) & cxgbe Xeon & Chelsio T520	4.83 Mpps	14.8 Mpps	204.3% +/- 4.80%
E5-2650v4 (2x12) & ml4en Xeon & Mellanox ConnectX-3 Pro	3.92 Mpps	8.06 Mpps	126.9% +/- 7.77%
E5-2650v4 (2x12) & ml5en Xeon & Mellanox ConnectX-4 Lx	0 Mpps	14.64 Mpps	*NA*
E5-2650v2 (8) & cxgbe Xeon & Chelsio T540	5.75 Mpps	11.15 Mpps	139.8% +/- 5.0%
E5-2650v2 (8) & oce Xeon & Emulex be3	1.33 Mpps	1.33 Mpps	No diff. proven at 95.0% confidence
C2758 (8) & cxgbe Atom & Chelsio T540	2.83 Mpps	4.19 Mpps	50.49% +/- 5.33%
C2758 (8) & ixgbe Atom & Intel 82599ES	2.29 Mpps	3.85 Mpps	66.97% +/- 2.7%
C2558 (4) & igb Atom & Intel I354	951 Kpps	1.13 Mpps	18.58% +/- 1.17%
GX412 (4) & igb AMD & Intel I210	726 Kpps	735 Kpps	1.03% +/- 0.56%

Illustration 22: Forwading tuning summary

III. SOME CONFIGURATIONS IMPACT

A) IPv6

All previous benches were done using IPv4 flows but what about IPv6 flows?

Setup CPU (cores) & NIC	inet4	inet6	ministat
E5-2650v4 (2x12) & ixgbe Xeon & Intel 82599ES	8.35 Mpps	8.12 Mpps	-3.25% +/- 1.7%
E5-2650v4 (2x12) & cxgbe Xeon & Chelsio T520	14.8 Mpps	14.47 Mpps	-2.18% +/- 0.02%
E5-2650v4 (2x12) & ml4en Xeon & Mellanox ConnectX-3 Pro	8.06 Mpps	7.71 Mpps	-3.35% +/- 3.26%
E5-2650v4 (2x12) & ml5en Xeon & Mellanox ConnectX-4 Lx	14.84 Mpps	14.29 Mpps	-3.70% +/- 0.02%
E5-2650v2 (8) & cxgbe Xeon & Chelsio T540	10.94 Mpps	9.18 Mpps	-16.12% +/- 0.19%
C2758 (8) & cxgbe Atom & Chelsio T540	4.29 Mpps	3.43 Mpps	-19.08% +/- 1.61%
C2758 (8) & ixgbe Atom & Intel 82599ES	3.81 Mpps	3.43 Mpps	-9.84% +/- 1.3%
C2558 (4) & igb Atom & Intel I354	1.23 Mpps	1.08 Mpps	-11.79% +/- 0.5%
GX412 (4) & igb AMD & Intel I210	734 Kpps	709 Kpps	-3.6% +/- 0.70%

Illustration 23: IPv4 vs IPv6 forwarding performance

The IPv6 forwarding stack is not as efficient as the IPv4 and can performance penalty are between -3 to -20%. Notice the exact same performance on the 8 core Atom servers: The bottleneck is no more into NIC drivers but moved into the IPv6 kernel stack.

B) *VLAN tagging*

Routers often use 802.1Q tagging on their network interfaces. And, as seen previously, modern NIC chipsets include VLAN tag accelerating features: So performance impact should be minimum.

- Configuration set 1: No VLAN

```
ifconfig_cxl0="inet 198.18.0.10/24"
ifconfig_cxl1="inet 198.19.0.10/24"
```

- Configuration set 2: VLAN tagging

```
vlans_cxl0="2"
ifconfig_cxl0="up"
ifconfig_cxl0_2="inet 198.18.0.10/24"
vlans_cxl1="4"
ifconfig_cxl1="up"
ifconfig_cxl1_4="inet 198.19.0.10/24"
```

```
x Xeon E5-2650v2 & cxgbe, no VLAN tagging: inet4 packets-per-second
+ Xeon E5-2650v2 & cxgbe, VLAN tagging: inet4 packets-per-second
+--------------------------------------------------------------------------+
|+                                                                         |
|+                                                                       xx|
|+++                                                                    xxx|
|                                                                      |A||
|MA|                                                                      |
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x   5      10917371      10970686      10945136      10946743     22298.313
+   5       9056449       9104195       9064032     9075563.7     21531.387
Difference at 95.0% confidence
        -1.87118e+06 +/- 31966.4
        -17.0935% +/- 0.267353%
        (Student's t, pooled s = 21918.2)
```

Illustration 24: VLAN tagging impact

The performance drop of -17% is massive but it's a known problem caused by the long path a tagged frame needs to cross into FreeBSD network stack. An experimental patch (once again from Yandex) fixing this problem is in progress [13].

C) *Jail/vnet (VIMAGE)*

VNET is a powerful feature allowing to create isolated network stack for jails. But it needs kernel option VIMAGE that is not enabled by default on FreeBSD 11.1. The first step is to bench impact of just enabling this kernel option, without using it.

E5-2650v2 & cxgbe Xeon & Chelsio T540	GENERIC (median) Mpps	VIMAGE (median) Mpps	ministat
inet 4 forwarding	10.9	10.2	-6.25% +/- 0.29%
inet 6 forwarding	9.18	9.39	2.24% +/- 0.33

Illustration 25: VIMAGE impact of forwarding performance

The performance degradation is very negligible (about -6% on this setup) versus the benefit of VIMAGE.

The second step is to create a simple jail/vnet lab setup to measuring the impact:

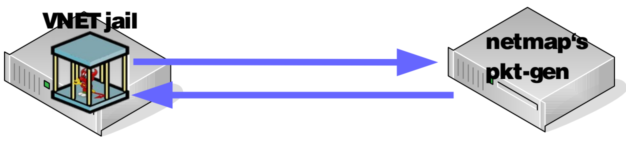


Illustration 26: Jail/vnet lab diagram

Configuration parameters for this lab:

/etc/rc.conf of the host:

```
ifconfig_cxl0="up -tso4 -tso6 -lro -vlanhwtso"
ifconfig_cxl1="up -tso4 -tso6 -lro -vlanhwtso"
jail_enable="YES"
jail_list="jrouter"
```

/etc/rc.conf of the jail/vnet:

```
gateway_enable=YES
ipv6_gateway_enable=YES
ifconfig_cxl0="inet 198.18.0.10/24"
ifconfig_cxl1="inet 198.19.0.10/24"
static_routes="generator receiver"
route_generator="-net 198.18.0.0/16
198.18.0.108"
route_receiver="-net 198.19.0.0/16 198.19.0.108"
```

E5-2650v2 & cxgbe Xeon & Chelsio T540	No Jail	VNET-Jail	Ministat
inet 4 forwarding	10.8 Mpps	11.0 Mpps	No diff. proven at 95.0% confidence
inet 6 forwarding	10.0 Mpps	10.0 Mpps	No diff. proven at 95.0% confidence

Illustration 27: jail/vnet forwarding performance

Very big surprise: There is no performance penalty if forwarding is done by a jail or the host system.

D) *if_bridge*

After creating multiple jail/vnet, the need for sharing the same VLAN between multiple jail/vnet will follow.

To sharing a LAN, if_bridge interface is the easiest solution. But how the insertion of if_bridge into the network stack impacts forwarding performance?

2 configuration sets are created: Once without bridge and one with a bridge.

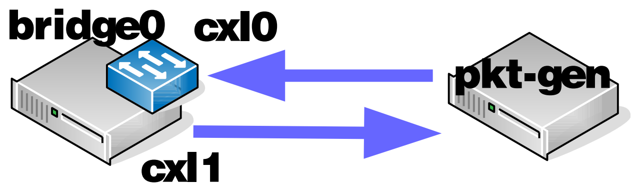


Illustration 28: if_bridge bench lab diagram

- Configuration set 1: No bridge

```
ifconfig_cxl0="inet 198.18.0.10/24"
ifconfig_cxl1="inet 198.19.0.10/24"
```

- Configuration set 2: Using a bridge

```
cloned_interfaces="bridge0"
ifconfig_bridge0="inet 198.18.0.8/24 addm cxl0
up"
ifconfig_cxl0="up"
ifconfig_cxl1="inet 198.19.0.10/24"
```

```
x Xeon E5-2650v2 & cxgbe, NO bridge: inet4 packets-per-second
+ Xeon E5-2650v2 & cxgbe, bridge: inet4 packets-per-second
+--------------------------------------------------------------------------+
|  +                                                                      x|
|++++                                                                    xx|
|                                                                        |A|
||AM|                                                                      |
+--------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x   5      11102006      11179490      11155098      11149783     28766.212
+   5       4040161       4322481     4201494.5     4178806.5     113801.03
Difference at 95.0% confidence
        -6.97098e+06 +/- 121051
        -62.5212% +/- 1.05729%
        (Student's t, pooled s = 83000.5)
```

Illustration 29: if_bridge bench results

The massive performance degradation (-63%) is a big surprise: if_bridge code is using lot's on non-optimised locking mechanism. Its usage needs to be avoided.

IV. TUNING FIREWALLS PERFORMANCE

Disclaimer: All benches in this section have the unique purpose of measuring the impact of firewalls configurations on forwarding throughput. None of these benches results can conclude than a firewall is better than another because a firewall can't be reduced to its only forwarding performance.

A) Firewalls impact on forwarding throughput

FreeBSD includes three firewalls (ipfw, pf and ipf) and this bench, by using minimum rule set for each is measuring their impact on the forwarding speed.

Configuration set 1: ipfw in stateful

```
#!/bin/sh
/sbin/ipfw -f flush
/sbin/ipfw add 3000 allow ip from any to any
keep-state
```

Configuration set 2: ipfw in stateless

```
#!/bin/sh
/sbin/ipfw -f flush
/sbin/ipfw add 3000 allow ip from any to any
```

Configuration set 3: pf in stateful

```
set skip on lo0
pass
```

Configuration set 4: pf in stateless

```
set skip on lo0
pass no state
```

Configuration set 5: ipf in stateful

```
pass in quick on lo0
pass out quick on lo0
pass in proto icmp from any to any keep state
pass out proto icmp from any to any keep state
pass out proto udp from any to any keep state
pass out proto udp from any to any keep state
pass in proto tcp from any to any flags S/SAFR
keep state
pass out proto tcp from any to any flags S/SAFR
keep state
```

Configuration set 6: ipf in stateless

```
pass out all
pass in all
```

Like all previous benches, 2000 UDP flows are generated to being forwarded by the 8-core Xeon and Chelsio setup for an objective of a 10Giga bit firewall

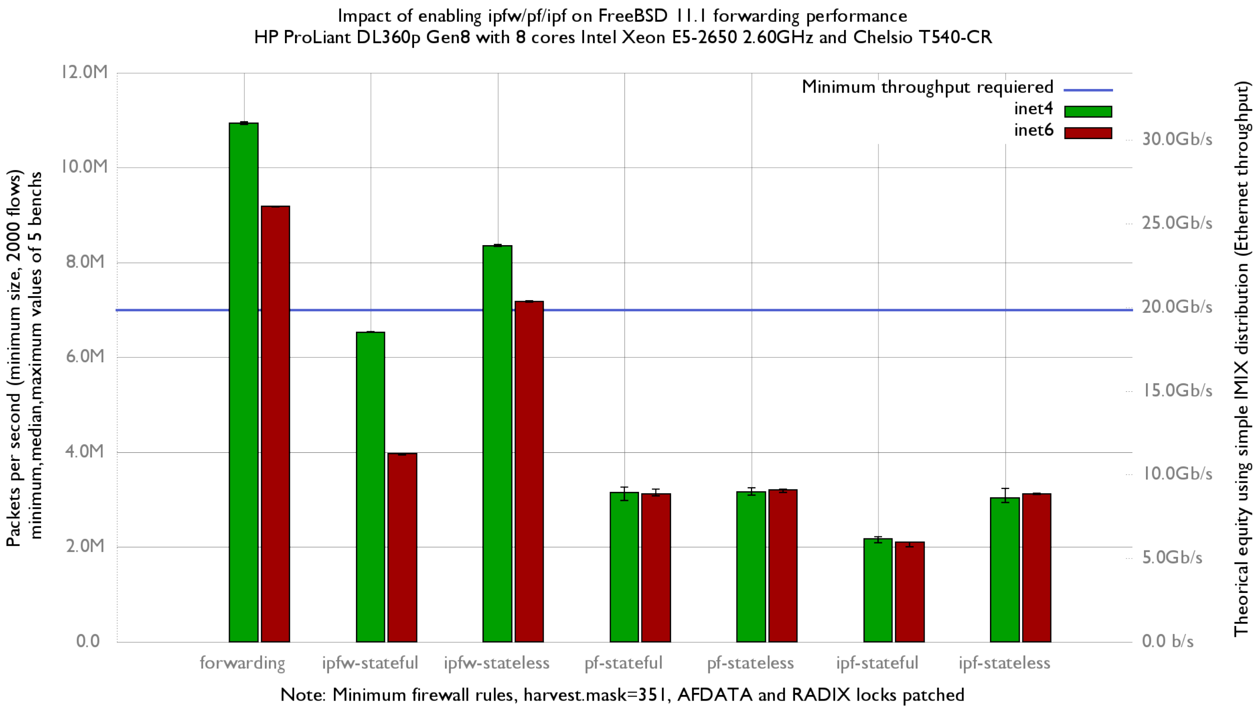


Illustration 30: firewalls impact on throughput

Only IPFW doesn't hurt too much the forwarding performance and allows this server to reach the minimum of 7Mpps for keeping its "10Gigabit IMIX stateless firewall" label.

Notice the bug regarding IPv6 performance with IPFW in stateful mode. This bug was related to a bad hash value and was fixed in head[14] and 11-stable.

B) Number of rules impact

Once firewalls enabled, next step is to measure the impact of number of rules. For each stateless firewall, new configuration sets are generated by inserting some number of non-matched rules before the last "allow all".

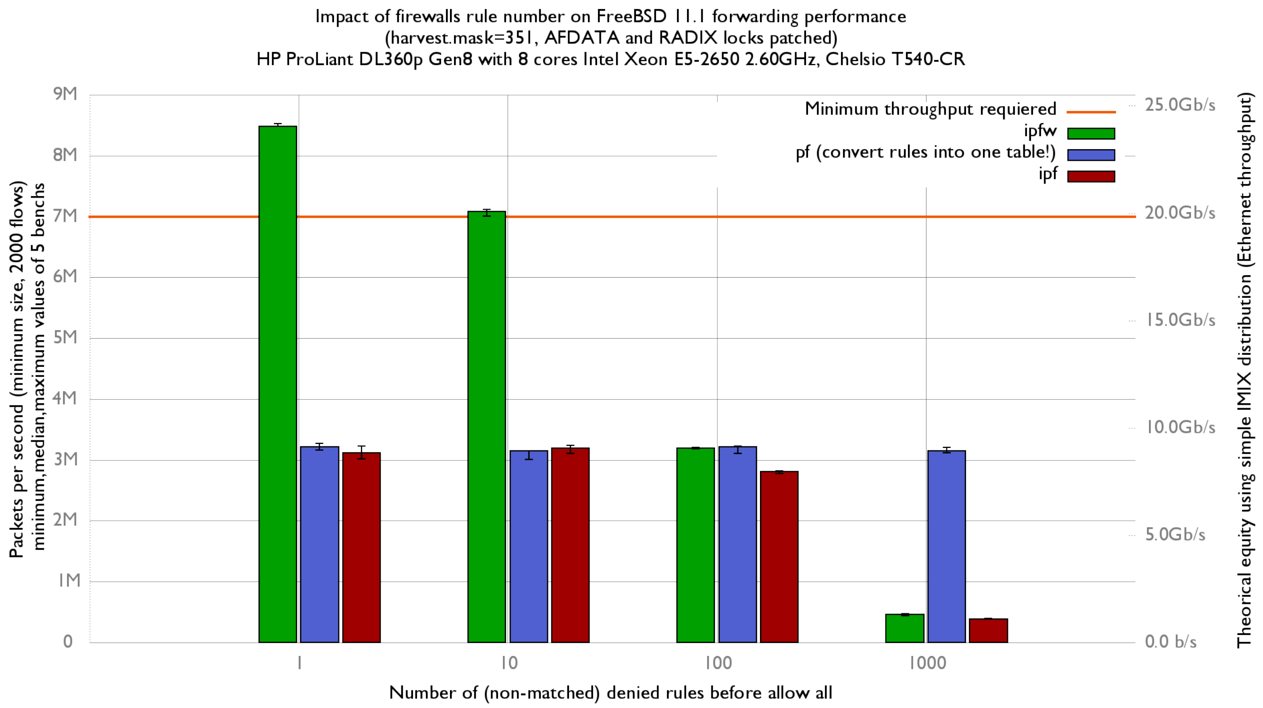


Illustration 31: Number of firewalls rules impact on forwarding performance

- IPFW is very sensitive to the number of rules: Starting at 10 rules we can already observe a degradation.
- IPFW and IPF became almost useless at 1000 rules.
- PF is converting all the simple rule set into a table. This bench **is wrong** because it didn't compare the same things: pf table vs ipfw & ipf rules number.

C) Table size impact

To fixing the previous bench (number of rules impact), a new bench is started but using the table concept. IPF firewall doesn't support table.

All deny rules used previously are replaced by a unique table with a variable number of entries and result in Illustration 32: Firewall table size impact on forwarding performance.

The behaviour between IPFW and PF is now equivalent and this bench shows the importance of using table. IPFW is useable as 10Gigabit IMIX stateless firewall.

D) Number of states impact

After the number of states or table size, the lookup speed of the state table needs to be benched too. IPFW and PF allow to configure 2 main parameters regarding their state table:

- Default maximum number of state
- Default size for their state hash table

The major difference between IPFW and PF is that PF creates 2 state entries for each flow (one state for each direction): This bench will generate up to 5M of unidirectional UDP flow, so:

- IPFW maximum state entry needs to be 5M
- PF maximum state entry needs to be 10M

But once increased the state table, the hash table needs to be increased too: A simple cross-multiplication between default values and targeted state table is used for calculating the size of the hash table for IPFW and PF.

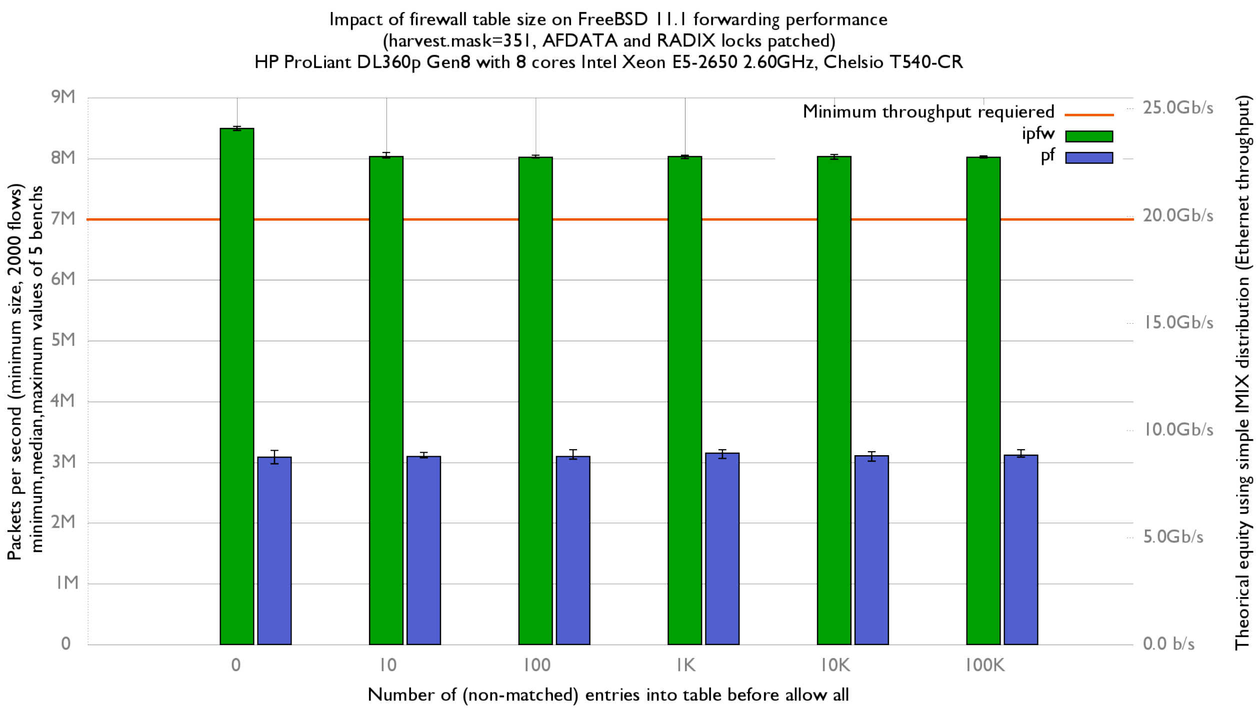


Illustration 32: Firewall table size impact on forwarding performance

keys	Default value	Increased value
dynamic rules net.inet.ip.fw.dyn_max	16 384	5 000 000
hash table size [max_dyn / 64 ?] (power of 2) net.inet.ip.fw.dyn_buckets	256	65 536 (max)

Illustration 33: IPFW state table limit and size of hash table

IPFW limit is 65 536 for its hash table size (net.inet.ip.fw.dyn_buckets), so theoretically the maximum number of states (net.inet.ip.fw.dyn_max) should be about 4M, but the value of 5M is used for this bench.

keys	Default value	Increased value
states limit set limit { states x }	10 000	10 000 000
Hash table size = state x 3 (power of 2) net.pf.pf_states_hashsize	32 768	33 554 432 *(max with 8GB RAM)*
RAM consummed (hashsize x 80) vmstat -m | grep pf_hash	2.5Mb	2.5Gb

Illustration 34: PF state table limit and size of hash table

PF needs a power-of-2 value for its hash table size and it allocates RAM for this table. So, once configured a value of 33 554 432 for it (net.pf.pf_states_hashsize), the maximum limit of number of state can be increased to 10M.

- Configuration set 1: IPFW

  /etc/sysctl.conf:

```
net.inet.ip.fw.dyn_max=5000000
net.inet.ip.fw.dyn_buckets=65535
```

  /etc/ipfw.rules

```
#!/bin/sh
/sbin/ipfw -f flush
/sbin/ipfw add 3000 allow ip from any to any
keep-state
```

- Configuration set 2: PF

  /boot/loader.conf:

```
net.pf.states_hashsize="33554432"
```

  /etc/pf.conf

```
set limit { states 10000000 }
set skip on lo0
pass
```

This bench result is in Illustration 35: Number of states impact on forwarding performance.

IPFW stateful engine didn't scale once reached 100K sessions while PF performance stay consistent.

A patch ("Make ipfw dynamic states lockless on fast path"), written by Andrey V. Elsukov (Yandex), fixes

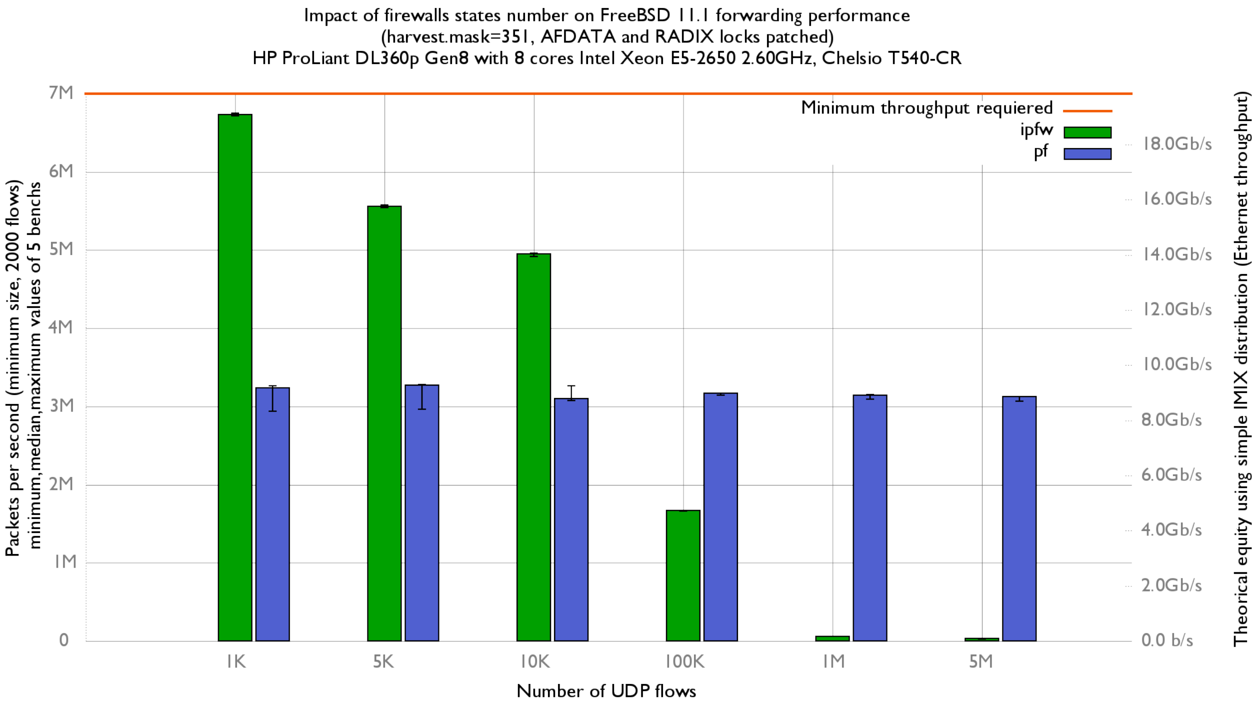


Illustration 35: Number of states impact on forwarding performance

IPFW stateful performance [15] and was committed into -head. Amongst many improvements, the hash table size didn't have limitation anymore, so the last bench with this patch applied on a FreeBSD 12-head is using these updated ipfw values:

```
net.inet.ip.fw.dyn_max=5000000
net.inet.ip.fw.dyn_buckets=5000000
```

This patch correctly fixes stateful IPFW behaviour, but still not enough to allow this 8-core Xeon server to be called a "10 Gigabit IMIX Stateful Firewall".

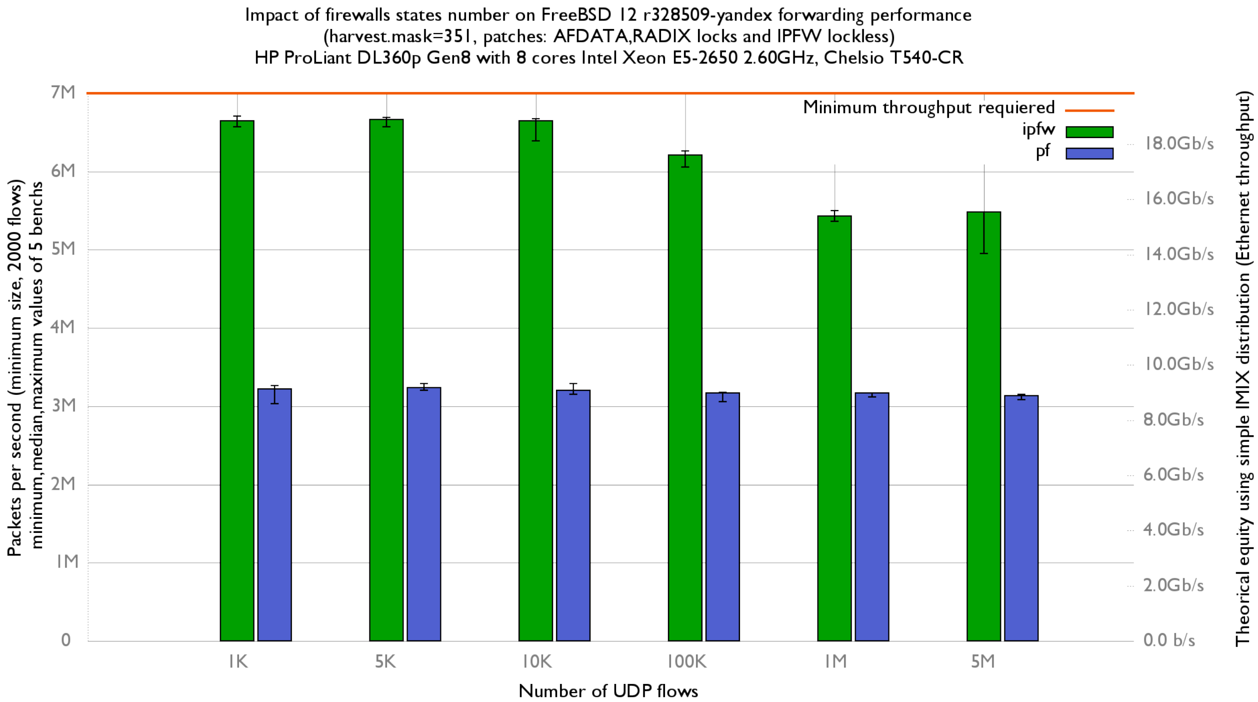


Illustration 36: Number of states impact with IPFW-lockless on forwarding performance

REFERENCES

1 https://www.freebsd.org/

2 http://www.ietf.org/rfc/rfc2544.txt

3 https://www.cisco.com/c/en/us/about/security-center/network-performance-metrics.html

4 https://en.wikipedia.org/wiki/Internet_Mix

5 https://github.com/luigirizzo/netmap

6 https://github.com/ocochard/BSDRP/blob/master/BSDRP/Files/usr/local/bin/nic-queue-usage

7 https://wiki.freebsd.org/ProjectsRoutingProposal

8 https://svnweb.freebsd.org/base/projects/routing/

9 https://people.freebsd.org/~ae/afdata.diff

10 https://people.freebsd.org/~ae/radix.diff

11 https://people.freebsd.org/~olivier/fbsd11.1.ae.afdata-radix.patch

12 https://github.com/ocochard/BSDRP/blob/master/BSDRP/Files/usr/local/etc/rc.d/chelsio_affinity

13 https://reviews.freebsd.org/D12040

14 https://svnweb.freebsd.org/base?view=revision&revision=309660

15 https://reviews.freebsd.org/D12685