Control and forwarding-plane separation of an open-source router

Robert Olsson, Uppsala University and KTH,
Olof Hagsand, KTH,
Jens Laas, Uppsala University,
Bengt Gördén, KTH
Sweden
robert@herjulf.net

1 Abstract

In previous work[1-3] it has been shown how open-source routers on new PC hardware allows for forwarding speeds of 10Gb/s and above. In this work we extend the applicability of the results by showing how the new 10Gb/s interface classification techniques can be used to separate packet forwarding from control plane operation.

It is important to isolate the control-plane from forwarding load, since it makes routing protocol and management operation independent of forwarding load. It also increases the resilience against denial-of-service attacks. In addition, it relates to the forwarding and control element separation proposed by the IETF ForCES work[4], where we use one CPU core as control element and the remaining cores as forwarding elements.

Many new interface cards have chipsets with advanced classification capabilities motivated by advances in virtualization and multicore architectures. We have chosen to study the Intel 82599 10Gb/s controller[5] and the Linux *ixgbe* driver. The 82599 has several mechanisms to control packet classification, including Receiver Side Scaling (RSS)[6], Flow director, and N-tuple filters. Other interface cards on the market use generic TCAMs providing similar functionality.

The approach we used was to implicitly configure the Flow director by outgoing control traffic, so that return flows aimed at the control plane were identified and could be directed to a designed control processor. Flows not destined to the control processor were load balanced among the remaining cores using RSS. We found this to be a simple and straight-forward approach, and we present results that verifies this method. However, we have seen some cases in overload scenarios where packet drops are made in hardware before classification which need to be further analyzed.

During the project we also explored some of the hardware capabilites new buses (PCIe gen2). We discovered with optimal setting that we could transmit (DMA) 92 Gb/s using 1500 byte packets.

2 Experimental platform

2.1 Motherboard and CPU

The experimental platform was a TYAN Thunder S7025 with two Quad-Core Intel Nehalem Processors at 2.4 GHz. There are 8 physical CPUs in total, 16 with hyperthreading, see Figure 1. The eight physical CPUs are arranged in two quad-cores, each having access to a local memory node, thus forming a simple NUMA architecture. Internal buses are Intel QPI.

The motherboard have two separate I/O devices with PCIe gen2 slots providing for parallel performance. There are four x16, one x4 and one x8 PCIe slots in total. The x8 slot in fact only has 4 lanes however.

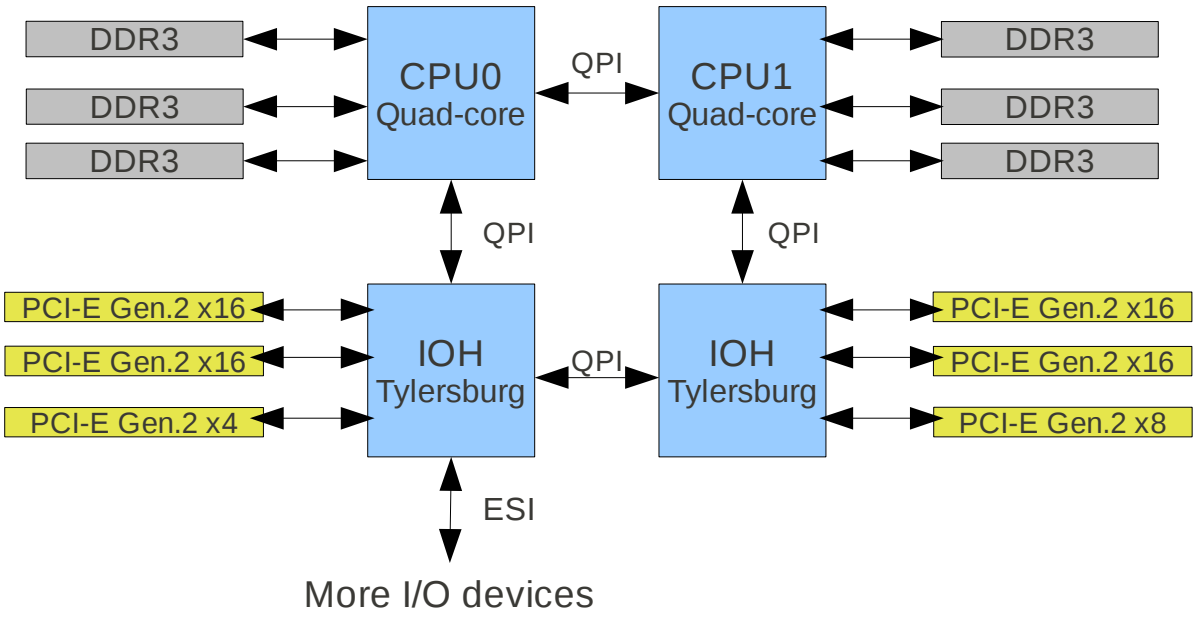

Figure 1: Architecture of the S7025 motherboard

Packets that come in on an interface card are classified by the hardware, assigned an RX queue and transferred usingDMA via PCIe, Tylersburg I/O Hub, QPI, and memory bus to memory. When the packet is in memory, a CPU core makes a forwarding decision based on examining the packet header, and then initiates a TX operation to an outgoing interface card. As can be seen in Figure 1, it is important to identify bottlenecks and to use the inherent parallelism and locality present in the hardware to achieve maximum performance.

2.2 Network interface cards

The box was equipped with five network interface cards with Intel dual 82599 chipset[5]. Theoretical simplex NIC bandwidth was therefore 100Gb/s, but since one card was placed in a 4x PCIe slot, the max was 96 Mb/s. In Section 4 the performance figures are described in more detail.

The Intel 82599 has multiple queues for both RX and TX which are tied to IRQ's and RX/TX-rings. By controlling the queue-selection and interrupt affinity setup it is possible to control and utilize traffic to the CPU cores.

The Intel 82599 has an advanced series of built-in classification filtering in hardware. Traffic classification of incoming traffic is made by defining appropriate filters and matching them with RX queues or by dropping traffic. The RX-queue is handled by the CPU core assigned by the affinity setup.

The traffic filters in the 82599 includes L2 Ethertype and TCP SYN match. There is also a general N-tuple filter matching for matching IP source, destination and TCP ports, for example. Maybe the most powerful mechanism is the *flow-director* where individual flows can be filtered. Finally, the hash-based receiver-side scaling(RSS) can be used as a last resort.

From user space in Linux, it is possible to program the N-tuple filters using ethtool which is supported by both the SUN *niu* driver and the Intel *ixgbe* driver. Actions are add, remove and list of hardware filters.

2.3 Operating system and software

The software was Bifrost[7] 6.0 based on Linux kernel 2.6.31-rc5. Appendix 1 shows the modifications we made to the kernel for the experiments. Bifrost is a Linux release aimed at providing an open source routing and packet filtering platform. Bifrost includes routing daemons, packet filtering, packet generators, etc.

Pktgen[8] was used to generate traffic. We extended the tool with an option to explicitly control memory allocation, the patch is now included in the kernel.

3 Experiment 1: Traffic classification

The goal of the first experiment was to separate control data from data traffic. One core (CPU0) was assigned to control tasks, while the remaining cores were assigned to forwarding. By identifying control

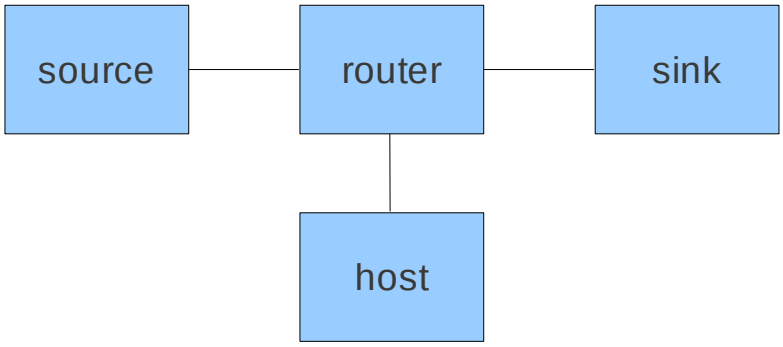

Figure 2: Traffic classification experimental setup

traffic in hardware in the 82599 NICs, the idea was that control traffic could be dispatched to CPU0 while all other traffic was load-balanced over the remaining cores. In this way, the control-plane of a router can continue to operate isolated from the data-plane. For example, denial-of-service attacks taking place in the data-plane will not be able to affect the route-processor.

The router was connected to a source and a sink, and an additional third node as shown in Figure 2. The source and sink generated and received background traffic, respectively, while the extra host communicated with the router using TCP transactions. For this purpose, we used Netperf's TCP transaction test TCP_RR[9]. The receiving part of Netperf was started with CPU affinity to CPU0 on the router. Netperf was run from the host.

The idea to use serial transactions was to highlight the communication to and from the control-plane from an external source. Transactions were serial, so that one was completed before the next was initiated. In this way we get a measure of response time from the control-plane that we believe mimics actual protocol operation. We expect that a modest increase of scheduling latency in the router decreases the transaction rate drastically. Thus this method should be a sensitive indicator of control-plane operation.

The router was loaded with a BGP routing table consisting of 280K routes. To furthermore load and stress the router the route cache was disabled so every packet had to be looked up in the fib_trie. We used a mix of packet-lengths of 64, 512 and 1500 bytes to model real-world traffic[1]. The rate actually being forwarded through the router was 1.33M packets per second and a bandwidth of 9.4 Gb/s of forwarded data.

3.1 N-tuples versus flow-director

Our first approach for separation of traffic was to use N-tuple filters to enforce traffic separation. In that case, filters for identifying local BGP and SSH traffic (for example) should be added and associated with the RX-queue assigned to CPU0.

One problem with this approach is that RSS used for distributing forwarding traffic was still assigning packets to CPU0. Both N-tuple filtered traffic as well as RSS traffic, which actually makes CPU0 have higher load than the other processors.

Therefore, we changed the initialization of the RSS table to not send any packets to CPU0, which was a simple patch to the *ixgbe* driver (see Appendix 1). Instead, the forwarding traffic was evenly spread over the remaining processors.

Another problem was that all flows to the control-plane must be explicitly programmed. We therefore found a way to simplify the setup by using the flow-director rather than explicit programming with N-tuple filters. Using this approach, outgoing flows from CPU0 are used to program the Flow-director so that the returning flows are dispatched back to CPU0. In this way, no explicit identification of flows destined to the

control-plane needs to be made. This approach works as long as there are duplex flows of some duration in time, which is in general the case for control-plane traffic. Therefore, the Flow-director was programmed by the *ixgbe* driver at TX in hard_xmit() to use the same queue for the receiving part of the flow

However, since the size of the flow-director is limited (32K flows), it is necessary that this method only applies to flows from the control-plane, not forwarded traffic. To achieve this another small patch to the *ixgbe* driver was added, the patch is also listed in Appendix 1.

3.2 Results

In a first baseline test, the transaction performance was measured without any forwarding load. The left-most bar in Figure 3 shows a transaction rate of 30000 transactions per second.

Next, the background traffic was started from source to sink causing RSS to distribute the forwarding load evenly across all CPU cores, including CPU core 0 where Netserver was running. This resulted in an increase in transaction latency causing a drastic reduction in transaction rate as seen in the middle bar in Figure 3.

Thereafter, we introduced the RSS-patch so that no packets are sent to CPU core 0. The result is shown in Table 1 and the rightmost bar in Figure 3. Note that no packets are forwarded by core 0 and that the RSS load balancing is reasonably fair between the forwarding cores.

It can be seen in Figure 3 that the transaction performance is very close to the idle machine performance in the baseline test, indicating that we achieved an efficient separation.

CPU-core	0	1	2	3	4	5	6	7
Number of packets	0	196830	200860	186922	191866	186876	190106	190412

Table 1: Effects of the RSS patch: No packets are forwarded by CPU0.

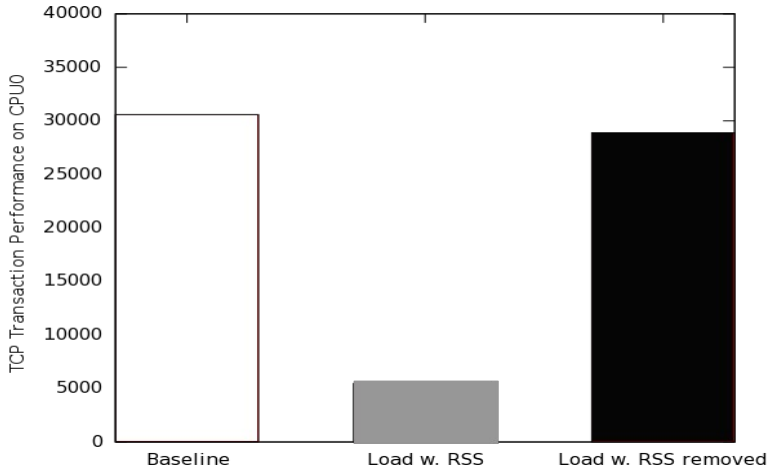

Figure 3: Forwarding capacity

By monitoring /proc/net/softnet_stat we could verify how the packets were handled per CPU core and verify the setup. We did not see any degradation of forwarding performance by using 7 of 8 cores in this case.

3.3 Discussion

We have not yet used the flow-director with multicast traffic. This would be needed for example in OSPF control communication. We have also noticed that traffic separation has some limitations at very high DOS rates using only 64 byte packets. The hardware seems to drop packets before classification. However, this needs further study.

4 Experiment 2: Bus bandwidth performance tests

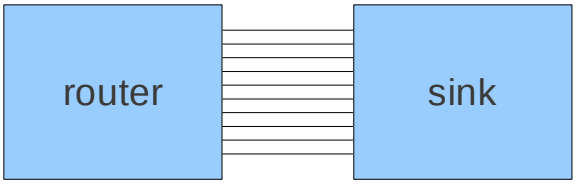

Figure 4: Setup of the performance experiment. Ten Ethernet interfaces were connected to an external system.

In the second experiment, the objective was to generate as much traffic as possible, thus exploring the upper limits of the architecture. All interfaces were connected to obtain running link, and the interface counters were recorded. This experiment was originally made in earlier work [1] where we reported a total throughput of 26 Gb/s. In this paper we reproduce it with newer hardware.

Our idea was to reach the maximum aggregated bandwidth for packet transmission using optimal setup using both Tylersburg bridges, having DMA take the best way according to the hardware topology. The traffic generator (pktgen) was extended with an option to explicitly control memory allocation by a node.

Component	**Bandwidth (simplex payload)**
PCIe gen2 x1	4 Gb/s
PCIe gen2 x4	16 Gb/s
PCIe gen2 x16	64 Gb/s
QPI 2.4GHz (5.8Gt/s)	76 Gb/s
DDR3	68 Gb/s
DDR3 triple chan	3x68Gb/s = 204 Gb/s

Table 2: Approximate payload performance of datapath components in the hardware.

However, exploiting the topology in the hardware as the one shown in Figure 1 is difficult. Not only should CPU cores, interrupt affinity and I/O channels match, the locality of memory and Tylersburg I/O bridges must be taken into consideration. To give a hint of the bottlenecks in the system, Table 2 lists the approximate bus/memory bandwidth of the components used in the experiment. The numbers are an over-simplification but provides an indication of the upper limit of payload traffic when forwarding packets from an interface card to memory and back.

Interface	eth0	eth1	eth2	eth3	eth4	eth5	eth6	eth7	eth8	eth9
CPU-core	0	1	2	3	4	5	6	7	12	13
Memory node	0	0	0	0	1	1	1	1	1	1

Table 3: Performance setup: CPU versus memory node and interface.

We started with a configuration as shown in Table 3, where eth0-eth3, and eth6-eth9 are 10Gb/s Ethernet on PCIe x16. Two interfaces (eth4 and eth5) were mounted on a x4 slot. Note that eth8 and eth9 use the CPU cores 12 and 13 which are hyper-threaded cores sharing physical CPU 1. The memory nodes are local to the CPUs, that is, CPU cores on CPU0 (0-3,8-11) use memory node 0, and CPU cores on CPU1 (4-7,12-15) use memory node 1.

We then let Pktgen send 1500 byte packets continuously out on all interfaces and monitored the bandwidth of the transmitted traffic. The result is plotted in Figure 5. In the figure, it can be seen that 10Gb/s wire-speed for all interfaces except for eth4 and eth5 which were installed in the x4 slot.

The aggregated performance adds up to an impressive 92 Gb/s. This indicates that multiple 10Gb/s cards can be handled and even higher bandwidth NIC's. Such as upcoming 40 Gb/s and 100 Gb/s

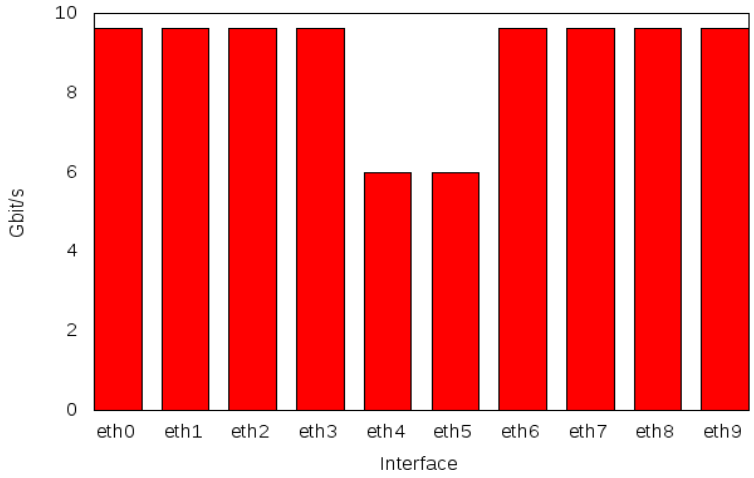

Figure 5: Capacity results with localized memory.

We then reversed the node setup, so that all cores in CPU 0 used memory node 1 and CPU 1 used memory node 0. The result is shown in Figure 6. As expected, the performance degraded somewhat, thus proving the effect of localized memory in forwarding situations.

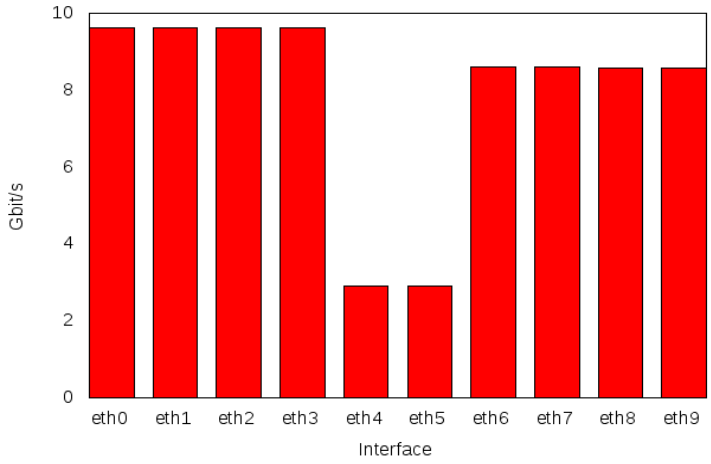

Figure 6: Capacity result with reversed memory nodes

5 Conclusion and future work

In this work we have explored how new hardware can provide better functionality and performance for a Linux-based open-source router on selected PC hardware.

We have shown how control and data traffic can be separated by using a combination of multi-core processors, multi-queues and hardware classification, all available on commercial PC hardware. Separating control and data traffic is important in a router since it makes the control-plane operation isolated from data-plane load, thus reducing the risk of denial-of-service attacks or load-induced degradation.

In our experiments, we used the flow-director and RSS capabilities on the Intel 82599 network interface card and observed the transaction performance between an external host and the router's control-plane in the presence of high forwarding load. The results show that traffic isolation is possible in our setting: The transaction performance is virtually unaffected by forwarding traffic when applying our method, whereas there is a significant performance degradation with an unaltered system.

A limitation of our experimental setting is that the control traffic was only present on a separate interface. We would like to repeat the experiment with in-line control-data. That is, mixing control and data traffic on the same interface.

We also noticed packet loss at very high data rates that we believe are hardware related. This needs further study.

As a second experiment, we explored the bus-bandwidth performance for a high-end PC router hardware, using a simple simplex transmission setup. We populated a PC with 10Gb/s interface cards and generated as much data as possible from memory to the interfaces. In this experiment we reached a total of 92 Gb/s one-way performance, which is promising for today's 10Gb/s and upcoming 40Gb/s and 100Gb/s links.

Linux has a flexible multi-queue implementation which supports many of the new chipsets, not only the one studied in this paper. It has been a great help that the vendors make their chip-documentation open to encourage development, experiments and research. We would not have been able to perform these experiments without open access to the Intel 82599 documentation[5], for example.

One experience we gained during this work is the complex task of setting up the new hardware equipped with multiple buses, memory nodes, CPU's, and interface cards. There is a need to develop new tools in Linux to model the underlying hardware better, and to optimize its forwarding performance.

6 Acknowledgements

This paper is a result of a project sponsored by .SE - The Swedish Internet Infrastructure Foundation. Thanks to Intel's Peter P Waskiewicz Jr and to Intel Sweden for providing network interface cards.

References

[1] O. Hagsand, Robert Olsson, Bengt Görden, *Towards 10Gb/s open-source routing*, Linux Kongress, Hamburg, October, 2008.

[2] O. Hagsand, Robert Olsson, Bengt Görden, O*pen-source routing at 10Gb/s*, Swedish National Computer Networking Workshop (SNCNW), Uppsala, May, 2009.

[3] J. D. Brouer, *10 Gbit/s Bi-Directional Routing on Standard Hardware Running Linux*, LinuxCon, Portland, 2009

[4] A. Doria J. H. Salim, et al. *Forwarding and Control Element (ForCES) Protocol Specification*, IETF RFC 5810, March, 2010

[5] *Intel 82599 10GbE Controller Datasheet, rev 2.3*, Intel corporation, 2010.

[6] Microsoft corporation, *Scalable Networking: Eliminating the Receive Processing Bottleneck-Introducing RSS*, WinHEC 2004 Version - April 14, 2004

[7] R. Olsson, H. Wassen, E. Pedersen, *Open Source Routing in High-Speed Production Use*, Linux Kongress, October 2008.

[8] R.Olsson. *Pktgen the linux packet generato*r. In Proceedings of the Linux Symposium, Ottawa, Canada, volume 2, pages 11 - 24, 2005.

[9] R. Jones et al, *Netperf – A network performance benchmark*, Hewlett-Packard, 1995

Appendix 1: Source code modifications

The kernel source code modifications for the results in this paper were surprisingly small. They are appended here based on linux kernel 2.6.31-rc5

1. Let RSS skip CPU0

```
diff --git a/drivers/net/ixgbe/ixgbe_main.c
b/drivers/net/ixgbe/ixgbe_main.c
index 1b1419c..08bbd85 100644
--- a/drivers/net/ixgbe/ixgbe_main.c
+++ b/drivers/net/ixgbe/ixgbe_main.c
@@ -2379,10 +2379,10 @@ static void ixgbe_configure_rx(struct ixgbe_adapter
*adapter)
        mrqc = ixgbe_setup_mrqc(adapter);

        if (adapter->flags & IXGBE_FLAG_RSS_ENABLED) {
-               /* Fill out redirection table */
-               for (i = 0, j = 0; i < 128; i++, j++) {
+               /* Fill out redirection table but skip index 0 */
+               for (i = 0, j = 1; i < 128; i++, j++) {
                        if (j == adapter->ring_feature[RING_F_RSS].indices)
-                               j = 0;
+                               j = 1;
                        /* reta = 4-byte sliding window of
                         * 0x00..(indices-1)(indices-1)00..etc. */
                        reta = (reta << 8) | (j * 0x11);
```

2. Program flow-director with only localhost traffic

```
@@ -5555,6 +5555,11 @@ static void ixgbe_atr(struct ixgbe_adapter *adapter,
struct sk_buff *skb,
        u32 src_ipv4_addr, dst_ipv4_addr;
        u8 l4type = 0;

+       if(!skb->sk) {
+         /* ignore nonlocal traffic */
+         return;
+       }
+
        /* check if we're UDP or TCP */
        if (iph->protocol == IPPROTO_TCP) {
                th = tcp_hdr(skb);
```