Low Level Gigabit Ethernet Analysis for the LHCb Computing Farm

Public Note

Issue: 1
Revision: 1
Reference: LHCb-2005-091
Created: September 5, 2005
Last modified: August 22, 2006
Prepared by: Cedric Walravens, Benjamin Gaidioz
Abstract

This note presents the concept of Receive Descriptor Recycling to significantly reduce the performance drop associated with small packet Gigabit Ethernet traffic. High reliability of small-sized transmissions is crucial for correct calibration runs of the LHCb experiment, at the CERN LHC accelerator. Previous work [1] applied to full link load Ethernet traffic, using UDP processes. This work covers more low-level details of the performance problem for small-sized traffic, using more lower-level Ethernet frames, and, with a deeper analysis at the PCI/PCI-X level. Measurements were performed at the LHCb online system, which is to a large extend made up of commodity equipment. Limits and trade-offs are inherent when optimising for small packet traffic. All important aspects in this context are covered. Results gathered show the Ethernet Controller’s driver currently is the major bottleneck, preventing the system from reaching maximal Gigabit Ethernet performance. Receive Descriptor Recycling is implemented in the driver, and is shown to successfully remedy this situation.

Document Status Sheet

1. Document Title: Low Level Gigabit Ethernet Analysis for the LHCb Computing Farm
3. Issue
4. Revision
5. Date
6. Reason for change
Draft 0 September 5, 2005 First draft version.
Final 1 October 1, 2005 Clearified content where necessary, improved plots, updated references.
Release 1 August 22, 2006 Public Release of Final version.

Contents

1 Introduction .......................................................... 3
   1.1 Performance Defined ............................................. 3
   1.2 Small Packet Performance Issues ................................. 4
   1.3 Goals .................................................................... 4

2 Hardware Setup ......................................................... 4

3 Linux Networking ..................................................... 6
   3.1 Ethernet Frame Format and Data Transfers ....................... 6
   3.2 Receiving Packets .................................................. 6
      3.2.1 Softirq ............................................................ 6
      3.2.2 NAPI ............................................................... 6
      3.2.3 ksoftirqd ........................................................ 7
   3.3 Sending Packets ..................................................... 8

4 Description of the Tests ............................................... 8

5 Receive Throughput and Delay ....................................... 10

6 Transmission Throughput and Delay ................................. 10
7 Response time ............................................. 12
8 PCI-X Analysis .................................................. 17
   8.0.1 A Typical Trace ........................................... 17
   8.0.2 PCI-X Traces ............................................... 18
9 Kernel Profiling ................................................ 20
10 Tuning of the e1000 NIC Driver ............................... 23
   10.1 Receive Descriptor Recycling .............................. 23
   10.2 Results .................................................... 23
11 Conclusion ..................................................... 26
12 References ..................................................... 27

A Receive Descriptor Recycling Patch ............................ 27

List of Figures

1 Typical layout of server systems in the LHCb DAQ Farm. .................. 5
2 The NAPI mechanism in Linux kernel 2.6. ................................ 7
3 Receive throughput from NP (port1) to SFC (eth2) for one port on a dual Intel NIC. .............. 11
4 Receive throughput from NP to SFC for two ports simultaneously on a quad Intel NIC. “sendx→ethy” identifies source (x) and destination (y) port. .................................................... 12
5 Receive throughput from NP to SFC for two ports simultaneously on a dual Intel NIC. (scripted benchmark) ................................................................. 13
6 Transmission throughput as function of delay for different payloads from SFC for a dual port Intel NIC. ................................................................. 14
7 Transmission throughput as function of delay for different payloads from SFC to NP for a quad port Intel NIC. ................................................................. 15
8 PingPong benchmark from SFC to SRV, with both Intel dual cards on PCI-X bus 64bit 133MHz. Notice the sudden increase when the MTU is reached (i.e. due to the extra fragmentation overhead). These points were not included in the linear fit. .................................................... 16
9 Transition between smooth and stumble PCI traffic. (1 of 2 ports receiving @ PCI 64bit 66MHz) ............................. 17
10 PCI-X trace from the beginning of the transition. (1 of 2 ports receiving @ PCI-X 64bit 100MHz) .................. 19
11 Interpreted Postscript plot of the PCI trace from Figure 9 (1 of 2 ports receiving @ PCI 64bit 66MHz) showing left to right and top to bottom the advancing time axis. wd indicates a descriptor is written to the driver, signalling the presence of a received frame in main memory. The reception of Receive Descriptor Tails (rdt) from the driver in blocks of 4 is very clear. The small gaps in the smooth regime signal that the NIC has employed a new RDT and is consecutively reading in the 16 RDs this RDT points to. .................................................... 20
12 Summary of relative occupation of the CPU during a 5 minute flood by the NP (1 of 2 ports @ PCI-X 64bit 133MHz) for a different weight values. 64 is the hard coded value of the unmodified e1000 driver. .................................................... 21
13 Interpreted Postscript plot of a PCI-X 64bit 100MHz trace (1 of 2 ports receiving). The write-back of descriptors has been hidden for readability. .................................................... 22
14 Receive throughput from NP to SRV for two ports on a dual Intel NIC. ............................. 24
15 Receive throughput from NP to SRV for one port on a dual Intel NIC. ............................. 25
16 Comparative plot of RDR improvement. .................................. 26
1 Introduction

This note first describes the tests carried out as part of the LHCb Trigger & DAQ Project to investigate and measure the performance of Gigabit Ethernet components using small raw Ethernet frames. Previous work [1] was more focused on full link load Ethernet traffic, using UDP. However, by performing more lower-level analyses, this note aims, secondly, at identifying the current bottleneck and finally presents and implements the concept of Receive Descriptor Recycling in the NIC’s driver to increase small-sized traffic performance.

The work was funded through the Summer Student 2005 Programme.

1.1 Performance Defined

The meaning of performance for this application includes the following, often related, elements:

1. **Throughput**: a measure of how much data an Ethernet device can transmit or receive per unit of time, for these tests measured in gigabits per second (Gbps). Larger throughput numbers are usually desirable.

2. **Packet rate**: the number of packets that an Ethernet device may send or receive per unit of time, measured in packets per second. Larger packet rates are usually desirable.

3. **Packet loss**: measures the number of packets lost en route. Smaller packet losses are usually desirable.

4. **Per-packet latency**: the delay associated with processing an incoming or outgoing packet. This is usually a function of processing power and efficiency of the local host, rather than any characteristic of the network itself. Smaller latencies are usually desirable.

5. **Response time**: the delay between sending a packet on the network and receiving a response to or acknowledgement of that packet. Smaller response times are usually desirable.

6. **Efficient resource utilisation**: resources will include CPU utilisation, memory utilisation and bus utilisation.

In this work, emphasis has been put on the analysis of small packet performance. Obviously, when the LHCb performs its experiments, data will be packed together as much as possible to create as large as frames as possible (i.e. limited to the MTU to prevent extra fragmentation overhead). However, during the regular calibrations of the experiment, small packet sizes are used. Furthermore, in this case, reliable transmission is often of greater concern than maximising throughput.
1.2 Small Packet Performance Issues

Limits and trade-offs are inherent when optimising for small packet traffic. Possible adverse effects are:

- **CPU utilisation**: Software needs to process incoming packets more quickly, thereby requiring more CPU interrupts and causing an increase in overall system utilisation. This is likely the most important trade-off issue.

- **Resource requirements**: Software needs to allocate more resources in order to accommodate high rates of small packets.

- **PCI-X bus utilisation**: An Ethernet controller needs to transfer packets across the local bus more quickly, increasing bus utilisation and contention.

- **Network utilisation and congestion**: The increased traffic rates that accompany small packet traffic will place greater demands on the network infrastructure.

- **Full-size packet performance**: Increases in small-packet performance may come at the cost of decreased full-size packet performance.

Transmit performance is not affected by small packet traffic to the same extent as receive traffic. This asymmetry exists because the local host cannot usually overwhelm the Ethernet controller with outgoing traffic. Thus, it is not usually necessary to adjust transmit resources to accommodate smaller packet sizes.

1.3 Goals

The aims of the work were to:

- devise useful benchmarks, using the available hardware and software tools.

- automate the benchmarking process as much as possible.

- make baseline network measurements, both on hard- and software level.

- provide understanding of the operation of current networking components.

In the continuation of the chapter we first describe the equipment used and the test environment followed by a short introduction to the way networking is implemented under Linux. Only these parts that are relevant to the LHCb setup are covered, for instance TCP/UDP will not be treated. Next, an overview of the different tests carried out and a more detailed description of the benchmark results are given.

2 Hardware Setup

Today’s commodity servers typically are implemented using motherboards based on the PCI-X bus, with CPUs from Intel or AMD. A typical layout of a SFC server is shown in Figure 1. The processor bus (Front Side Bus, FSB) is coupled to the system bus via the Northbridge. A Southbridge connects the system bus to peripheral buses such as the serial, keyboard and mouse port drivers. Chips connected to the system bus constitute what is referred to as the chipset.

All benchmarks were performed on a local test-bed taken out from the LHCb DAQ Prototype Farm of 50 Dell PowerEdge servers running two 32-bit Intel Xeon processors. The DAQ LAN is a Gigabit Ethernet switched network, connected by Cat 5e Copper UTP wires.

The local benchmark setup consisted of 1 tower server (SRV06), 1 rack server (SFC04) and 1 network processor (SRC13/14) \(^1\). Two other PCs, one running Linux and one running Windows XP, provided
support for the measurement analysis. A detailed description of the systems and motherboards is given in Table 1. All the Intel CPUs (except for the Pentium MMX) had **hyper-threading** enabled. This means that one CPU behaves as two “logical” processors each with its own independent machine states, data registers, control registers and interrupt controller (APIC). However they share the core resources including execution engine, systembus interface, and level-2 cache. Hyper-threading allows the core to execute two or more separate code streams concurrently, using out-of-order instruction scheduling to maximise the use of the execution units [2].

Both Intel Pro/1000 MT Dual and Quad port NICs were used. On SRV06 version 6.0.54-k2-NAPI of the e1000 network driver was used, on SFC04 this was version 5.6.10.1-k2-NAPI. The Ethernet controllers’ specifications are summarised in Table 2. Note that both cards only use 64 bit transfers for frame data, not for addresses, register manipulation or descriptor transfers.

---

1. SRC13 suffered a hard disk failure and was replaced by SRC14.
Table 2 Specifications of NICs used.

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Ports</th>
<th>Chipset</th>
<th>Host bus</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pro/1000 MT</td>
<td>dual</td>
<td>Intel 82546EB</td>
<td>up to PCI-X 64b 133MHz</td>
<td>e1000</td>
</tr>
<tr>
<td>Intel Pro/1000 MT</td>
<td>quad</td>
<td>Intel 82546EB</td>
<td>up to PCI-X 64b 133MHz</td>
<td>e1000</td>
</tr>
<tr>
<td>IBM PowerNP NP4GS3</td>
<td>tri</td>
<td>BCM5700</td>
<td>PCI</td>
<td>n/a</td>
</tr>
</tbody>
</table>

3 Linux Networking

3.1 Ethernet Frame Format and Data Transfers

Raw Ethernet frames using the IEEE 802.3 MAC format [3] were used for all the tests described; they consist of the 14 byte header followed by variable length of user data. The header is made up of the 6 byte destination address, the 6 byte source address and the 2 byte data length indicating the number of bytes following. At the end a 4 byte CRC-checksum is added, resulting in a total physical data overhead of 18 bytes per frame.

3.2 Receiving Packets

Packet reception causes the following sequence of events:

1. NIC generates an interrupt
2. Interrupt handler (e1000_intr()) schedules a softirq
   (_netif_rx_schedule())
3. Interrupt handler exits
4. A check is made for all scheduled softirqs (do_softirq())

We will now explore this outline in more detail. Before the NIC raises an interrupt, it will have copied the packet to main memory through Direct Memory Access (DMA). Each device driver maintains a DMA ring (circular buffer) and the device interrupts the CPU only when the packet is ready in memory. This effectively tackles down the main memory bottleneck.

3.2.1 Softirq

In order to keep the kernel response time small, system interrupts must be disabled for as little time as possible. To achieve this, tasks that need to be performed upon interrupt handling are divided into two: critical and deferrable tasks. During the critical tasks interrupts are masked, deferrable tasks are executed with all interrupts enabled.

Softirqs [4] implement these deferrable tasks to comply to the following properties: a softirq cannot be interrupted to run another softirq on the same CPU, and softirqs can run concurrently on multiple CPUs. When a NIC raises an interrupt, a softirq is scheduled. The softirq code is then executed with interrupts enabled when the interrupt handler finishes (by net_rx_action()). For received network packets, the softirq code type is NET_RX_SOFTIRQ. Note that NET_TX_SOFTIRQ has a higher priority than NET_RX_SOFTIRQ.

3.2.2 NAPI

Another problem is what is commonly referred to as livelock. When a system is flooded by packets, a torrent of interrupts is generated which in turn prevents the kernel from processing the packets, since the CPU spends all its time in interrupt handling. NAPI (New API) [5] tries to resolve this situation.
NAPI's most important goals are to, in the event of an overload, reduce interrupts and early drop packets (i.e., early in the network stack). An overview of NAPI is given in Figure 2. It applies adaptive interrupt coalescing to offer a middle ground between an interrupt driven scheme and polling scheme.

A NIC driver that supports NAPI only interrupts the system on the arrival of the first packet in a batch. Subsequently, it registers to the system that it has work and turns off interrupts concerning receiving packets or running out of receive buffers (referred to as receive descriptors) in its DMA ring. This clearly reduces the number of interrupts generated by the device. Furthermore, any packets arriving after the DMA ring is filled will be dropped without disturbing the system. This approach meets the second design goal.

![Figure 2](image.png)  
*Figure 2 The NAPI mechanism in Linux kernel 2.6.*

Softirq plays a role in the NAPI as such that a softirq is activated to poll all devices that registered to offer packets. They are allowed to set up to a configurable number of packets known as a quota. When this quota is exceeded and if it still has packets to offer (work_done < work_to_do), it is put back at the end of the polling queue. Otherwise, the device is removed from the queue and allowed to interrupt again.

The net effect of NAPI is that under low load, the system converges towards a interrupt driven scheme. Under heavy load, however, it will evolve to a polling scheme which effectively reduces the interrupt-to-packet ratio.

The situation so far is the following: supposing the system is flooded by incoming packets, NAPI's adaptive interrupt coalescing prevents a flood of interrupts. If packets are coming faster than the softirq can process them, i.e. the device's DMA ring is full, then the device silently drops the new packets.

The kernel is thus able to process incoming packets as fast as it can. However, remember that softirqs are invoked upon return from an interrupt handler. With this infrastructure, the kernel will just try to keep up with packet processing and user level code will never get any CPU time. Thus, the system will starve.

### 3.2.3 ksoftirqd

To get around this situation, Linux has implemented a low priority softirq daemon (ksoftirqd [4]) which basically checks for any pending softirqs, if there is free CPU time. Recall that when a NIC has raised an interrupt on packet reception, upon return from the interrupt handler, the softirq polling function do_softirq() is invoked. This function will go through the list of pending softirq's, each time calling their respective handlers. In the case of packet reception, NET_RX_SOFTIRQ is the scheduled softirq and net_rx_action() it's corresponding handler. If, for example, do_softirq() notices that after net_rx_action() returns there is another NET_RX_SOFTIRQ pending, then it does not call net_rx_action() again. Instead it wakes up one of the ksoftirqd processes (there is one ksoftirqd for each logical CPU) by calling wakeup_softirqd(), which will process any remaining packets when it gets CPU time.

---

2see hyper-threading, page 5.
The ksoftirq mechanism ensures that user processes do not suffer under heavy traffic. On the other hand, if the system acts as a network node, as is the case in the LHCb DAQ Project, then ksoftirqd will virtually have all the CPU at its disposal.

3.3 Sending Packets

From a system performance point of view, sending of packets is much simpler. The NIC driver provides a function through consistent DMA mapping which instructs the hardware to begin transmission. Concurrent access to this function is prevented by use of a spinlock. Under heavy load, when the hardware is unable to keep up, packet transmission may be deferred to a low priority ksoftirqd thread.

Each device maintains a queue of packets to be transmitted, using a FIFO queue discipline by default. In case of any problems, such as the device being locked by another thread or CPU (the xmit_lock spinlock), or the transmit DMA ring being full, a softirq (NET_TX_SOFTIRQ) is scheduled and packet transmission will be postponed until a later point in time, regulated by the ksoftirqd.

4 Description of the Tests

Five different tests were carried out:

1. receiving of frames,
2. sending of frames,
3. response time measurement,
4. low-level analysis of PCI-X bus traffic,
5. kernel profiling.

All these tests are related to the different performance elements as set out under Section 1.1.

The first test was done with the SFC in combination with an IBM PowerNP NP4GS3 programmable network processor [6]. It has 3 Gigabit Ethernet ports (Broadcom NetXtreme BCM5700 chipset) and an on-board FPGA (Xilinx Spartan) which allows configuring the board in various ways for sending or receiving frames, e.g. endless, numbered... The NP4GS3 produces useful statistics (RxFrameRate, RxDataRate, FrameErrors,...) for analysing traffic.

For the sending of frames test, the SFC now operated as a source by running the Linux packet generator PKTGEN [7]. It provides a loadable Linux module for high-performance benchmarking the TX process of both the device driver and NIC. Because PKTGEN is "in-kernel", it can generate very high packet rates. This can be seen as the software counterpart of the network processor.

The third test measured response time while directly connecting the SFC and SRV and running a back-to-back pingpong test of 1,000 packets and plotting the minimum round-trip time (rtt) as a function of the packet size. As the SuperMicro motherboard allows for bus speed selection through jumper settings, measurements were done for different PCI-X bus speeds.

The aforementioned benchmarks measure the real performance of the network, but generally do not allow for identification of bottlenecks present. Therefore, lower level tests as PCI-X traffic analysis and kernel profiling were subsequently carried out, emphasising hardware and software, respectfully.

PCI-X traffic was analysed using the VMETRO PBT-615 PCI/PCI-X Bus Analyser. It extracts the bus traffic during a certain window of time. The accompanying BusView software allows for downloading the data from the board to a Windows PC. The measurements can be converted to raw ASCII files, ideal for automated bash scripting analysis.

The last test employed the OProfile package [8], a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. It consists of a kernel driver and a daemon for collecting...
sample data, and several post-profiling tools for turning data into information. OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications. Next, all tests and their accompanying results will be described in great detail.

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Type</th>
<th>Range</th>
<th>SW version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalyst</td>
<td>TA700</td>
<td>100MHz PCI</td>
<td>1.54</td>
</tr>
<tr>
<td>VMETRO</td>
<td>PBT-615</td>
<td>100MHz PCI-X</td>
<td>3.37</td>
</tr>
</tbody>
</table>

Table 3 Specifications of PCI(-X) Analysers used.
5 Receive Throughput and Delay

This test resulted in several ‘throughput versus payload’ and ‘throughput versus delay’ plots. Both the dual and quad card were benchmarked by this test. Note that in the case of the quad card we were limited to 3 ports due to the NP. For every plot, the theoretical Gigabit Ethernet limit is plotted. The calculation goes as follows. As mentioned in Section 3.1, the Ethernet header is 14 bytes. In the datalink layer this is augmented with a 4 byte CRC-checksum. Thus, the total header is 18 bytes. In the physical layer, a preamble of 8 byte and an inter-frame gap of 12 byte is observed, resulting in an extra 20 byte overhead. All this can be written down in the following expression as function of payload $p$:

$$\frac{p + 14 + 4}{p + 14 + 4 + 20}$$

Figure 3(a) shows the bit rate measured when sending to one port of the SFC from one port of the NP with a variable frame size. The bit rate measured at the output of the NP matches pretty well the theory, apart from regular drops. According to the NP documentation, this behaviour is to be expected when pushing this device to the Gigabit Ethernet limit. The SFC receive rate drops accordingly of course. In the range of 0 through 200, this bit rate measured at the input of the SFC is much lower (data is lost). Then, it reaches the sent bit rate (no losses). Receiving on both ports of the dual card at the same time, produces an identical plot.

Figure 3(b) also shows the theoretical limit of the PCI bus, calculated as

$$\frac{p + 14 + 4}{p + 14 + 4 + 128/4}$$

where the 128 bytes in the denominator represent the PCI minimum delay of 16 cycles of 64 bit between bursts, and the division by 4 as conversion factor of 64 bit $\times$ 66 MHz $\approx$ 4 Gbps PCI bandwidth with respect to 1 Gbps Ethernet. We see that, in the case of operating one single port, this PCI bandwidth will never pose any limit on Gigabit Ethernet transfers, even taking into account the fact that this is a theoretical value which will never actually be reached due to numerous non-idealities.

The same figure also shows a linear fit of the first part of the curve. The slope is 0.004, which is in agreement with measurements in [9]. In the continuation of this paper, we will however reach a different conclusion with regard to what contributes to this slope.

The same measurements were done on an Intel quad port NIC, the resulting plot is shown in Figure 4. The slope of the linear part now amounts to 0.002, a value insensitive to the inter-frame delay setting of the NP. This slope is clearly worse than the 0.004 slope of the dual card. A more careful look at the layout of the NIC board reveals that, whereas the dual port is basically one single Intel FW82546EB chip, the quad card has -naturally- two such chips, but also a lot of supporting chips such as a SST 39VF020 2Mbit Flash memory bank, a Tundra Tsi310 PCI-X bridge and an Altera MAX 7000A PLD. This may give cause to the quad card chipset being not as ‘streamlined’ as the single-chip dual card. During this test, no PCI-X transfer bottleneck was observed.

Later, the benchmarking process was automated using bash scripts. Each benchmark resulted in a ‘throughput versus payload’ plot, a ‘throughput versus delay’ plot and a report file describing the complete test performed, the NIC, the driver and the settings used. An example plot is given in Figure 5. For the receive throughput, the ‘rcvd x’ curves are the important ones, with the ‘send x’ plotted for reference; and vice versa for the transmission throughput presented in the next section. All measurements agreed with the above conclusions.

6 Transmission Throughput and Delay

Using the knowledge of the previous benchmark automation, we were able to immediately implement a bash script to run the PKTGEN benchmarks. These also provided us with a transmission ‘throughput
Figure 3  Receive throughput from NP (port1) to SFC (eth2) for one port on a dual Intel NIC.
versus payload’ and ‘throughput versus delay’ plots, each for different delays and payloads, respectfully.

For the dual cards, no significant decrease in performance was observed. Sending on one or two ports gave basically the same plot, as can be seen in Figure 6.

For the quad cards, some more interesting results are obtained, see Figure 7. Going from from 1 to 2 ports sending, the behaviour is basically the same. Starting from 3 transmitting ports onwards, a change in performance is observed. Small delays of 3 µs and 5 µs come together and, for higher delays, we see that the throughput actually increases.

There is an important remark to make concerning the tests conducted under Section 5 and 6. Due to the extensive time it takes to perform a full length sweep, the points of measurement were greatly reduced to do so-called quick runs. This means that the importance of one point of a plot increases, while its accuracy of course does not. This goes for both the receive and transmission test. In addition, the former test has to deal with the NP being pushed to its limits, and the latter suffers from variable operating system conditions.

7 Response time

The aim of this test is to retrieve the slope of the delay as function of payload. The minimum round trip time (rtt) was measured using a scripted ‘ping -A’ command, for gradually increased payload size, each time taking 1,000 measurements. Note that the effective payload is 46 bytes larger now since the final packets also contain an ICMP header (8 bytes), an IP header (20 bytes), an Ethernet header (14 bytes) and CRC-checksum (4 bytes). The resulting data points were plotted and linearly fitted with gnuplot, giving the slope and intersect.

Following [10], this slope compared to an expected value gives an insight in the total amount of time spent in other parts of the system. An overview of the expected slopes is given in Table 4.

The measured values are presented in Table 5, along with their difference compared to the total expected rtt going from SFC04 to SRV06 and back, effectively going through every component twice. All the values are well in agreement with the expected ones, given the rather large margin of error for
(a) Throughput as function of payload for different delays.

(b) Throughput as function of delay for different payloads.

Figure 5  Receive throughput from NP to SFC for two ports simultaneously on a dual Intel NIC. (scripted benchmark)
Figure 6 Transmission throughput as function of delay for different payloads from SFC for a dual port Intel NIC.
Low Level Gigabit Ethernet Analysis for the LHCb Computing Farm

Ref: LHCb-2005-091

Date: August 22, 2006

Figure 7 Transmission throughput as function of delay for different payloads from SFC to NP for a quad port Intel NIC.
Figure 8 PingPong benchmark from SFC to SRV, with both Intel dual cards on PCI-X bus 64bit 133MHz. Notice the sudden increase when the MTU is reached (i.e. due to the extra fragmentation overhead). These points were not included in the linear fit.

Apart from the slope, also the intersect was calculated by the fit. All tests had a small intersect of about 23 to 24 $\mu$s, confirming that interrupt throttling was disabled on the Ethernet controller.

Intel Gigabit Ethernet controllers implement three ways of moderating the interrupt rate [11]:

- An absolute timer, which starts upon reception (transmission) of the first packet. An interrupt is issued when the timer has expired. Its purpose is to assure interrupt moderation in heavy traffic conditions. The timer is set in steps of 1,024 $\mu$s.

- A packet timer, which starts upon reception (transmission) of every packet. An interrupt is generated only when this timer expires. Since it is reset at every packet received (transmitted), it

### Table 4  Expected slopes of different transfer elements.

<table>
<thead>
<tr>
<th>Transfer Element</th>
<th>Expected slope ($\mu$s/byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI-X 64b 66MHz</td>
<td>0.00189</td>
</tr>
<tr>
<td>PCI-X 64b 100MHz</td>
<td>0.00125</td>
</tr>
<tr>
<td>PCI-X 64b 133MHz</td>
<td>0.00094</td>
</tr>
<tr>
<td>1 Gigabit Ethernet</td>
<td>0.00800</td>
</tr>
</tbody>
</table>

### Table 5  Measured slopes of different setups.

<table>
<thead>
<tr>
<th>Ping Network</th>
<th>Pong Network</th>
<th>Minimum rtt slope ($\mu$s/byte)</th>
<th>Difference ($\mu$s/byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI-X 64b 66MHz</td>
<td>1 Gbps 133MHz</td>
<td>0.02739</td>
<td>5.4e-3</td>
</tr>
<tr>
<td>100MHz</td>
<td>1 Gbps 133MHz</td>
<td>0.02575</td>
<td>5.7e-3</td>
</tr>
<tr>
<td>133MHz</td>
<td>1 Gbps 133MHz</td>
<td>0.02531</td>
<td>5.5e-3</td>
</tr>
</tbody>
</table>
is not guaranteed to generate an interrupt if the network traffic is high. Its purpose is rather to lower the latency in low traffic situations. The timer is set in steps of 1,024 $\mu$s.

- An interrupt throttle mechanism can be used to set an upper boundary on the interrupt rate generated by the controller. The corresponding parameter is the maximum interrupt rate.

While the first two timers work independently for reception and transmission of frames, the throttle mechanism can be used to ensure that despite quickly varying traffic conditions in both directions, the total interrupt rate does not increase above the given value.

It is clear that one can view NAPI as a software implementation of interrupt throttling, but actually NAPI goes much further than just that, since it is much more dynamic and also implements a complete polling mechanism (see page 6).

In the case of a DAQ system, it is obviously preferred that the system limits itself to what it can do (with NAPI), possibly being in a very busy loop of “packet processing”, in order to not lose packets. Note that thanks to the softirq mechanism, the system will still not starve. With interrupt throttling, the Ethernet controller will discard packets before the PC is at its limit, which is clearly not optimal.

In fact, the whole point of hardware interrupt throttling is actually to assist operating systems which do facilitate any NAPI-support (e.g. Linux kernel 2.4 and earlier).

## 8 PCI-X Analysis

Similar to network overhead issues, PCI/PCI-X bus protocols also impose penalties on small packet traffic. In addition to the actual data transfer, each bus transaction requires extra control cycles which introduce overhead (e.g., arbitration latency, address phases, attribute phases, wait states). This overhead reduces the efficiency of each transaction as well as overall bus performance. For example, placing a 64-byte data packet over PCI-X Bus, requires 8 bus cycles while the overhead of a transaction requires a minimum of 4 cycles (3 to initiate and 1 cycle to terminate). A “worst case” will add up to 16 wait cycles. Bus overhead costs become more pronounced with small packet traffic as bus control and wait cycles consume a larger fraction of each bus transaction.

All PCI/PCI-X measurements shown in this section are taken on the SRV06 server, since it allowed for a flexible PCI/PCI-X bus type and speed setting. Also, it was not possible to perform PCI-X measurements in the SFC, because it has only one PCI-X slot, and the VMETRO PBT-615 does not feature a Device-Under-Test slot on top of the card.

### 8.0.1 A Typical Trace

![Figure 9](image)

**Figure 9** Transition between smooth and stumble PCI traffic. (1 of 2 ports receiving @ PCI 64bit 66MHz)
Figure 9 shows the two regimes of transmission across the PCI bus in one single trace. The dark areas mean that the signal varies a lot, indicating frames are transmitted across the bus. Notwithstanding that this plot was taken in PCI mode, exactly the same behaviour was observed when analysing the PCI-X traffic. This section summarises the flow of traffic on the bus, going from the smooth to the stumble regime.

During the smooth regime, all frames are nicely put on the PCI bus behind each other (inter-frame delay of 54 CLks or 0.8 μs). The Intel Pro/1000 MT Ethernet Controller uses receive and transmit descriptors to keep the books. Such a descriptor is basically a pointer to memory space, indicating a block of configurable size for the NIC to store a received frame or read in a frame to send. From here on, we will concentrate on reception of frames. When a frame is received, the Ethernet controller uses a receive descriptor (RD) to find out where to store the frame. After the DMA transfer finishes, this receive descriptor is sent back to the driver to advertise the presence of a frame at that location (delay of 7 CLks). Under normal circumstances, the driver will provide the NIC with newly allocated RDs in time before the card runs out of descriptors. This is what happens in the smooth regime, where the driver sends a new bunch of RDs every 3800 CLks (58 μs). A RD bunch transfer continues until all allocated descriptors for that batch have been provided, or, the card’s frame receive buffer is nearly full and the card terminates the descriptor transfer by raising the STOP# signal in order not to lose any frames. Since a NP is flooding the card during this test (with approx. 1 million packets per second), the latter will always happen, as we see on the STOP# signal line. After the descriptor transfer, the frame receive buffer is quickly emptied to regain the time lost during this transfer. All frames are now put on the PCI bus with a minimum delay of 7 CLks in between.

During the smooth regime, all frames are nicely put on the PCI bus behind each other (inter-frame delay of 54 CLks or 0.8 μs). The Intel Pro/1000 MT Ethernet Controller uses receive and transmit descriptors to keep the books. Such a descriptor is basically a pointer to memory space, indicating a block of configurable size for the NIC to store a received frame or read in a frame to send. From here on, we will concentrate on reception of frames. When a frame is received, the Ethernet controller uses a receive descriptor (RD) to find out where to store the frame. After the DMA transfer finishes, this receive descriptor is sent back to the driver to advertise the presence of a frame at that location (delay of 7 CLks). Under normal circumstances, the driver will provide the NIC with newly allocated RDs in time before the card runs out of descriptors. This is what happens in the smooth regime, where the driver sends a new bunch of RDs every 3800 CLks (58 μs). A RD bunch transfer continues until all allocated descriptors for that batch have been provided, or, the card’s frame receive buffer is nearly full and the card terminates the descriptor transfer by raising the STOP# signal in order not to lose any frames. Since a NP is flooding the card during this test (with approx. 1 million packets per second), the latter will always happen, as we see on the STOP# signal line. After the descriptor transfer, the frame receive buffer is quickly emptied to regain the time lost during this transfer. All frames are now put on the PCI bus with a minimum delay of 7 CLks in between.

This is what happens in the second so-called stumble regime, where huge gaps of 4000 CLks show the lack of RDs is preventing any further traffic. This results in many FIFO receive buffer overruns and a huge packet loss of up to 500k packets per second. Furthermore, the omnipresent STOP# indicates that any transfer that takes longer that absolutely necessary is abruptly terminated by the NIC.

In the smooth part, the system can process 1 frame per microsecond (i.e. what the NP sends). When tumbling into stumble regime, this number degrades to a value of merely 0.3 frames per microsecond or 0.28 Gb/s, cf. earlier results, e.g. Figure 3. This also confirms that the linear part for small packet sizes is caused by this stumble behaviour. Where one might be tempted to think the PCI bus or NIC is the cause for this bottleneck, calculations on PCI bus utilisation proved otherwise. During the whole trace, peak data rates did not reach any higher than 250 MB/s, which is still far away from the PCI limit (i.e. 533 MB/s in this case). This was already a clear indication that nor the PCI bus, nor the card were responsible for this stumble behaviour. Note that this trace was taken for a single port receiving frames. It is clear that a PCI 64bit 66MHz will pose a bottleneck when more than 1 port is operating on the same bus.

8.0.2 PCI-X Traces

Figure 10 visualises the PCI-X trace from the start of the transmission by the NP to SRV06. A closer analysis of the bus traffic revealed the following pattern in the smooth receive regime:

1. Four frames are transferred to main memory through DMA bursts with an inter-frame delay of about 45 CLks or 0.4 μs.
2. Right afterwards (i.e. 25 CLks or 0.2 μs), 4 corresponding receive descriptors are written back to the driver.
3. Step 1 and 2 are repeated 16 times (containing a total of 64 frames), after which four new Receive Descriptor Tail (RDT) are received from the driver, each allowing the Ethernet controller to fetch 16 new consecutive descriptors.
Figure 10  PCI-X trace from the beginning of the transition. (1 of 2 ports receiving @ PCI-X 64bit 100MHz)
Compared to PCI, it was observed that PCI-X traffic contained less errors and less accompanying WAIT states. Along with higher clock frequencies, the more intelligent use of WAIT states and so-called split transactions instead of delayed transactions are among the most important features of the PCI-X protocol.

Analysing plots as shown in Figure 9 and 10 is tedious and time consuming. Luckily, the BusView software allowed for data to be converted into plain ASCII files which in turn can be manipulated using numerous Linux tools such as grep and awk to extract the desired information. Using Postscript language, the extracted data was then converted into custom made plots, e.g. Figure 11.

While analysing these custom made plots, some arguments implying a software bottleneck were gathered. A look at the kernel’s memory management information, provided by the /var/slabinfo file, made clear that memory access was responsible for the large gaps in stumble regime. The gaps can be explained by the DMA slab freeings. Normally, a CPU keeps a freed DMA slab in its bucket (one bucket per CPU). Occasionally, however, a large bunch of main memory is reallocated by the driver and then the main memory needs to be accessed, preventing others, e.g. DMA accesses by the NIC, from accessing the main memory. Beware, not the memory bank technology but the way the driver seems to handle memory access is causing the bottleneck. Simple on the back-of-the-envelope calculations show that the DDR-2 400MHz (PC2-3200) provides enough raw bandwidth.

It is clear that the Intel Pro/1000 MT Ethernet Controller card, when used in combination with a fast PCI-X host bus, will not become a bottleneck, even for quad port operation. Therefore, the next section will look closer to the software side of the network infrastructure, i.e. the Linux operating system and the e1000 Ethernet controller driver.

9 Kernel Profiling

It is instructive to understand where the system is spending most of its time. The OProfile report file outputs the number of times a certain function (in user space, kernel, driver,..) has been called and what percentage of the total samples taken this represents. This benchmark was run under a uniprocessor compiled version of kernel 2.6.12 with hyper-threading disabled. This proved to be beneficial
for our purposes as all calls are then made on one single CPU, allowing for a true sequential analysis of function calls. Furthermore, as a positive side effect, this simplified the report file to only one single data set. One drawback when running the uniprocessor kernel, was the system becoming really starved when flooded with packets by the NP. This, unfortunately, renders the gathered statistics less representative, since under 100% CPU usage the scheduling has a non-negligible influence. To somewhat diminish this indeterminism, the IRQ affinity was manually set and the irqbalance daemon disabled.

While performing this benchmark, a first attempt was made to tune the e1000 NIC driver by increasing the netdev->weight value in main_e1000.c (see [12]) from its original hard coded value of 64. One Receive Descriptor Tail (RDT) is a pointer indicating the presence of 16 consecutive Receive Descriptors in main memory. By default, 4 such RDTs will be sent in one NAPI poll (see Figure 11), since 16 × 4 = 64. Thus, increasing this value will increase the number of RDTs transmitted in one period, e.g. 12 in Figure 13 for a weight of 196. Any remaining RDTs less than the budget (i.e. quota allowed by NAPI polling function) are sent afterwards when the scheduled low priority ksoftirq daemon is called. At first sight this does increase the number of RDTs sent to the NIC at once, and as such should reduce the packet loss rate. However, this is not the case. On the contrary, on the short-term more RDTs are sent in one NAPI poll, but there is now a larger gap between every NAPI poll, effectively reducing the number of RDTs in the long run: in 98,500 CLKs (1 ms @ 100 MHz) from 28 RDTs for the default weight value of 64, to 25 RDTs for the same period of time, for a weight of 196. This is a decrease of 11%, which is clearly not desirable. Also, the transition from smooth to stumble regime will happen sooner. This is supported by the observed increase in packetloss from 43% to 54%.

Results of the OProfile test are summarised in Figure 12. It shows that by increasing the weight, less time is spent in cleaning descriptors by the driver (e1000_clean_rx_irq()) and socket buffers by the kernel (skb_release_data()). On the other hand, more time is spent allocating (new) RDs (e1000_alloc_rx_buffers()). This is as expected when increasing the short-term amount of RDTs provided. The decrease of eth_type_trans() calls again provides some indication that there are less packets to process, i.e. packets have been lost.

Given the above results of trying to tune the weight value, we will now concentrate on the profile of the original hard coded weight of 64. Analysing the exact content of the most frequently made calls in the original case, it is established that all are involve with freeing and reallocating RDs. This gave cause for the idea to tune the e1000 driver to more cleverly handle this RD processing, and implement some kind of Receive Descriptor recycling. This idea, along with its implementation in the e1000 source code, will be covered in the next section.
Figure 13
Interpreted Postscript plot of a PCI-X 64bit 100MHz trace (1 of 2 ports receiving). The write-back of
descriptors has been hidden for readability.
10 Tuning of the e1000 NIC Driver

The e1000 driver source code can be found in your kernel source directory or [12], of which the e1000_main.c file contains most of the driver’s basic functionality. For example, e1000_intr() is the interrupt handler which is invoked when an interrupt is raised by the Ethernet controller. In this function, the Interrupt Cause Register (ICR) is read, and if the NIC indicated it has received packets and wrote them to memory, all interrupt involving packet reception are disabled and the device schedules itself in the NAPI queue. At some later point, a softirq is activated and will poll all devices that registered to offer packets. The adapter->netdev->poll member points to the e1000 NAPI polling function, e1000_clean(). It determines the amount of work_to_do, based on the minimum of either the allowed quota received from NAPI or the configurable adapter->netdev->weight value (now left unchanged at the original hard coded value of 64). Now, e1000_clean() basically calls e1000_clean_rx_irq() to send received data up the network stack. e1000_clean_rx_irq() finishes by calling e1000_alloc_rx_buffers() to reallocate used receive descriptors. It is in this function that we find the include/linux/skbuff.h::dev_alloc_skb() function call.

This short overview points out that the most frequently called functions of Section 9 (Figure 12, weight 64) are all related to one single very time consuming operation: the freeing of large heaps of memory. Especially when talking small payload sizes, it is clear that the situation only worsens due to more descriptors and thus more kfree() overhead.

It is this overhead that prevents the driver from quickly sending new RDs to the Ethernet controller, as they need to be freed and reallocated first. This is why we implemented a Receive Descriptor Recycling mechanism. The idea is to allocated a fixed number of permanent descriptors which are reused every time around, effectively taking away the need for the costly kfree() overhead. The only processing that remains to be done is resetting some fields in the descriptors. The remainder of this section will outline the details of this implementation.

10.1 Receive Descriptor Recycling

First we needed to store the permanent buffers in the e1000_adapter struct (see e1000.h), so they become associated with each Ethernet controller present in the system. For this, it was extended by two members: an array to store the preallocated socket buffers, and an array to point to the associated data fields. We opted for a fixed array size of 2,048 which was the RxDescriptors e1000 driver parameter used in all previous tests.

During the driver initialisation, we loop through these arrays to allocated memory and setup the data pointers. To prevent the driver from freeing our allocated array, we artificially increase the number of users for each socket buffer to 2 by calling skb_get(). As long as the number of users remains higher than 2, kfree_skb() will never free them, since it believes someone is still using the socket buffer and its data.

This is realised by altering the driver function call flow in e1000_alloc_rx_buffers(). Here the call to dev_alloc_skb(), which would return a fresh allocated socket buffer (skb), is replaced by returning one of our preallocated skbs in the array. Next, the skb reset is implemented by the newly added reset_skb() function.

Please refer to Appendix A for a detailed overview of exact code implementations.

10.2 Results

To check for any performance increase for small packet sizes, the ‘Receive Throughput and Delay’ test (see Section 5) was repeated. For small frame sizes, the RDR-enabled driver was able to reduce packet loss by 40%, see Figure 14 and 15. The influence is the most clear for short delay packets. The performance of higher payloads remained unchanged with RDR.
(a) Throughput as function of payload for different delays, using the RDR-enabled driver.

(b) Throughput as function of payload for different delays, using the official e1000 driver v6.0.54-k2-NAPI.

Figure 14  Receive throughput from NP to SRV for two ports on a dual Intel NIC.
10 Tuning of the e1000 NIC Driver

0.1

0.2

0.3

0.4

0.5

0.6

0.7

0.8

0.9

1

0

200

400

600

800

1000

1200

1400

1600

bit rate (Gb/s)

payload (bytes)

(a) Throughput as function of payload for different delays, using the RDR-enabled driver.

(b) Throughput as function of payload for different delays, using the official e1000 driver v6.0.54-k2-NAPI.

Figure 15 Receive throughput from NP to SRV for one port on a dual Intel NIC.
11 Conclusion

Several benchmarks were performed on Gigabit Ethernet hardware and the Ethernet controller driver. Among the aspects analysed in this work were throughput, packet loss, response time and efficient resource utilisation. Emphasis was put on small packet size network traffic, to be used for calibration of the LHCb experiment. It was shown that the current bottleneck lies in the way the e1000 driver handles the receive descriptor memory management, which proofs to be fatal for small packet sizes. In order to remedy the situation, a Receive Descriptor Recycling mechanism was proposed and implemented in the official e1000 driver. Results have shown an improvement of small packet size performance by 40% in terms of increase in throughput and reduction of packet loss.

Figure 16 Comparative plot of RDR improvement.
12 References


[10] Performance of 1 and 10 Gigabit Ethernet Cards with Server Quality Motherboards, R. Hughes-Jones et al.


A Receive Descriptor Recycling Patch

Remark: This patch is written for kernel 2.6.12 and e1000 version 6.0.54-k2-NAPI. This patch is provided "as-is", without express or implied warranty of any kind. For more details, see the GNU General Public License. Furthermore, note that the patched driver has not been tested to the greatest of extend.

```c
*** e1000_orig.h
--- e1000_hack.h
*************** struct e1000_adapter {
*** 280,285 ****
--- 280,289 ----
    uint32_t test_icr;
    struct e1000_desc_ring test_tx_ring;
    struct e1000_desc_ring test_rx_ring;
+   struct sk_buff * prealloc_buff[2048];
+   void* data[2048];
    int msg_enable;

*** e1000_main_orig.c
--- e1000_main_hack.c
*************** e1000_sw_init(struct e1000_adapter *adap
*** 852,857 ****
--- 852,877 ----
    spin_lock_init(&adapter->stats_lock);
    spin_lock_init(&adapter->tx_lock);
+   /*
+   + We keep skb->users > 1, so that kfree() will never actually free anything.
+   + (note for later: kfree() will decrement users by one, so we should increase every time.)
```
int i;
unsigned int bufsz = adapter->rx_buffer_len + NET_IP_ALIGN;
struct skbuff* skb;

for (i=0;i<2048;i++) {
  // prealloc a set of buffers which are ready for use
  skb = dev_alloc_skb(bufsz);
  adapter->prealloc_buff[i] = skb;
  // increase users by one, so it becomes 2. (see skb_alloc() for init value)
  skb = skb_get(skb);
  // store copy of data
  adapter->data[i] = skb->data;
}

return 0;

*************** e1000_configure_rx(struct e1000_adapter
*** 1318,1333 ****
    uint64_t rdba = adapter->rx_ring.dma;
    uint32_t rdlen, rctl, rxcsum;

! if(adapter->rx_ps) {  
!   rdlen = adapter->rx_ring.count *  
!   sizeof(union e1000_rx_desc_packet_split);
!   adapter->clean_rx = e1000_clean_rx_irq_ps;
!   adapter->alloc_rx_buf = e1000_alloc_rx_buffers_ps;
! } else {
!   rdlen = adapter->rx_ring.count * sizeof(struct e1000_rx_desc);
!   adapter->clean_rx = e1000_clean_rx_irq;
!   adapter->alloc_rx_buf = e1000_alloc_rx_buffers;
! }

    /* disable receives while setting up the descriptors */
    rctl = E1000_READ_REG(&adapter->hw, RCTL);
--- 1338,1353 -----
    uint64_t rdba = adapter->rx_ring.dma;
    uint32_t rdlen, rctl, rxcsum;

! // if(adapter->rx_ps) {
!    rdlen = adapter->rx_ring.count *  
!    sizeof(union e1000_rx_desc_packet_split);
!    adapter->clean_rx = e1000_clean_rx_irq_ps;
!    adapter->alloc_rx_buf = e1000_alloc_rx_buffers_ps;
! // } else {
!    rdlen = adapter->rx_ring.count * sizeof(struct e1000_rx_desc);
!    adapter->clean_rx = e1000_clean_rx_irq;
!    adapter->alloc_rx_buf = e1000_alloc_rx_buffers;
! // }

    /* disable receives while setting up the descriptors */
    rctl = E1000_READ_REG(&adapter->hw, RCTL);
*************** e1000_free_rx_resources(struct e1000_ada
*** 1483,1488 ****
--- 1503,1521 -----
    pci_free_consistent(pdev, rx_ring->size, rx_ring->desc, rx_ring->dma);

    rx_ring->desc = NULL;
    int i;
Low Level Gigabit Ethernet Analysis for the LHCb Computing Farm

Ref: LHCb-2005-091

Public Note

A Receive Descriptor Recycling Patch

Date: August 22, 2006

```c
+    unsigned int bufsz = adapter->rx_buffer_len + NET_IP_ALIGN;
+    struct sk_buff* skb;
+
+    for (i=0; i<2048; i++) {
+      skb = adapter->prealloc_buff[i];
+      // decrease users to 1
+      dev_kfree_skb(skb);
+      // now we will free
+      dev_kfree_skb(skb);
+    }
+  }

+++ 3093,3099 ****
--- 3126,3171 ----
+ static void
+    reset_skb(struct sk_buff* skb, struct e1000_adapter* adapter, unsigned int i)
+    {
+      u8* data = adapter->data[i];
+      unsigned int size = adapter->rx_buffer_len + NET_IP_ALIGN;
+      
+      /* Get the DATA. Size must match skb_add_mtu(). */
+      size = SKB_DATA_ALIGN(size);
+      
+      //if (!data)
+      //  goto nodata;
+      memset(skb, 0, offsetof(struct sk_buff, truesize));
+      skb->truesize = size + 16 + sizeof(struct sk_buff);
+      atomic_set(&skb->users, 2);
+      skb->head = data;
+      skb->data = data;
+      skb->tail = data;
+      skb->end = data + size;
+      atomic_set(&(skb_shinfo(skb)->dataref), 1);
+      skb_shinfo(skb)->nr_frags = 0;
+      skb_shinfo(skb)->tso_size = 0;
+      skb_shinfo(skb)->tso_segs = 0;
+      skb_shinfo(skb)->frag_list = NULL;
+      
+      return;
+    }

+   **
+   * e1000_alloc_rx_buffers - Replace used receive buffers; legacy & extended
+   * @adapter: address of board private structure
+   ************** e1000_alloc_rx_buffers(struct e1000_adapter* adapter)
+++ 3115,3122 ****
--- 3187,3201 ----
+    while(!buffer_info->skb) {
+      skb = dev_alloc_skb(bufsz);
+      ifunlikely(!skb)) {
+        /* Better luck next round */
+        break;
+      }
+    }
+    buffer_info = &rx_ring->buffer_info[i];
+    while(!buffer_info->skb) {
+      skb = dev_alloc_skb(bufsz);
+      cedric
+      // use our buffers
```
! skb = adapter->prealloc_buff[i];
! // reset skb
! reset_skb(skb, adapter, i);
! // fool kfree (reaches 3)
! //skb = skb_get(skb);

if(unlikely(!skb)) {
    /* Better luck next round */

EOF