Modernizing NetBSD Networking Facilities and Interrupt Handling

Ryota Ozaki
ozaki-r@iij.ad.jp
Internet Initiative Japan Inc.

Kengo Nakahara
k-nakahara@iij.ad.jp
Internet Initiative Japan Inc.

Abstract

Networking facilities of NetBSD still have a lot of room for improvement, especially the scalability of its network processing is one of big issues that we have to address. We have been working on making the network stack and network device drivers MP-safe to run NetBSD networking facilities run in parallel. Supporting MSI/MSI-X and multi-queue (with interrupt affinity) features of network devices in NetBSD device drivers is also an important task towards the goal.

This paper describes the current status of NetBSD networking facilities and our recent work to improve them. The paper also reports the results of preliminary evaluations on our implementation and shows that our implementation scales up well on a multi-core system.

1 Introduction

Nowadays the direction of computer processor evolution has moved to multiple processor cores in one chip from high frequency clock rate of one processor core. Recent commodity x86 processors on server machines gain 16 cores and even more typical smart phones have two or four cores. Utilizing multi-cores is one of important tasks for operating systems (OS).

NetBSD supports multi-core processors on many platforms such as x86, ARM and PowerPC. User processes on NetBSD can run in parallel on multiple CPUs and many kernel components such filesystems and a lot of device drivers can also utilize multiple CPUs. However, most of the networking facilities of NetBSD do not work in parallel.

We have been working on making the network stack, mainly Layer 2, and device drivers MP-safe to run NetBSD networking facilities run in parallel. Furthermore, we have been implementing better interrupt handling facilities including MSI/MSI-X support, interrupt affinity and multi-queue support.

This paper is organized as follows. Section 2 describes the current status of network processing in NetBSD. Section 3 describes how to make networking facilities MP-safe; we pick `vioif` and `bridge` to explain that in detail. Section 4 and Section 5 describe how we support hardware mentioned above in NetBSD. Section 6 reports the results of preliminary performance evaluations on our implementation. At last, Section 7 concludes the paper.

2 Current Status of Network Processing

In this section, we describe the current status of network processing of NetBSD, including how the network facilities of NetBSD work on multi-core systems, how traditional mutual exclusion facilities work and what are problems of the network processing.

2.1 Basic network processing

Here we describe about only Layer 2 and below of the network stack.

When a user process tries to send a packet to a network, the socket layer, the protocol stack and a TX procedure of a network device driver are executed in a context of the user process. Before calling the TX procedure of the network device driver, sending packets are pushed to the sender queue of the driver called `if_snd`. The TX procedure dequeues a packet from the queue and sends it one by one, or dequeues and sends multiple packets at once. In either case, the driver may stop sending in some cases, for example, there is no TX descriptor to send more packets. The driver holds pending packets and later sends them once it is possible again.

RX procedures are a bit complex. An interrupt handler of a network device driver handles network packets in hardware interrupt context. The handler passes the packets up to the network protocol stack. In NetBSD, the handler does not directly handle Layer 3 and above protocols. Instead, it calls a software interrupt (hereinafter

called softint) handler to perform such protocol processing; it pushes the packets to the queue of a softint for each protocol such as IPv4, IPv6 and ARP[1].

Each softint has a priority that is lower than hardware interrupt handler (we will describe priorities of softints in Section 2.2.2). A softint has its own context and can sleep or block unlike hardware interrupt context. Switching to softint context from hardware interrupt context is faster than switching to a user process (we use LWP to refer to it hereinafter) context.

Network processing of Layer 2 and below is always executed with holding the big kernel lock of NetBSD, called KERNEL_LOCK described in the next section. The output routine of Layer 2 (`ether_output`), hardware interrupt handlers and softint handlers such as `bridge_forward` are executed with holding KERNEL_LOCK. `softnet_lock` which is a mutex (described in Section 3.1.1) is also held in softint handlers. Therefore, most network processing of Layer 2 and below is serialized.

2.2 Traditional mutual exclusion facilities

2.2.1 KERNEL_LOCK

KERNEL_LOCK is a kind of big (giant) kernel locks in the literature. By holding it, a LWP can prevent other LWPs (including softint) that try to hold the lock from running on other CPUs.

The lock can be used in any contexts including hardware interrupt contexts. A LWP can sleep with holding the lock; the lock is released temporarily when the LWP is going to sleep. Constraints of the lock are relatively lax and we can easily use it widely in the kernel. However, it is a biggest obstruction of parallel network processing and we have to remove them all from the kernel to utilize multi-core capabilities.

2.2.2 IPL and SPL

NetBSD defines interrupt priority levels (IPL): `IPL_HIGH` (a.k.a., `IPL_SERIAL`), `IPL_SCHED` and `IPL_VM` (a.k.a., `IPL_NET`) for hardware interrupts in descending order, `IPL_SOFTSERIAL`, `IPL_SOFTNET`, `IPL_SOFTBIO` and `IPL_SOFTCLOCK` for software interrupts in descending order, and `IPL_NONE` for the lowest level of IPL.

`spl*` APIs, such as `splhigh` and `splnet`, allow changing the system interrupt priority level (SPL). The SPL is an IPL under which the system is running, and the system cannot be interrupted by a lower interrupt than the SPL. For example, it is used to protect a data shared between the network stack and an interrupt hander of a network device by raising the SPL to `IPL_NET` that is an IPL for hardware interrupts of network devices. `splnet` is used to change the SPL to `IPL_NET`. `splx` restores the SPL with a passed SPL that is normally a return value of a `spl` function. Note that we cannot lower the SPL; if we try to do so, the operation is just ignored.

The limitation of the mechanism is that it prevents only interrupts that happen on the current CPU from executing. Thus, it does not work at all as a mutual exclusion between multiple CPUs. For MP-safe networking, we should replace most of `splnet` with mutexes.

2.3 How each component works

2.3.1 Network device drivers

NetBSD does not have per-cpu interrupt affinity and distribution facility. Every interrupts go to CPU#0, which means that every interrupt handlers and subsequent network processing including softint for each protocol processing described in Section 2.1 run on CPU#0. Thus, the current implementations of the device drivers do not provide any mutual exclusion between interrupt handlers. On the other hand, packet transmissions can be executed on any CPUs. We always hold KERNEL_LOCK before calling a packet transmission routine of a device driver so that only one instance of the routine is executed in the system at the same time.

Shared resources between LWPs and hardware interrupt handlers are simply protected by `splnet`. Mutual exclusion between LWPs and softint are currently done by KERNEL_LOCK and/or `softnet_lock`.

2.3.2 Layer 2 forwarding

Here we describe about `bridge` networking facility that connects two or more network interfaces at Layer 2 and forwards Ethernet frames between them. Figure 1 shows a basic forwarding flow of the Layer 2 forwarding.

When an incoming frame comes to a network interface attached to a bridge, it passes it up to the bridge. `bridge_input` is first called to handle the frame. If the frame directs to the machine, it receive the frame via `ether_input`. If not, the bridge forwards the frame via `bridge_forward`.

`bridge` is a half-baked component in NetBSD; it can run in any of hardware interrupt, softint and normal LWP contexts. Most of forwarding routines run in softint context (`bridge_forward`) while the packet receive processing (`bridge_input`) runs in hardware interrupt context and we have to use a spin mutex to protect shared resources between the two contexts. Layer 3 protocol processing runs in a softint context or a normal LWP con-

[1] Note that there is an exception; the *fastforward* mechanism directly calls a TX routine of a network device driver from a receive interrupt handler if possible.

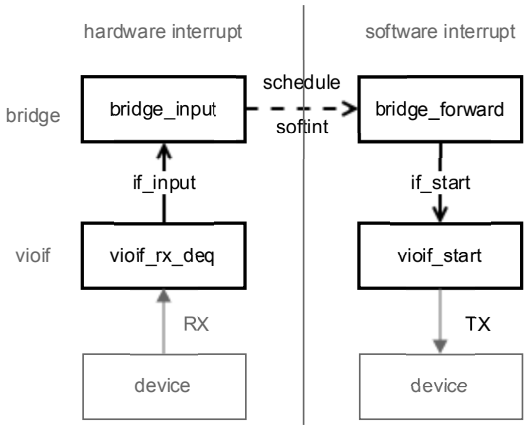

Figure 1: Layer 2 forwarding

text as mentioned in Section 2.1 so that we can use any mutual exclusion facilities.

Currently any procedures in bridge are serialized by the KERNEL_LOCK and additionally `softnet_lock` is used for `bridge_forward`. Shared resources between the hardware interrupt context and the other contexts are protected by `splnet`.

3 MP-safe Networking

In this section, we first describe mutual exclusion facilities that are usable to make networking facilities of NetBSD MP-safe. Next, we describe how to make networking facilities of NetBSD MP-safe by giving examples.

3.1 Mutual exclusion facilities for MP-safe

Table 1 summarizes all mutual exclusion facilities of NetBSD and their characteristics.

3.1.1 mutex

NetBSD's mutexes are almost identical to Solaris's ones [3]. There are two types of mutexes: spin mutex and adaptive mutex. A spin mutex is used as mutual exclusion between LWPs (including softint) and hardware interrupt handlers running on different CPUs. It busy-waits until it succeeds to take the lock. A LWP cannot sleep with holding a spin mutex.

An adaptive mutex is used as mutual exclusion between LWPs (including softint). It first tries busy-waiting if a mutex holder of the mutex is running on another CPU. If it fails to take the lock after some periods or in the first place the mutex holder is (was) running on the same CPU, it sleeps until the mutex holder releases the mutex. Thus, an adaptive mutex cannot be used in hardware interrupt handlers. A LWP is able to sleep with holding an adaptive mutex (at least as of this writing), however, that is not recommended. Sleep-able operations, such as memory allocations, should be done out of critical sections, otherwise performance may be degraded on multi-core systems.

Note that NetBSD does not support reentrant mutexes unlike FreeBSD, so we have to carefully use mutexes to avoid deadlock.

3.1.2 rwlock

`rwlock` is a kind of readers-writer lock in the literature. A critical section protected by an rwlock allows multiple readers to access it concurrently while a single writer has exclusive access to the critical section. Readers and writers may block on an rwlock, thus, rwlocks cannot be in hardware interrupt context. Like adaptive mutexes, readers and writers spin on an rwlock when there is no waiter and an owner of the lock is running on another CPU.

An rwlock is suited for cases that there are many readers and a few writers, in other word, there are frequent reads and infrequent writes to a protected data. However, if a data has a higher ratio of read(er)s to write(r)s (e.g., a case of `ifnet_list` that is modified only when an interface is added or deleted), `pserialize` that we will mention in the next section is more appropriate to use for that case.

3.1.3 pserialize

Making MP-safe by serializing concurrent processes works but it is known that it does not scale [4]. Using a lockless data structure and implementing fast paths without any locks can work and scale well. However, there is an issue to solve; how we safely destroy an object that users may be holding. One solution for the issue is to have a reference counting mechanism, although it also prevents scaling up because a counter can be a point of cache contentions.

`pserialize` is a readers-writer synchronization designed for lock-less read operations. The basic idea of pserialize is similar to `RCU` [5, 4] that is a passive synchronization mechanism of Linux. It allows readers to access shared resources with low overhead while a writer has to wait slightly long periods until it can guarantee that readers have released holding shared resources. To surely guarantee the condition, pserialize enforces that a reader inside a critical section must not sleep and block. With the assumption, we can know that readers have released holding objects once a context switch on all CPU

	Usable on MP (1)	HW interrupt (2)	Sleep/block-able	Reentrant	Callable facilities (3)
KERNEL_LOCK	yes	yes	yes	yes	all
spl*	no	yes	yes	yes	all (4)
mutex (spin)	yes	yes	no	no	mutex (spin)
mutex (adaptive)	yes	no	yes (5)	no	all
rwlock	yes	no	yes (5)	no	all
pserialize (reader)	yes	no	no	no (6)	mutex (spin)

(1) Does it affect to activities on other CPUs? (2) Can it be used in hardware interrupt context? (3) Facilities that can be used in its critical section (4) It should not lower SPL (5) Possible but not recommended (6) Possible but not expected

Table 1: Mutual exclusion facilities

cores happens twice. There is another constraint that a shared data collection, for example a linked list, needs to be lock-free and allow a writer to remove a data from the collection without interfering readers.

On a writer side, we use `pserialize_perform` to wait for readers going out from a critical section. A typical usage of `pserialize_perform` is like this:

```
mutex_enter(&writer_lock);
/* remove a object from the collection */
pserialize_perform(psz);
/*
 * Here we can guarantee that nobody is
 * touching the object
 */
mutex_exit(&writer_lock);
/* So we can free the object safely */
```

The lock (adaptive mutex) is required to protect the shared data collection. After `pserialize_perform`, pserialize guarantees a removed object is not being accessed by anyone.

3.2 Case studies

In this section, we describe how to make network device drivers and network facilities MP-safe. We pick `if_vioif` network device driver and `bridge` as examples for our explanation in this paper.

3.2.1 Making `vioif` MP-safe

The basic strategy is quite simple: get rid of KERNEN_LOCK and introduce fine-grain locking [2].

We introduce two spin mutexes, one for TX and the other for RX. The TX mutex is always held when a TX procedure is sending packets (`vioif_start`) and the TX interrupt handler is cleaning up used `mbufs` (`vioif_tx_vq_done`). The RX mutex is held when the RX interrupt is handling packets (`vioif_rx_deq`) and filling `mbufs` to RX descriptors (`vioif_populate_rx_mbufs`). The RX mutex is released when `vioif_rx_deq` calls a upper layer routine `if_input` (normally it is `ether_input`), and re-taken after returning from the upper routine.

With the preparations, the device driver can work without global locks. We enable the interrupt handlers to run without KERNEL_LOCK by setting the `PCI_INTR_MPSAFE` to `pci_intr_establish`. Additionally, we use the interrupt distribution facility that we have implemented and will describe later. By doing so, we can run TX procedures and RX procedures in parallel, and also run multiple RX interrupt handlers on different CPUs. However, TX procedures on multiple CPUs are still serialized due to KERNEL_LOCK as we mentioned before. To this end, we have to make upper layer components MP-safe. We have done this only for the bridge forwarding facility.

Shutdown processes are great concerns when making a component MP-safe. We have to guarantee that, for example, a freed packet is not accessed by anybody and a stopped device driver does not process any more packets. We introduce a flag to `if_vioif` that indicates the driver is stopping. The shutdown routine (`vioif_stop`) has to set the flag before starting the shutdown sequence. The TX and RX procedures have to check the flag to know whether the driver can proceed its procedure. Especially we have to check it after coming back from a upper layer routine in `vioif_rx_deq`.

3.2.2 Making `bridge` MP-safe

A bridge has two important resources: a bridge member list and a MAC address table. A bridge member list is a linked list and each item points to an interface object. The list is not changed in fast paths, i.e., during packet processing, so we can use a pserialize for the list[2]. We also need a reference counting mechanism for each

[2]Note that `pserialize` does not intend to be used in hardware interrupt context. However, in practice, we can use it for readers in hardware interrupt context using `splhigh`. This is a temporal solution until we make the entire Layer 2 network stack run in softint context.

bridge member to work the pserialize correctly because bridge members can be held by users over sleep-able operations, which violates a constraint of `pserialize` that is mentioned in Section 3.1.3. On a reader side, we increment a reference count of a bridge member during `pserilize_read` and release the reference after using the bridge member. On a writer side, we first remove a bridge member from the member list and then use `pserialize_perform` to guarantee nobody is trying to hold the bridge member anymore. Then, we synchronize on the reference count of the bridge member to wait for a possible user to release the bridge member. After that, we can ensure that nobody is holding the bridge member and safely free the bridge member.

A MAC address table is a hash table with linked lists. Linked lists of a table are not changed in fast paths so that we can use a pserialize for linked lists. Unlike a bridge member list, an entry of a table will not be held over sleep-able operations, thus, we do not need a reference counting mechanism in addition to the pserialize. Nevertheless, we have another issue to tackle; we need to release a bunch of entries at once, for example, a system administrator wants to flush all MAC address entries from a bridge. If we try to get rid of each entry one by one through a pserialize, it would take longer time than we desire because the pserialize is extremely slow. Therefore, we remove entries at one pserialize. To this end, we need to allocate a temporal buffer to accommodate releasing entries *before* taking a mutex for the MAC address table because allocating memory with holding a mutex is not recommended nowadays in NetBSD. The fact enforces that we have to retry allocating a buffer when we know the allocated memory is short since the releasing entries have been increased between the allocation and the mutex acquisition.

This is a pseudo code around `pserialize_perform` in `if_bridge`.

```
retry:
  allocated = sc->entry_count;
  list = kmem_alloc(sizeof_entry * allocated, ...);

  /* Here the # of entries can be changed */

  mutex_enter();
  if (sc->entry_count > allocated) {
    /* The entries increased, we need more memory */
    mutex_exit();
    kmem_free(list, ...);
    goto retry;
  }

  /* Remove entries from the list */
  i = 0;
  LIST_FOREACH_SAFE(entry, sc->list, ...) {
    bridge_rtnode_remove(sc, entry);
    list[i++] = entry;
  }

  /* Waiting for users having gone */
  if (i > 0)
    pserialize_perform();
  mutex_exit();

  /* Destroy entries here */
  while (--i >= 0)
    bridge_rtnode_destroy(brt_list[i]);

  kmem_free(list, ...);
```

The code gets more complex than ever, however, lock contentions are greatly eliminated by using a pserialize instead of mutexes.

3.3 An alternative architecture for network device drivers

NetBSD has a new architecture of network device drivers designed by Matt Thomas, a core NetBSD developer. The concept of the architecture is to run most packet processing in softint context. A softint handler processes *both* TX and RX of a device driver; both a upper protocol layer and an hardware interrupt handler just queues sending/incoming packets to the softint and calls it to perform TX/RX.

The benefits of the architecture are:

- It reduces execution periods of hardware interrupt handlers and prevents interrupt overload causing livelock [6].
- It allows whole the protocol stack run in softint context, which especially simplifies tasks on making Layer 2 protocol processing MP-safe.
- It simplifies locking for MP-safe because only softint processes TX and RX in the architecture.
- It would easily migrate to the polling model of packet processing such as Linux NAPI.

A drawback of the architecture would be an overhead of switching to a softint on any packet processing. We suppose that switching to a softint is enough fast [3]. Actually the NAPI framework of Linux, which uses `softirq` for receiving packets, has proved that it can archive good throughput with switching to softirq on every packet receptions if it receives incoming packets in one switch.

We consider migrating all network device drivers to the architecture by providing a framework for it.

4 Interrupt Process Scaling

In order to achieve high performance parallel network processing, we need to run interrupt handlers in parallel on multiple CPUs. Furthermore, to avoid contentions

[3] In NetBSD, some CPU architecture implements *fast* softint and others does not implement it. We assume a *fast* version here.

against packet buffers between interrupt handlers, we need to support the multi-queue facility of modern network devices, which requires MSI/MSI-X.

In this section, we describe the technologies and what we have done to support them in NetBSD.

4.1 MSI/MSI-X support

Message Signaled Interrupt (MSI/MSI-X) support is optional for PCI 3.0 devices and required for PCI Express devices [7]. They use memory read/write instead of interrupt signal line to notify interrupts to CPUs, so they are not restricted by hardware as much as legacy interrupts. Some devices (and hypervisors) do not support legacy INTx interrupts and support only MSI/MSI-X. Therefore, MSI support is required by OSes that want to use such devices.

Both MSI and MSI-X can use multiple interrupts for one device. MSI-X allows issuing multiple interrupts from one device at the same time while MSI can send only one interrupt at the same time. To distribute interrupts from one device at the same time to multiple CPUs, it is required to support not only MSI but also MSI-X.

Currently, NetBSD supports only MSI on only PowerPC while FreeBSD and DragonFlyBSD support both MSI and MSI-X, and OpenBSD supports only MSI. We design APIs for MSI and MSI-X and implement these drivers for NetBSD/x86. The APIs (shown in Appendix A) are similar to FreeBSD's MSI/MSI-X APIs to port device drivers easily.

4.2 Interrupt distribution

A device can distribute interrupts to multiple CPUs by using MSI-X, however the device does not distribute interrupts automatically. In the case of FreeBSD, MSI-X APIs automatically bind each interrupt to CPU in a round robin manner. Furthermore, FreeBSD has `cpuset(1) -x` option and `intr_setaffinity` kernel API, but they just bind an interrupt thread to a specified CPU and do not use MSI/MSI-X functions.

We implement `intrctl(8)` and `intr_distribute(9)` which let NetBSD support interrupt distribution. The command and kernel API are defined as MI but currently implemented only for amd64 and i386. A device driver that wants to distribute interrupts can do it as the driver author desires by using this API. Furthermore, the system administrators can distribute interrupts as they desire by using `intrctl(8)`.

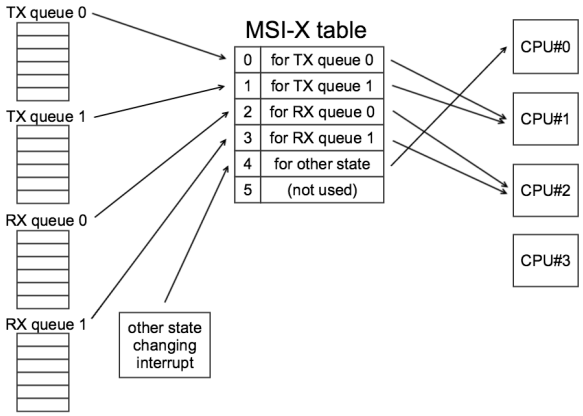

Figure 2: Example of a MSI-X setting

5 Multi-queue

Modern 1 Gigabit or more Ethernet controllers have more than one TX/RX queues, which is called multi-queue. When we want to use multi-queue, we need to setup Ethernet controller registers and a `MSI-X table` that is defined by PCI-SIG specification. Figure 2 is a setting example. In this example, we assign a MSI-X vector for each TX queue, RX queue and other interrupts that include link state change interrupts. A `MSI-X table` is mapped in PCI configuration space and has entries of a MSI-X vector. Each entry consists of PCI-SIG defined fields and system-specified fields. The system-specified fields include, for example, `Message Data`, `Message Upper Address` and upper 30 bits of `Message Address`. In x86 case, the `Destination ID` [1] in the `Message Address` binds the MSI-X vector to a CPU core.

To bind a TX or RX queue to a MSI-X vector, we set an index of the MSI-X vector in the MSI-X table to one of a set of registers, called `Interrupt Vector Allocation Registers`, for the queue on the Ethernet controller. The case of Figure 2, we set as follows:

- index 0 to the register for TX queue 0
- index 1 to the register for TX queue 1
- index 2 to the register for RX queue 0
- index 3 to the register for RX queue 1
- index 4 to the register for other interrupts

In our implementation, these bindings are setup in the initialization of each device driver and cannot be changed during OS running.

For `MSI-X table`, we setup MSI-X table entries for MSI-X vectors, for example, mask bits in the PCI-SIG

defined fields and binding MSI-X vectors to CPU cores in the x86-specified fields. The case of Figure 2, we set as follows:

- cpuid 1 to the entries of index 0 and 2
- cpuid 2 to the entries of index 1 and 3
- cpuid 0 to the entry of index 4

In out implementation, these bindings are setup in MI code and can be changed by intrctl(8) during OS running.

5.1 Scaling by Multi-queue

A system can use multi-queue with the above setting, however it does not scale without appropriate CPU affinity. If all TX queue and RX queue affinity to the same CPU, TX procedures and RX procedures do not scale. As an example, we describe the relationship between RX, TX procedures and CPU affinity for `bridge`.

A RX procedure runs on a CPU that a RX queue interrupt occurs and `bridge_forward` also runs on the same CPU, therefore it scales by assigning each MSI-X vector for a RX queue to a different CPU.

In contrast, a TX procedure runs on a CPU which `bridge_forward` runs, but the TX procedure does not necessarily runs on a CPU which TX queue interrupts occur. Therefore, to scale TX procedures, it is required not to assign each MSI-X vector for a TX queue to a different CPU, but to assign each TX procedure of `bridge_forward` to a different CPU using TX side multi-queue. We will describe this detail in Section 5.4.

We implemented multi-queue support for two Ethernet drivers that are required for our projects. One is `if_vmx` and the other is `if_wm`.

`If_vmx` is the driver of VMXNET3 that is a paravirtualized network device of VMware. We implemented both RX side multi-queue and TX side multi-queue for `if_vmx`, then we tested and measured the performance of it on VMware ESXi.

`If_wm` is the driver of Intel Ethernet controller series including GbE. The supported devices of NetBSD's `if_wm` are equal to the sum of FreeBSD's `if_em` and `if_igb`. We implement RX side and TX side multi-queues for `if_wm`.

5.2 Common part

As described above, to scale up by using multi-queue, the driver is required to support MSI-X. Meanwhile, there are multiple ways to bind each hardware queue and MSI-X vector. In our implementation, we bind one MSI-X vector for each RX queue handler and each TX queue handler. The other way such as FreeBSD's if_igb, a pair of one TX queue and RX queue is bound to one MSI-X vector.

Furthermore, it needs for `if_wm` to support fallback to MSI or INTx, because `if_wm` supports also the old devices that do not support MSI-X. We implement tiny INTx code to keep clean fallback code.

Note that old PCI host bridges do not support MSI/MSI-X. Some PCI Express root complexes should not use MSI or MSI-X because of their errata. So, we implement a framework to disable MSI or MSI-X for such host bridges and devices.

5.3 RX side

We implement I354 RX side multi-queue support to `if_wm`. The implementation consists of two parts, separating RX-related variables from `wm_softc` and setting Ethernet controller registers.

Separating RX-related variables from `wm_softc` helps us to decide the number of using hardware queues dynamically. We implement `struct wm_rxqueue` to manage each RX queue. We cannot determine the suitable number of using hardware queues statically, because the number depends on the number of CPUs, hardware queues and MSI-X vectors, which vary according to hardware.

We implement the code setting Ethernet controller registers. In particular, we extend the code to use registers for multiple queues. We implement setting the registers that decide which incoming frames use a hardware queue. The registers are not used if the driver uses only one queue.

At this point, we have implemented I354 support code only. It may be able to be used for other Ethernet controllers that use the same MSI-X setting registers, but we have not tested yet. At least, 82574's MSI-X setting is different from I354. We should implement register setting codes for such devices.

5.4 TX side

We implement I354 TX side multi-queue support to `if_wm` as well as RX side multi-queue. The TX implementation is similar to the RX implementation except for new `ifnet` interface.

With current TX interface `if_start`, Layer 2 enqueues a `mbuf` to `if_snd` queue bound to the interface, then the driver dequeues the `mbuf` in `if_start` function. In contrast, with new interface that we implement (`if_transmit`), Layer 2 calls `if_transmit` only. `if_transmit` consists of two parts, enqueue and dequeue.

Before describing the enqueue process, we describe the relation between the number of CPUs and the num-

ber of TX queues that the driver is actually using. When the number of the TX queues is more than the number of CPUs, it means that some CPUs are assigned two or more TX queues. Even if one CPU has multiple TX queues, the CPU cannot scale more than one TX queue because the CPU can work only on one queue at a time. When the number of the TX queues is less than the number of CPUs, it means that some CPUs share the same TX queue. In this case, the performance is lower than that all CPUs are assigned a dedicated queue because of lock contention on TX queue operations. Therefore, it is ideal that the number of TX queues is equal to the number of CPUs and each TX queue is assigned to a different CPU.

In the enqueue part, the driver selects `if_snd` binding a TX queue to use on the CPU. On an ideal situation as described above, the driver use a dedicated `if_snd` and TX queue assigned to the CPU. Otherwise, we need some selection logic of TX queues. Our implementation selects a TX queue in a simple round-robin manner. When there are 8 CPUs and 4 TX queues, we set as follows:

- CPU 0 uses TX queue 0
- CPU 1 uses TX queue 1
- CPU 2 uses TX queue 2
- CPU 3 uses TX queue 3
- CPU 4 uses TX queue 0
- CPU 5 uses TX queue 1
- CPU 6 uses TX queue 2
- CPU 7 uses TX queue 3

In the dequeue part, the driver can dequeue a `mbuf` from the `if_snd` binding one of TX queues with holding only the TX queue lock. As a result, the dequeue part can also run concurrently.

In our current implementation, `if_transmit` is used by `bridge` only. Other Layer 2 and 3 components use `if_start` yet.

6 Performance

We evaluate our implementation in terms of throughput with different CPU affinities and frame sizes. This section describes the results of our experiments.

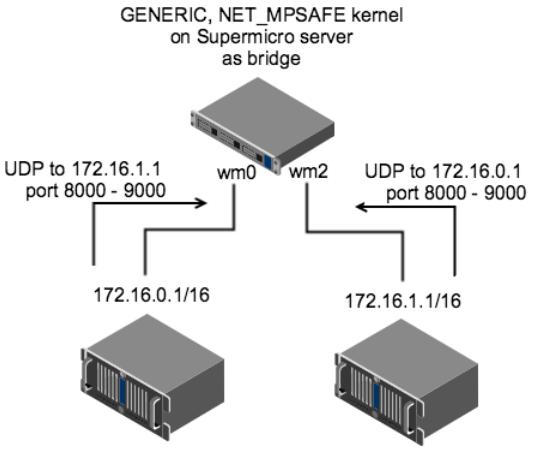

Figure 3: measurement environment

6.1 Experimental setup

We set up Supermicro A1SRi-2758F for our measurements as a DUT (device under test), which has an 8 core Atom C2758 SoC including a 4 port I354 Ethernet controller.

We set up two kernels; one is a kernel built based on NetBSD-current source code at 2015/01/07 with the GENERIC configuration (GENERIC kernel), and the other is a kernel built based on our MP-safe implementation with the GENERIC configuration plus `NET_MPSAFE` enabled (NET_MPSAFE kernel). We use each kernel as a bridge, and send UDP packets over the bridge bi-directionally (Figure 3).

We changed CPU affinities of TX and RX queues for NET_MPSAFE kernel by using `intrctl(8)`. The affinity variations are 1 core, 2 cores, 4 cores and 8 cores; the detailed affinities are shown in Table 2. Note that the tables show only RX queues that affect the results of the measurements.

We setup the driver of I354 (`if_wm`) to select the TX queue whose interrupt is bound to the CPU to be running `bridge_forward` in this measurement. Therefore, `mbuf`s do not move between CPU cores.

In NET_MPSAFE implementation for Atom C2758 and I354, the driver use 4 RX queues and 4 TX queues, that is, use 4 MSI-X vectors for RX queue handlers, 4 MSI-X vectors for TX queue handlers, and 1 MSI-X vector for link state changing handler. It is not cared to affinity of MSI-X vector for link state changing interrupt, because the interrupt rarely occurs.

Table 2: CPU core affinities

The four tables show affinities of RX queues on 1 core, 2 cores, 4 cores and 8 cores from top to bottom respectively.

CPU#0	CPU#1	CPU#2	CPU#3	CPU#4	CPU#5	CPU#6	CPU#7
				wm0 RX0			
				wm0 RX1			
				wm0 RX2			
				wm0 RX3			
				wm2 RX0			
				wm2 RX1			
				wm2 RX2			
				wm2 RX3			

CPU#0	CPU#1	CPU#2	CPU#3	CPU#4	CPU#5	CPU#6	CPU#7
				wm0 RX0			
				wm0 RX1			
				wm0 RX2			
				wm0 RX3			
wm2 RX0							
wm2 RX1							
wm2 RX2							
wm2 RX3							

CPU#0	CPU#1	CPU#2	CPU#3	CPU#4	CPU#5	CPU#6	CPU#7
				wm0 RX0			
					wm0 RX1		
				wm0 RX2			
					wm0 RX3		
wm2 RX0							
	wm2 RX1						
wm2 RX2							
	wm2 RX3						

CPU#0	CPU#1	CPU#2	CPU#3	CPU#4	CPU#5	CPU#6	CPU#7
				wm0 RX0			
					wm0 RX1		
						wm0 RX2	
							wm0 RX3
wm2 RX0							
	wm2 RX1						
		wm2 RX2					
			wm2 RX3				

Table 3: Kilo frames per second

frame size	GENERIC	NET_MPSAFE			
		1 core	2 cores	4 cores	8 cores
74	99.3	329.0	527.7	705.4	1,147.1
128	81.3	327.7	528.2	704.8	1,157.6
192	118.9	333.5	530.4	706.6	1,142.9
256	89.1	331.0	530.8	700.3	905.7
384	115.1	332.1	528.5	618.8	613.9
512	109.3	329.0	469.0	469.9	469.9
768	110.3	316.9	317.2	317.2	317.2
1024	119.9	239.4	239.4	239.4	239.4
1280	104.7	192.3	192.3	192.3	192.3
1408	104.7	175.0	175.0	175.0	175.0
1518	108.2	162.5	162.5	162.5	162.5

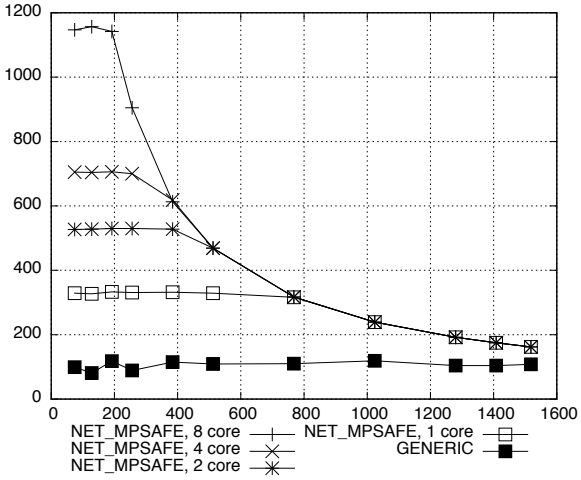

Figure 4: Frame size vs. kilo frames per second

6.2 Experimental results

We use the throughput test defined in RFC 2544 [8] to measure our implementation with different CPU affinities and frame sizes.

Table 3 and Table 4 show the results of our measurements in kilo frames per second (kfps) and Mbps respectively. Figure 4 to Figure 7 show the results in various ways: frame size versus kfps, the number of cores versus kfps, frame size versus Mbps and the number of cores versus Mbps.

The GENERIC kernel uses only one interrupt for one Ethernet port, so it cannot use more than 2 cores. In contrast, NET_MPSAFE kernel can use multiple interrupts using MSI-X. Additionally, NET_MPSAFE kernel can run in parallel because intrctl(8) can set affinity those MSI-X vectors to different CPUs. Furthermore, our MP-safe `bridge` implementation is scalable. In this measurement, the interrupt affinity setting lets `mbuf` not move between CPU cores, this setting also helps MP-safe `bridge` to scale up.

All of the throughputs of the GENERIC kernel do not reach to the wire rate. Even at 1518 byte frame size, the performance is 66.6% of the wire rate. In contrast, all of the throughputs of the NET_MPSAFE kernel reach the wire rate at 768 byte frame size. Hence the NET_MPSAFE lines in Figure 4 are overlapped at 768 byte or lager. As the same reason, 768 byte lines in Figure 5 is flat. Furthermore, 2 core or higher performance reaches the wire rate at 512 byte, 4core or higher does at 384 byte, and 8 core does at 256 byte. Each graphs in Figure 4 are overlap at frame size that the performance reaches the wire rate, moreover each graph in Figure 5 is flat. Figure 6 and Figure 7 more clearly show that all of throughputs of the NET_MPSAFE kernel reach the wire rate.

At frame size 74 byte, the throughput of the GENERIC kernel is only 3.74% of the wire rate. The throughput of NET_MPSAFE kernel at 1 core is 12.4% of the wire rate. Likewise, the throughput on 2 core is 19.8%, 4 core is 26.5%, and 8 core is 43.1%. These results show that our implementation scales up well, however, it does not reach the wire rate on the experimental machine.

Table 3 shows the throughput of NET_MPSAFE kernel (1 core) is 3.2 times higher than the throughput of GENERIC kernel in 74 byte frame size. This is due to

[4]GENERIC kernel can use 1 core only. 2 core, 4 core and 8 core data is copy of 1 core one.

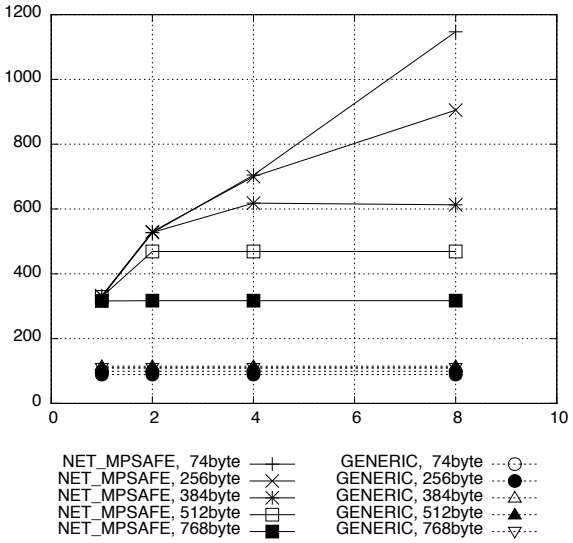

Figure 5: # of cores vs. kilo frames per second [4]

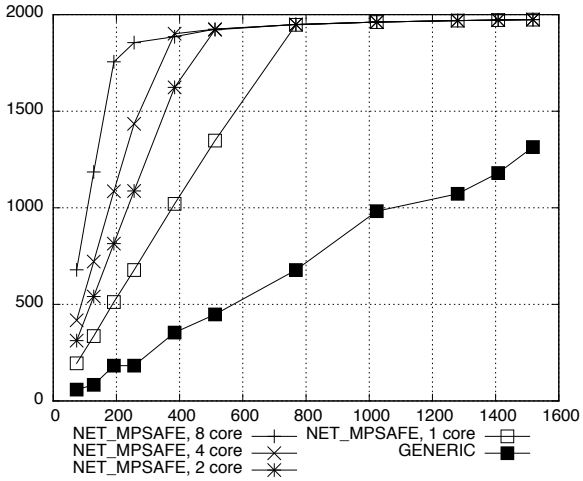

Figure 6: frame size vs. Mbps

Table 4: Mbps per second

frame size	GENERIC	NET_MPSAFE			
		1 core	2 core	4 core	8 core
74	58.81	194.81	312.43	417.60	679.14
128	83.34	335.64	540.88	721.79	1185.47
192	182.72	512.26	814.74	1085.37	1755.60
256	182.63	677.90	1087.13	1434.42	1855.06
384	353.83	1020.48	1623.63	1900.97	1885.95
512	447.93	1347.93	1921.42	1924.79	1924.80
768	678.03	1947.51	1949.22	1949.22	1949.22
1024	982.34	1961.67	1961.67	1961.67	1961.67
1280	1072.66	1969.22	1969.22	1969.21	1969.22
1408	1179.89	1971.97	1971.98	1971.97	1971.97
1518	1314.65	1973.98	1973.98	1973.97	1973.98

not using MSI-X itself but the difference between the implementations of the MSI-X handler and the INTx handler. The MSI-X handler processes one incoming frame on a single interrupt. In contrast, `if_wm` INTx handler processes all available incoming frames on a single interrupt by loop until TX queue is empty, therefore the period of hardware interrupt context can be long. For `bridge`, this too long hardware context disturbs `bridge_forward` softint. Furthermore, the loop processes cause `if_snd` overflow. For these reasons, the performance degraded. By modifying `if_wm` INTx handler to process one frame on a single interrupt, the INTx performance is improved to 256 kfps. Because the current implementation has separate MSI-X vector handlers for TX and RX, NET_MPSAFE is more improved up to about 330 kfps.

7 Conclusion

We have modified several components of NetBSD and developed new components to achieve parallel network processing as described in this paper. Some of them are already in the NetBSD-current tree, but others are not yet. For example, modifications of `bridge` are already in the tree while MSI/MSI-X and multi-queue supports are not (as of this writing).

Our work has not finished yet and there remain a lot of tasks to do, especially tasks of making Layer 3 and above in the network stack MP-safe are important towards our goal. The tasks will take more efforts than so far and will need helps by other developers.

We also have several plans other than the above remaining work to do in the future for the goal. We will implement interrupt distribution facilities for architectures other than x86. As our project use ARM and MIPS, we will undertake those architectures.

References

[1] *Intel 64 and IA-32 Architectures Software Developer Manuals*, `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[2] Greg Lehey, *Improving the FreeBSD SMP Implementation*, USENIX Annual Technical Conference, FREENIX Track, 2001.

[3] Jim Mauro and Richard McDougall, *Solaris Internals: Core Kernel Components Vol.1*, Prentice Hall Professional, 2001.

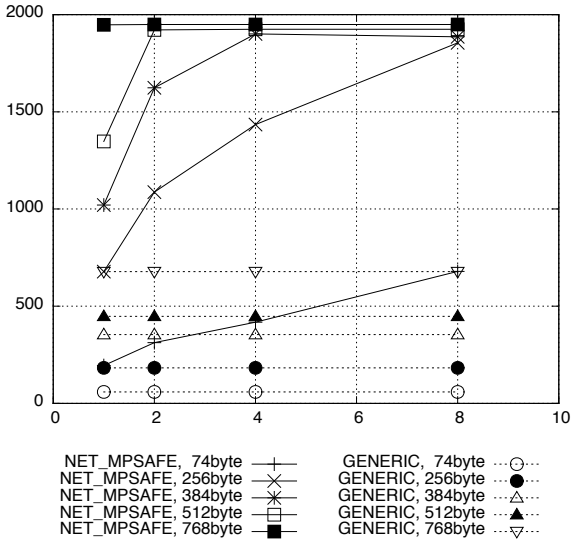

Figure 7: # of cores vs. Mbps [4]

[4] Paul E. McKenney, *RCU vs. Locking Performance on Different CPUs*, linux.conf.au 2004, 2004.

[5] Paul E. McKenney and Jonathan Walpole, *Introducing Technology Into the Linux Kernel: A Case Study*, ACM SIGOPS Operating Systems Review 42.5, 4–17, 2008.

[6] Jeffrey C. Mogul and Kadangode K. Ramakrishnan, *Eliminating Receive Livelock in an Interrupt-Driven Kernel*, ACM Transactions on Computer Systems 15.3: 217–252, 1997.

[7] PCI-SIG, `https://www.pcisig.com/`.

[8] RFC 2544, `https://tools.ietf.org/rfc/rfc2544.txt`, 1999.

Appendix A MSI/MSI-X APIs

```
/* for MSI */

int
pci_msi_count(struct pci_attach_args *pa);

int
pci_msi_alloc(struct pci_attach_args *pa, pci_intr_handle_t **ihps, int *count);

int
pci_msi_alloc_exact(struct pci_attach_args *pa, pci_intr_handle_t **ihps, int count);

void
pci_msi_release(pci_intr_handle_t **pihs, int count);

void *
pci_msi_establish(pci_chipset_tag_t pc, pci_intr_handle_t ih, int level, int (*func)(void *), void *arg);

void *
pci_msi_establish_xname(pci_chipset_tag_t pc, pci_intr_handle_t ih,
    int level, int (*func)(void *), void *arg, const char *xname);

void
pci_msi_disestablish(pci_chipset_tag_t pc, void *cookie);

/* for MSI-X */

struct msix_entry {
    pci_intr_handle_t me_handle; // set by internal implementation of API
    u_int me_index; // MSI-X table index which API user want to bind "me_handle"
};

int
pci_msix_count(struct pci_attach_args *pa);

int
pci_msix_alloc(struct pci_attach_args *pa, struct msix_entry *entries, int *count);

int
pci_msix_alloc_exact(struct pci_attach_args *pa, struct msix_entry *entries, int count);

void
pci_msix_release(pci_intr_handle_t **pihs, int count);

void *
pci_msix_establish(pci_chipset_tag_t pc, pci_intr_handle_t ih,
    int level, int (*func)(void *), void *arg, const char *xname);

void *
pci_msix_establish_xname(pci_chipset_tag_t pc, pci_intr_handle_t ih,
    int level, int (*func)(void *), void *arg, const char *xname);

void
pci_msix_disestablish(pci_chipset_tag_t pc, void *cookie);
```