Scaling in the Linux Networking Stack

Introduction

This document describes a set of complementary techniques in the Linux networking stack to increase parallelism and improve performance for multi-processor systems.

The following technologies are described:

- RSS: Receive Side Scaling
- RPS: Receive Packet Steering
- RFS: Receive Flow Steering
- Accelerated Receive Flow Steering
- XPS: Transmit Packet Steering

RSS: Receive Side Scaling

Contemporary NICs support multiple receive and transmit descriptor queues (multi-queue). On reception, a NIC can send different packets to different queues to distribute processing among CPUs. The NIC distributes packets by applying a filter to each packet that assigns it to one of a small number of logical flows. Packets for each flow are steered to a separate receive queue, which in turn can be processed by separate CPUs. This mechanism is generally known as "Receive-side Scaling" (RSS). The goal of RSS and the other scaling techniques is to increase performance uniformly. Multi-queue distribution can also be used for traffic prioritization, but that is not the focus of these techniques.

The filter used in RSS is typically a hash function over the network and/or transport layer headers-- for example, a 4-tuple hash over IP addresses and TCP ports of a packet. The most common hardware implementation of RSS uses a 128-entry indirection table where each entry stores a queue number. The receive queue for a packet is determined by masking out the low order seven bits of the computed hash for the packet (usually a Toeplitz hash), taking this number as a key into the indirection table and reading the corresponding value.

Some advanced NICs allow steering packets to queues based on programmable filters. For example, webserver bound TCP port 80 packets can be directed to their own receive queue. Such "n-tuple" filters can be configured from ethtool (--config-ntuple).

RSS Configuration

The driver for a multi-queue capable NIC typically provides a kernel module parameter for specifying the number of hardware queues to configure. In the bnx2x driver, for instance, this parameter is called num_queues. A typical RSS configuration would be to have one receive queue for each CPU if the device supports enough queues, or otherwise at least one for each memory domain, where a memory domain is a set of CPUs that share a particular memory level (L1, L2, NUMA node, etc.).

The indirection table of an RSS device, which resolves a queue by masked hash, is usually programmed by the driver at initialization. The default mapping is to distribute the queues evenly in the table, but the indirection table can be retrieved and modified at runtime using ethtool commands (--show-rxfh-indir and --set-rxfh-indir). Modifying the indirection table could be done to give different queues different relative weights.

RSS IRQ Configuration

Each receive queue has a separate IRQ associated with it. The NIC triggers this to notify a CPU when new packets arrive on the given queue. The signaling path for PCIe devices uses message signaled interrupts (MSI-X), that can route each interrupt to a particular CPU. The active mapping of queues to IRQs can be determined from /proc/interrupts. By default, an IRQ may be handled on any CPU. Because a non-negligible part of packet processing takes place in receive interrupt handling, it is advantageous to spread receive interrupts between CPUs. To manually adjust the IRQ affinity of each interrupt see Documentation/core-api/irq/irq-affinity.rst. Some systems will be running irqbalance, a daemon that dynamically optimizes IRQ assignments and as a result may override any manual settings.

Suggested Configuration

RSS should be enabled when latency is a concern or whenever receive interrupt processing forms a bottleneck. Spreading load between CPUs decreases queue length. For low latency networking, the optimal setting is to allocate as many queues as there are CPUs in the system (or the NIC maximum, if lower). The most efficient high-rate configuration is likely the one with the smallest number of receive queues where no receive queue overflows due to a saturated CPU, because in default mode with interrupt coalescing enabled, the aggregate number of interrupts (and thus work) grows with each additional queue.

Per-cpu load can be observed using the mpstat utility, but note that on processors with hyperthreading (HT), each hyperthread is represented as a separate CPU. For interrupt handling, HT has shown no benefit in initial tests, so limit the number of queues to the number of CPU cores in the system.

RPS: Receive Packet Steering

Receive Packet Steering (RPS) is logically a software implementation of RSS. Being in software, it is necessarily called later in the datapath. Whereas RSS selects the queue and hence CPU that will run the hardware interrupt handler, RPS selects the CPU to perform protocol processing above the interrupt handler. This is accomplished by placing the packet on the desired CPU's backlog queue and waking up the CPU for processing. RPS has some advantages over RSS:

1. it can be used with any NIC
2. software filters can easily be added to hash over new protocols
3. it does not increase hardware device interrupt rate (although it does introduce inter-processor interrupts (IPIs))

RPS is called during bottom half of the receive interrupt handler, when a driver sends a packet up the network stack with netif_rx() or netif_receive_skb(). These call the get_rps_cpu() function, which selects the queue that should process a packet.

The first step in determining the target CPU for RPS is to calculate a flow hash over the packet's addresses or ports (2-tuple or 4-tuple hash depending on the protocol). This serves as a consistent hash of the associated flow of the packet. The hash is either provided by hardware or will be computed in the stack. Capable hardware can pass the hash in the receive descriptor for the packet; this would usually be the same hash used for RSS (e.g. computed Toeplitz hash). The hash is saved in skb->hash and can be used elsewhere in the stack as a hash of the packet's flow.

Each receive hardware queue has an associated list of CPUs to which RPS may enqueue packets for processing. For each received packet, an index into the list is computed from the flow hash modulo the size of the list. The indexed CPU is the target for processing the packet, and the packet is queued to the tail of that CPU's backlog queue. At the end of the bottom half routine, IPIs are sent to any CPUs for which packets have been queued to their backlog queue. The IPI wakes backlog processing on the remote CPU, and any queued packets are then processed up the networking stack.

RPS Configuration

RPS requires a kernel compiled with the CONFIG_RPS kconfig symbol (on by default for SMP). Even when compiled in, RPS remains disabled until explicitly configured. The list of CPUs to which RPS may forward traffic can be configured for each receive queue using a sysfs file entry:

```
/sys/class/net/<dev>/queues/rx-<n>/rps_cpus
```

This file implements a bitmap of CPUs. RPS is disabled when it is zero (the default), in which case packets are processed on the interrupting CPU. Documentation/core-api/irq/irq-affinity.rst explains how CPUs are assigned to the bitmap.

Suggested Configuration

For a single queue device, a typical RPS configuration would be to set the rps_cpus to the CPUs in the same memory domain of the interrupting CPU. If NUMA locality is not an issue, this could also be all CPUs in the system. At high interrupt rate, it might be wise to exclude the interrupting CPU from the map since that already performs much work.

For a multi-queue system, if RSS is configured so that a hardware receive queue is mapped to each CPU, then RPS is probably redundant and unnecessary. If there are fewer hardware queues than CPUs, then RPS might be beneficial if the rps_cpus for each queue are the ones that share the same memory domain as the interrupting CPU for that queue.

RPS Flow Limit

RPS scales kernel receive processing across CPUs without introducing reordering. The trade-off to sending all packets from the same flow to the same CPU is CPU load imbalance if flows vary in packet rate. In the extreme case a single flow dominates traffic. Especially on common server workloads with many concurrent connections, such behavior indicates a problem such as a misconfiguration or spoofed source Denial of Service attack.

Flow Limit is an optional RPS feature that prioritizes small flows during CPU contention by dropping packets from large flows slightly ahead of those from small flows. It is active only when an RPS or RFS destination CPU approaches saturation. Once a CPU's input packet queue exceeds half the maximum queue length (as set by sysctl net.core.netdev_max_backlog), the kernel starts a per-flow packet count over the last 256 packets. If a flow exceeds a set ratio (by default, half) of these packets when a new packet arrives, then the new packet is dropped. Packets from other flows are still only dropped once the input packet queue reaches netdev_max_backlog. No packets are dropped when the input packet queue length is below the threshold, so flow limit does not sever connections outright: even large flows maintain connectivity.

Interface

Flow limit is compiled in by default (CONFIG_NET_FLOW_LIMIT), but not turned on. It is implemented for each CPU independently (to avoid lock and cache contention) and toggled per CPU by setting the relevant bit in sysctl net.core.flow_limit_cpu_bitmap. It exposes the same CPU bitmap interface as rps_cpus (see above) when called from procfs:

```
/proc/sys/net/core/flow_limit_cpu_bitmap
```

Per-flow rate is calculated by hashing each packet into a hashtable bucket and incrementing a per-bucket counter. The hash function is the same that selects a CPU in RPS, but as the number of buckets can be much larger than the number of CPUs, flow limit has finer-grained identification of large flows and fewer false positives. The default table has 4096 buckets. This value can be modified through sysctl:

```
net.core.flow_limit_table_len
```

The value is only consulted when a new table is allocated. Modifying it does not update active tables.

Suggested Configuration

Flow limit is useful on systems with many concurrent connections, where a single connection taking up 50% of a CPU indicates a problem. In such environments, enable the feature on all CPUs that handle network rx interrupts (as set in /proc/irq/N/smp_affinity).

The feature depends on the input packet queue length to exceed the flow limit threshold (50%) + the flow history length (256). Setting net.core.netdev_max_backlog to either 1000 or 10000 performed well in experiments.

RFS: Receive Flow Steering

While RPS steers packets solely based on hash, and thus generally provides good load distribution, it does not take into account application locality. This is accomplished by Receive Flow Steering (RFS). The goal of RFS is to increase datacache hitrate by steering kernel processing of packets to the CPU where the application thread consuming the packet is running. RFS relies on the same RPS mechanisms to enqueue packets onto the backlog of another CPU and to wake up that CPU.

In RFS, packets are not forwarded directly by the value of their hash, but the hash is used as index into a flow lookup table. This table maps flows to the CPUs where those flows are being processed. The flow hash (see RPS section above) is used to calculate the index into this table. The CPU recorded in each entry is the one which last processed the flow. If an entry does not hold a valid CPU, then packets mapped to that entry are steered using plain RPS. Multiple table entries may point to the same CPU. Indeed, with many flows and few CPUs, it is very likely that a single application thread handles flows with many different flow hashes.

rps_sock_flow_table is a global flow table that contains the *desired* CPU for flows: the CPU that is currently processing the flow in userspace. Each table value is a CPU index that is updated during calls to recvmsg and sendmsg (specifically, inet_recvmsg(), inet_sendmsg(), inet_sendpage() and tcp_splice_read()).

When the scheduler moves a thread to a new CPU while it has outstanding receive packets on the old CPU, packets may arrive out of order. To avoid this, RFS uses a second flow table to track outstanding packets for each flow: rps_dev_flow_table is a table specific to each hardware receive queue of each device. Each table value stores a CPU index and a counter. The CPU index represents the *current* CPU onto which packets for this flow are enqueued for further kernel processing. Ideally, kernel and userspace processing occur on the same CPU, and hence the CPU index in both tables is identical. This is likely false if the scheduler has recently migrated a userspace thread while the kernel still has packets enqueued for kernel processing on the old CPU.

The counter in rps_dev_flow_table values records the length of the current CPU's backlog when a packet in this flow was last enqueued. Each backlog queue has a head counter that is incremented on dequeue. A tail counter is computed as head counter + queue length. In other words, the counter in rps_dev_flow[i] records the last element in flow i that has been enqueued onto the currently designated CPU for flow i (of course, entry i is actually selected by hash and multiple flows may hash to the same entry i).

And now the trick for avoiding out of order packets: when selecting the CPU for packet processing (from get_rps_cpu()) the rps_sock_flow table and the rps_dev_flow table of the queue that the packet was received on are compared. If the desired CPU for the flow (found in the rps_sock_flow table) matches the current CPU (found in the rps_dev_flow table), the packet is enqueued onto that CPU's backlog. If they differ, the current CPU is updated to match the desired CPU if one of the following is true:

- The current CPU's queue head counter >= the recorded tail counter value in rps_dev_flow[i]
- The current CPU is unset (>= nr_cpu_ids)
- The current CPU is offline

After this check, the packet is sent to the (possibly updated) current CPU. These rules aim to ensure that a flow only moves to a new CPU when there are no packets outstanding on the old CPU, as the outstanding packets could arrive later than those about to be processed on the new CPU.

RFS Configuration

RFS is only available if the kconfig symbol CONFIG_RPS is enabled (on by default for SMP). The functionality remains disabled until explicitly configured. The number of entries in the global flow table is set through:

```
/proc/sys/net/core/rps_sock_flow_entries
```

The number of entries in the per-queue flow table are set through:

```
/sys/class/net/<dev>/queues/rx-<n>/rps_flow_cnt
```

Suggested Configuration

Both of these need to be set before RFS is enabled for a receive queue. Values for both are rounded up to the nearest power of two. The suggested flow count depends on the expected number of active connections at any given time, which may be significantly less than the number of open connections. We have found that a value of 32768 for rps_sock_flow_entries works fairly well on a moderately loaded server.

For a single queue device, the rps_flow_cnt value for the single queue would normally be configured to the same value as rps_sock_flow_entries. For a multi-queue device, the rps_flow_cnt for each queue might be configured as rps_sock_flow_entries / N, where N is the number of queues. So for instance, if rps_sock_flow_entries is set to 32768 and there are 16 configured receive queues, rps_flow_cnt for each queue might be configured as 2048.

Accelerated RFS

Accelerated RFS is to RFS what RSS is to RPS: a hardware-accelerated load balancing mechanism that uses soft state to steer flows based on where the application thread consuming the packets of each flow is running. Accelerated RFS should perform better than RFS since packets are sent directly to a CPU local to the thread consuming the data. The target CPU will either be the same CPU where the application runs, or at least a CPU which is local to the application thread's CPU in the cache hierarchy.

To enable accelerated RFS, the networking stack calls the ndo_rx_flow_steer driver function to communicate the desired hardware queue for packets matching a particular flow. The network stack automatically calls this function every time a flow entry in rps_dev_flow_table is updated. The driver in turn uses a device specific method to program the NIC to steer the packets.

The hardware queue for a flow is derived from the CPU recorded in rps_dev_flow_table. The stack consults a CPU to hardware queue map which is maintained by the NIC driver. This is an auto-generated reverse map of the IRQ affinity table shown by /proc/interrupts. Drivers can use functions in the cpu_rmap ("CPU affinity reverse map") kernel library to populate the map. For each CPU, the corresponding queue in the map is set to be one whose processing CPU is closest in cache locality.

Accelerated RFS Configuration

Accelerated RFS is only available if the kernel is compiled with CONFIG_RFS_ACCEL and support is provided by the NIC device and driver. It also requires that ntuple filtering is enabled via ethtool. The map of CPU to queues is automatically deduced from the IRQ affinities configured for each receive queue by the driver, so no additional configuration should be necessary.

Suggested Configuration

This technique should be enabled whenever one wants to use RFS and the NIC supports hardware acceleration.

XPS: Transmit Packet Steering

Transmit Packet Steering is a mechanism for intelligently selecting which transmit queue to use when transmitting a packet on a multi-queue device. This can be accomplished by recording two kinds of maps, either a mapping of CPU to hardware queue(s) or a mapping of receive queue(s) to hardware transmit queue(s).

1. XPS using CPUs map

The goal of this mapping is usually to assign queues exclusively to a subset of CPUs, where the transmit completions for these queues are processed on a CPU within this set. This choice provides two benefits. First, contention on the device queue lock is significantly reduced since fewer CPUs contend for the same queue (contention can be eliminated completely if each CPU has its own transmit queue). Secondly, cache miss rate on transmit completion is reduced, in particular for data cache lines that hold the sk_buff structures.

2. XPS using receive queues map

This mapping is used to pick transmit queue based on the receive queue(s) map configuration set by the administrator. A set of receive queues can be mapped to a set of transmit queues (many:many), although the common use case is a 1:1 mapping. This will enable sending packets on the same queue associations for transmit and receive. This is useful for busy polling multi-threaded workloads where there are challenges in associating a given CPU to a given application thread. The application threads are not pinned to CPUs and each thread handles packets received on a single queue. The receive queue number is cached in the socket for the connection. In this model, sending the packets on the same transmit queue corresponding to the associated receive queue has benefits in keeping the CPU overhead low. Transmit completion work is locked into the same queue-association that a given application is polling on. This avoids the overhead of triggering an interrupt on another CPU. When the application cleans up the packets during the busy poll, transmit completion may be processed along with it in the same thread context and so result in reduced latency.

XPS is configured per transmit queue by setting a bitmap of CPUs/receive-queues that may use that queue to transmit. The reverse mapping, from CPUs to transmit queues or from receive-queues to transmit queues, is computed and maintained for each network device. When transmitting the first packet in a flow, the function get_xps_queue() is called to select a queue. This function uses the ID of the receive queue for the socket connection for a match in the receive queue-to-transmit queue lookup table. Alternatively, this function can also use the ID of the running CPU as a key into the CPU-to-queue lookup table. If the ID matches a single queue, that is used for transmission. If multiple queues match, one is selected by using the flow hash to compute an index into the set. When selecting the transmit queue based on receive queue(s) map, the transmit device is not validated against the receive device as it requires expensive lookup operation in the datapath.

The queue chosen for transmitting a particular flow is saved in the corresponding socket structure for the flow (e.g. a TCP connection). This transmit queue is used for subsequent packets sent on the flow to prevent out of order (ooo) packets. The choice also amortizes the cost of calling get_xps_queues() over all packets in the flow. To avoid ooo packets, the queue for a flow can subsequently only be changed if skb->ooo_okay is set for a packet in the flow. This flag indicates that there are no outstanding packets in the flow, so the transmit queue can change without the risk of generating out of order packets. The transport layer is responsible for setting ooo_okay appropriately. TCP, for instance, sets the flag when all data for a connection has been acknowledged.

XPS Configuration

XPS is only available if the kconfig symbol CONFIG_XPS is enabled (on by default for SMP). If compiled in, it is driver dependent whether, and how, XPS is configured at device init. The mapping of CPUs/receive-queues to transmit queue can be inspected and configured using sysfs:

For selection based on CPUs map:

```
/sys/class/net/<dev>/queues/tx-<n>/xps_cpus
```

For selection based on receive-queues map:

```
/sys/class/net/<dev>/queues/tx-<n>/xps_rxqs
```

Suggested Configuration

For a network device with a single transmission queue, XPS configuration has no effect, since there is no choice in this case. In a multi-queue system, XPS is preferably configured so that each CPU maps onto one queue. If there are as many queues as there are CPUs in the system, then each queue can also map onto one CPU, resulting in exclusive pairings that experience no contention. If there are fewer queues than CPUs, then the best CPUs to share a given queue are probably those that share the cache with the CPU that processes transmit completions for that queue (transmit interrupts).

For transmit queue selection based on receive queue(s), XPS has to be explicitly configured mapping receive-queue(s) to transmit queue(s). If the user configuration for receive-queue map does not apply, then the transmit queue is selected based on the CPUs map.

Per TX Queue rate limitation

These are rate-limitation mechanisms implemented by HW, where currently a max-rate attribute is supported, by setting a Mbps value to:

```
/sys/class/net/<dev>/queues/tx-<n>/tx_maxrate
```

A value of zero means disabled, and this is the default.

Further Information

RPS and RFS were introduced in kernel 2.6.35. XPS was incorporated into 2.6.38. Original patches were submitted by Tom Herbert (therbert@google.com)

Accelerated RFS was introduced in 2.6.35. Original patches were submitted by Ben Hutchings (bwh@kernel.org)

Authors:

- Tom Herbert (therbert@google.com)
- Willem de Bruijn (willemb@google.com)