10 years with DragonFlyBSD network stack

Yanmin Qiao
sephe@dragonflybsd.org
DragonFlyBSD project

Abstract

In this paper we are going to introduce the evolution of DragonFlyBSD's network stack in the past 10 years: what's the current state of its network stack, the important changes we did to it, why the important changes, and the lessons we learned. Finally, I'd like to list the areas that DragonFlyBSD's network stack can enjoy help hands.

1. The current state of DragonFlyBSD's network stack

1.1. Message passing and thread serialized

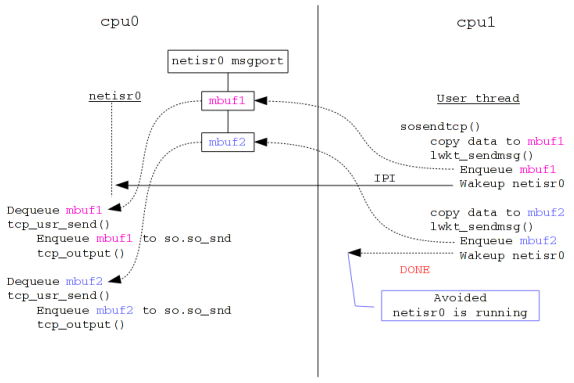


Figure 1.1

DragonFlyBSD's network stack uses message passing and is thread serialized. Most of the message passing used on hot code path is asynchronous. Each CPU has one network thread bound to it. This network thread will be referred as netisr-N (N is the CPU id) in the rest of this paper. Majority of the network protocol processing (IP/TCP/UDP) runs locklessly in netisr-N. UIPC syscalls and NICs just send messages to netisr-N, and netisr-N does the hard lift. A simplified version of UIPC sendto syscall is shown in Figure 1.1, and the simplified version of NIC received packet processing is shown in Figure 1.2.

1.2. Network data replication and partition

DragonFlyBSD's are replicated to, partitioned between, or grows independently on, each CPU, so accessing to the network data from the netisr-N is lockless (the exception is UDP wildcard INPCB table, which requires mostly uncontended per-cpu token to promise the data stability. Due to the flexibility of socket APIs, DragonFlyBSD's UDP implementation will not be introduced in this paper). For example, the static routing entries is replicated to each CPU, as shown in Figure 1.3. The TCP INPCBs are partitioned to each CPU when they are connected, as shown in Figure 1.4. The "cloned" routing entries grow independently on each CPU, as shown in Figure 1.5. The replication of wildcard sockets, e.g. TCP listen sockets, is different from other network data replication: only reference to the wildcard sockets is replicated, but the wildcard sockets themselves are not, as shown in Figure 1.6. The wildcard sockets lookup is lockless when they are replicated in this fashion. However, enqueuing datagrams to a UDP

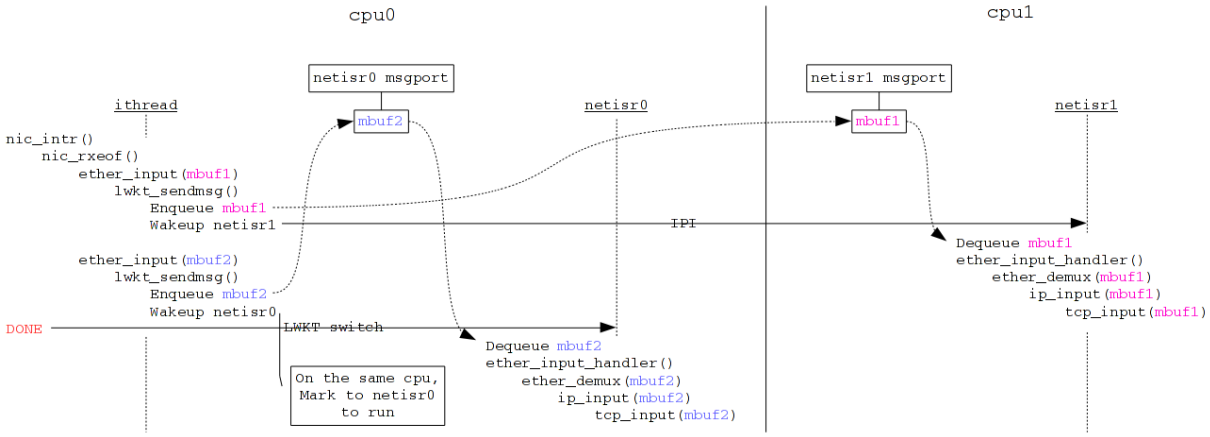


Figure 1.2

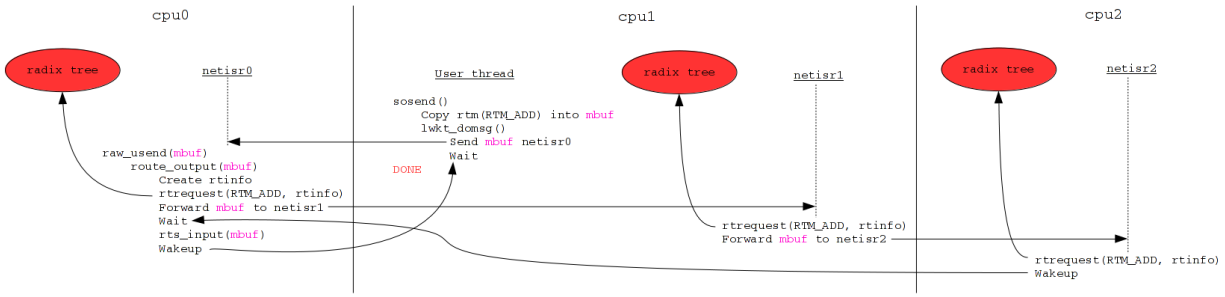


Figure 1.3

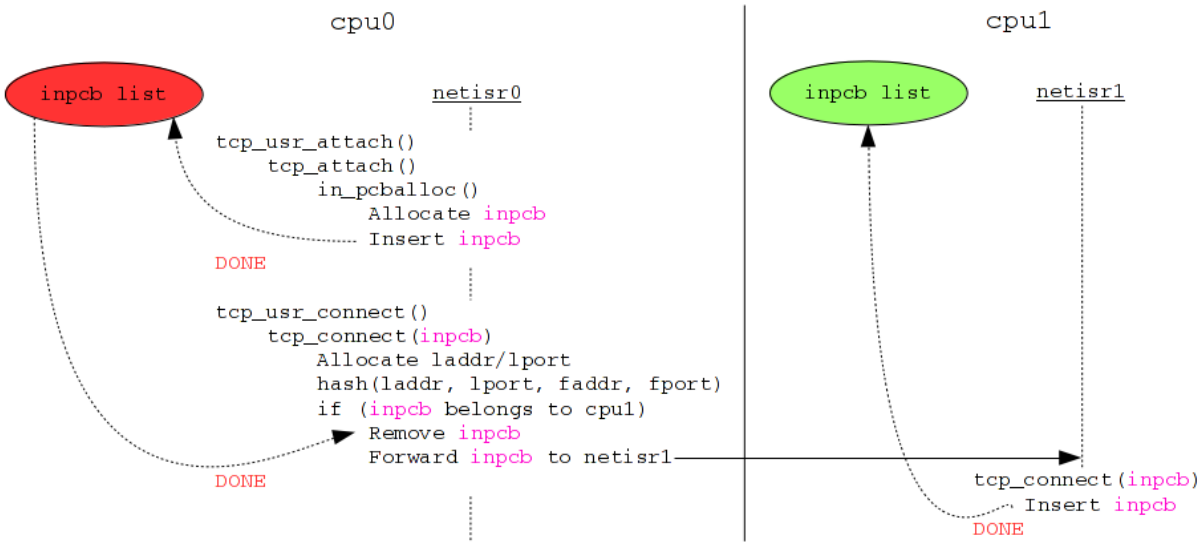


Figure 1.4

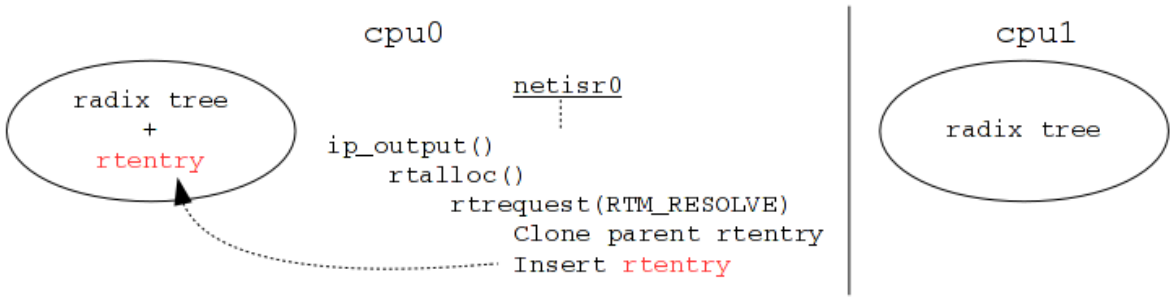


Figure 1.5

wildcard socket or enqueuing connected TCP INPCBs to a TCP listen socket accept queue will cause serious contention as shown in Figure 1.6; we will explain how we address this contention in section 1.6.

1.3. Symmetric receive side scaling (RSS)

DragonFlyBSD uses symmetric receive side scaling (RSS) to partition network data. RSS calculates the packet hash using the following formula:

```
toeplitz(laddr, lport, faddr, fport, key)
```

What's the difference between RSS and symmetric RSS? Symmetric RSS duplicates 2 bytes to create the 40 bytes RSS key, so it allows the addresses-ports 4-tuple to be commutative. For example, if the example key in the Microsoft RSS specification [1] is used:

```
toeplitz(192.168.1.1, 3007, 192.168.1.2, 2003, key) =
0x9e51fb2a

toeplitz(192.168.1.2, 2003, 192.168.1.1, 3007, key) =
0x4c472df4
```

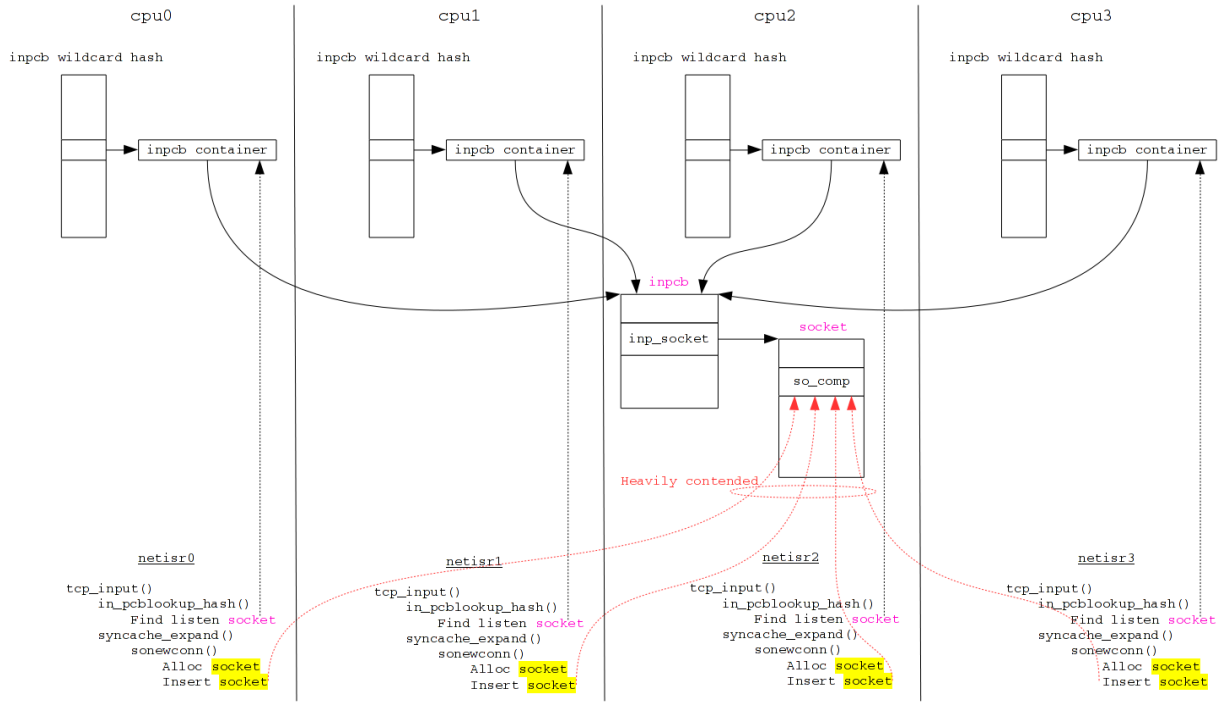


Figure 1.6

As you can see the addresses-ports 4-tuple is not commutative for non-symmetric RSS. Let's see the example of the symmetric RSS, which creates a 40 bytes key by duplicating 0x6d and 0x5a:

```
toeplitz(192.168.1.1,    3007,    192.168.1.2,    2003,
key_dup) = 0xe501e501

toeplitz(192.168.1.2,    2003,    192.168.1.1,    3007,
key_dup) = 0xe501e501
```

Since symmetric RSS only affects how the RSS key is generated, applying it to the NIC chips does not posing any issues. NIC drivers just use toeplitz_get_key() KPI to extract the system RSS key and configure it to the chips. And symmetric RSS reduces the Toeplitz hash computation burden on the host side by using 2 precalculated result arrays. On DragonFlyBSD, the TCP INPCBs connected through connect(2) are distributed to the CPUs based on the RSS hash value of the 4-tuple: toeplitz(laddr, lport, faddr, fport). However, the hash of the received packets belonging to a specific connection is calculated by the NIC chips using toeplitz(faddr, fport, laddr, lport). The symmetric RSS solves this issue in a simple and much less error prone fashion. And the symmetric RSS paves the way to the per-cpu state table for the stateful firewalling, which is not implemented by DragonFlyBSD as of this write yet.

1.4. Illustration of the data sending and receiving

Typical TCP data sending is shown in Figure 1.7. Typical TCP data receiving is shown in Figure 1.8.

1.5. Accessing network data from non-netisr threads

The accessing to the network data are lockless from netisr threads, however, sometimes non-netisr threads also need to access the network data, e.g. the sysctl to dump the INPCB list. Let's use the INPCB list dumping sysctl as an example:

- The non-netisr kernel thread is migrated to CPU-N; N starts from 0.
- For the INPCB on the per-cpu INPCB list, make a kernel space copy in XINPCB format, which is used by the userland too. Before copying the in kernel XINPCB out to the userland, which may block, a "cursor INPCB" is inserted after the INPCB we made the in kernel XINPCB copy. Then the in kernel XINPCB is copied out to the userland. Next INPCB is located using the "cursor INPCB" inserted before the copyout, so that even the

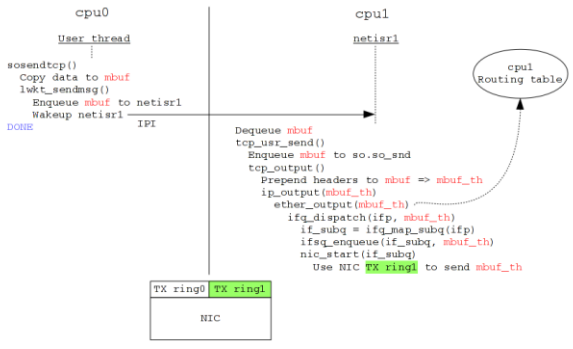


Figure 1.7

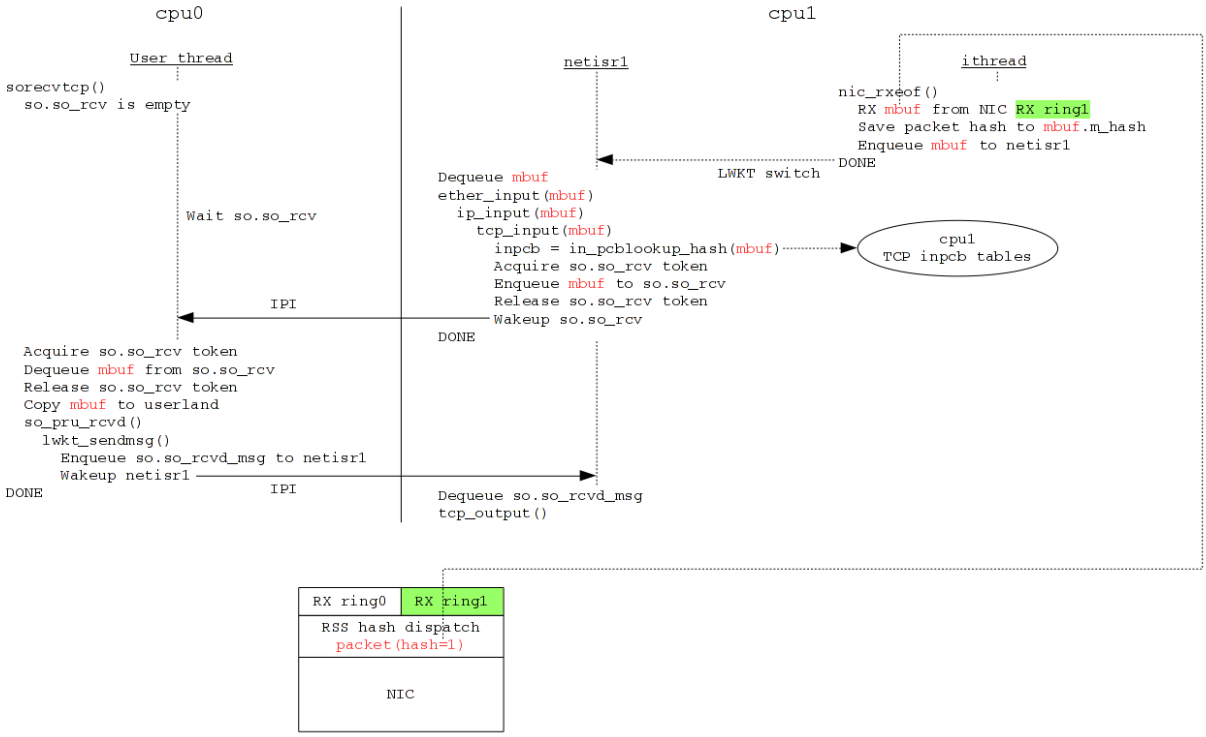


Figure 1.8

INPCB list is altered during the copyout, we still can move on safely. As shown in Figure 1.9.

- Increase N by 1, if N is less than number of CPUs on the system, then restart. Else, INPCB list dumping is done.

Several DragonFlyBSD specific kernel scheduling features make this possible:

- Kernel thread will only be preempted by the interrupt kernel threads. The INPCB list is not accessing or altered by the interrupt threads. So everything happens before/after the copyout will not be interrupted.
- Kernel thread will not be moved to other CPUs by kernel thread scheduler.

1.6. Google's SO_REUSEPORT

The basic idea of Google's SO_REUSEPORT is shown in Figure 1.10. For TCP, it is intended to improve userland workload distribution, since the accepted socket is distributed to different listen sockets based on SYN's packet hash. And obviously, it significantly reduces the TCP listen socket accept queue contention. While on DragonFlyBSD, the implementation of Google's SO_REUSEPORT is shown in Figure 1.11. There is one drawback in the SO_REUSEPORT implemented by Google for Linux: if one of the SO_REUSEPORT TCP listen sockets is closed, then all accepted sockets on its accept queue is aborted. On DragonFlyBSD, we addressed this issue by moving the accepted sockets of the closing SO_REUSEPORT TCP listen socket to other SO_REUSEPORT TCP socket listening on the same local port. Figure 1.12 shows the performance improvement and latency reduction of the

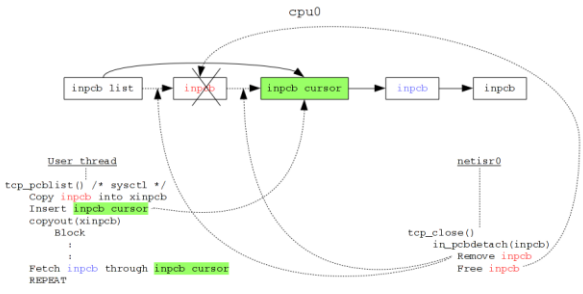


Figure 1.9

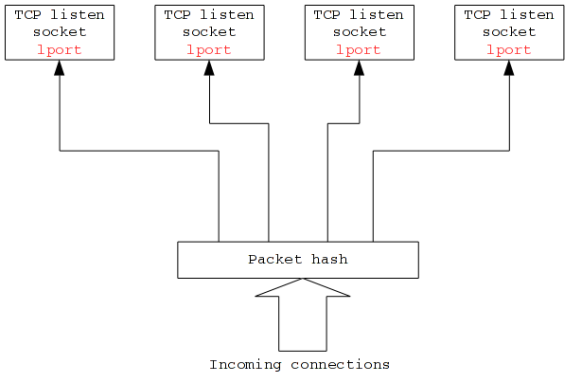


Figure 1.10

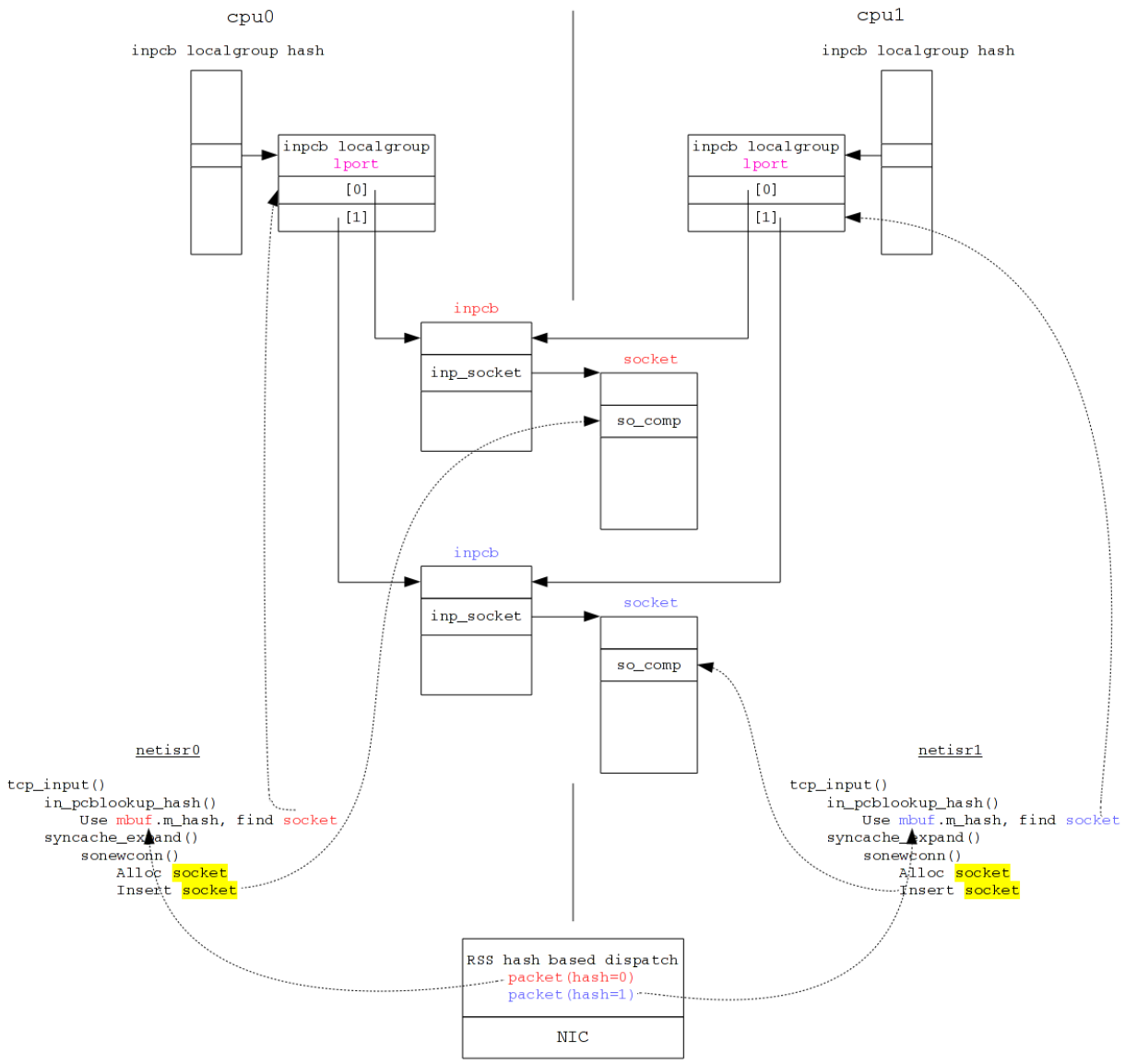

cpu0
inpcb localgroup hash
cpu1
inpcb localgroup hash
inpcb localgroup
lport
[0]
[1]
inpcb localgroup
lport
[0]
[1]
inpcb
inp_socket
socket
so_comp
inpcb
inp_socket
socket
so_comp
netisr0
tcp_input()
in_pcblookup_hash()
Use mbuf.m_hash, find socket
syncache_expand()
sonewconn()
Alloc socket
Insert socket
netisr1
tcp_input()
in_pcblookup_hash()
Use mbuf.m_hash, find socket
syncache_expand()
sonewconn()
Alloc socket
Insert socket
RSS hash based dispatch
packet (hash=0)
packet (hash=1)
NIC

Figure 1.11

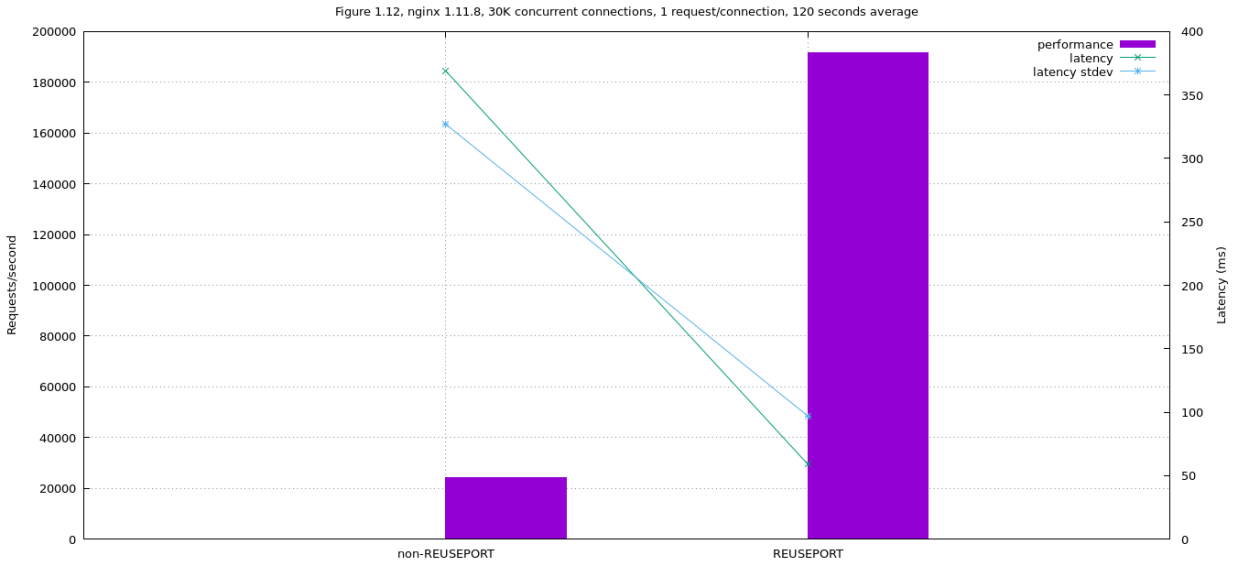


Figure 1.12, nginx 1.11.8, 30K concurrent connections, 1 request/connection, 120 seconds average
performance
latency
latency stdev
Requests/second
Latency (ms)
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
400
350
300
250
200
150
100
50
0
non-REUSEPORT
REUSEPORT

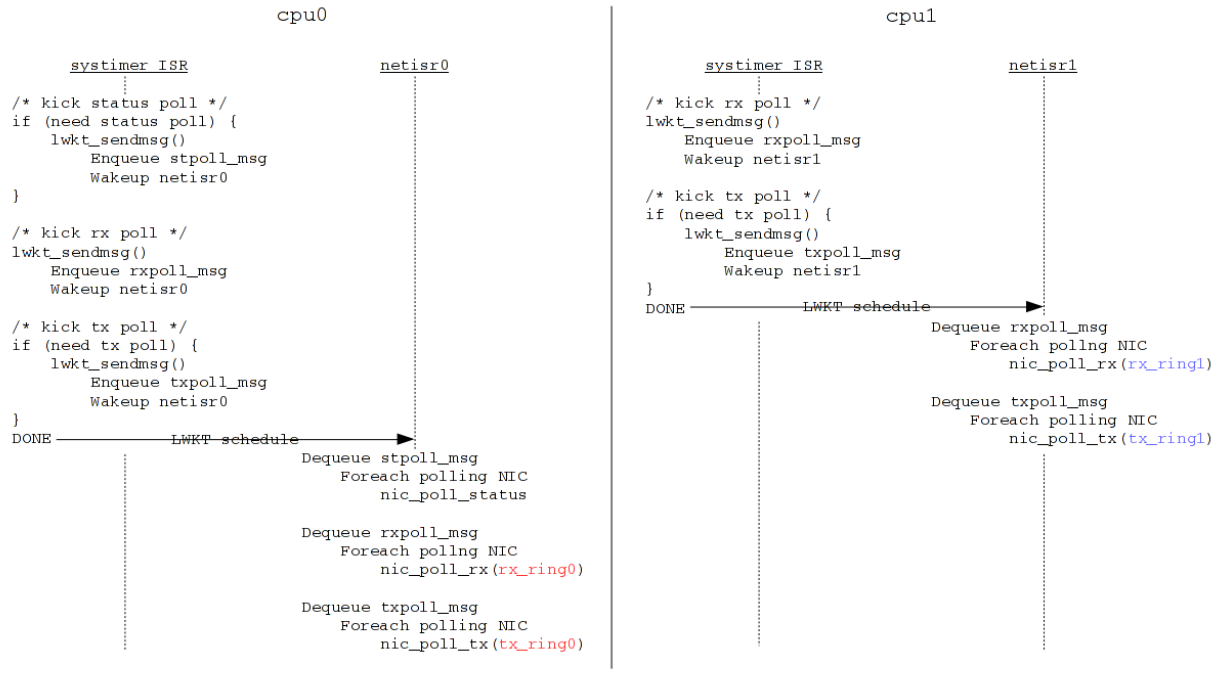


Figure 1.13

1.7. NIC ring based polling(4)

The polling(4) inherited from FreeBSD is NIC based. However, all modern NICs support multiple RX rings at least, and most NICs for the server market support multiple TX rings in addition to multiple RX rings, so the NIC based polling(4) is no longer sufficient. The original polling(4) seems to support non-NIC devices, though only NIC drivers have polling(4) support. We then rework the polling(4), dropped the non-NIC device support, and make it based on NIC ring. The basic NIC ring based polling(4) is shown in Figure 1.13. The NIC drivers supporting the NIC ring base polling(4) only need to implement the ifnet.if_npoll method like the following pseudo code:

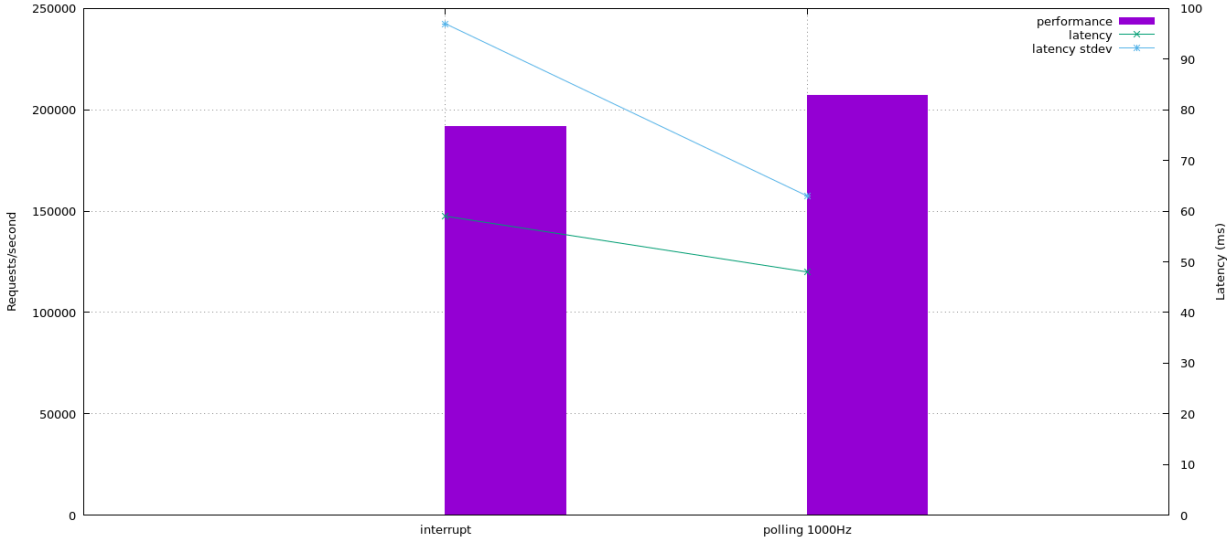
```
void
nic_npoll(struct ifnet *ifp, struct ifpoll_info *info)
{
   if (!info) {
       Enable interrupt;
       Return;
   }

   /* Poll status, e.g. link status. */
   info->ifpi_status.status_func = nic_poll_status;
   /* Per TX ring setup. */
   for (i = 0; i < tx_ring_cnt; ++i) {
       /* Poll TX-done, and kick start if_start. */
       info->ifpi_tx[i].poll_func = nic_poll_tx;
       info->ifpi_tx[i].arg = tx_ring[i];
   }
   /* Per RX ring setup. */
   for (i = 0; i < rx_ring_cnt; ++i) {
       /* Poll RXed packets. */
       info->ifpi_rx[i].poll_func = nic_poll_rx;
       info->ifpi_rx[i].arg = rx_ring[i];
   }
   Disable interrupt;
}
```

nic_poll_status, nic_poll_tx and nic_poll_rx is quite self-explain. Another benefit of the NIC ring based polling(4) is that more NIC rings can be enabled, i.e. the number of usable RX/TX rings is not constrained by the available MSI-X vectors. For example, even though Intel 82576 has 16 RX/TX rings [2], it only has 10 MSI-X vectors, which means 8 TX/RX rings are usable in interrupt mode. If NIC ring based polling(4) is used, then all 16 RX/TX rings can be utilized with Intel 82576. NIC ring based polling(4) helps performance and reduces latency as shown in Figure 1.14.

1.8. Some useful TCP features

RFC6675 instead of NewReno is used if SACK can be negotiated. DragonFlyBSD uses Eifel detection (RFC3522) to detect false retransmission, and uses Eifel response (RFC4015) to recover from the false timeout. NCR (RFC4653) is enabled by default to avoid false fast retransmission. Initial RTO is reduced to 1 seconds

Figure 1.14, nginx 1.11.8, 30K concurrent connections, 1 request/connection, 120 seconds average



according to RFC6298. Per-routing IW settings are available according to RFC6928 (IW10), though DragonFlyBSD only increases the default IW from 3 to 4.

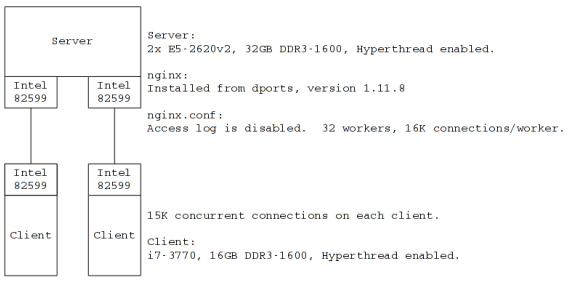
1.9. Performance

The performance and latency of HTTP/1.1 short lived connection, i.e. one request per TCP connection, are shown in Figure 1.14. As shown in Figure 1.14, NIC ring based polling(4) gives the best performance (207Ktps) and lowest average latency (48ms). The test environment and parameters are shown in Figure 1.15. The IPv4 forwarding performance is 5Mpps. The test environment and parameters are shown in Figure 1.16.

2. The evolution of DragonFlyBSD's network stack

It did take quite some time for DragonFlyBSD's network stack to reach its current state. The important changes will be detailed in this section.

2.1. Back in 2006

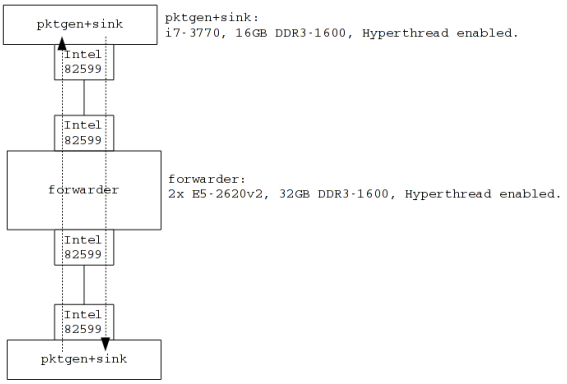



```
MSL on clients and server are changed to 10ms by:
route change -net net -msl 10

/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 1.15

These were what we have in 2006:

- Mbuf object cache, which greatly accelerated the mbuf cluster allocation.
- Three network threads on each CPU. One handles TCP, one handles UDP and another thread handles other network protocols.
- Message passing was synchronous for all UIPC syscalls.
- Message passing used DragonFlyBSD's IPIQ mechanism.




```
/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

Flow control is disabled on all testing machines:
ifconfig ix0 -mediaopt flowcontrol

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

Performance counting:
Only packets sent by 'forwarder' are counted.
```

Figure 1.16

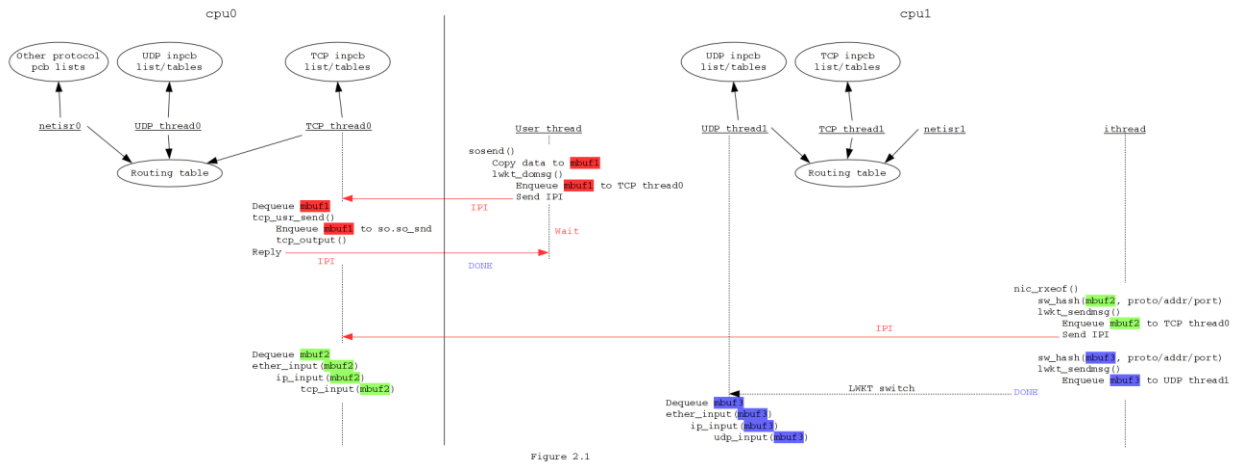


Figure 2.1

- Network data were replicated and partitioned properly.
- Home-made 4-tuple commutative hash was used to partition network data.
- Everything was still under BGL.

These were designed and implemented by Matthew Dillon and Jeffrey Hsu. The big picture is shown in Figure 2.1.

2.2. Packet batching

In 2008, network stuffs became mature enough, so we began to remove BGL from the per-cpu TCP threads and per-cpu UDP threads. The first issue unveiled was the higher than expected IPI rate between the NIC interrupt thread (only legacy interrupt was supported at that time) and the per-cpu netisr. The solution: aggregating packets in NIC driver's interrupt handler and dispatching the packet batch to the per-cpu netisr, as shown in Figure 2.2. Thanks to the interrupt moderation, and interrupt thread cannot be preempted on DragonFly-

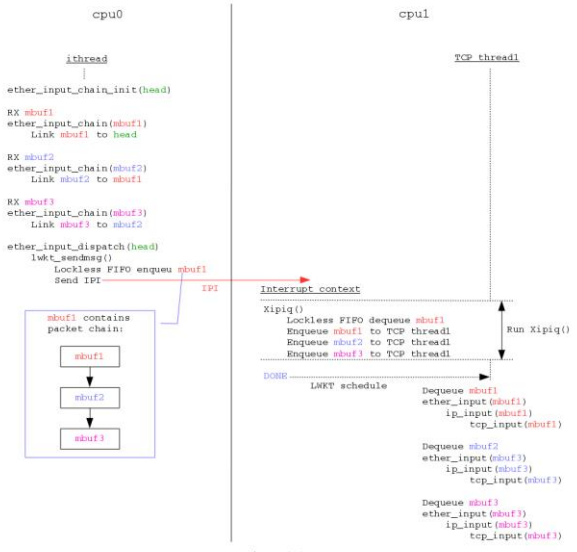


Figure 2.2

BSD, no extra latency was noticed after this change, and IPv4 forwarding performance was improved bit. Lesson we learned: avoid IPI if possible, and aggregation in DragonFlyBSD is good.

2.2. Introduction of symmetric RSS

In 2009, Hasso Tepper raised a question about the multiple RX ring support. And at that time Intel's 82571 had already been widely available, and it supports 2 RX rings. After studying the RSS specification, we decided that the RSS and DragonFlyBSD's data partition mechanism are good match. The first symmetric RSS hash implementation used Intel 82571, and the symmetric RSS hash was used as the default network data partition hash in 2010. A lot of work had been put the NIC drivers to add the symmetric RSS support since then; as of today, we have emx(4), igb(4), jme(4), bce(4), bnx(4), mxge(4) and ix(4) supporting symmetric RSS.

2.3. Single per-cpu network thread

In 2010, with more bits of the kernel becoming MP-safe, the issues of multiple networking threads per-cpu were found. First of all, since non-interrupt kernel threads do not preempt each other, networking threads on the same CPU tended to lock out of each other, which caused issues like lacking of mbufs (they were sitting on the locked-out network threads' message port). Thus the per-cpu TCP thread, UDP thread and netisr became one per-cpu netisr. After this, the network threading is exact same as what we have nowadays. Lesson we learned: multiple per-cpu threads handling almost the same kind of tasks do not play well.

2.4. Asynchronous message passing

In 2011, the synchronous message passing for the UIPC syscalls became the bottleneck for further improvement, as shown in Figure 2.3. We took on the socket sending as the first step: since most of the return values for

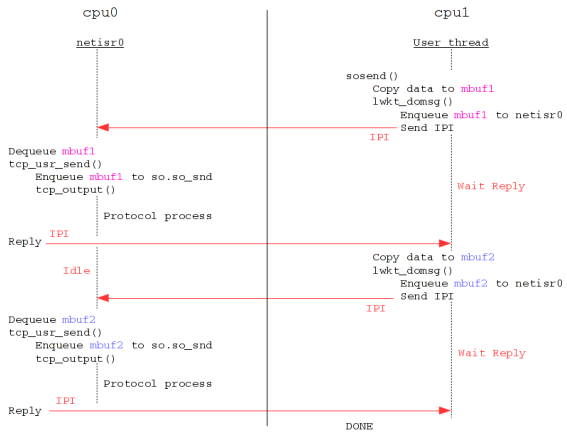


Figure 2.3

sending is either not widely used, e.g. UDP socket sending errors (Linux always returns 0), or is actually asynchronous, e.g. TCP socket's sending errors caused by connection dropping, we made the sending asynchronous and always return 0 for non-blocking socket if its sending buffer is not full as shown in Figure 1.1. This takes the advantage of the full pipe-lining effect, improves performance a lot, and greatly reduces the scheduling cost as mentioned in [3]. Once the message passing for send UIPC syscall was made asynchronous, IPIQ for the message passing could no longer be used due to the message ordering issue discovered later on. The issue is shown in Figure 2.4. We then switched to spinlock based message port for netisr-N, since the protocol processing is much more time consuming compared with the dequeue/enqueue to the netisr-N's message port, not much contention was/is observed on the netisr-N message port's spinlock. And packet batching in the NIC interrupt is no longer necessary, since spinlock based message port sends IPI only is the target thread owning the message port is waiting. Ever since the discovery that asynchronous message passing is tremendous, great amount of effort has been made to use asynchronous message passing for UIPC syscalls used by application hot code path, and the result is always encouraging. For example, after we made the setsockopt(TCP_NODELAY) asynchronous, the performance of short lived HTTP/1.1 (one request per TCP connection) connection increased 19%, from 156Ktps to 186Ktps. And we are still moving on in that direction. Lesson we learned: asynchronous message passing is one of the critical parts for a performant message passing system.

2.5. Statistics maniac

In 2013, during the round-up of IPv4 forwarding performance improvement cycle, we found that the largest performance improvement was from making the ifnet statistics per-cpu; its performance improvement ratio is even higher than adding multiple NIC TX ring support! The lesson we learned: don't overlook the globally shared statistics, especially ones that are updated at high frequency.

2.6. Rebirth of SO_REUSEPORT

Again in 2013, while we were having headache to reducing the contention on the TCP listen socket accept queue, Google proposed a new SO_REUSEPORT semantics to Linux (Thank Aggelos Economopoulos for bringing it up to me). We were so excited about that it; we leveraged that idea, implemented it within one week, and addressed one drawback mentioned in Google's original proposal (as pointed out by nginx folks). And our patch to nginx to support SO_REUSEPORT was accepted (several years later though). The improvement

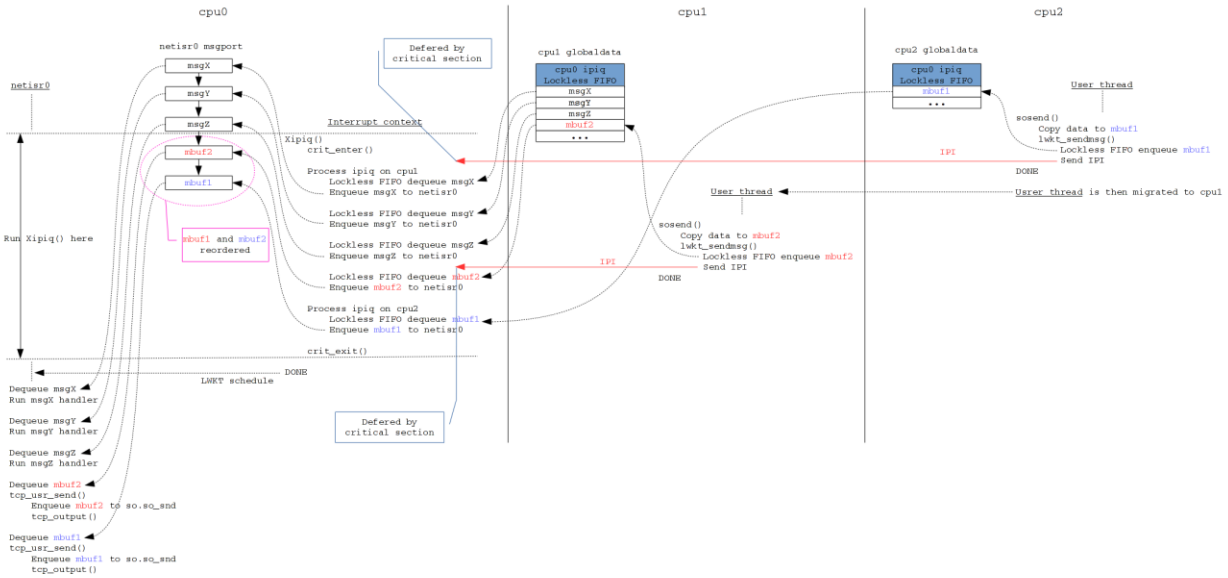


Figure 2.4

is shown Figure 1.12. A good lesson to me at least: open more eyes to what's going on around.

2.7. Acrobatic UDP

In 2014, UDP INPCB tables were made MP-safe. This was a complex process due to the flexibility of UDP socket APIs, we'd discuss it someplace else.

3. The parts of the DragonFlyBSD's network stack that can really enjoy help hands.

- The IPsec stack.
- The IPv6 stack; though I am making progress, the progress is slow.
- Per-cpu pf static rule table replication.
- Per-cpu pf/ipfw state table partition.

4. Reference

[1] Verifying the RSS Hash Calculation https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/verifying-the-rss-hash-calculation

[2] Intel® 82576EB Gigabit Ethernet Controller Datasheet http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82576eg-gbe-datasheet.pdf

[3] An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems https://www.usenix.org/legacy/events/usenix06/tech/full_papers/willmann/willmann_html/index.html