Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

pktgen the linux packet generator

Robert Olsson
Uppsala Universitet & SLU
robert.olsson@its.uu.se

Abstract

pktgen is a high-performance testing tool included in the Linux kernel. pktgen is currently the best tool to test the TX process of device driver and NIC. pktgen can also be used to generate ordinary packets to test other network devices. Especially of interest is the use of pktgen to test routers or bridges which often also use the Linux network stack. Because pktgen is "in-kernel," it can generate high bandwith and very high packet rates to load routers, bridges, or other network devices.

1 Introduction

This paper describes the novel rework of pktgen in Linux 2.6.11. Much of the rework has been focused on multi-threading and SMP support. The main goal is to have one pktgen thread per CPU which can then drive one or more NICs. An in-kernel pseudo driver offers unique possibilities in performance and capabilities. The trade-off is additional responsibility in terms of robustness and avoiding kernel bloat (vs user mode application).

Pktgen is not an all-in-one testing tool. It offers a very efficient direct access to the host system NIC driver/chip TX-process and bypasses most of the Linux networking stack. Because of this, use of pktgen requires root access. The packet stream generated by pktgen can be used as input to other network devices. Pktgen also exercises other subsystems such as packet memory allocators and I/O buses. The author has done tests sending packets from memory to several GIGE interfaces on different PCI-buses using several CPU's. Aggregate Rates > 10 GBit/s have been seen.

1.1 Other testing tools

There are lots of good testing tools for network and TCP testing. netperf and ttcp are probably among the most widespread. Pktgen is not a substitute for those tools but complements for some types of tests. The test possibilities is described later in this paper. Most importantly, pktgen cannot do any TCP testing.

2 Pktgen performance

Performance varies of course with hardware and type of test. Some examples follow. A single flow of 1.48 Mpps is seen with a XEON 2.67 GHz using a patched e1000 driver (64 byte packets). High numbers are also reported with bcm5703 with tg3 driver. Aggregated performance of >10 Gbit/s (1500 byte packets) comes from using 12 GIGE NIC's and DUAL XEON

2.67 MHz with hyperthreading enabled (motherboard has 4 independent PCI-X buses). Similarly, DUAL 1.6ăGHz Opterons can generate 2.4 Mpps (64 byte packets). Tests involving lots of alloc's results in lower sending performance (see `clone_skb()`).

Many other things also affect performance: PCI bus speed, PCI vs PCI-X, PCI-PCI Bridge, CPU speed, memory latency, DMA latency, number of MMIO reads/writes per packet or per interrupt, etc.

Figure 1 compares performance of Intel's DUAL Port NIC (2 x 82546EB) with Intel's QUAD NIC (4 x 82546EB; Secondary PCI-X Bus runs at 120 Mhz). on a Dual Opteron 242 (Linux 2.6.7 32-bit).

The graph shows a faster I/O bus gives higher performance as this probably lowers DMA latency. The effects of the PCI-X bridge are also evident as the bridge is the difference between the DUAL and QUAD boards.

It's interesting to note that even bus bandwidth is much faster than 1 Gbit/s it degrades the small packet performance as seen from the experiment. 133 MHz would theoretically correspond to 8.5 Gbit/s. The patched version of e1000 driver adds data prefetching and skb refill at `hard_xmit()`.

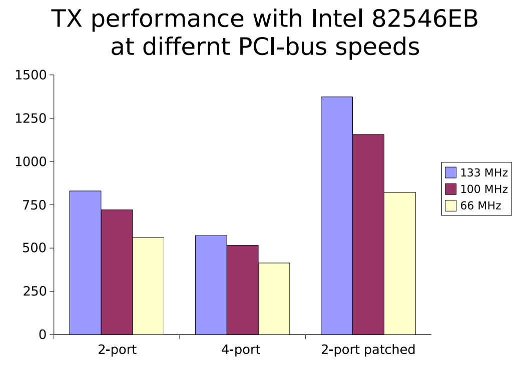

Figure 1: PCI Bus Topology vs TX perf

3 Getting pktgen to run

Enable `CONFIG_NET_PKTGEN` in the .config, compile and build pktgen.o either in-kernel or as module, insmod pktgen if needed. Once running, pktgen creates a kernel thread and binds thread to that CPU. One can the register a device to exactly one of those threads.This to give full control of the device to CPU relationship. Modern platforms allow interrupts to be assigned to a CPU (aka IRQ affinity) and this is necessary to minimize cache-line bouncing.

Generally, we want the same CPU that generates the packets to also take the interrupts given a symmetrical configuration (CPU:NIC is 1:1).

On a dual system we see two pktgen threads: [pktgen/0], [pktgen/1]

pktgen is controlled and monitored via the /proc file system. To help document a test configuration and parameters, shell scripts are recommended to setup and start a test. Again referring to our dual system, at start up the files below are created in ă `/proc/net/pktgen/` kpktgend_0, kpktgend_1, pgctrl

Assigning devices (e.g. eth1, eth2) to kpktgend_X thread, makes new instances of the devices show up in `/proc/net/pktgen/` to be further configured at the device level.

A test can be configured to run forever or terminate after a fixed number of packets. Ctrl-C aborts the run.

pktgen sends UDP packets to port 9 (discard port) by default. IP, MAC addresses, etc. can be configured. Pktgen packets can hence be identified within the kernel network stack for profiling and testing.

4 Pktgen versioninfo

The pktgen version is printed in dmesg when pktgen starts. Version info is also in `/proc/net/pktgen/pgctrl`.

5 Interrupt affinity

When adding a device to a specific pktgen thread, one should also set `/proc/irq/X/smp_affinity` to bind the NIC to the same CPU. This reduces cache line bouncing in several areas: when freeing skb's and in the NIC driver. The `clone_skb` parameter can in some cases mitigate the effect of cache line bouncing as skb's are not fully freed. One must experiment a bit to achieve maximum performance.

The irq numbers assigned to particular NICs can be seen in `/proc/interrupts`. In the example below, eth0 uses irq 26, eth1 uses irq 27 etc.

```
26: 933931      0 IO-APIC-level eth0
27: 936392      0 IO-APIC-level eth1
28:      8 936457 IO-APIC-level eth2
29:      8 939310 IO-APIC-level eth3
```

The example below assigns eth0, eth1 to CPU0, and eth2, eth3 to CPU1:

```
echo 1 > /proc/irq/26/smp_affinity
echo 1 > /proc/irq/27/smp_affinity
echo 2 > /proc/irq/28/smp_affinity
echo 2 > /proc/irq/29/smp_affinity
```

The graph below illustrates the performance effects of affinity assignment of PII system.

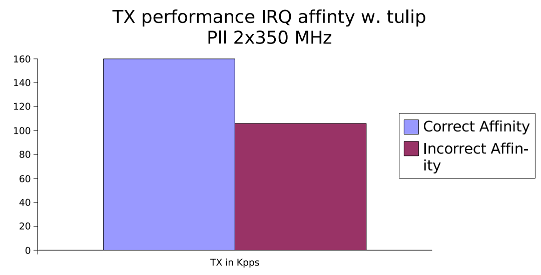

Figure 2: Effects of irq affinity

5.1 clone_skb: limiting memory allocation

pktgen uses a trick to increment the skb's refcnt to avoid full path of kfree and alloc when sending identical skb's. This generally gives very high sending rates. For Denial of Service (DoS) and flow tests this technique can not be used as each skb has to be modified.

The parameter `clone_skb` controls this functionality. Think of `clone_skb` as the number of packet clones followed by a master packet. Setting `clone_skb=0` gives no clones, just master packets, and `clone_skb=1000000` givs 1 master packet followed by one million clones.

`clone_skb` does not test normal use of a NIC. While the kfree and alloc are avoided by using `clone_skb`, one also avoids sending packets from dirty cachelines. The clean cache can contribute as much as 20% in performance as shown in Table 1.

Data in Table 1 was collected on HP rx2600-Itanium2 with BCM5703 (PCI-X) NIC running 2.6.11 kernel. The difference in performance between columns (RC on vs. off) shows how much dirty cache can affect DMA. Numbers are in packets per second. Read Current (RC) is a Mckinley bus transaction that allows the CPU to respond to a cacheline request directly from cache and retain ownership of the dirty cacheline. I.e., the cacheline can stay dirty-private

clone_skb	RC on	RC off	% Drop
on	947315	913768	–3.54%
off	630736	506711	–19.66%

Table 1: clone_skb and cache effects (pps)

and the CPU can write the same cacheline again without having to acquire ownership first.

It's likely cache effects contribute to the difference in performance between rows too (with and without `clone_skb`). But it's just as likely `clone_skb` reduces the CPU's use of memory bus bandwidth and thus contends less with DMA. This data is contributed by Grant Grundler.

5.2 Delay: Decreasing sending rate

pktgen can insert an extra artificial delay between packets. The unit is specified in nanoseconds. For small delays, pktgen busywaits before putting this skb on the TX-ring. This means traffic is still bursty and somewhat hard to control. Experimentation is probably needed.

6 Setup examples

Below a very simple example of pktgen sending on eth0. One only needs to bring up the link.

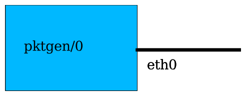

Figure 3: Just send/Link up

pktgen can send if the device is UP but many derives also requires that link is up can be done using a crossover cable connected to another NIC in the same box. If generated packets should be seen (i.e. Received) by the same host, set dstmac to match the NIC on the cross over cable as shown in Figure 4. Using a "fake" dstmac value (e.g. 0) will cause the other NIC to just ignore the packets.

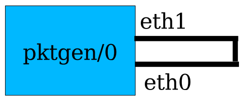

Figure 4: RX/TX in one Host

On SMP systems, it's better if the TX flow (pktgen thread) is on a different CPU from the RX flow (set IRQ affinity). One way to test Full Duplex functionality is to connect two hosts and point the TX flows to each other's NIC.

Next, the box with pktgen is used just a packet source to inject packets into a local or remote system. Note you need to configure dstmac of localhost or gateway appropriate.

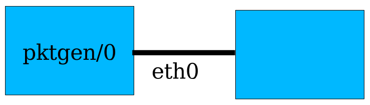

Figure 5: Send to other

Below pktgen in a forwarding setup. The sink host receives and discards packets. Of course, forwarding has to be configured on all boxes. It might be possible to use a dummy device instead of sink box.

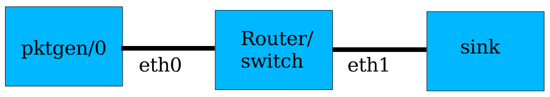

Figure 6: Forwarding setup

Forwarding setup using dual devices. Pktgen can use different threads to achieve high load in terms of small packets or concurrent flows.

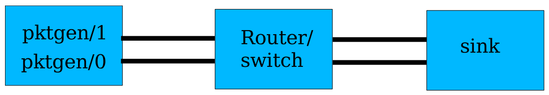

Figure 7: Parallel Forwarding setup

7 Viewing pktgen threads

Thread information as which devices are handled by this thread as actual status for each device is seen. `max_before_softirq` is used to avoid pktgen to avoid pktgen monopolize kernel resources. This will probably be removed as this of less problem with the threaded design. Result: is the "return" code "from the last `/proc` write.

/proc/net/pktgen/kpktgend_0

```
Name: kpktgend_0
max_before_softirq: 10000
Running:
Stopped: eth1
Result: OK: max_before_softirq=10000
```

7.1 Viewing pktgen devices

'Parm' sections holds configured info. 'Current' holds running stats. Result is printed after run or after interruption for example: See Appendix.

8 Configuring

Configuring is done via the /proc interface this is easiest done via scripts. Select a suitable script and customize. This paper includes one full example in Section 8. Additional example scripts are available from:

..1-1	# 1 CPU 1 dev
..1-2	# 1 CPU 2 dev
..2-1	# 2 CPU's 1 dev
..2-2	# 2 CPU's 2 dev
..1-1-rdos	# 1 CPU 1 dev route DoS
..1-1-ip6	# 1 CPU 1 dev ipv6
..1-1-ip6-rdos	# 1 CPU 1 dev ipv6 route DoS
..1-1-flows	# 1 CPU 1 dev multiple flows

Table 2: Script Filename Extensions

`ftp://robur.slu.se/pub/Linux/net-development/pktgen-testing/examples/`

Additional examples have been contributed by Grant Grundler `<grundler@parisc-linux.org>` `ftp://gsyprf10.external.hp.com/pub/pktgen-testing/`

See Appendix A for a quick-reference guide for currently implemented commands. It's divided into three parts: Pgcontrol, Threads, and Device. Each part has corresponding files in the `/proc` file system.

A collection of small tutorial scripts for pktgen are in examples dir. The file name extension is described in Table reffilename-ext.

Run in shell: ./pktgen.conf-X-Y

It does all the setup and then starts/stops TX thread. The scripts will need to be adjusted based on which NICs one wishes to test.

8.1 Configuration examples

Below is concentrated anatomy of the example scripts. This should be easy to follow.

pktgen.conf-1-2 A script fragment assigning eth1, eth2 to CPU on single CPU system.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"
pgset "add_device eth2"
```

pktgen.conf-2-2 A script fragment assigning eth1 to CPU0 respectivly eth2 to CPU1.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
pgset "add_device eth2"
```

pktgen.conf-2-1 A script fragment assigning eth1 and eth2 to CPU0 on a dual CPU system.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"
pgset "add_device eth2"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
```

pktgen.conf-1-2 A script fragment assigning eth1, eth2 to CPU on single CPU system.

```
PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth1"
pgset "add_device eth2"
```

pktgen.conf-1-1-rdos A script fragment for route DoS testing. Note clone_skb 0

```
PGDEV=/proc/net/pktgen/eth1
pgset "clone_skb 0"
# Random address with in the
# min-max range
# pgset "flag IPDST_RND"
pgset "dst_min 10.0.0.0"
pgset "dst_max 10.255.255.255"
```

pktgen.conf-1-1-ipv6 Setting device ipv6 addresses.

```
PGDEV=/proc/net/pktgen/eth1
pgset "dst6 fec0::1"
pgset "src6 fec0::2"
```

pktgen.conf-1-1-ipv6-rdos

```
PGDEV=/proc/net/pktgen/eth1
pgset "clone_skb 0"
# pgset "flag IPDST_RND"
pgset "dst6_min fec0::1"
pgset "dst6_max fec0::FFFF:FFFF"
```

pktgen.conf-1-1-flows A script fragment for route flow testing. Note clone_skb 0

```
PGDEV=/proc/net/pktgen/eth1
pgset "clone_skb 0"
# Random address within the
# min-max range
# pgset "flag IPDST_RND"
pgset "dst_min 10.0.0.0"
pgset "dst_max 10.255.255.255"
# 8k Concurrent flows at 4 pkts
pgset "flows 8192"
pgset "flowlen 4"
```

2x4+2 script

```
#Script contributed by Grant Grundler
# <grundler@parisc-linux.org>
# Note! 10 devices

PGDEV=/proc/net/pktgen/kpktgend_0
pgset "rem_device_all"
pgset "add_device eth3"
pgset "add_device eth5"
pgset "add_device eth7"
pgset "add_device eth9"
pgset "add_device eth11"
pgset "max_before_softirq 10000"

PGDEV=/proc/net/pktgen/kpktgend_1
pgset "rem_device_all"
pgset "add_device eth2"
pgset "add_device eth4"
pgset "add_device eth6"
pgset "add_device eth8"
pgset "add_device eth10"
pgset "max_before_softirq 10000"

# Configure the individual devices

for i in 2 3 4 5 6 7 8 9 10 11
do
   PGDEV=/proc/net/pktgen/eth$i
   echo "Configuring $PGDEV"

   pgset "clone_skb 500000"
   pgset "min_pkt_size 60"
   pgset "max_pkt_size 60"
   pgset "dst 192.168.3.10$i"
   pgset "dst_mac 01:02:03:04:05:0$i"
   pgset "count 0"
done
echo "Running... CTRL-C to stop"
PGDEV=/proc/net/pktgen/pgctrl
pgset "start"

tail -2 /proc/net/pktgen/eth*
```

9 Tips for driver/chip testing

When testing a particular driver/chip/platform, start with TX. Use pktgen on the host system to get a sense of which ptkgen parameters are optimal and how well a particular NIC can perform TX. Try with a range of packet sizes from 64 bytes to 1500 bytes or jumbo frames.

Then start looking at the RX on the target platform by using pktgen to inject packets either direct via crossover cable or via pktgen from another host.

Again, vary the packet size etc To isolate driver/chip from other parts of kernel stack pktgen packets can be counted and dropped at various points. See section on detecting pktgen packets.

Depending on the purpose of the test repeat the process with additional devices, one at a time. Multiple devices are trickier since one needs to know I/O bus topology. Typically one tries to balance I/O loads by installing the NICs in the "right" slots or utilizing built-in devices appropriately.

9.1 Multiple Devices

With multiple devices, it is best to use CTRL-C to stop a test run. This prevents any pktgen thread from stopping before others and skewing the test results. Sometimes, one NIC will TX packets faster than another NIC just because of bias in the DMA latency or PCI bus arbiter (to name only two of several possibilities). Using CTRL-C to stop a test run aborts all pktgen threads at once. This results in a clean snapshot of how many packets a given configuration could generate over the same period of time. After the CTRL-C is received, pktgen will print the statistics the same as if the test had been stopped by a counter going to zero.

9.2 Other testing aspects

To isolate driver/chip from other parts of kernel stack, pktgen packets can be counted and dropped at various points. See Section 9.3 on detecting pktgen packets.

If the tested system only has one interface, the dummy interface can be setup as the output device. The advantage is we can test the system at very high load and the results are very reproduceable. Of course, other variables such as different types of offload and checksumming should be tested as well.

Besides knowing the hardware topology, one should know what workloads are expected to be present on the target system when placed in production (i.e. real world use). An FTP server can see quite a different workload than a web server, mail handler, or router, etc. Roughly 160 Kpps seems to fill a Gigabit link when running an FTP server. While this can vary, it gives an useful estimate of required packet per second (pps) versus bandwidth for this type of production system.

For routers the number of routes in the routing table is also an issue as lookup times and other behaviour may be affected. The author has taken snapshots from current Internet routing table IPV4 and IPV6 (BGP) and formed into scripts for this purpose. The routes are added via the ip utility so the tested system does not need any routing connectivity nor routing daemon. Some scripts are available from:

`ftp://robur.slu.se/pub/Linux/net-development/inet_routes/`

At last use your fantasy when testing, elaborate with new setups try to understand how things are functioning, monitor interested and related variables add printouts etc. Testing understanding and development are closely related.

9.3 Detecting pktgen packets in kernel

Sometimes it's very useful to monitor/drop pktgen packets within the driver/network stack either at ingress or egress. The technique for both is essentially the same. The patchlet in Section 13.1 drops pktgen packets at ingress and uses an unused counter.

Also it should be possible to capture pktgen packets via the tc command and the u32 classifier which might be a better solution in most cases.

10 Thanks to…

Thanks to Grant Grundler, Jamal Hadi Salim, Jens Låås, and Hans Wassen for comments and useful insights. This paper covers several years of work and conversations with all of the above.

Relevant site:
`ftp://robur.slu.se://pub/Linux/net-development/pktgen-testing/`

Good luck with the linux net-development!

11 Appendix A

Table 3: Command Summary

Commands	
Pgcontrol commands	
start	Starts sending on all threads
stop	
Threads commands	
add_device	Add a device to thread i.e eth0
rem_device_all	Removes all devices from this thread
max_before_softirq	do_softirq() after sending a number of packets
Device commands	
debug	
clone_skb	Number of identical copies of the same packet 0 means alloc for each skb. For DoS etc we must alloc new skb's.
clear_counters	normally handled automatically
pkt_size	Link packet size minus CRC (4)
min_pkt_size	Range pkt_size setting If < max_pkt_size, then cycle through the port range.
max_pkt_size	
frags	Number of fragments for a packet
count	Number of packets to send. Use zero for continious sending
delay	Artificial gap inserted between packets in nanoseconds
dst	IP destination address i.e 10.0.0.1
dst_min	Same as dst If < dst_max, then cycle through the port range.
dst_max	Maximum destination IP. i.e 10.0.0..1
src_min	Minimum (or only) source IP. i.e. 10.0.0.254 If < src_max, then cycle through the port range.
src_max	Maximum source IP.
dst6	IPV6 destination address i.e fec0::1
src6	IPV6 source address i.e fec0::2
dstmac	MAC destination adress 00:00:00:00:00:00
srcmac	MAC source adress. If omitted it's automatically taken from source device
src_mac_count	Number of MACs we'll range through. Minimum' MAC is what you set with srcmac.
dst_mac_count	Number of MACs we'll range through. Minimum' MAC is what you set with dstmac.
Flags	
IPSRC_RND IPDST_RND TXSIZE_RND UDPSRC_RND	IP Source is random (between min/max), Etc

	Commands continued
UDPDST_RND MACSRC_RND MACDST_RND	
udp_src_min	UDP source port min, If < udp_src_max, then cycle through the port range.
udp_src_max	UDP source port max.
udp_dst_min	UDP destination port min, If < udp_dst_max, then cycle through the port range.
udp_dst_max	UDP destination port max.
stop	Aborts packet injection. Ctrl-C also aborts generator. **Note**: Use count 0 (forever) and stop the run with Ctrl-C when multiple devices are assigned to one pktgen thread. This avoids some devices finishing before others and skewing the results. We are primarily interested in how many packets all devices can send at the same time, not absolute number of packets each NIC sent.
flows	Number of concurrent flows
flowlen	Length of a flow

12 Appendix B

12.1 Sample pktgen output

/proc/net/pktgen/eth1 output after run

```
Params: count 10000000  min_pkt_size: 60  max_pkt_size: 60
 frags: 0  delay: 0  clone_skb: 1000000  ifname: eth1
 flows: 0 flowlen: 0
 dst_min: 10.10.11.2  dst_max:
 src_min:   src_max:
 src_mac: 00:00:00:00:00:00  dst_mac: 00:07:E9:13:5C:3E
 udp_src_min: 9  udp_src_max: 9  udp_dst_min: 9  udp_dst_max: 9
 src_mac_count: 0  dst_mac_count: 0
 Flags:
Current:
 pkts-sofar: 10000000  errors: 39192
 started: 1076616572728240us  stopped: 1076616585502839us idle: 1037781us
 seq_num: 11  cur_dst_mac_offset: 0  cur_src_mac_offset: 0
 cur_saddr: 0x10a0a0a  cur_daddr: 0x20b0a0a
 cur_udp_dst: 9  cur_udp_src: 9
 flows: 0
Result: OK: 12774599(c11736818+d1037781) usec, 10000000 (64byte)
 782840pps 382Mb/sec (400814080bps)  errors: 39192


Results show 10 millon 64 byte packets were sent on eth1 to 10.10.11.2
with a rate at 783 kpps
```

```
\section{Appendix C}
\subsection{pktgen.conf-1-1 script}

Below is the full pktgen.conf-1-1 script

\begin{footnotesize}
\begin{verbatim}
#!/bin/sh

#modprobe pktgen

function pgset() {
  local result

  echo $1 > $PGDEV

  result=‘cat $PGDEV | fgrep "Result: OK:"‘
  if [ "$result" = "" ]; then
       cat $PGDEV | fgrep Result:
  fi
}

function pg() {
    echo inject > $PGDEV
    cat $PGDEV
}

# Config Start Here -------------------------------------

# thread config
# Each CPU has own thread. Two CPU exammple.
# We add eth1, eth2 respectively.

PGDEV=/proc/net/pktgen/kpktgend_0
  echo "Removing all devices"
 pgset "rem_device_all"
  echo "Adding eth1"
 pgset "add_device eth1"
  echo "Setting max_before_softirq 10000"
 pgset "max_before_softirq 10000"

# device config
# delay is inter packet gap. 0 means maximum speed.

CLONE_SKB="clone_skb 1000000"
# NIC adds 4 bytes CRC
PKT_SIZE="pkt_size 60"

# COUNT 0 means forever
#COUNT="count 0"
COUNT="count 10000000"
delay="delay 0"
```

```
PGDEV=/proc/net/pktgen/eth1
  echo "Configuring $PGDEV"
 pgset "$COUNT"
 pgset "$CLONE_SKB"
 pgset "$PKT_SIZE"
 pgset "$delay"
 pgset "dst 10.10.11.2"
 pgset "dst_mac  00:04:23:08:91:dc"

# Time to run
PGDEV=/proc/net/pktgen/pgctrl

 echo "Running... ctrl^C to stop"
 pgset "start"
 echo "Done"

# Result can be vieved in /proc/net/pktgen/eth1
```

13 Appendix D

13.1 Patchlet to ip_input.c

Below is the patchlet to count and drop pktgen packets.

```
--- linux/net/ipv4/ip_input.c.orig      Mon Feb 10 19:37:57 2003
+++ linux/net/ipv4/ip_input.c   Fri Feb 21 21:42:45 2003
@@ -372,6 +372,23 @@
            IP_INC_STATS_BH(IpInDiscards);
            goto out;
    }

+           {
+                 __u8 *data = (__u8 *) skb->data+20;
+
+                 /* src and dst port 9 --> pktgen */
+
+                 if(data[0] == 0 &&
+                    data[1] == 9 &&
+                    data[2] == 0 &&
+                    data[3] == 9) {
+                        netdev_rx_stat[smp_processor_id()].fastroute_hit+
+;
+                         goto drop;
+                 }
+           }
+
    if (!pskb_may_pull(skb, sizeof(struct iphdr)))
            goto inhdr_error;
```