Improving the DragonFlyBSD Network Stack

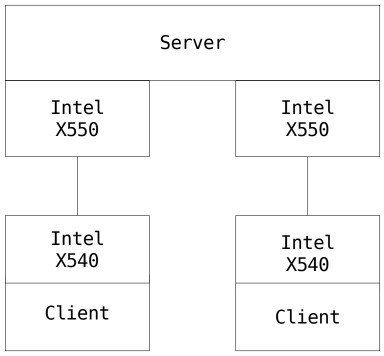
Yanmin Qiao
sephe@dragonflybsd.org
DragonFlyBSD project

Abstract

In this paper, we are going to describe various improvements we have made to DragonFlyBSD to help reduce and stabilize network latency, and increase network performance. How it works and why it works in DragonFlyBSD will be explained.

The configuration for nginx HTTP request/response performance and latency evaluation is shown in Figure.1. 1KB web objects are used in the evaluation, and a TCP connection only carries one HTTP request/response, i.e. short-lived TCP connections.

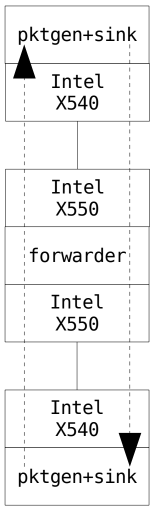
```
Server:
2x E5-2620v2, 32GB DDR3-1600, Hyperthreading enabled.

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
```

Figure.1

The configuration used to evaluate IP forwarding performance is shown in Figure. 2.



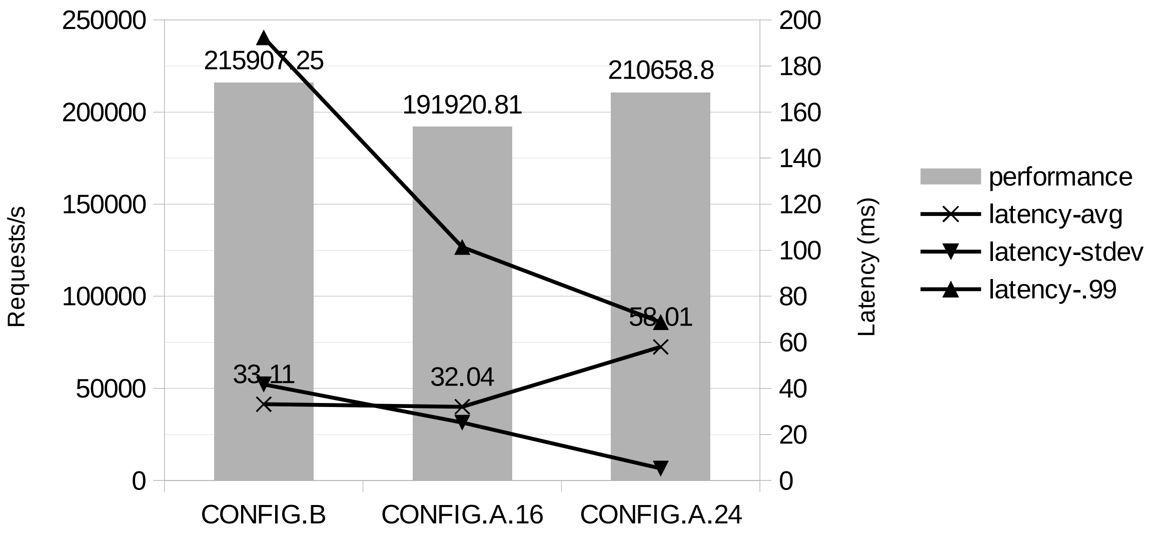
```
pktgen+sink:
i7-3770, 16GB DDR3-1600,
Hyperthreading enabled.

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.
```

Figure.2

1. Use all available CPUs for network processing

When DragonFlyBSD's network stack was originally adapted to use per-cpu thread serialization, it could only use a power-of-2 count of CPUs on the system. For example, a system with 24 CPUs would only use 16 of them for network processing. Though this integrated well with the RSS hash calculations of the time, it imposed significant limitations to the further exploration of the stack's potential. Also, 6, 10, 12 and 14 cores CPU packages are quite common these days.

Figure.3



1.1. Issues when using only a power-of-2 CPU count for network processing

One issue is the under-utilization of available computing power for kernel forwarding and bridging. For example a system with 24 CPUs will only use 16, while the rest of the CPUs (CPU17~CPU23) remain idle! Another issue occurs mainly with the userland applications which support CPU localized network operation. The userland application in this category that we exploit in this paper is nginx [1]. It has two CPU localization options on DragonFlyBSD. The first option is used to specify the number of workers, which handle HTTP requests from clients. The second option allows nginx to bind the workers to their respective CPUs. In order to properly support CPU localized network operation, nginx workers must be bound to their respective CPUs, and the number of workers must be same as the number of CPUs handling network processing. We label this configuration as CONFIG.A.16. If the number of CPUs in the system is not power-of-2, another choice we have is to use twice as many nginx workers as available CPUs and leave them unbound. We label this configuration CONFIG.B. This configuration does not CPU-localize its network operation. The performance and latency comparison is shown in Figure.3. Though CONFIG.A.16 gives a lower 99th percentile latency and latency standard deviation compared with CONFIG.B, its performance is lower, since the system's computing power is under utilized; only 16 out of the 24 CPUs are used in CONFIG.A.16. While the performance of CONFIG.B is higher, its 99th percentile latency and latency standard deviation is much worse due to non CPU-localized network operation which cause excessive wakeup IPIs and contention on the netisr's message ports. Sadly, none of these configurations are optimal.

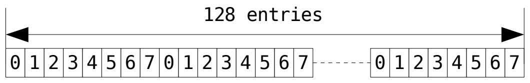
1.2. Use all available CPUs for network processing

It is easier for the OS itself, i.e. from the kernel side, to pick up a CPU from all available CPUs based on RSS hash:

```
(RSS_hash & 127) % ncpus
```

However, configuring the NIC's RSS redirect table, and RX ring to CPU bindings require more consideration. We avoid unnecessary IPIs when the input packets are redispatched to their respective netisrs ([2], Figure 1.2). If (ncpus % nrxrings) == 0, then seq(0, nrxrings-1) can be replicated N times to fill the RSS redirect table. The RX rings can be bound to CPUs belong to ((M * nrxrings) + seq(0, nrxrings-1)). For example, on a system with 24 CPUs, 3 NICs, the maximum number of RX rings of each NIC is 8:

- The RSS redirect table of each NIC:



- The RX rings of NIC0 can be bound to CPU0~CPU7, the RX rings of NIC1 can be bound to CPU8~CPU15, while the RX rings of NIC2 can be bound to CPU16~CPU23.

The difficulty comes, when (ncpus % nrxrings) != 0. If we still replicate seq(0, nrxrings-1) onto the RSS redirect table, the number of unnecessary IPIs during input packet redispatch would be increased substantially. For example, on a system with 24 CPUs, 1 NIC, the maximum number of RX rings of the NIC is 16:

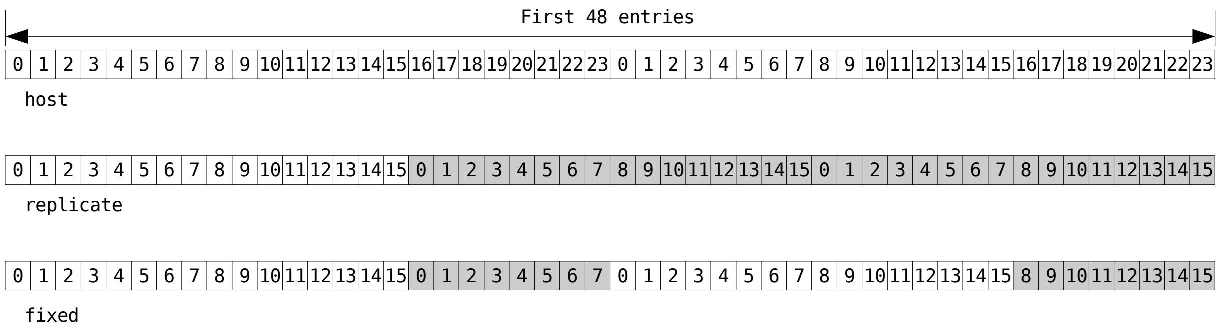
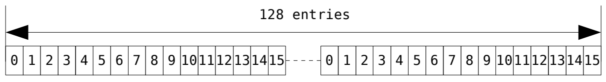


Figure.4

- The RSS redirect table:



- The RX rings are bound to CPU0~CPU15.

In Figure.4 the host's view is marked as 'host'. The above example is shown as 'replicate' in Figure.4. The mismatches are shaded. Each mismatch causes unnecessary IPIs during input packet redispatch.

But if the RSS redirect table is configured in the way shown in Figure.4 as 'fixed', large amounts of unnecessary IPIs during the input packet redispatch is avoided. In the example shown in Figure.4, the number of mismatches is halved, 32 mismatches are cut down to 16 in the first 48 RSS redirect table entries.

We thus developed a set of functions (if_ringmap_*() in sys/net/if.c) to help the drivers to create an optimal RSS redirect table, and help the drivers bind the RX/TX rings to CPUs in an optimal way. The basic ideas behind these functions are:

1) The available CPUs are divided into equal sized sets. The size of these sets is merely big enough to hold the rings. Rings can be bound to one set of CPUs starting from the first CPU in the set.

2) The RSS redirect table is divided into grids according to the set size calculated in 1). The first nrings entries of the grid are filled with seq(0, nrings-1). The rest of the entries for the grid are filled with subset of seq(0, nrings-1) in a round-robin fashion, e.g. if the grid is 5 with seq(0, 3), 0 will be used in the last entry of the first grid; 1 will be used in the last entry of the second grid, and so on and so forth.

1.3. Introduction to if_ringmap APIs

`if_ringmap_alloc()`

`if_ringmap_alloc2()`

Allocate a ringmap for RX or TX rings. The ringmap contains optimal configuration of RSS redirect table for RX ring, and optimal CPU binding information for RX or TX rings.

For some NICs, e.g. jme(4), the number of RX rings must be a power-of-2. `if_ringmap_alloc2()` is designed for these NICs.

`if_ringmap_free()`

Free the ringmap.

`if_ringmap_align()`

For some NICs, e.g. bce(4) and bnx(4), RX rings and TX rings must be assigned to the same set of CPUs, in the same order. If the number of RX rings and TX rings are same, the ringmaps allocated by `if_ringmap_alloc()` will meet this constraint fairly well. However, if the number of RX rings and TX rings are not same, then the CPU binding information in the ringmaps may not meet this constraint. This function takes RX and TX ringmaps and enforces the constraint.

`if_ringmap_match()`

For some NICs, e.g. igb(4) and ix(4), RX rings and TX rings can be operated completely independently. However, if the number of RX rings and TX rings are not same, the ringmaps allocated by `if_ringmap_alloc()` may not be able to distribute rings to CPUs evenly. This function takes RX and TX ringmaps and tries to make an even CPU distribution.

`if_ringmap_cpumap()`

Take a ringmap and a ring index, returns the CPU binding information for that ring.

`if_ringmap_rdrtable()`

Return a RSS redirect table. The RSS redirect table's size must be greater than or equal to 128.

1.4. The performance improvement and latency reduction

- IP forwarding performance is improved by 5%~6%:

	Before	After
Normal	9.2Mpps	9.8Mpps
Fast	11.3Mpps	11.9Mpps

- Nginx HTTP request/response improvement is shown in Figure.3 as CONFIG.A.24. Though the latency is drastically stabilized, performance drops a little (less than 2.5%), and the average latency increases. We believe this is a desireable tradeoff and the drawbacks of this change will be addressed in section 3.

2. Direct input with polling(4)

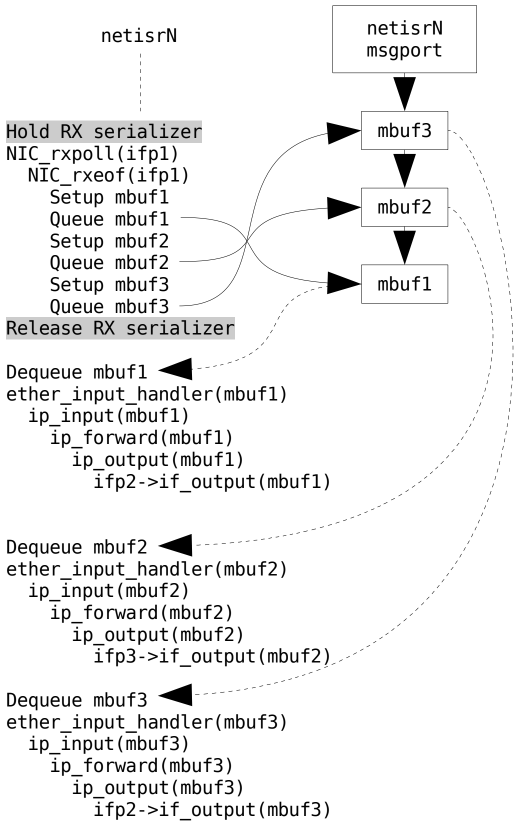
In DragonFlyBSD polling(4) is the primary mechanism to improve network performance in high-performance situations, reduce and stabilize latency, and prevent the system from being monopolized by packet flooding. Unlike the polling(4) we inherited from FreeBSD, the DragonflyBSD polling code understands multiple TX/RX rings ([2], section 1.7).

2.1. Excessive L1 cache eviction and refetching with non-direct input polling(4)

Figure.5 illustrates what happens when polling(4) gets several input packets with indirect input polling(4). Though the data portion of the mbuf is not touched if the NIC provides a RSS hash, the mbuf itself requires several modification on the NIC's input path, i.e. before the packet is queued for network protocol processing, the mbuf itself is cache hot, and probably is L1 cache hot. Since the NIC's input path will queue up to 50 packets on it's input path by default, the chance that the previously queued mbufs will be evicted from the L1 cache is high. When the network protocol processes the queued mbufs, they will have to be fetched into L1 cache again, thus reducing L1 cache utilization.

2.2. Direct input polling(4)

The main reason for doing indirect input polling(4) is that the NIC RX ring's serializer must be held when



```
Hold RX serializer
NIC_rxpoll(ifp1)
  NIC_rxeof(ifp1)
    Setup mbuf1
    Queue mbuf1
    Setup mbuf2
    Queue mbuf2
    Setup mbuf3
    Queue mbuf3
Release RX serializer

Dequeue mbuf1
ether_input_handler(mbuf1)
  ip_input(mbuf1)
    ip_forward(mbuf1)
      ip_output(mbuf1)
        ifp2->if_output(mbuf1)

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

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

Figure.5

the NIC RX ring's polling handler is running; requeuing input mbufs for network protocol processing prevents the NIC RX ring's serializer from being held for too much time, and also prevents a dangerous deadlock against the NIC RX ring's serializer. Let's investigate again why the NIC RX ring's serializer should be held when the NIC RX ring's polling handler is running. It is used to prevent various races from happening if the NIC is brought down while the NIC RX ring is handling heavy input traffic. The NIC RX polling handler is running in netisr and can be easily synchronized using a synchronous message. As long as the NIC driver can synchronize with the RX ring polling handler on it's stop path, RX ring's serializer does not need to be held before its polling handler, thus the network processing can run directly from the NIC RX ring's input path without worrying about the issues brought about by the RX ring's serializer. The direct input polling(4) is shown in Figure.6.

2.3. Resulting performance improvement

- IP forwarding performance is improved by 10%~12%:

```
                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)
```

Figure.6

	Before	After
Normal	9.8Mpps	11.0Mpps
Fast	11.9Mpps	13.2Mpps

- Not much nginx HTTP request/response improvement is observed, because unlike IP forwarding, the TCP input and userland processing takes the majority of the input processing time. For this path the only real improvement is that the latency standard deviation is reduced from 5.20ms to 4.60ms.

3. Kqueue(2) accept queue length report

In all BSDs kqueue(2) reports accept queue length when the EVFILTER_READ kevent for a listen(2) socket is ready. Most of the userland applications ignore kevent's accept queue length by using nonblocking listen(2) socket and polling until nothing can be accepted:

```
kevent(kq, NULL, 0, &kvt, 1);
for (;;) {
    s = accept(ls, &addr, &addrlen);
    if (s < 0) {
        if (errno == EWOULDBLOCK)
            break;
    }
}
```

We have no cure for this kind of userland application. However, some kqueue(2)-aware userland applications use the accept queue length provided by kevent, most noticeably nginx. Nginx uses the accept queue length like this:

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

The "`Setup the accepted socket`" part could be time consuming, which may destabilize and increase HTTP request handling latency, if the connection is short-lived and the accept queue length is too long.

3.1. Double edged sword: listen(2) socket's backlog

The backlog of the listen(2) socket is used to absorb temporary jitter and allows userland application to do more work before calling accept(2), so setting this to a low value, e.g. 32, will cause too many connection drops. However, setting the backlog to a relatively large value, e.g. 256, will also have a negative impact on the latency of short-lived request/response, as we have shown at the beginning of this section; the accept queue length reported by the kqueue(2) can wind up being very large.

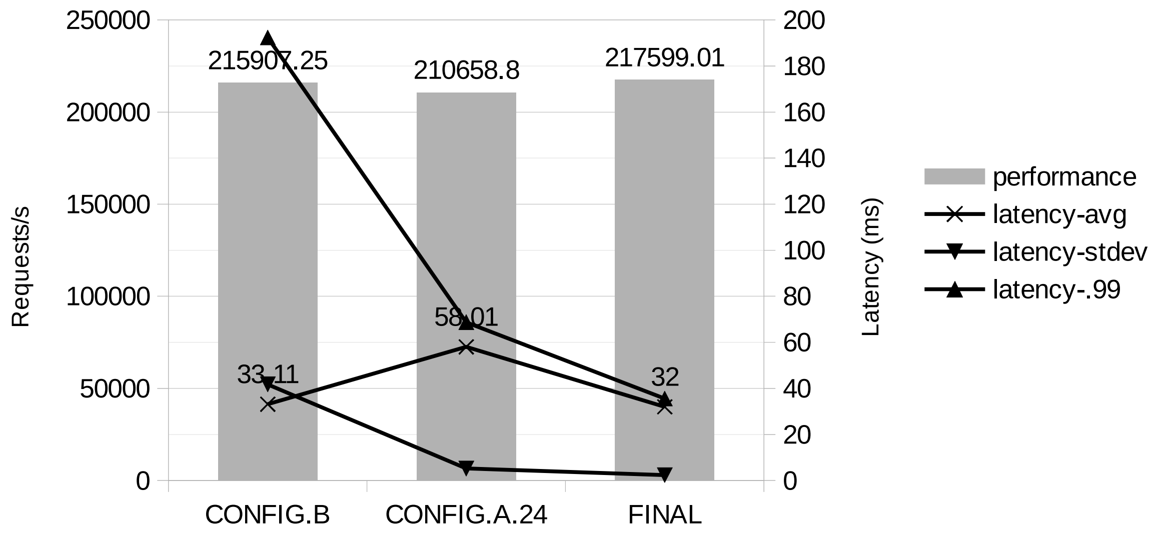
3.2. Don't be honest about the accept queue length

It's actually quite simple to enjoy both the benefit of a relatively large listen(2) socket backlog, while keeping the time consumption of the accept(2) loop controlled. We just put an upper limit on how long an accept queue length kqueue(2) can report. In DragonFlyBSD, we added a global sysctl(9) for this upper limit, and the default value for this upper limit is 32. It's arguable whether the userland application should enforce an upper limit on the accept(2) loop, or whether the kernel should put an upper limit on the kqueue(2)'s accept queue length. Currently, we believe it's much easier to do this on the kernel side, and all applications can enjoy the improvement without modification.

3.3. Resulting performance improvements and latency reduction

Nginx HTTP request/response improvements are shown in Figure.7, marked as FINAL. We borrow CONFIG.B and CONFIG.A.24 from section 1 for comparison. Compared with CONFIG.B, where we started, FINAL's performance is improved a bit, and the latency is greatly stabilized and reduced.

Figure.7



4. Sending aggregation from the network stack

Various studies [3] have already shown that writing the NIC TX ring's doorbell register costs a lot on the sending path. Though most of the high performance drivers have already deployed various mechanism to reduce their TX ring's doorbell register writing, the effectiveness still depends a lot on how the network stack dispatches the packets to the driver. In many cases, the network stack passes only one packet to the driver at a time, e.g. on the IP forwarding path, thus the TX ring's doorbell register ends up being written for each packet.

4.1. Explicit sending aggregation in netisr

In DragonFlyBSD, we don't call driver's if_start() method for each packet; the packet is just queued to the NIC TX ring's IFQ. IFQ maintains two counters: a byte counter and packet counter, which count how many bytes and packets are pending for sending. These counters are updated when a packet is queued to the IFQ, and are reset when the if_start() is called with the IFQ. The driver's if_start() method will be called with the IFQ, when either of the following conditions meet:

- IFQ's packet counter goes beyond a threshold, which is controlled by a sysctl(9). The threshold is 16 by default.
- IFQ's byte counter gets greater than 1500. This threshold can't be adjusted, and is heavily influenced by CoDel's settings.
- The netisr is going to sleep, i.e. there is no more network related work to do. This is achieved through a registerable 'rollup',

```
                        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
```

Figure.8

which is also used for TCP ACK aggregation. The registered 'rollup' list will be called before the netisr thread blocks for more messages. This works, since all network sends only happen via netisrs in DragonFlyBSD. This is shown in Figure.8.

- NIC TX ring's interrupt happens.

Because DragonFlyBSD's kernel threads (netisrs are plain kernel threads) are not preemptable by non-interrupt kernel threads, explicit sending aggregation does not cause latency issues.

4.2. Resulting performance improvements

- IP forwarding performance improvement

	pktcnt=4	pktcnt=16 (**)
Normal	11.0Mpps	11.3Mpps
Fast	13.8Mpps (*)	14.2Mpps

(*) Increased from 13.2Mpps (as shown in section 2.3) to 13.8Mpps, after mbuf objcache caching limit was increased and Matthew Dillon's VM improvements.

(**) Setting pkgcnt to anything above 16 gives no observable performance improvement.

- No observable improvement for nginx HTTP request/response, after the pktcnt is increased from 4 to 16.

5. Per CPU IPFW states

In DragonFlyBSD, IPFW was made MPSAFE back in 2008. At that time, the static rules were replicated to each netisr, but the states were read-write-locked as shown in Figure.9. This is mainly because the UDP MPSAFE work was not ready in 2008: UDP input and output for the same UDP 4-tuples were not running in the same netisr; this was fixed in 2014. Though read-write-locked states work quite well for relatively persistent states, e.g. TCP connections on which several HTTP requests/responses are exchanged, the performance degenerates drastically and the latency increases dramatically for short-lived states, e.g. TCP connections service only one HTTP request/response exchange, which is still quite common nowadays.

5.1. Independent per-CPU states

In DragonFlyBSD, the packets associated with a TCP/UDP address/port 4-tuple or other IP protocol's address 2-tuple are always processed in their respective netisrs, so we can remove the read-write state lock completely, and use one state table for each netisr; they never interfere with each other. Each state table can be operated (expand and shrink) independently and locklessly by their owning netisr as shown in Figure.10. For short-lived states, this avoids both heavy contention on the original global read-write lock and reduces cache-line bouncing between CPUs.

The only shared datum is the global state counter, which prevents the total number of states from going

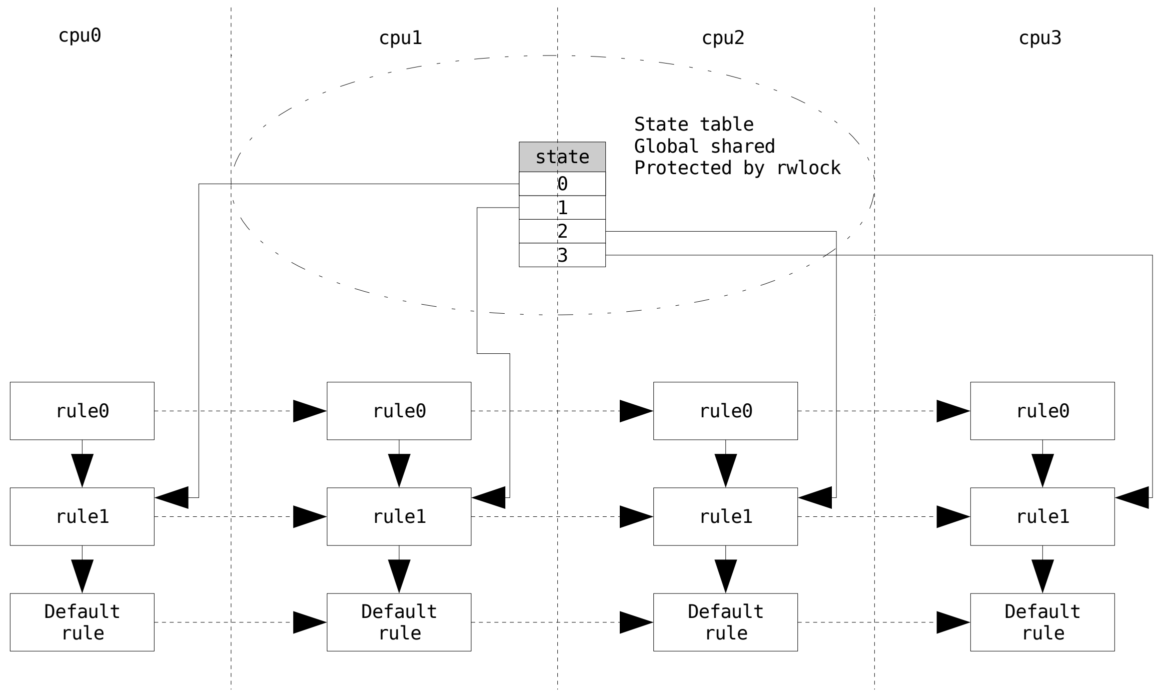


Figure.9

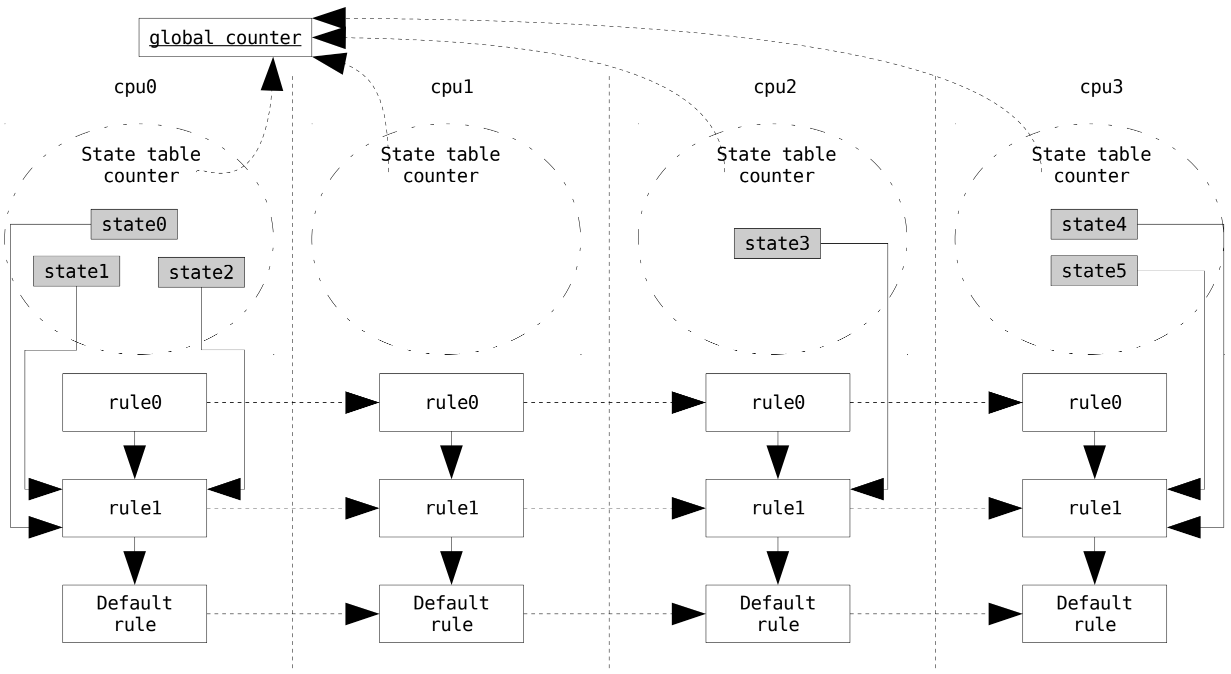


Figure.10

beyond a certain threshold. This datum is loosely updated, which allows 5% overflow. Given that memory is quite cheap nowadays, we believe this is fine. The loose update works like this:

- Each state table has a per-netisr counter, which is updated when a state is installed or removed. This update is lockless.
- Once per state table counter reaches certain threshold, it is added to the global state counter. Since this does not happen often, the effect of cache trashing by updating global state counter should be pretty minimal. Atomic operations are not used here though.
- At state installation, the global state counter is checked, which is a read-only operation. If it reaches the limit, the global state counter will be refreshed by merging all of the state counters from the per-cpu state tables, and will be checked against the limit again. If it broke its limit, the current CPU's state table will be GC'd first. If there are still too many states, IPIs are dispatched to all netisrs to GC their own state tables. This operation is constrained to once per second. However, on a properly configured system, the global state counter should not break its limit even when operating at a high frequency.

This loose counter updating mechanism was initially implemented by Matthew Dillon for the DragonFly-BSD's slab allocator, and was borrowed for IPFW and a number of other kernel subsystems.

5.2. The 'limit'

IPFW 'limit' command sets an upper limit for a set of states based on the conditions configured by the user, e.g. 10 states for a given address pair. When IPFW states were made per-CPU, the 'limit' got reworked and abstracted into two tiers: the track counter tier, which is globally shared, and the track tier, which is per CPU. The name 'track' is taken from pf(4). The track counter maintains the limit and a counter which counts how many states are tracked by it. The track points to a track counter and maintains a list of states tracked by it, as shown in Figure.11. If a tracked state is about to be installed, a track will be looked up locklessly, since the track table is per-CPU, and the track counter will be checked and updated using atomic-ops through the track. Contention can still happen in the track counter tier, if a new track is being added to the per-CPU track table, or the last track of the track counter is about to be removed. Loose counter updating can't be used for tracks, since the limit is usually pretty low.

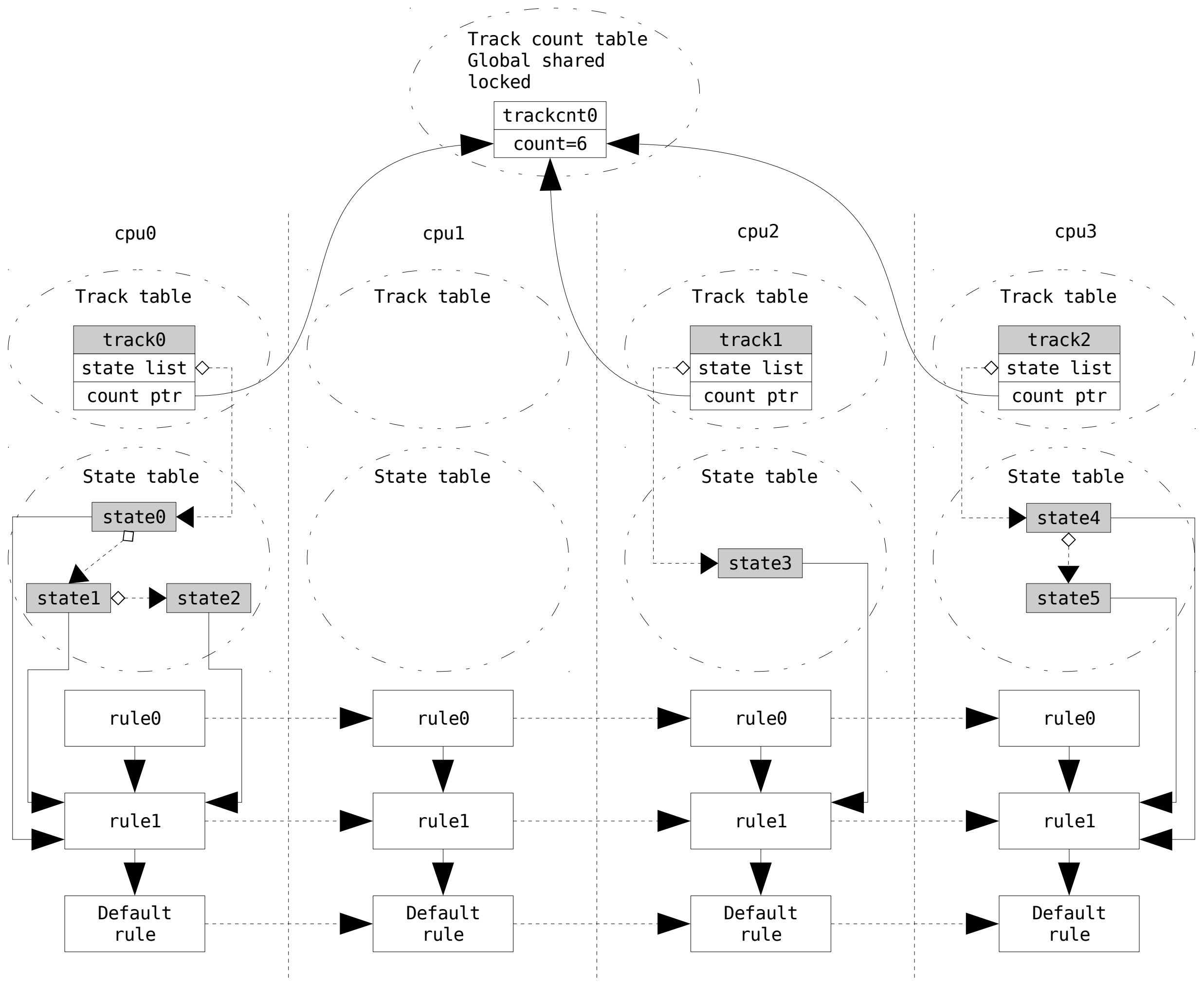


Figure.11

5.3. Resulting performance improvements and latency reduction

The performance/latency evaluation was conducted using nginx HTTP request/response test. IPFW is running on the server side. IPFW settings:

- The maximum number of read-writ-locked states is set to 500K. The hash table bucket size is 64K.
- The maximum number of states for per-CPU states is set to 100K (~14MB memory). No hash table settings, since per-CPU state tables uses red-black trees.
- Two IPFW rules are added:

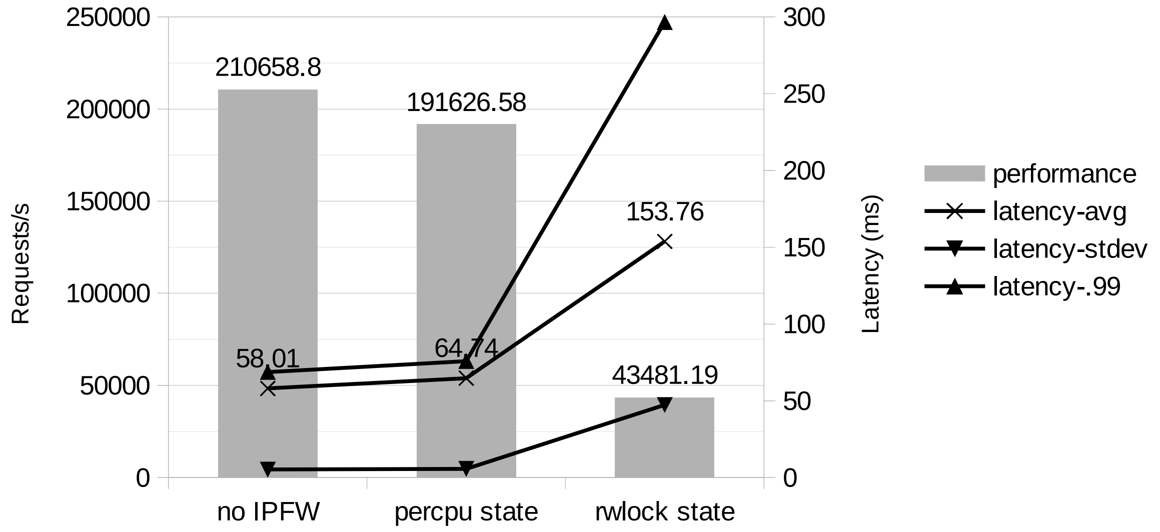
```
ipfw add 1 check-state
ipfw add allow tcp from any to me 80 setup keep-state
(default to deny)
```

The performance/latency comparison is shown in Figure.12. The results of CONFIG.A.24 (section 1.4) are shown as ‘no IPFW’, because the per-CPU IPFW state was implemented immediately after the changes in section 1. Though there are some performance/latency gaps between ‘no IPFW’ and ‘percpu state’, they are pretty close. When ‘percpu state’ is compared with ‘rwlock state’, you can see that the performance is greatly improved, while latency is significantly stabilized and reduced.

5.4. State based redirection

Since the per-CPU states gave pretty good results, we moved on and tried to use per-CPU states to build NAT/redirection. Redirection has been implemented as of this writing, and NAT work is still going on, so only redirection will be described here. Two states are used for a given redirected flow: the master state is identified by {saddr:sport, odaddr:odport}, i.e. before the translation; the slave state is identified by

Figure.12



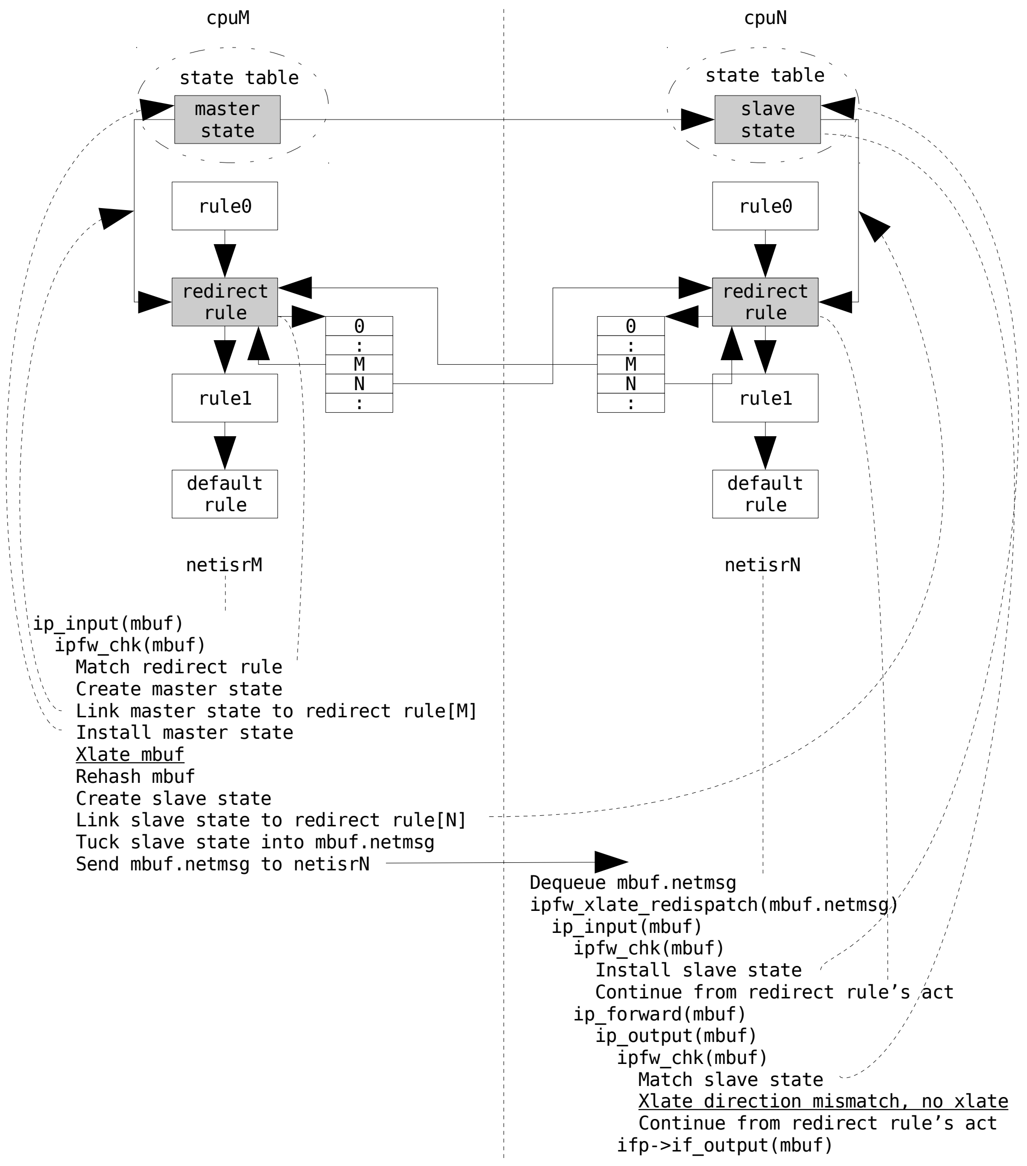
{ndaddr:ndport, saddr:sport}, i.e. after the translation. However, most often (almost 99% of the time), these two states will not be hashed to the same CPU, so we need ways to:

- Install the slave state on the remote CPU.
- Continue walking the rules on the remote CPU after translation.
- Remove the slave state on the remote CPU, once the master state is deleted.

In order to achieve this at IPFW redirection rule creation time, we save the addresses of its replica into each of the replicated rules, so that the corresponding IPFW redirection rule on the remote CPU can be located easily. The initial translation for the redirection is shown in Figure.13. Each IPFW redirection rule replica maintains a cross-reference counter. In the packet originating netisr, the cross-reference counter is incremented by 1, while in the remote netisr, i.e. the targeting netisr, the corresponding rule replica's cross-reference counter is decremented by 1. The rule replicas are only deleted in their respective netisrs: when the sum of all IPFW redirection rule replicas reaches 0, the replicas are no longer referenced, and the IPFW redirection rule replicas can be freed safely. This mechanism avoids expensive atomic reference counting at high frequencies. Deletion of states created by IPFW redirection rules is initiated by the deletion of master state, i.e. the slave state would never initiate deletion, even if the slave state expired. Master state deletion consists of 2 steps:

1. The master state is removed from its owning per CPU state table.
2. The master state is dispatched to the netisr owning the slave state for the final destruction.

The master/slave states also deploy cross reference counting mechanism themselves to make sure that they can be invalidated timely, and inflight packets can still check them safely.



```
ip_input(mbuf)
  ipfw_chk(mbuf)
    Match redirect rule
    Create master state
    Link master state to redirect rule[M]
    Install master state
    Xlate mbuf
    Rehash mbuf
    Create slave state
    Link slave state to redirect rule[N]
    Tuck slave state into mbuf.netmsg
    Send mbuf.netmsg to netisrN

Dequeue mbuf.netmsg
ipfw_xlate_redispatch(mbuf.netmsg)
  ip_input(mbuf)
    ipfw_chk(mbuf)
      Install slave state
      Continue from redirect rule’s act
    ip_forward(mbuf)
      ip_output(mbuf)
        ipfw_chk(mbuf)
          Match slave state
          Xlate direction mismatch, no xlate
          Continue from redirect rule’s act
        ifp->if_output(mbuf)
```

Figure.13

5. Reference

[1] nginx, http://nginx.org/

[2] 10 years with DragonFlyBSD network stack, https://leaf.dragonflybsd.org/~sephe/AsiaBSDCon%20-%20Dfly.pdf

[3] netmap, http://info.iet.unipi.it/~luigi/papers/20120503-netmap-atc12.pdf