Open-Source PC-Based Software Routers: A Viable Approach to High-Performance Packet Switching*

Andrea Bianco[1], Jorge M. Finochietto[1], Giulio Galante[2], Marco Mellia[1], and Fabio Neri[1]

[1] Dipartimento di Elettronica, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Torino, Italy
{bianco,finochietto,mellia,neri}@polito.it
[2] Networking Lab, Istituto Superiore Mario Boella, via Pier Carlo Boggio 61, 10138 Torino, Italy
galante@ismb.it

Abstract. We consider IP routers based on off-the-shelf personal computer (PC) hardware running the Linux open-source operating system. The choice of building IP routers with off-the-shelf hardware stems from the wide availability of documentation, the low cost associated with large-scale production, and the continuous evolution driven by the market. On the other hand, open-source software provides the opportunity to easily modify the router operation so as to suit every need. The main contribution of the paper is the analysis of the performance bottlenecks of PC-based open-source software routers and the evaluation of the solutions currently available to overcome them.

1 Introduction

Routers are the key components of IP packet networks. The call for high-performance switching and transmission equipment in the Internet keeps growing due to the increasing diffusion of information and communication technologies, and the deployment of new bandwidth-hungry applications and services such as audio and video streaming. So far, routers have been able to support the traffic growth by offering an ever increasing switching speed, mostly thanks to the technological advances of microelectronics granted by Moore's Law.

Contrary to what happened for PC architectures, where, at least for hardware components, de-facto standards were defined, allowing the development of an open multivendor market, networking equipment in general, and routers in particular, have always seen custom developments. Proprietary architectures are affected by incompatibilities in configuration and management procedures, scarce programmability, lack of flexibility, and the cost is often much higher than the actual equipment value.

Appealing alternatives to proprietary network devices are the implementations of software routers based on off-the-shelf PC hardware, which have been recently made available by the open-source software community, such as Linux [1], Click [2] and FreeBSD [3] for the data plane, as well as Xorp [4] and Zebra [5] for the control plane,

* This work has been carried out in the framework of EURO, a project partly funded by the Italian Ministry of University, Education, and Research (MIUR).

just to name a few. Their main benefits are: wide availability of multi-vendor hardware and documentation on their architecture and operations, low cost and continuous evolution driven by the PC market's economy of scale.

Criticisms to software routers are focused on limited performance, instability of software, lack of system support, scalability problems, lack of functionalities. Performance limitations can be compensated by the natural evolution of the performance of the PC architecture. Current PC-based routers and switches have the potentiality for switching up to a few Gbit/s of traffic, which is more than enough for a large number of applications. Today, the maturity of open-source software overcomes most problems related to stability and availability of software functionalities. It is therefore important to explore the intrinsic limitations of software routers.

In this paper we focus only on the data plane, ignoring all the (fundamental) issues related to management functions and to the control plane. Our aim is to assess the routing performance and the hardware limitations of high-end PCs equipped with several Gigabit Ethernet network interface cards (NICs) running at both 1 Gbit/s and 100 Mbit/s under the Linux operating system.

The remainder of the paper is organized as follows. Provided that the Linux IP stack is implemented partly in hardware and partly in software in the operating system kernel, in Sect. 2 we give an overview of the architecture and the operation of hardware commonly available on high-end PCs, whereas in Sect. 3 we describe the current implementation of the IP stack in the Linux kernel. In Sect. 4 we describe our experimental setup, the tests performed, and the results obtained. Finally, in Sect. 5 we conclude and give some directions for future work.

2 PC Architectural Overview

A bare bones PC consists of three main building blocks: the central processing unit (CPU), random access memory (RAM), and peripherals, glued together by the *chipset*, which provides complex interconnection and control functions.

As sketched in Fig. 1, the CPU communicates with the chipset through the *system bus*, also known as front side bus (FSB) in Intel's jargon. The RAM provides temporary data storage for the CPU as long as the system is on, and can be accessed by the memory controller integrated on the chipset through the memory bus (MB). The peripherals are connected to the chipset by the peripheral component interconnect (PCI) shared bus, which allows to expand the system with a huge number of devices, including, but not limited to, permanent data storage units, additional expansion buses, video adapter cards, sound cards, and NICs. All interconnections are bidirectional, but, unfortunately, use different parallelisms, protocols, and clock speeds, requiring the implementation of translation and adaption functions on the chipset.

The following sections detail the operation of the CPU, the RAM the PCI bus, NICs and explain how the these components can be used to implement a software router.

2.1 CPU

State-of-the-art CPU cores run at frequencies up to 3.8 GHz, whereas next-generation CPUs run at 4 GHz. The front side bus is 64-bit wide and runs at either 100 MHz or

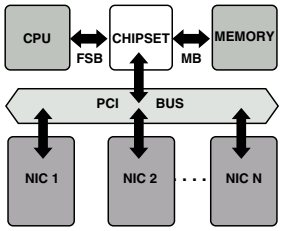

Fig. 1. PC architectural overview.

133 MHz with *quad pumped* transfers, meaning that data are transferred at a speed four times higher than the nominal clock speed. Therefore, the peak transfer bandwidth achievable ranges between 3.2 Gbyte/s and 4.2 Gbyte/s. Note that in Intel's commercial jargon, systems with FSBs clocked at 100 MHz and 133 MHz are marketed as running at 400 MHz and 533 MHz, because of quad pumping. High-end PCs are equipped with chipsets supporting multiple CPUs connected in a symmetric multiprocessing (SMP) architecture. Typical configurations comprise 2, 4, 8 or even 16 identical CPUs.

2.2 RAM

The memory bus is usually 64-bit wide and runs at either 100 MHz or 133 MHz with *double pumped* transfers, meaning that data are transferred on both rising and falling clock edges. Thus, the peak transfer bandwidth available ranges between 1.6 Gbyte/s and 2.1 Gbyte/s, and, in Intel's jargon, the two solutions are named *PC1600* and *PC2100* double data rate (DDR) RAM. In high-end PCs the memory bandwidth is doubled bringing the bus width to 128 bits by installing memory banks in pairs. Note that this allows to match the memory bus peak bandwidth to that of the front side bus.

2.3 PCI Bus

Depending on the PCI protocol version implemented on the chipset and the number of electrical paths connecting the components, the bandwidth available on the bus ranges from 1 Gbit/s for PCI 1.0, when operating at 33 MHz with 32-bit parallelism, to 2 Gbyte/s for PCI-X 266, when transferring 64 bits on both rising and falling edges of a double pumped 133 MHz clock.

The PCI protocol is designed to efficiently transfer the contents of large blocks of contiguous memory locations between the peripherals and the RAM, without requiring CPU intervention. Data and address lines are time multiplexed, therefore each *transaction* starts with an *addressing cycle*, continues with the actual data transfer which may take several *data cycles*, and ends with a *turnaround cycle*, where all signal drivers are three-stated waiting for the next transaction to begin. Some more cycles may be wasted if the *target* device addressed by the transaction has a high initial latency and introduces several *wait states*. This implies that the throughput experienced by the actual data transfer increases as the burst length gets larger, because the almost constant-size protocol overhead becomes more and more negligible with respect to useful data.

As the bus is shared, no more than one device can act as a *bus-master* at any given time; therefore, an *arbiter* is included in the chipset to regulate the access and fairly share the bandwidth among the peripherals.

2.4 NICs

Gigabit Ethernet and Fast Ethernet NICs are high-performance PCI cards equipped with at least one direct memory access (DMA) engine that can operate as bus-masters to offload the CPU from performing back-and-forth bulk data transfers between their internal memory and the RAM. An adequate amount of on-card transmission (TX)/reception (RX) first-in first-out (FIFO) buffer memory is still needed to provide storage for data directed to (received from) the RAM.

The most common operation mode for streaming data transfers is *scatter-gather* DMA, which allows to spread small buffers all over the available RAM rather then allocating a single large contiguous RAM region which may be difficult, if not impossible, to find because of memory fragmentation. During RAM-to-card transfers, the card fetches data *gathering* them from sparse RAM buffers; conversely, in the opposite direction, data originating from the card are *scattered* over available RAM buffers.

The simplest way to implement scatter-gather DMA is with two linked lists, one for transmission which requires RAM-to-card transfers and the other for reception which triggers card-to-RAM transfers. Each list element, dubbed *descriptor*, contains a pointer to the first location of the RAM buffer allocated by the operating system, the buffer size, a command to be executed by the card on the data, a status field detailing the result of such operation, and a pointer to the next descriptor in the list, possibly null if this is the last element. Alternatively, descriptors can be organized in two fixed-size arrays, managed by the card as circular buffers, named *rings*. The performance improvement granted by this solution is twofold: first, descriptors become smaller; second, descriptors on the same ring are contiguous and can be fetched in one burst, allowing for a higher PCI bus efficiency.

During normal operation, the card *i*) fetches descriptors from both rings to determine which RAM buffers are available for reading/writing data and how the related data must be processed, *ii*) transfers the contents of the buffers, *iii*) performs the required operations, and *iv*) updates the status in the corresponding descriptors.

Outgoing packets are read from buffers on the *transmission ring*, whereas incoming packets are written to buffers on the *reception ring*. Buffers pointed by descriptors are usually sized to fit both incoming and outgoing packets, even though, it is often possible to split outgoing packets among multiple buffers. Transmission stops whenever the transmission ring empties, whereas incoming packets are dropped by the card either when the reception ring fills up or when the on-card FIFO overruns because of prolonged PCI bus unavailability due to congestion.

Each NIC is connected to one (possibly shared) hardware interrupt request (IRQs) line and collects in a *status register* information on TX/RX descriptor availability and on events needing attention from the operating system. It is then possible to selectively enable the generation of hardware IRQs to notify the CPU when a given bit in the status register is cleared/set to indicate an event, so that the operating system can take the appropriate action. Interrupts are usually generated when the reception ring is either

full or almost full, when the transmission ring is either empty or almost empty, and after every packet transmission/reception.

In addition, it is usually possible to turn IRQ generation off altogether, leaving to the operating system the burden of periodically *polling* the hardware status register and react accordingly. More details on these packet reception schemes are provided later in the paper.

2.5 Putting All the Pieces Together

The hardware available on a PC allows to implement a shared bus, shared memory router, where NICs receive and store packets in the main RAM, the CPU routes them to the correct output interface, and NICs fetch packets from the RAM and transmit them on the wire. Therefore, each packet travels twice on the PCI bus, halving the bandwidth effectively available for NIC-to-NIC packet flows.

3 Linux Network Stack Implementation

The networking code in the Linux kernel is highly modular: the hardware-independent IP stack has a well defined application programming interface (API) toward the hardware-dependent device driver, which is the glue making the IP layer operate with the most diverse networking hardware.

When a NIC is receiving traffic, the device driver pre-allocates packet buffers on the reception ring and, after they have been filled by the NIC with received packets, hands them to the IP layer. The IP layer examines each packet's destination address, determines the output interface, and invokes the device driver to enqueue the packet buffer on the transmission ring. Finally, after the NIC has sent a packet, the device driver unlinks the packet buffer from the transmission ring. The following sections discuss briefly the operations performed by the memory management subsystem, the IP layer, and detail how the network stack is invoked upon packet reception.

3.1 Memory Management

In the standard Linux network stack implementation, buffer management is performed resorting to the operating system general-purpose memory management system, which requires CPU expensive operations. Some time can be saved if the buffer deallocation function is modified so as to store unused packet buffers on a recycling list in order to speed up subsequent allocations, allowing the device driver to turn to the slower general-purpose memory allocator only when the recycling list is empty.

This has been implemented in a patch [6] for 2.6 kernels, referred to as *buffer recycling* patch in the reminder of the paper, which adds buffer recycling functionalities to the `e1000` driver for Intel Gigabit Ethernet NICs. As of today, the buffer recycling patch has not been officially included in the kernel source, but it could be easily integrated in the core networking system code, making it available for all network device drivers without any modification to their source code.

3.2 IP Layer

The Linux kernel networking code implements a standard RFC 1812 [7] router. After a few sanity checks such as IP header checksum verification, packets that are not addressed to the router are processed by the routing function which determines the IP address of the next router to which they must be forwarded, and the output interface on which they must be enqueued for transmission.

The kernel implements an efficient routing cache based on a hash table with collision lists; the number of hash entries is determined as a function of the RAM available when the networking code is initialized at boot time. The route for outgoing packets is first looked up in the routing cache by a fast hash algorithm, and, in case of miss, the whole routing table stored in the forwarding information base (FIB) is searched by a (slower) longest prefix matching algorithm.

Next, the time-to-live (TTL) is decremented, the header checksum is updated and the packet is enqueued for transmission in the RAM on the drop-tail TX queue associated with the correct output interface. Then, whenever new free descriptors become available, packets are transferred from the output TX queue to the transmission ring of the corresponding NIC. The maximum number of packets that can be stored in the output TX queue is limited and can be easily modified at runtime; the default maximum length for current Linux implementations is 1000 packets.

Note that, for efficiency's sake, whenever it is possible, packet transfers inside the kernel networking code are performed by moving pointers to packet buffers, rather than actually copying buffers' contents. For instance, packet forwarding is implemented unlinking the pointer to a buffer containing received packets from the reception ring, handing it to the IP layer for routing, linking it to the output TX queue, and moving it to the NIC transmission ring before processing the next pointer. This is commonly referred to as *zero-copy* operation.

3.3 Stack Activation Methods

Interrupt. NICs notify the operating system of packet reception events by generating hardware IRQs for the CPU. As shown in Fig. 2, the CPU invokes the driver hardware IRQ handler which acknowledges the NIC request, transfers all packets currently available on the reception ring to the kernel *backlog queue* and schedules the network software-IRQ (softIRQ) handler for later execution.

SoftIRQs are commonly used by many Unix flavors for deferring to a more appropriate time the execution of complex operations that have a lower priority than hardware IRQs and that cannot be safely carried out by the IRQ handler, because they might originate race conditions, rendering kernel data structures inconsistent.

The network softIRQ handler extracts packets from the backlog queue, and hands them to the IP layer for processing as described in Sect. 3.2. There is a limit on the maximum number of packets that can be stored in the backlog queue by the IRQ handler so as to upper bound the time the CPU spends for processing packets. When the backlog queue is full, the hardware IRQ handler just removes incoming packets from the reception ring and drops them. In current Linux implementations, the default size of the backlog queue is 300 packets.

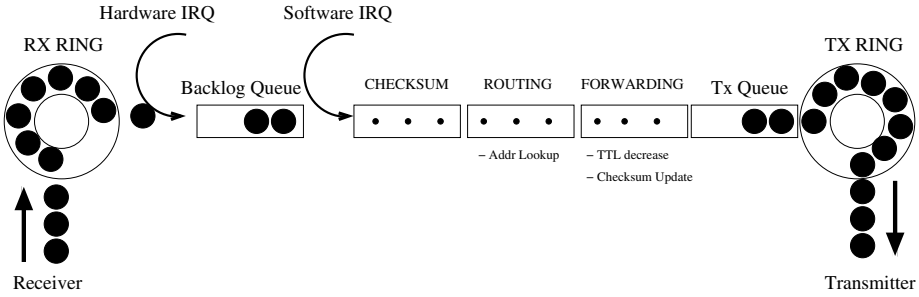

Fig. 2. Operation of the interrupt-driven network stack.

Under heavy reception load, a lot of hardware IRQs are generated from NICs. The backlog queue fills quickly and all CPU cycles are wasted extracting packets from the reception ring just for dropping them after realizing that the backlog queue is full. As a consequence, the softIRQ handler, having lower priority than the hardware IRQ handler, never gets a chance of draining packets from the backlog queue, practically zeroing the forwarding throughput. This phenomenon was first described in [8] and dubbed *receive livelock*.

Hardware IRQs are also used to notify the operating system of the transmission of packets enqueued on the transmission ring so that the driver can free the corresponding buffers and move new packets from the TX queue to the transmission ring.

Interrupt Moderation

Most modern NICs provide *interrupt moderation* or *interrupt coalescing* mechanisms (see, for example, [9]) to reduce the number of IRQs generated when receiving packets. In this case, interrupts are generated only after a batch of packets has been transmitted/received, or after a timeout from the last IRQ generated has expired, whichever comes first. This allows to relieve the CPU from IRQ storms generated during high traffic load, improving the forwarding rate.

NAPI. Receive livelock can be easily avoided by disabling IRQ generation on all NICs and letting the operating system decide when to poll the NIC hardware status register to determine whether new packets have been received. The NIC polling frequency is determined by the operating system and, as a consequence, a polling-driven stack may increase the packet forwarding latency under light traffic load.

The key idea introduced in [8] and implemented in the Linux network stack in [10] with the name new API (NAPI), is to combine the robustness at high load of a polling-driven stack with the responsiveness at low load of an interrupt-driven stack.

This can be easily achieved by enabling IRQ status notification on all NICs as in an interrupt-activated stack. The driver IRQ handler is modified so that, when invoked after a packet reception event, it enables polling mode for the originating NIC by switching IRQ generation off and by adding that NIC to the NAPI *polling list*. It then schedules the network softIRQ for execution as usual.

As shown in Fig. 3, a new function `poll` is added to the NIC driver to *i*) remove packets from the reception ring, *ii*) hand them to the IP layer for processing as described in Sect. 3.2, *iii*) refill the reception ring with empty packet buffers, *iv*) detach from the

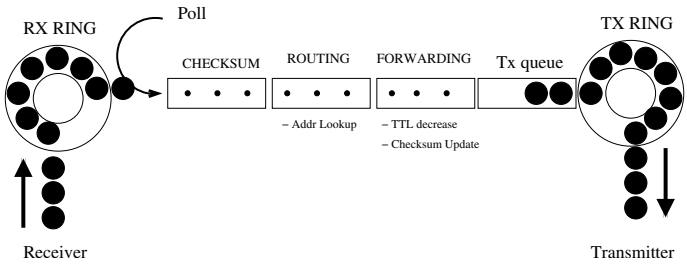

Fig. 3. Operation of the NAPI network stack.

transmission ring and free packets buffers after their content has been sent on the wire. The `poll` function never removes more than a *quota* Q of packets per invocation from the NIC reception ring.

The network softIRQ is modified so as to run `poll` on all interfaces on the polling list in a round-robin fashion to enforce fairness. No more than a *budget* B of packets can be extracted from NIC reception rings in a single invocation of the network softIRQ, in order to limit the time the CPU spends for processing packets. This algorithm produces a max-min fair share of packet rates among the NICs on the polling list.

Whenever `poll` extracts less than Q packets from a NIC reception ring, it reverts such NIC to interrupt mode by removing it from the polling list and re-enabling IRQ notification. The default values for B and Q in current Linux implementations are, respectively, 300 and 64.

Currently, not all NIC drivers are NAPI aware, but it is easy to write the NAPI `poll` handler by just borrowing code from the hardware IRQ handler.

4 Performance Evaluation

In Sect. 4.1, we describe the testbed setup we used and report some preliminary results from experiments on the NIC maximum reception and transmission rate, which motivated the adoption of a commercial router tester for subsequent tests.

Many metrics can be considered when evaluating the forwarding performance of a router: for instance, the RFC 2544 [11] defines both steady-state indices such as the maximum lossless forwarding rate, the packet loss rate, the packet delay and the packet jitter, as well as transient quantities such as the maximum-length packet burst that can be forwarded at full speed by the router without losses. In this paper, however, the different configurations were compared in terms of the *steady-state saturation forwarding throughput* obtained when all router ports are offered the maximum possible load. We considered both *unidirectional* flows, where each router port, at any given time, only receives or transmits data, as well as the *bidirectional* case, where all ports send and receive packets at the same time. All the results reported are the average of five 30-second runs of each test.

The IP routing table used in all tests is minimal and only contains routes to the class-C subnetworks reachable from each port. As a consequence, the number of IP destination addresses to which the router-tester sends packets on each subnetwork is always less than 255, so that the routing cache never overflows and the routing overhead is marginal.

4.1 Testbed Setup

The router tested is based on a high-end PC with a SuperMicro X5DPE-G2 mainboard equipped with one 2.8 GHz Intel Xeon processor and 1 Gbyte of PC1600 DDR RAM consisting of two interleaved banks, so as to bring the memory bus transfer rate to 3.2 Gbyte/s.

Almost all experiments were performed running Linux 2.4.21, except for buffer recycling tests, which were run on a patched 2.6.1 kernel. No major changes occurred in the networking code between kernel version 2.4 and 2.6. The only modification needed to make a fair performance comparison between 2.6 and 2.4 kernels is to lower the clock interrupt frequency from 1000 Hz (default for 2.6 kernels) to 100 Hz (default for 2.4 kernels).

Although Fast and Gigabit Ethernet NICs offer a raw rate of 100 Mbit/s and 1 Gbit/s, respectively, the data throughput at IP layer actually achievable may be much lower because of physical and data-link layer overhead, due to Ethernet overhead. Indeed, besides MAC addresses, protocol type and CRC, also the initial 8-bytes trailer and the final 12-byte minimum inter-packet gap, must be taken into account. Carried payload ranges from a 46-bytes minimum size up to 1500-bytes of maximum size. As a consequence, Gigabit (Fast) Ethernet NICs running at full speed must handle a packet rate ranging from 81 274 (8 127) to 1 488 095 (148 809) packets/s as the payload size decreases from 1500 byte to 46 byte.

Most of the results presented in this section were obtained for minimum-size Ethernet frames because they expose the effect of the per-packet processing overhead. However, we also ran a few tests for frame sizes up to the maximum, in order to check that both the PCI bus and the memory subsystem could withstand the increased bandwidth demand.

A number of tests were performed on Gigabit Ethernet NICs produced by Intel, 3Com (equipped with a Broadcom chipset), D-Link and SysKonnect, using open-source software generators (see [12] for a good survey) operating either in user- or in kernel-space. We compared `rude` [13], which operates in user-space, with `udpgen` [14] and `packetgen` [15], that, instead, live in kernel-space. The aim was to assess the maximum transmission/reception rate of each NIC-driver pair, when only one NIC was active and no packet forwarding was taking place. In order to allow for a fair comparison and to avoid a state-space explosion we left all driver parameters to their default values. The highest generation rate of 650 000 minimum-size Ethernet frames per second (corresponding to almost 500 Mbit/s) was achieved by `packetgen` on Intel PRO 1000 Gigabit Ethernet NICs running with the Intel `e1000` driver [16] version 5.2.52.

As it was not possible to generate 64-byte Ethernet frames at full speed on any of the off-the-shelf NIC we considered, we ran all subsequent tests on an Agilent N2X RouterTester 900 [17], equipped with 2 Gigabit Ethernet and 16 Fast Ethernet ports, that can transmit and receive at full rate even 64-byte Ethernet frames.

The maximum reception rate of the different NICs was also evaluated, when only one NIC was active and packet forwarding was disabled, by generating 64-byte Ethernet frames with the RouterTester, and counting the packets received by the router with `udpcount` [14]. Again, the best results were obtained with the `e1000` driver and Intel PRO 1000 NICs, which were able to receive a little bit more than 1 000 000 packets/s

on an IRQ stack and about 1 100 000 packets/s on a NAPI stack. Given the superior performance of Intel NICs, all tests in the next two sections were performed equipping the router with seven Intel PRO 1000 MT dual port Gigabit Ethernet NICs running at either 1 Gbit/s or 100 Mbit/s. All driver parameters were left to their default values, except for interrupt moderation which was disabled when NAPI was used.

From the packet reception and transmission rates measured when forwarding was disabled, it is possible to extrapolate that, very likely, in a bidirectional two-port scenario, a router will not be able to forward more than 650 000 packets/s each way, because of the NIC's transmission rate limit.

4.2 Two-Port Configuration

We first considered the simplest unidirectional case, where packets are generated from the tester, received by the router from one NIC, and sent back to the tester through the other NIC.

In Fig. 4 we plot the forwarding rate achieved for minimum-size Ethernet frames by IRQ, IRQ moderated, and NAPI activated stacks under unidirectional traffic, as a function of the offered load. Both IRQ and IRQ moderated stacks suffer from receive livelock: when the offered load becomes greater than 230 000 packets/s, or 500 000 packets/s, respectively, the throughput quickly drops to zero. Only NAPI is able to sustain a constant forwarding rate of 510 000 packets/s (corresponding to 340 Mbit/s) under high load. A more accurate analysis of the latter case shows that all packet drops occur on the reception ring rather than from the RX queue. Such forwarding rate is more than reasonable, because a back-of-the-envelope calculation indicates that the per-packet overhead is around 2 μs or about 5500 clock cycles. On the other hand, a 2.8 GHz CPU for forwarding 1 488 095 packets/s would have to take less than 2000 clock cycles (corresponding to approximatively 700 ns) per packet. Given the superior performance of NAPI with respect to other packet forwarding schemes, in the remainder of the paper, we will consider only results obtained with NAPI.

It is now interesting to repeat the last test for different Ethernet payload sizes to assess whether the forwarding rate limitation observed in Fig. 4 depends on the per-packet processing overhead or on a more unlikely PCI bandwidth bottleneck.

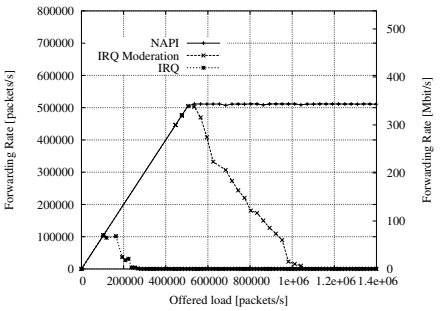

Fig. 4. Comparison of the router forwarding rate under unidirectional traffic for an IRQ, an IRQ moderated and a NAPI stack.

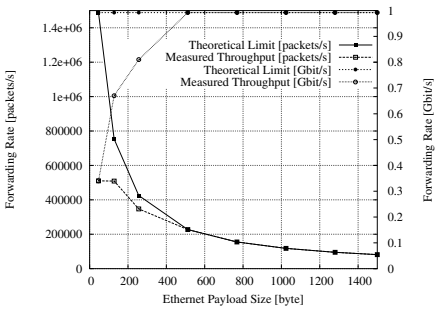

Fig. 5. Comparison of the router saturation forwarding rate for a NAPI stack under unidirectional traffic and different IP packet sizes.

In Fig. 5, we report in both packets/s and Mbit/s the forwarding rate we measured (empty markers) and the value the forwarding rate would have taken in absence of packet drops (filled markers) versus the Ethernet payload size, under NAPI. For 64- and 128-byte payloads the forwarding rate is most likely limited to about 500 000 packets/s by the per-packet CPU processing overhead. Conversely, for payloads of 512 byte or more, it is possible to reach 1 Gbit/s, because the corresponding packet rate is low enough and the PCI bus is not a bottleneck. The result for 256-byte frames is difficult to explain and may be related to some PCI-X performance impairment. Indeed, in [18], the authors, using a PCI protocol analyzer, show that the bus efficiency for bursts of 256 byte or less is pretty low.

Since the PCI bus seems not to be a bottleneck for minimum-size packet transfers, in Fig. 6 we compare the aggregated forwarding rate of unidirectional and bidirectional flows plotted versus the aggregated offered load. Notice that, for bidirectional traffic, the forwarding rate improves from 510 000 packets/s to 620 000 packets/s, corresponding to approximatively 400 Mbit/s. This happens because, at high offered load under bidirectional traffic, the NAPI `poll` function, when invoked, services both the reception and the transmission ring of each NIC it processes, greatly reducing the number of IRQs generated with respect to the unidirectional case, where only one NIC receives packets in polling mode and the other one sends them in IRQ mode. Indeed, measurements performed when the aggregated offered load is around 1 000 000 packets/s show that the transmitting NIC generates, under unidirectional traffic, 27 times as many interrupts than in the bidirectional case.

The curves in Fig. 7 show that the buffer recycling optimization improves the forwarding rate of a bidirectional flow of 64-byte Ethernet frames to 730 000 packets/s, roughly corresponding to 500 Mbit/s. Measurements on the router reveal that, after an initial transient phase when all buffers are allocated from the general-purpose memory allocator, in stationary conditions all buffers are allocated from the recycling list, which contains around 600 buffers, occupying approximatively 1.2 Mbyte of RAM.

All tests in the following section were performed on a Linux 2.4.21 kernel and, therefore, could not take advantage of the buffering recycling patch.

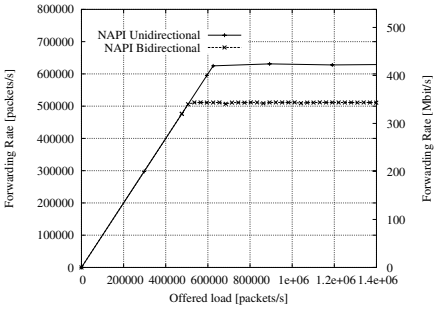

Fig. 6. Comparison of the router forwarding rate achieved by unidirectional and bidirectional traffic flows for a NAPI stack.

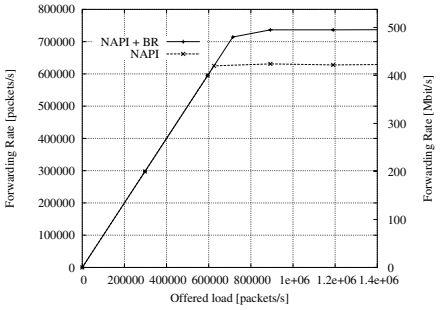

Fig. 7. Comparison of the effect of the buffer recycling (BR) patch on the forwarding rate under bidirectional traffic for a NAPI stack.

4.3 Multi-port Configurations

In this section we consider both a *one-to-one* traffic scenario, where all packets arriving at one input port are addressed to a different output port, so that no output port gets packets from more than one input port, as well as a *uniform* traffic pattern where packets received by one NIC are spread over all the remaining ports uniformly.

We first consider a router with two ports running at 1 Gbit/s and two ports running at 100 Mbit/s, under one-to-one unidirectional traffic. The offered load is distributed according to a 10:1 ratio between the ports running at 1 Gbit/s and the ports running at 100 Mbit/s,and varied from 0 to 100% of the full line rate. The quota Q and the budget B are set to their default values of 64 and 300. Figure 8 shows the total and per-flow forwarding rate versus the aggregated load offered to the router. The forwarding rate for the 100 Mbit/s flow increases steadily, stealing resources from the 1 Gbit/s flow according to a max-min fair policy, when the router bandwidth is saturated, and the aggregated forwarding rate keeps constant.

This behavior can be easily altered to implement other fairness models just changing the quota assigned to each port. Figure 9 shows the results obtained in the same scenario when the quota Q was set to 270 for ports running at 1 Gbit/s and to 27 for ports running at 100 Mbit/s. In this way, `poll` can extract from the former 10 times as many packets than from the latter and, assigning more resources to more loaded ports, aggregated performance is improved.

In Fig. 10 we plot the aggregated forwarding rate versus the aggregated offered load for 6, 10 and 14 ports running at 100 Mbit/s under bidirectional one-to-one traffic. There is a slight decrease in the saturation forwarding rate as the number of ports receiving traffic increases, which may be due to the greater overhead incurred by the NAPI `poll` handler when switching among different ports.

In Fig. 11 we compare the forwarding rate for the one-to-one and the uniform traffic pattern when 14 ports are running at 100 Mbit/s. Again, we observe a small difference in the forwarding rate achieved in the two scenarios, which is probably due to a major processing overhead incurred by the latter.

This may depend on the fact that, under uniform traffic, packets consecutively extracted from the reception ring of one port are headed for different IP subnetworks and

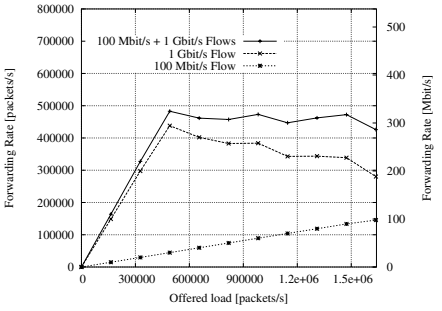

Fig. 8. Max-min fair behavior of two unidirectional traffic flows coming from ports running at different speed under NAPI.

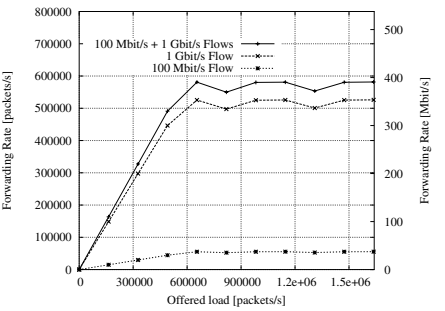

Fig. 9. Impact of different per-port quota setting on the fair share of two unidirectional traffic flows.

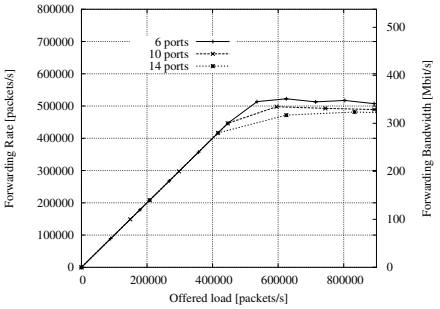

Fig. 10. Forwarding performance for one-to-one bidirectional traffic versus number of 100 Mbit/s ports.

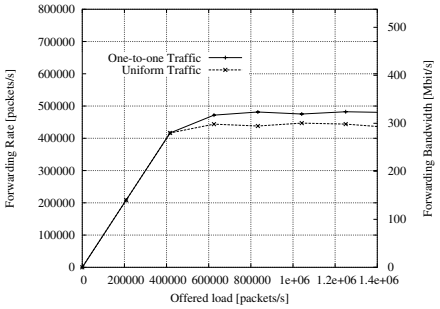

Fig. 11. Forwarding performance comparison between one-to-one and uniform bidirectional traffic with 14 100 Mbit/s ports.

must be spread among different TX queues, contrarily to what happens under one-to-one traffic, where all packets are forwarded to the same TX queue.

5 Conclusions and Future Work

In this paper we evaluated the viability of building a high-performance IP router out of common PC hardware and the Linux open-source operating system. We ran a number of experiments to assess the saturation forwarding rate in different scenarios, completely ignoring all issues related to the control plane.

We showed that a software router based on a high-end off-the-shelf PC is able to forward up to 600 000 packets/s, when considering minimum-size Ethernet frames, and to reach 1 Gbit/s when larger frame sizes are considered. Configurations with up to 14 ports can be easily and inexpensively built at the price of a small decrease in the forwarding rate. A number of tricks, such as packet buffer recycling, and NAPI quota tuning are also available to improve the throughput and alter the fairness among different traffic flows.

In the future we plan to compare the routing performance of Click and FreeBSD with that of Linux. Provided that the major bottleneck in the systems seems to be the per-packet processing overhead introduced by the CPU, we are also profiling the Linux kernel networking code so as to identify the most CPU intensive operations and implement them on custom NICs enhanced with field programmable gate arrays (FPGAs).

Acknowledgment

We would like to thank M.L.N.P.P. Prashant for running the tests on different packet sizes and the experiments for router configurations with more than 6 ports, and Robert Birke for modifying `udpcount` to run on top of NAPI and providing the results on the NIC reception rates. We would also like to thank the partners of the Euro project for the useful discussions, and the anonymous reviewers for helping to improve the paper.

References

1. Torvalds, L.: Linux. (URL: http://www.linux.org)
2. Kohler, E., Morris, R., Chen, B., Jannotti, J.: The click modular router. ACM Transactions on Computer Systems **18** (2000) 263–297
3. FreeBSD. (URL: http://www.freebsd.org)
4. Handley, M., Hodson, O., Kohler, E.: Xorp: An open platform for network research. In: Proceedings of the 1st Workshop on Hot Topics in Networks, Princeton, NJ, USA (2002)
5. GNU: Zebra. (URL: http://www.zebra.org)
6. Olsson, R.: skb recycling patch. (URL: ftp://robur.slu.se/pub/Linux/net-development/skb_recycling)
7. Baker, F.: RFC 1812, requirements for IP version 4 routers. URL: ftp://ftp.rfc-editor.org/in-notes/rfc1812.txt (June 1995)
8. Mogul, J.C., Ramakrishnan, K.K.: Eliminating receive livelock in an interrupt-driven kernel. ACM Transactions on Computer Systems **15** (1997) 217–252
9. Intel: Interrupt moderation using Intel Gigabit Ethernet controllers (Application Note 450). URL: http://www.intel.com/design/network/applnots/ap450.htm
10. Salim, J.H., Olsson, R., Kuznetsov, A.: Beyond softnet. In: Proceedings of the 5th Annual Linux Showcase & Conference (ALS 2001), Oakland, CA, USA (2001)
11. Bradner, S., McQuaid, J.: RFC 2544, benchmarking methodology for network interconnect devices. URL: ftp://ftp.rfc-editor.org/in-notes/rfc2544.txt (March 1999)
12. Zander, S.: Traffic generator overview. (URL: http://www.fokus.gmd.de/research/cc/glone/employees/sebastian.zander/private/trafficgen.html)
13. Laine, J.: Rude/Crude. (URL: http://www.atm.tut.fi/rude)
14. Zander, S.: UDPgen. (URL: http://www.fokus.fhg.de/usr/sebastian.zander/private/udpgen)
15. Olsson, R.: Linux kernel packet generator for performance evaluation. (URL: /usr/src/linux-2.4/net/core/pktgen.c)
16. Intel: Intel PRO/10/100/1000/10GbE linux driver. (URL: http://sourceforge.net/projects/e1000)
17. Agilent: N2X routertester 900. (URL: http://advanced.comms.agilent.com/n2x)
18. Brink, P., Castelino, M., Meng, D., Rawal, C., Tadepalli, H.: Network processing performance metrics for IA- and IXP-based systems. Intel Technology Journal **7** (2003)