Toward MP-safe Networking in NetBSD

Ryota Ozaki <ozaki-r@iij.ad.jp>
Kengo Nakahara <k-nakahra@iij.ad.jp>

EuroBSDcon 2016
2016-09-25
Contents

- Background and goals
- Approach
- Current status
- MP-safe Layer 3 forwarding
- Performance evaluations
- Future work
Background

- The Multi-core Era
- The network stack of NetBSD couldn’t utilize multi-cores
  - As of 2 years ago
Our Background and Our Goals

- **Internet Initiative Japan Inc. (IIJ)**
  - Using NetBSD in our products since 1999
  - Products: Internet access routers, etc.

- **Our goal**
  - Better performance of our products, especially Layer 2/3 forwarding, tunneling, IPsec VPN, etc.
    → MP-safe networking
Our Targets

- **Targets**
  - 10+ cores systems
  - 1 Gbps Intel NICs and virtualized NICs
    - `wm(4), vmx(4), vioif(4)`
  - Layer 2 and 3
    - IPv4/IPv6, `bridge(4), gif(4), vlan(4), ipsec(4), pppoe(4), bpf(4)`
- **Out of targets**
  - 100 cores systems and above
  - Layer 4 and above
    - and any other network components except for the above
Approach

● MP-safe and then MP-scalable

● Architecture
  ○ Utilize hardware assists
  ○ Utilize lightweight synchronization mechanisms

● Development
  ○ Restructure the code first
  ○ Benchmark often
Approach: Architecture

- **Utilize hardware assists**
  - Distribute packets to CPUs by hardware
    - NIC multi-queue and RSS

- **Utilize software techniques**
  - Lightweight synchronization mechanisms
    - Especially pserialize(9) and psref(9)
  - Existing facilities
    - Fast forwarding and ipflow
Forwarding Utilizing Hardware Assists

Packets are distributed by hardware based on flow (5-tuples).

Packets are processed on a received CPU to the last.
Approach: Development

- **Restructure the code first**
  - Hard to simply apply locks to the existing code
    - E.g., hardware interrupt context for Layer 2, cloning/cloned routes, etc.
  - Need tests to detect regressions

- **ATF tests**
  - Automated and isolated tests of the network stack
    - Thanks to rump kernels
  - 320 test cases in total related to networking
    - 130 test cases have been added since NetBSD 7
    - New categories: IP forwarding, ARP/NDP, IPv6, routes (flags and messages), bridge, gif, tun, tap, pppoe, ifconfig (commands/options)
  - Only 2 minutes to run all the test cases
Approach : Development (cont’d)

● Benchmark often
  ○ Measure speedup
  ○ Measure (single-thread) overheads

● Performance evaluation environments
  ○ Easy to retry tests by anyone else
  ○ Easy to replicate the environment
    ■ Ansible

● ipgen (https://github.com/iij/ipgen)
  ○ netmap-based packet generator running on FreeBSD
  ○ Support RFC 2544 tests
Current Status

NetBSD’s Goal

Our Goal

- Our local changes
- Planned work
- Our contributions

Our Goal: current
Our Contributions in -current:

Device Drivers

- MSI/MSI-X support
  - i386 and amd64
- Interrupt distribution / affinity
  - intrctl(8) changes interrupt destination CPUs
- MP-safe network device drivers
  - wm(4), vioif(4) and vmx(4)
- Hardware multi-queue support of wm(4)
Our Contributions in -current:

Network Components

- MP-safe bridge(4) and gif(4)
- Partial MP-safe Layer 3
  - Interfaces, IP address, etc.
- Important restructuring
  - Separate ARP/NDP caches from the routing table
    - Based on Iitable/Ilentry of FreeBSD
  - Softint-based packet Rx processing
    - Except for net80211 and bpf(4)
- Lots of ATF tests for the network stack

Check our presentation at AsiaBSDCon 2015 (http://www.netbsd.org/gallery/presentations/) for details
Our Local Changes

- Experimental MP-safe Layer 3 forwarding
- Packet Rx processing optimization

CAVEAT: the changes don’t get consensus in the community yet and are not guaranteed to be merged
Planned Work

● Complete MP-ification of Layer 3
  ○ Remaining non MP-safe stuffs: statistic counters, nd_defrouter, nd_prefix, etc.

● MP-ifications
  ○ vlan(4)
  ○ ipsec(4) including opencrypto
  ○ pppoe(4)
MP-safe Layer 3 Forwarding

- **Tools**
  - Hardware assists
  - Software techniques

- **Changes for MP-safe Layer 3 Forwarding**
Tools

- **Hardware assists**
  - NIC hardware multi-queues
  - MSI/MSI-X
  - Interrupt distribution / affinity

- **Software techniques**
  - Lightweight synchronization
  - Fast forwarding and ipflow
Hardware assists

- Recent Ethernet controllers have multiple hardware queues for packet Tx/Rx
- MSI-X allows to have an interrupt on each queue
- We can set an interrupt affinity to a CPU
  - Set by device drivers or intrctl(8)
- We can distribute packets between CPU by RSS (receive-side scaling)
  - Classified by 5-tuples
Interrupt Distributions

Old

Packet distribution by hardware assists
intrctl(8)

- Enable to change the interrupt affinity
- Support only x86 for now

Example: Direct interrupts of queue 1 to CPU 0

intrctl affinity -i 'msix0 vec 1' -c 0
intrctl(8) : Screenshots

```
# intrctl list
interrupt id      CPU0      CPU1
ioapic0 pin 9     0*        0
msix0 vec 0       1696*     0
msix0 vec 1       0         621*
msix0 vec 2       0         0
msix0 vec 3       0         0
msix0 vec 4       2*        0

# intrctl affinity -i 'msix0 vec 1' -c 0
# intrctl list
interrupt id      CPU0      CPU1
ioapic0 pin 9     0*        0
msix0 vec 0       2732*     0
msix0 vec 1       177*      621
msix0 vec 2       0         0
msix0 vec 3       0         0
msix0 vec 4       2*        0
```
Synchronization Techniques

- pserialize(9) and psref(9)
- An example of pserialize(9)
- An example of psref(9)
pserialize(9) and psref(9)

- **Lightweight synchronization primitives**
  - Sort of *deferred processing* in the literature
    - Cf. RCU of Linux and hazard pointers
  - Lightweight read and heavyweight write

- **Reader critical sections**
  - pserialize can be used for those that don’t sleep
  - psref can be used for those that may sleep

- **Writer side**
  - Both provide a mechanism that waits until readers release referencing objects
An Example of pserialize(9) : reader

Iterate addresses of an interface with pserialize

```
s = pserialize_read_enter();
IFADDR_READER_FOREACH(ifa, ifp) {
    /* Do something on ifa, which doesn’t sleep */
}
pserialize_read_exit(s);
```
An Example of psref(9) : reader

Acquiring a reference of a bridge member entry by psref(9) on an interaction of the bridge member list

```c
s = pserialize_read_enter();
BRIDGE_IFLIST_READER_FOREACH(bif, sc) {
    struct psref psref;
    psref_acquire(&psref, &bif->bif_psref, bridge_psref_class);
    pserialize_read_exit(s);
    /* Do something may sleep */
    psref_release(&psref, &bif->bif_psref, bridge_psref_class);
}
pserialize_read_exit(s);
```
An Example of pserialize(9) and psref(9) : writer

Remove a member from the list with holding a lock and wait for readers left

BRIDGE_LOCK (sc);

BRIDGE_IFLIST_WRITER_FOREACH (bif, sc) {
    if (strcmp(bif->bif_ifp->if_xname, name) == 0)
        break;
}

PSLIST_WRITER_REMOVE (bif, bif_next);

pserialize_perform (sc->sc_iflist_psref.bip_psz);

BRIDGE_UNLOCK (sc);

psref_target_destroy (&bif->bif_psref, bridge_psref_class);

/* Free bif safely */
Fast Forwarding and ipflow

- Data structures for the routing table
- Fast forwarding
- ipflow
Data Structures for the routing table

- The routing table (backend)
  - Radix tree
- rtentry
  - Representation of a route
- rtcache
  - Caches to reduce looking up a route from the radix tree
  - Per-CPU rtcache for Layer 3 forwarding
Fast Forwarding

Normal forwarding

Use rtcache or lookup routing table

Routing table

ip_forward

Create an entry

ip_input

ether_input

wm0

wm1

ip_output

ether_output

Fast forwarding

Routing table

ip_forward

Create an entry

ip_input

ether_input

wm0

If ipflow hit, skip Layer 3

wm1

ip_output

ether_output

Ipflow hash list

wm0

wm1
ipflow

- Route caches used by fast forwarding
- ipflow is a hash list
  - Key: struct ip (source and destination addresses)
  - Value: rtcache
- Per-flow rtcache
  - More scalable compared to per-CPU rtcache for Layer 3 forwarding

Calculated from struct ip
Changes for MP-safe Layer 3 Forwarding

- Packet Rx/Tx processing and queuing
- MP-safe interfaces and addresses
- MP-safe routing table
- Scaling up Layer 3 forwarding
- Optimizing packet Rx processing

Our local changes
Packet Rx/Tx Processing and Queuing

- **Rx processing mess**
  - Layer 2 processing including bridge(4), vlan(4), fast forwarding, bpf(4) run in hardware interrupt context
  - Hardware interrupt context is an enemy of MP-ification
    - No sleep is allowed
    - Only spin mutex can be used

- **Softint-based Rx processing**
  - Run Layer 2 (and above) in softint (per-CPU)
    - Except for bpf(4)...
  - Interrupt handlers of device drivers just put packets to a per-CPU queue and schedule softint
Packet Rx/Tx Processing and Queuing

● Tx processing
  ○ From Layer 2 to an interface (device driver)
  ○ If the driver supports hardware multi-queue, the upper layer just passes packets directly
    ■ If not, it enqueues packets into the traditional if_snd queue of the interface

● Tx processing in wm(4)
  ○ wm(4) has multiple queues corresponding to hardware queues to temporarily store packets passed from the upper layer
Softint and Queuing on L3 Forwarding

Old

Driver → L2 → L3 → L2 → Driver

- H/W interrupt
- Input queue for IP
- Output queue per interface

-current

Driver → L2 → L3 → L2 → Driver

- H/W interrupt
- softint

- H/W Rx queues
- Per-cpu queues
- Multiple queues (only wm(4))
- H/W Tx queues

Oops!
MP-safe Interfaces and addresses

● Applied pserialize(9) and psref(9)
● Interfaces (struct ifnet)
  ○ Iterating interfaces with pserialize or psref
  ○ Calling ioctl to an interface with holding psref of it
  ○ rcvif (*ifp) of mbuf is changed to an interface index to avoid dangling pointers
● Addresses (struct ifaddr)
  ○ Three data stores for IPv4
    ■ Global list, global hash list and list per interface
  ○ Get an address from either with pserialize or psref
MP-safe Routing Table

- An experimental design of MP-safe routing table
  - Use rwlock
    - psz and psref are difficult to apply to the routing table because it's not a lockless data structure
  - Limited scalability

- rwlocks
  - A global rwlock for each the backend and rtcaches
  - A rwlock for each rtentry
MP-safe Routing Table (cont’d)

- **If rt caches hit:**
  - No need to hold any writer locks
  - Resulting in good scalability

- **If not:**
  - Performance of Layer 3 forwarding decreases heavily
  - It can easily happen because NetBSD has just one rtcache per CPU in -current
    - Multiple flows on one CPU cause contentions on the rtcache
Scaling up Layer 3 Forwarding

- Reuse ipflow
  - Apply ipflow to Layer 3 (normal) forwarding as well as fast forwarding
- Make ipflow per-CPU
  - Apply both normal forwarding and fast forwarding

Local changes
Scaling up Layer 3 Forwarding (cont’d)

Before:
- Use rtcache or lookup routing table
- ip_forward
  - ip_input
    - ether_input
  - ip_output
    - ether_output
- Routing table
- wm0
- wm1

After:
- Use per-CPU ipflow at first
- ip_forward
  - ip_input
    - ether_input
  - ip_output
    - ether_output
- Ipflow hash list
- Routing table
- wm0
- wm1
Poll Mode

● An optimization technique of Rx processing
  ○ Inspired by NAPI of Linux and the like
  ○ Also one of DoS/livelock mitigation

● Overview
  ○ Disable interrupts during Rx processing
  ○ No queuing

● Support only for wm(4) for now
Poll Mode (cont’d)

[Original]

HW Interrupt handler Softint

Interrupt Disable interrupt

Rx queue Schedule softint

Pass to upper layers

[Poll Mode]

HW Interrupt handler Softint

Interrupt Disable interrupt

Schedule softint

Enable interrupt

Pass to upper layers
Softint and Queuing with Poll Mode

- current

Driver | L2 | L3 | L2 | Driver

- H/W interrupt

- softint

Ours

- H/W Rx queues

- Multiple queues (only wm(4))

- H/W Tx queues
Performance Evaluations : Settings

● Hardware
  ○ DUT (device under test): Supermicro A1SRi-2758F
    ■ 8 core Atom C2758 SoC (2.4 GHz)
    ■ 4 port I354 Ethernet adapter (each port has 8 TX/RX queues)
  ○ Packet generator box: BPV4 (our product)
    ■ 4 core Atom C2558 SoC (2.4 GHz)
    ■ 4 port I354 Ethernet adapter (each port has 8 TX/RX queues)

● DUT kernel
  ○ Based on NetBSD-current at 2016-08-24
    with our local changes

CAVEAT: the changes are incomplete and resulting performance would degrade by further developments
Performance Evaluations: Settings

- **Targets**
  - Layer 3 forwarding
  - 1, 2, 4, 5 and 8 cores

- **Tests**
  - RFC 2544 throughput by ipgen
  - UDP/IPv4 packets
  - Unidirectional

- **Note that packet distributions**
  - We adjust IP addresses to distribute packets almost equally between CPUs
Setups for L3 forwarding Evaluation

L3 forwarding

DUT

forwarding

wm0

172.16.0.1/24

252 flows

igb0

172.16.0.2/24

wm1

172.16.1.1/24

igb1

172.16.1.2/24

ipgen
Throughput vs. # of cores: Normal forwarding

Without per-CPU ipflow, throughputs are around 50-60 Mbps
Throughput vs. # of cores: Fast forwarding

Throughput [Mbps]

number of cores

- 30%
- 98%
- 100%
Summary of Experimental Results

- Frames per second per # of cores
  - At 64 bytes

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>5 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wire rate</td>
<td>1,488,095</td>
<td>1,488,095</td>
<td>1,488,095</td>
<td>1,488,095</td>
<td>1,488,095</td>
</tr>
<tr>
<td>Fast forwarding</td>
<td>455,727</td>
<td>688,241</td>
<td>1,190,617</td>
<td>1,460,193</td>
<td>1,488,095</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>46%</td>
<td>80%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>Normal forwarding</td>
<td>224,375</td>
<td>372,022</td>
<td>674,290</td>
<td>762,646</td>
<td>1,078,865</td>
</tr>
<tr>
<td></td>
<td>15%</td>
<td>24%</td>
<td>45%</td>
<td>51%</td>
<td>72%</td>
</tr>
</tbody>
</table>
Future Work

● Evaluate with 10 Gbps NICs
● Explore alternatives of the routing table backend
● Poll mode with either of softint and LWP
  ○ To prevent userland starvation
Backup
Expected Questions

● Q: Why don’t you measure with 10 GbE?
  ○ A: Our 10 GbE aren’t MP-safe yet
  ○ A: Our immediate target is 1 GbE

● Q: Why don’t you use netmap/DPDK if performance really matters?
  ○ A: Difficult to cooperate with existing tools including ours for our products

● How do you test MP-safe?
  ○ A: Run ATF with LOCKDEBUG
  ○ Do ioctl repeatedly while applying fluctuating traffic
    ■ If there are bugs, the kernel panic for about 10 minutes
    ■ So, when the kernel run completely a few days, it probably ok
Expected Questions

- Q: Why do you use rwlock for the routing table despite NetBSD has pserialize/psref?
  - A: We cannot simply apply serialize/psref to the routing table
    - because the radix tree isn’t a lockless data structure
    - because a route can be deleted in softint but pserialize_perform and psref_target_destroy cannot be used in softint
    - A big restructuring is required
  - A: We don’t have a good alternative to the radix tree yet
  - A: We (IIJ) want to make Layer 3 MP-safe right away
    - To MP-ify other components like ipsec(4)
Q: What about the overhead of ipflow?

- A: Not trivial (see the next slide)
- A: Hash table size is 64 while there are 31 flows
The Case of gif(4)

- What’s gif?
  - A generic tunneling pseudo device

- Why gif?
  - It’s a good first step prior to other complex tunneling facilities
    - gif(4) is a very basic tunneling device
  - It uses a common IP tunneling utility, ip_encap
    - ip_encap is used by ipsec(4)
gif(4) and ip_encap

Reader processing
1. Iterate on the encaps_table
   ○ Find the highest priority struct encap
2. Get a gif(4) softc
   ○ On receiving a packet
   ○ Not iterate gif_softc_list in fast path

The objects can be freed at anytime in packet processing using ip_encap

encaptab_list (pslist(9))

e.g. stf_softc_list

The objects can be freed at anytime in gif(4) packet processing
gif(4) to gif(4) Forwarding

Old

Driver → L2 → outer L3 → inner L3 → outer L3 → L2 → Driver

Input queue for gif → Input queue for IP → Output queue for gif → Output queue per interface

H/W interrupt → softint A → softint B → softint C

-per-CPU queues → Output queue per interface

-current + our local changes
Setups for gif(4) Evaluation

- Adjustment
  - 1 gif(4) per 1 CPU
  - 1 flow per 1 gif(4)
Throughput vs. # of cores: gif(4)
Overhead of ipflow

1 flow per CPU w/o ipflow (i.e., per-CPU rtcache)

85%

72%
### Summary of Experimental Results with 5 cores

- **Frames per second per # of cores**
  - at 64 bytes

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>2 cores</th>
<th>4 cores</th>
<th>5 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wire rate</strong></td>
<td>1,488,095</td>
<td>1,488,095</td>
<td>1,488,095</td>
<td>1,488,095</td>
<td>1,488,095</td>
</tr>
<tr>
<td><strong>bridge(4)</strong></td>
<td>381,322</td>
<td>613,837</td>
<td>1,191,634</td>
<td>1,488,095</td>
<td>1,488,095</td>
</tr>
<tr>
<td></td>
<td>25%</td>
<td>41%</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Fast forwarding</strong></td>
<td>455,727</td>
<td>688,241</td>
<td>1,190,617</td>
<td>1,460,193</td>
<td>1,488,095</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>46%</td>
<td>80%</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Normal forwarding</strong></td>
<td>224,375</td>
<td>372,022</td>
<td>674,290</td>
<td>762,646</td>
<td>1,078,865</td>
</tr>
<tr>
<td></td>
<td>15%</td>
<td>24%</td>
<td>45%</td>
<td>51%</td>
<td>72%</td>
</tr>
<tr>
<td><strong>gif(4)</strong></td>
<td>138,492</td>
<td>242,251</td>
<td>425,502</td>
<td></td>
<td>772,528</td>
</tr>
<tr>
<td></td>
<td>9%</td>
<td>16%</td>
<td>28%</td>
<td></td>
<td>51%</td>
</tr>
</tbody>
</table>
Dangling Pointers on struct ifnet

- Many structures have a pointer of an ifnet (ifp)
- In the MP-safe world, *ifp can be freed
- Solution
  - Replace a pointer with an interface index and get ifp from the interface database
    - With pserialize(9) or psref(9)
    - It adds some overhead
- Structures we needed the change
  - mbuf (rcvif)
    - Need to gain a reference of ifp on ip_input
  - ip_moptions, ip6_moptions
Dangling Pointers on struct ifnet (cont’d)

● A case when an interface pointer is always valid
  ○ If an object having ifp always lives shorter than the ifnet object, we can assume that *ifp is always valid
  ○ IOW, if an referencing object is destroyed on ifnet destruction, *ifp is always valid
  ○ Examples of this case
    ■ rtentry->rt_ifp
    ■ rtentry->rt_ifa (struct ifaddr)