Implementation and Experimental Performance Evaluation of a Hybrid Interrupt-Handling Scheme

K. Salah[**] A. Qahtan
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
Email: {salah,kahtani}@kfupm.edu.sa

Abstract

The performance of network hosts can be severely degraded when subjected to heavy traffic of today's Gigabit networks. This degradation occurs as a result of the interrupt overhead associated with the high rate of packet arrivals. NAPI, a packet reception mechanism integrated into the latest version of Linux networking subsystem, was designed to improve Linux performance to suit today's Gigabit traffic. NAPI is definitely a major step up from earlier reception mechanisms; however, NAPI has shortcomings and its performance can be further enhanced. A hybrid interrupt-handling scheme, which was recently proposed in [3], can better improve the performance of Gigabit network hosts. The hybrid scheme switches between interrupt disabling-enabling (DE) and polling (NAPI). In this paper, we present and discuss major changes required to implement such a hybrid scheme in the latest version of Linux kernel 2.6.15. We prove experimentally that the hybrid scheme can significantly improve the performance of general-purpose network desktops or servers running network I/O-bound applications, when subjecting such network hosts to both light and heavy traffic load conditions. The performance is measured and analyzed in terms of throughput, packet loss, latency, and CPU availability.

KEYWORDS: High-Speed Networks, Operating Systems, NAPI, Linux, Interrupts, Livelock

1. Introduction

1.1. Background

Using today's general-purpose Gigabit network adapters (also termed as NICs), an incoming packet gets transferred (or DMA'd) through the PCI bus from the NIC to a circular buffer in the kernel space known as DMA Rx Ring. After the packet has been successfully DMA'd, the NIC generates an interrupt to notify the kernel to start protocol processing of the incoming packet. During protocol processing, other packets may arrive and get queued into the DMA Rx Ring. Protocol processing typically involves TCP/IP processing of the incoming packet and delivering it to user applications. The packet does not need to be delivered to user applications if the receiving host is configured for IP forwarding, routing, filtering or NATing.

[**] Corresponding Author: Prof. K. Salah, PO Box 5066, ICS Department, KFUPM, Dhahran 31261, Saudi Arabia

The performance of network hosts can be significantly degraded when subjected to heavy traffic load such as that of Gigabit networks, and thus resulting in poor host performance perceived by the user. This is because every incoming packet triggers a hardware interrupt, which involves a context switching of saving and restoring processor's state and also in a potential cache/TLB pollution. More importantly, interrupt-level handling, by definition, has an absolute priority over all other tasks. If the interrupt rate is high enough, the system will spend all of its time responding to interrupts, while nothing else would be performed. This will cause the system throughput to drop to zero. This situation is called *receive livelock* [1]. In this situation, the system is not deadlocked, but it makes no progress on any of its tasks, causing any task scheduled at a lower priority to starve or have no chance to run.

A number of schemes to mitigate interrupt overhead and resolve receive livelock exists in the literature. Among the most popular ones are normal interruption, interrupt disabling and enabling, interrupt coalescing, and polling. In normal interruption, every incoming packet causes an interrupt to trigger protocol processing by the kernel. Typically protocol processing is performed by a deferrable and reentrant high-priority kernel function (e.g., tasklet in Linux). The idea of interrupt disable-enable (a.k.a. DE) scheme [2,3] is to have the received interrupts of incoming packets turned off (or disabled) as long as there are packets to be processed by kernel's protocol stack, i.e., the protocol buffer is not empty. When the buffer is empty, the interrupts are turned on again (or re-enabled). This means that protocol processing of packets by the kernel is processed immediately and at interrupt priority level. Any incoming packets (while the interrupts are disabled) are DMA'd quietly to protocol buffer without incurring any interrupt overhead. With the scheme of interrupt coalescing (IC) [4], the NIC generates a single interrupt for a group of incoming packets. This is opposed to normal interruption mode in which the NIC generates an interrupt for every incoming packet. There are two IC types: count-based IC and time-based IC. In count-based IC, the NIC generates an interrupt when a predefined number of packets has been received. In time-based IC, the NIC waits a predefined time period before it generates an interrupt. Finally, the basic idea of polling is to disable interrupts of incoming packets altogether and thus eliminating interrupt overhead completely. In polling [1,5-8], the OS periodically polls its host system memory (i.e., protocol processing buffer or DMA Rx Ring) to find packets to process. In general, exhaustive polling is rarely implemented. Rather, polling with quota or budget is usually the case whereby only a maximum number of packets is processed in each poll in order to leave some CPU power for application processing.

In [3], we utilized both mathematical analysis and discrete-event simulation to study the performance of those most popular interrupt-handling schemes which included normal interruption, polling, interrupt disabling and enabling, and interrupt coalescing. For polling, we studied both pure (or FreeBSD-style) polling and Linux

NAPI polling. The performance was studied in terms of key performance indictors which included throughput, system latency, and CPU availability (i.e., the residual fraction of CPU bandwidth left for user applications). Based on the study carried out in [3], it was concluded that no particular interrupt handling scheme gives the best performance under all load conditions. Under light and heavy traffic loads, it was shown that the scheme of disabling and enabling interrupts (DE) outperforms, in general, all other schemes in terms of throughput and latency. However, when it comes to CPU availability, polling is the most appropriate scheme to use, particularly at heavy traffic load. Based on these key observations and in order to compensate for the disadvantages of DE scheme of poor CPU availability, we proposed in [3] a hybrid scheme that combines both DE and polling. Such a hybrid scheme would be able to attain peak performance under both light and heavy traffic loads by employing DE at light load and polling at heavy load.

In [11], we presented preliminary experimental results to show that the performance of I/O bound applications can be enhanced when using the Hybrid scheme. In this paper we considerably extend the work presented in [11] and provide in-depth details of implementing the Hybrid scheme in the latest Linux version 2.6.15. The paper also shows how to measure experimentally the performance of Hybrid, DE, and NAPI. In addition, the paper addresses important implementation issues which include identifying the switching point between DE and NAPI, real-time measurement of incoming traffic rate, and the selection of proper NAPI budget or quota size during the polling period. The performance of the Hybrid scheme is evaluated in terms of throughput, latency, CPU availability, packet loss, and interrupt rate.

The rest of the paper is organized as follows. Section 2 gives a brief background and related work on the hybrid scheme. It discusses how different our hybrid scheme from those proposed in the literature. Section 3 details the inner-workings of Linux NAPI. Section 4 presents major changes required by network device driver to implement DE and Hybrid in latest version of Linux 2.6.15. Also the section addresses important issues as operation overhead, adjusting NAPI budget, identifying the cliff point, and switching mechanism. Section 5 describes experimental setup, then Section 6 presents performance measurements. Finally, Section 7 concludes the study and identifies future work.

2. Hybrid Scheme

A hybrid scheme of normal interruption and polling was first proposed in [2]. Later, it was implemented and utilized in [5,8,9,10]. In this hybrid scheme, normal interrupt was used under both light and normal network traffic load, whereas polling was used under heavy network traffic load. In sharp contrast, our hybrid scheme (which we initially proposed in [3]) differs from previously proposed scheme in three significant ways: (1) Under light and normal loads, our hybrid scheme utilizes the scheme of DE as opposed to normal interruption which was used in [2,5,8-10]. (2) Our hybrid scheme switches between DE and NAPI based on the estimated incoming traffic rate. (3) The switching point is identified *experimentally,* rather than arbitrarily.

It was demonstrated in [3] that normal interruption performs relatively poorly under light and normal traffic loads in terms of system throughput, CPU availability, and latency. This was due to the fact that normal interruption introduces interrupt overhead for each packet arrival and thereby leaving limited CPU power for IP processing and user applications. On the other hand, DE gave acceptable performance in terms of system throughput, CPU availability, and latency under low and normal traffic loads.

To identify the severity of traffic load conditions, and as opposed to other hybrid schemes proposed in [2,5,8-10], our hybrid scheme switches between DE and NAPI based on the estimated incoming packet arrival rate. In particular, our hybrid scheme estimates the traffic rate periodically and uses two thresholds for switching. As will be discussed in Section 4.2, two thresholds are used to minimize repeated switchings (or oscillation) around the saturation point in the presence of a traffic load that is highly fluctuating at the point of saturation. Moreover, the saturation point was not identified arbitrary but using before-hand experimentation discussed in Section 4.2. On the other hand, different ways to identify severity of traffic load conditions as well as the switching point were used in [2,5,8-10].

In [2,9,10], the utilization (or level of occupancy) of application or socket receive buffers was used. A receive buffer that has a utilization of 5% indicates low traffic load, while a buffer that has a utilization of 70% indicates high utilization. Switching based on buffer utilization has major drawbacks. First, a host system has multiple buffers of interest that could potentially be used to indicate traffic overload conditions, and therefore making it impractical to implement a comprehensive solution. For example, for IP forwarding hosts, socket or protocol processing buffer can be used. In application hosts such as web servers, user application buffer is more appropriate. In particular, the protocol receiver buffer was used in [2], while in [9,10], the application receive buffer was used. Secondly, the buffer utilization can be high for a number of reasons other than traffic load, e.g. starvation or bursty traffic. Starvation of a consumer process (i.e., user application or protocol processing) is a possibility as the CPU power is being allocated for other highly important tasks or processing. In addition, instantaneous traffic burst can fill up the buffer very quickly and thus cause sharp increase in buffer utilization. Thirdly, selection of utilization levels or thresholds is somewhat and always done arbitrarily. According to [2], determining the upper and lower level of occupancy thresholds is arbitrary and in reality not a trivial task.

In [8], a watchdog timer is used for switching between normal interruption and polling. Upon a packet arrival a predefined watchdog time is started. If a packet is not removed within this predefined time through polling, an interrupt is generated. In today's network adapters or host hardware, such a feature of monitoring a packet consumption is not supported. A special hardware support has to be introduced to generate an interrupt if a packet is not removed from the DMA Rx Ring within the predefined amount of time. In addition, determining a proper value for the watchdog timer is not trivial, and often defined arbitrarily. A Small value will resort to normal interruption while a large value will resort to polling.

In [5], the rate of incoming traffic is periodically estimated to switch between normal interruption and polling. The estimation is based on the packet interarrival times. Arbitrary thresholds were used for determining the level of frequency and predictability of packets. Simulation was used to study the performance of such a scheme. In reality, such a scheme is not practical and not implemented in today's network adapters. Estimation of packet arrival rate based on interarrival times is computationally expensive and requires hardware support by network adapters. Network adapters would require to timestamp each packet when received. Estimation of the average packet arrival rate can be done by the adapter or the kernel. Clock synchronization may be required between the network adapters and the kernel if estimation is to be carried out by the kernel.

In contrast with [5], our hybrid scheme is more practical. It is computationally inexpensive, and requires no hardware support. Our hybrid scheme uses a simple estimator of packet arrival rate [12]. The estimator is highly cost-effective as its computation is not carried out upon the arrival of each incoming packet and does not require measurement of interrarival times of incoming packets. The estimator makes use of the number of packets received within a predefined time window. Almost all today's network adapters provide a total count of received packets. In [12], the quality and performance of the estimator was evaluated *experimentally* and shown to be highly effective in terms of computational cost, accuracy, agility, and stability.

3. Linux NAPI

New API (NAPI) [6] is a packet reception mechanism which is implemented in Linux 2.6 to alleviate the problem of receive livelock. Figure 1 illustrates the algorithm of how NAPI works. Upon the arrival of a packet, NAPI disables the received interrupt (RxInt) of the network device to stop the generation of further interrupts and then raises a softirq of NET_RX_SOFTIRQ type to schedule polling. Softirq is a non-urgent (or deferrable) interruptible kernel event that gets handled by *__do_softirq* kernel function. *__do_softirq* gets invoked (if there are any pending softirqs) typically at the end of I/O and timer interrupts [13]. Other than NET_RX_SOFTIRQ, a total of six softirq types (0-5) are currently defined in Linux 2.6. During the execution of *__do_softirq*, softirq types of lower values (such as HI_SOFTIRQ, TIMER_SOFTIRQ, and NET_TX_SOFTIRQ) are processed before NET_RX_SOFTIRQ which has a type of 3 [13]. This means that the beginning of the polling period in Linux is not deterministic and a non-uniform delay can be encountered before the actual processing of packets.

The actual polling and processing of packets take place during the execution of *net_rx_action* which is the handling function of NET_RX_SOFTIRQ softirq. Up to budget *B* packets are processed in *net_rx_action* for all network devices or interfaces and up to quota *Q* packets are processed per interface. This is done to ensure fairness of packet processing in case of a host having multiple network interfaces. *poll_list* is used to process packets per network interface in a round robin fashion. In Linux 2.6, the budget *B* and quota *Q* are set to 300 and 64, respectively.

RxInt:

```
1. Disable RxInt of this
   network device
2. Add device to poll list
3. Raise sofirq of type
   NET_RX_SOFTIRQ
```

net_rx_action:

```
Start timer T
Initialize budget B = 300, quota Q = 64
while poll_list is not empty do
  1. if B <= 0 or T expired then
         raise again softirq of type NET_RX_SOFTIRQ
         break
     end if
  2. Extract a device from poll_list
  3. For this device, process up to min(Q,B) packets or
     until no more packets in DMA Ring of device
  4. Decrement B by the number of packets processed
  5. if device's DMA Rx Ring is empty then
         remove device from poll_list and enable its RxInt
     else
         place device at the end of poll_list
     end if
end while
```

Figure 1. NAPI polling algorithm

NAPI's polling period in Linux is not exhaustive and is designed to leave CPU processing power for other tasks in order to prevent starvation. This is accomplished using budget and handling time bounds (as shown in Step 1 of *net_rx_action* algorithm). Handling time is configured to be at least one *jiffy*. Budget and time basically limit the number of packets that get processed per NET_RX_SOFTIRQ handling. If there are remaining packets in the DMA Rx Rings to be processed or handling time has expired, softirq of NET_RX_SOFTIQ type will be raised and therefore will be shown pending in *__do_softirq*. Pending softirqs will be processed by *__do_softirq* repeatedly up to a maximum of MAX_SOFTIRQ_RESTART (set to 10 by default). After reaching this maximum limit and if there are still pending softirqs, *__do_softirq* will no longer handle softirqs. Softirqs will be handed by *ksoftirqd* which is a lower priority kernel thread with a nice value of

19. This will prevent starvation of user-level processes which run with a default nice priority value of 0 (which is much higher priority than 19).

In order to prevent starvation of user processes under severely high traffic load, *ksoftirqd* will typically get awakened. When *ksforitqd* is awakened, it is to be noted that user processes will be given more CPU share than *ksoftriqd*, as *ksoftriqd* will be running with lower priority than user processes. User processes will preempt *ksoftirqd* before the basic quantum of *ksoftirqd* gets exhausted. In some situations, e.g. IP forwarding, this is not desirable as it will have a negative impact on the performance of kernel processing of received packets, and thus degrading the overall forwarding capacity. Therefore, adjusting the priority of user processes or *ksoftirqd* is required to achieve the desirable user-softirq balance.

In addition, the current default configurations of budget B and MAX_SOFTIRQ_RESTART may not be proper for user processes. With budget B and MAX_SOFTIRQ_RESTART set to 300 and 10, respectively, a total of 3000 packets can possibly get processed by the kernel per polling period. In moderate or low-end CPU hosts, this constitutes a major consumption of CPU cycles and thus may cause starvation to user processes at light and normal traffic loads, i.e., way before awakening *ksoftirqd*. To alleviate such a problem, budget B has to be adjusted to suit user processes. According to our experimental results (shown in Section 6 and Figure 7) for determining a proper budget to suit the execution of a network I/O process running at the user-level, we found that a budget of 2 or 6 will outperform the default configuration of 300 in terms of throughput and latency. Other hosts with more emphasis on underlying protocol processing by the kernel than user processes may require a different budget. For example a PC-based router would require a large budget. Thus, the best way to determine the proper budget is through experimentation, as will be demonstrated in Section 6.

4. Implementation

This section describes modifications and major changes to be introduced to network device drivers to integrate and implement both schemes of DE and Hybrid. With our implementation, the kernel code was not modified. The changes were only made to the driver code, which was compiled as a kernel loadable module. In this section, an adequate level of details is provided so that such changes can be easily integrated in different or newer versions of drivers. In particular we present and discuss major changes made to BCM5752 network drivers to support DE and Hybrid. We also discuss switching mechanism and how to experimentally identify the proper switching point.

4.1. Driver's Support

In our implementation, we used BCM5700 network driver [14]. BCM5700 driver was used as opposed to the default tg3 because BCM5700 source code, available from [14], was better commented and offered more features. One of these features is the support for both non-NAPI and NAPI. We found this feature attractive to experiment with at an early stage. A second feature is that BCM5700 offers code to disable RxInt individually whereas tg3 disables all NIC interrupts altogether. Such an option was very useful to properly implement NAPI. In order to understand the major changes needed to support DE and Hybrid, we first present the sequence of function calls that are already in place to support the default NAPI. Figure 2 presents the sequence of function calls for both interrupt and softirq handlings. Interrupt handling is part of the driver code, while softirq handling is part of the kernel code.

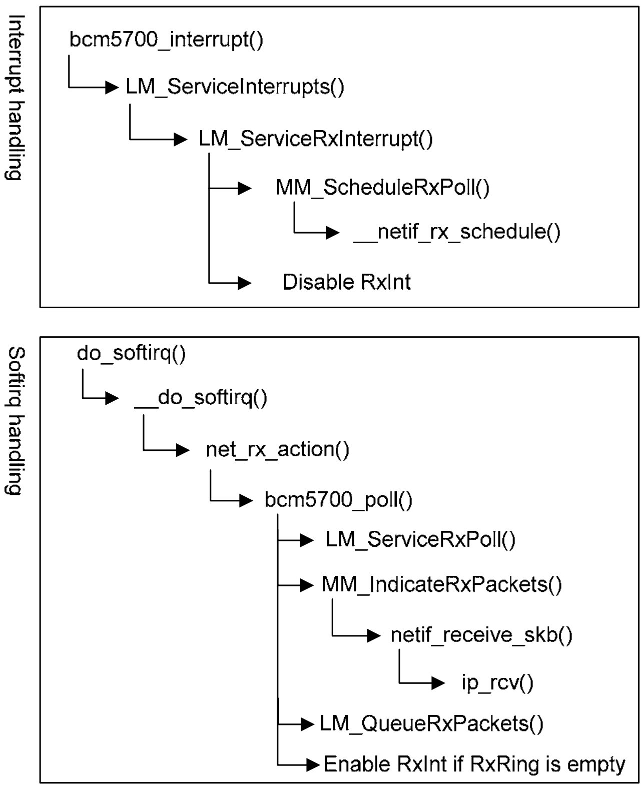

Figure 2. Sequence of driver and kernel functions calls to support NAPI

At the driver level, all NIC interrupts are handled by *bcm5700_interrupt()* function which calls *LM_ServiceInterrupts()* to read NIC's status registers to find the cause of interruption. If the interrupt is caused by the arrival of a new packet into the DMA Rx Ring, *LM_ServiceRxInterrupt()* is called to decide to process the packet using non-NAPI or NAPI. In case of NAPI, *MM_ScheduleRxPoll()* is called which then calls *__netif_rx_schedule()* which in turn raises a softirq of NET_RX_SOFTIRQ type to schedule polling. When control returns back to *LM_ServiceRxInterrupt(),* RxInt is disabled. It is to be noted that every NIC has a single interrupt line and thus all types of interrupts that belong to a particular NIC are shared and masked during the execution of *bcm5700_interrupt()* function. Therefore, it dos not matter if RxInt is disabled early on or later during the handling of *LM_ServiceRxInterrupt().*

At the kernel level, NET_RX_SOFTIRQ softirq is handled by *net_rx_action()* which gets called by *__do_softirq()* to process pending softirqs. *net_rx_action()* extracts a network device from *poll_list* and its respective *poll* virtual function. For the BCM5700 driver, *bcm5700_poll()* function is called which in turn calls *LM_ServiceRxPoll().* These two latter functions are part of the BCM5700 driver. In *LM_ServiceRxPoll(),* the descriptors of packets are dequeued from the DMA Rx Ring and inserted in a device queue *RxPacketReceivedQ* for further protocol processing. *LM_ServiceRxPoll()* terminates when the minimum of quota *Q* or budget B packets is processed or when the Ring becomes empty, as illustrated in the algorithm shown in Figure 1. *MM_IndicateRxPackets()* then gets called to dequeue each packet (using pointer manipulation) from *RxPacketReceivedQ* into *sk_buff* buffer and calls the kernel function *netif_receive_skb()* for further processing. *netif_receive_skb()* routes packets to its respective packet handler function for protocol processing. For IP processing, the kernel function *ip_rcv()* is called. Refilling of Rx DMA Ring is carried out by *LM_QueueRxPackets().* Typically refilling is not done on every poll, but rather when a group of 64 packets is consumed. Finally it is to be noted that RxInt gets re-enabled in *bcm5700_poll()* for a device whose respective Rx Ring became empty and was removed from *poll_list.*

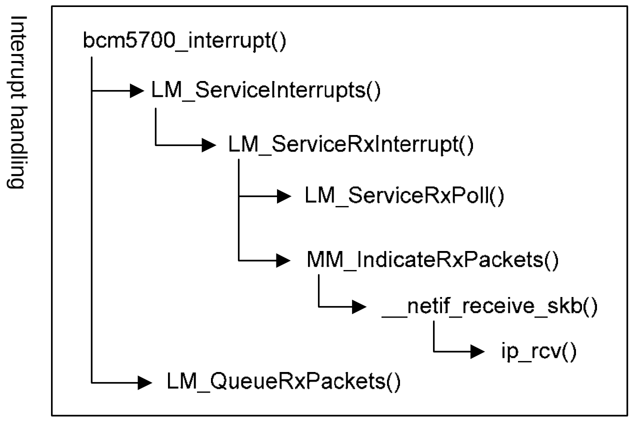

Figure 3. Sequence of driver and kernel functions calls to support DE

DE Implementation. To implement DE scheme, interrupt handling has to be changed as shown in Figure 3. The driver has to be modified so that protocol processing of all queued packets in the DMA Rx Ring is done during *LM_ServiceRxInterrupt().* Any incoming packets (while the interrupts are disabled) are DMAed quietly to Rx Ring without incurring any interrupt overhead. This means that processing of received packets by the kernel is started immediately and executed at interrupt priority level. And this way, deferring protocol processing of received packets using softirq is mitigated. This change is only done for RxInt, and all other interrupts including errors and TxInt are left untouched. It is also to be noted with this change packet reception is given more priority than packet transmission, as packet transmission remains deferred with softirq. Figure 3 shows briefly the sequence of function calls. *LM_ServiceRxInterrupt()* directly calls *LM_ServiceRxPoll()*. *LM_ServiceRxPoll()* terminates after *all* packets in Rx Ring are dequeued and inserted in *RxPacketReceivedQ* for further protocol processing. *LM_ServiceRxInterrupt()* then calls *MM_IndicateRxPackets()* to dequeue the packets into a *sk_buff* structure and calls *netif_receive_skb()* for further protocol processing. Lastly, and before returning from interrupt handling, *bcm5700_interrupt* checks the NIC status register and if more packets got DMA'd during packet processing, *LM_ServiceRxInterrupt* handling will be activated again. In BCM5700 driver, there is no need to specifically disable RxInt and then enable it again in interrupt handler *bcm5700_interrupt()*. The interrupt handler disables all types of device interrupts at beginning of handling and re-enables them all again just before retuning.

Overhead of DE vs. NAPI. With this implementation, it is critical to note that DE incurs far less overhead than NAPI. First, as opposed to DE in which processing of packets is executed immediately and at interrupt level, processing of packets in NAPI is deferred and executed at a lower priority using softirq. Second, all softirqs are reentrant, i.e., they run with interrupts enabled and therefore can be preempted at any time to handle a new incoming interrupt, therefore handling of a softirq may stretch due to other interrupt handling activities in the system. Third, a softirq may also stretch considerably due to processing of other softirqs, as *__do_softirq* does not handle only softirq of NET_RX_SOFTIRQ type for received packets, but also five other softirq types which include soft timers, high and low-priority tasklets, transmission of packets, and SCSI handling. Fourth, *net_rx_action* algorithm of NAPI is more computationally expensive than DE. NAPI incurs non-ignorable I/O write latencies to disable and enable RxInt [15], and also requires enforcing upper bounds for budget, quota, and handling time, besides the management of *poll_lis*t to provide fairness and avoid starvation. In contrast, DE simply performs exhaustive processing of received packets.

Hybrid Implementation. Hybrid scheme operates at any given time either as DE or NAPI. This requires maintaining the current state of operation: DE or NAPI. For this a global variable *RxScheme* is defined and assigned an enumerated value of DE or NAPI. Driver functions (specifically *bcm5700_interrupt*, *LM_ServiceRxInterrupt*, and *LM_ServiceRxPoll*) are instrumented to check *RxScheme* in order to properly operate in either DE or NAPI. It is to be noted when Hybrid operates as NAPI, its budget has to be configured to a value of 2. This does not require modification of the kernel's budget *netdev_budget*, as budget is configurable through the /proc file system, specifically with /proc/sys/net/core/netdev_budget. Changing operation mode is based on the estimated traffic rate. An experimental evaluation and detailed implementation of the packet rate estimation for BCM5700 driver is given in [12]. The estimation is based on a simple packet rate estimator which exhibits a good estimation quality in terms of accuracy, agility, and stability, and more importantly requires minimal computational cost. Minimal cost was achieved by using shift operations and thus avoiding CPU expensive operations of multiplication and division.

4.2. Identifying the Cliff Point and Switching Mechanism

Identifying the proper switching point is critical to achieve sustained performance at different traffic load conditions. There are primarily two methods to experimentally identify the cliff point: internal and external measurements. Internal measurement includes profiling and instrumentation of Linux kernel code [16] and may involve the use of logic analyzer or oscilloscope [17]. The objective is to measure the cost of interrupt overhead, protocol processing, and application processing. Closed-form solutions derived in [3,18] can then be used to identify the cliff point for kernel's protocol processing or application processing. For example, to identify the cliff point λ_{cliff} for kernel's protocol processing, the following formula derived in [3] can be used:

$$\lambda_{cliff} = \frac{r}{2}\left(\sqrt{1+4\frac{\mu}{r}} - 1\right),$$

where $1/\mu$ is the mean time the kernel takes to process one packet and deliver it to user application. $1/r$ is the mean time for interrupt handling an incoming packet. This mean time is essentially the overall interrupt cost which includes interrupt overhead and handling.

Internal measurement is tedious and can be impractical when dealing with closed operating systems such as Windows or Mac OS. In such a case external measurement is the best choice. With external measurement, the host is dealt with as a black box and is subjected to various network traffic loads generated by specialized traffic generators. Performance metrics such as throughput, latency, packet loss, and system CPU utilization are then measured and plotted in relation to generated traffic rate. By eyeballing the curves of plotted relations, the cliff or saturation point can be identified. The saturation point can be identified by multiple signs. It is approximately the arrival rate at where the throughput starts to flatten off or reaches its peak, or where packet loss or latency starts increasing sharply, or more precisely where the *system* CPU utilization approaches 100%. A CPU with 100% *system* utilization indicates that the kernel is consuming most of the CPU power and leaving little CPU to other tasks. For example in our experiment described later in Section 5, the CPU utilization consumed by system and shown with Linux "*top*" and "*mpstat*" utilities approaches close to 100% when traffic rate is around 100 Kpps. Therefore, we will use 100 Kpps as the cliff point in our implementation of the Hybrid scheme.

Both commercial and open-source traffic generators are available. Commercial traffic generators are available from IXIA, Agilent, Spirent, Simena, and Cisco. Open-source traffic generators are not as powerful in terms of generating traffic rate but can also suffice. We experimented with various traffic generators and found out that kernel-based generators such as KUTE [19] and pktgen [20] can produce up to 700 Kpps using a Pentium IV 3.2 GHz with 512 MB RAM and a BCM5752 network adapter. In addition we found out experimentally that D-ITG (which is an application-level generator) [21] can produce up to 115 Kpps using the same hardware. With a dual-processor machine running at 3.6 GHz and 4 GB RAM, we were able to produce 235 Kpps with D-ITG. In order to generate further more powerful traffic load using publicly available generators, it is possible to use multiple machines to generate individual traffic, and then use a switch to aggregate these individual traffic into a more powerful load, as was done in [22]. This works nicely for studying the performance measurement for throughput, packet loss, but not for latency. With latency, there is a need for clock synchronization between all generators and receiver. Unfortunately and according to our experience, publicly available network synchronization protocols such as NTP do not provide accurate and fine-grain timing measurements.

It is possible that repeated switchings (or oscillation) between normal interruption and NAPI can occur around the saturation point in the presence of a traffic load that is highly fluctuating at the point of saturation. In order to alleviate this, two thresholds for arrival rate are used around the saturation or cliff point λ_{cliff}. Particularly, these two threshold can be expressed by the following formulas: $cliff1 = (1-\varepsilon)\times\lambda_{cliff}$ and $cliff2 = (1+\varepsilon)\times\lambda_{cliff}$, where ε is a tunable design parameter. *cliff1* acts as a lower threshold and *cliff2* acts as an upper threshold. Switching from DE to NAPI only occurs when the estimated traffic arrival rate is greater than *cliff2*. Switching from NAPI to DE only occurs when the estimated arrival rate is lower than *cliff1*. No switching takes place if estimated arrival rate is between *cliff1* and *cliff2*. A simulation study was conducted in [23] to determine the proper relation of these thresholds with respect to a given arrival rate for a cliff point such that minimal switching is achieved when generating Poisson and highly fluctuating bursty traffic with a mean rate of the given cliff point λ_{cliff}. Based on the reported results in [23], the least number of switchings was achieved when $\varepsilon = 0.15$.

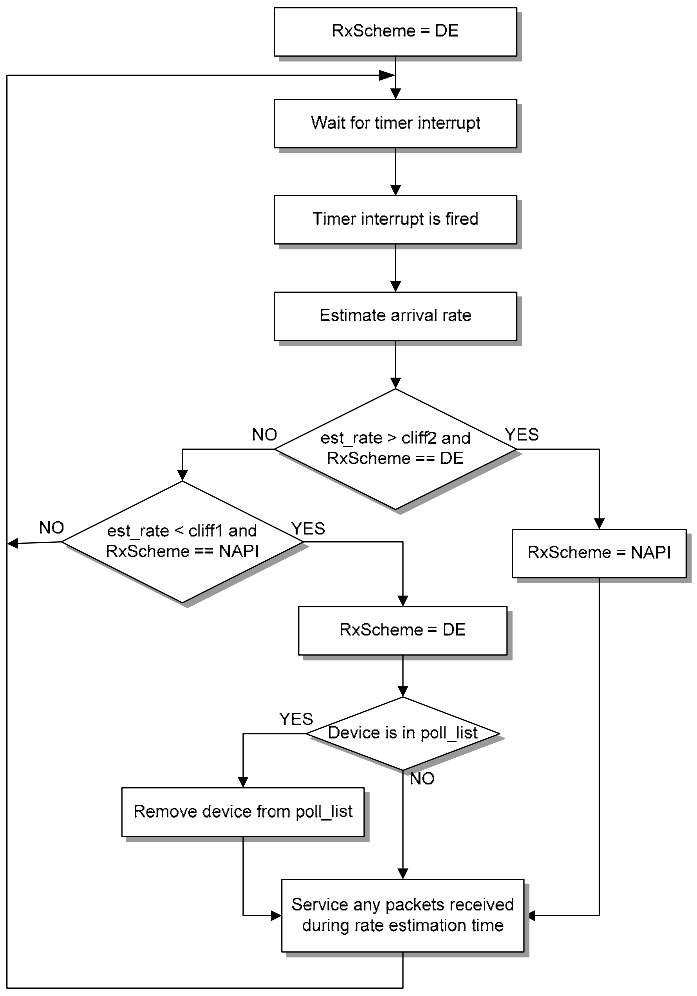
RxScheme = DE
Wait for timer interrupt
Timer interrupt is fired
Estimate arrival rate
NO
est_rate > cliff2 and RxScheme == DE
YES
NO
est_rate < cliff1 and RxScheme == NAPI
YES
RxScheme = NAPI
RxScheme = DE
YES
Device is in poll_list
NO
Remove device from poll_list
Service any packets received during rate estimation time

Figure 4. Switching mechanism in Hybrid

A flowchart of the switching mechanism that was implemented in Linux kernel is shown in Figure 4. At initialization, specifically in *bcm5700_open()*, the estimated timer is configured to fire every 8 ms and DE scheme is set by default. Periodically on every 8 ms, the arrival rate is estimated and checked against *cliff2* and *cliff1*. It is possible that some packets can arrive during rate estimation, so it is important to process those packets before the start time of next arrival or scheduled poll. When switching to DE, the network device has to be removed from *poll_list* if it is already there.

5. Experimental Setup

In order to test our implementation and evaluate the performance of the three schemes (viz. DE, NAPI, and Hybrid), we set up an experiment comprised of two Linux machines of a sender and a receiver connected with 1 Gbps Ethernet crossover cable (as shown in Figure 5). The sender has two Intel Xeon processors running at

3.6 GHz with 4 GB of RAM. It has an embedded Intel 82541GI Gigabit Ethernet NIC running with the e1000 driver. The receiver is an Intel Pentium 4 processor running at 3.2 GHz with 512 MB of RAM. It has a 3COM Broadcom NetXtreme Gigabit Ethernet card with BCM5752 controller. This NIC is running with a loadable kernel module of the modified BCM5700 driver version 8.2.18 that implements the schemes of DE and Hybrid. Both sender and receiver use Fedora Core 5 Linux 2.6.15. To minimize the impact of other system activities on performance and measurement, we boot up both machines with run level 3, and we made sure that no services are running in the background. We also disabled Ethernet link flow control. For both machines, the timer interrupt frequency or *HZ* was set to 250. The timer for packet rate estimation was set to fire every two *jiffies* (i.e., every 8 ms).

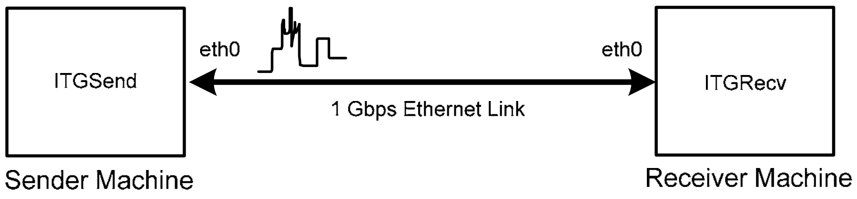

Figure 5. Experimental Setup

To generate traffic from the sender machine, we used the open-source D-ITG 2.4.4 generator [21]. D-ITG is a user-level generator and requires installation of ITGSend at the sender and ITGRecv at the receiver. ITGSend has the ability to generate different types of network traffic with defined random distribution for interarrival times and packet sizes. For all of our generated traffic, we used UDP packets with a constant 64-byte packet sizes and constant interarrival times. The reason we used UDP is to ensure that at the receiver we have 1:1 mapping between incoming packets and generated interrupts. This makes it easier to analyze results and source of delays and overhead. Unlike TCP, multiple packets can be generated by sender to successfully transmitting a single TCP packet, and thus causing multiple interrupts to be generated at the receiver. With this setup, ITGSend was able to produce up to 235 Kpps for one flow.

6. Performance Measurements

For evaluating performance, several measurements of various metrics were taken in relation to generated traffic load. These metrics include the average throughput, packet loss, latency, CPU availability, and interrupt frequency. For all of experimental results reported and shown in this section, we performed three experimental trials and final results are the average of these three trials. For each trial, we recorded the results after the generation of a flow with a specific rate for a satisfactory duration of 30 seconds. Longer durations gave little or no difference. The average throughput and packet loss were recorded by decoding the logs produced by

ITGRecv. D-ITG provides a decoding utility called ITGDec for this purpose. Average latency is the average round-trip time for a packet to be sent by ITGSend at the sender to ITGRecv at the receiver and returned back by ITGRecv to ITGSend at the sender. In order to avoid clock synchronization between the sender and receiver, we used the sender to send and receive packets. In our experiment, we did not use NTP protocol for clock synchronization between sender and receiver, as NTP did not give acceptable synchronization. Clock drifting was a persistent problem and required frequent synchronization. As for measuring the average CPU availability and interrupt rate, we used the "*sar*" Linux utility at the receiver. There was no synchronization between starting ITGSend and starting the measurement of "*sar*". For this reason, we started "*sar*" manually on the receiver after 5 seconds from starting ITGSend on the sender. "*sar*" also has to be terminated 5 seconds before the termination of ITGSend flow. So for a flow duration of 30 seconds, "*sar*" was thus run for a duration of 20 seconds.

Before embarking on taking performance measurement, it is imperative to find the appropriate value of the NAPI's budget that can yield the best performance using our hybrid scheme. As discussed in Section 3, the default NAPI's budget may not be appropriate and needs to be adjusted depending on what the system is primarily utilized for. In NAPI and with the default budget of 300 and under high traffic rate, up to 3000 packets can be processed by the kernel before giving a chance (via *ksoftirqd*) to user-applications to run. Under our experimental setup shown in Figure 5, we found out that *ksoftirqd* does not get awakened under the maximum possible generated traffic rate by D-ITG, which is 235 Kpps. We used KUTE [19] on the sender machine to generate a traffic of a maximum rate of 700 Kpps. Under this setup, we observed that *ksoftirqd* starts being awakened at a rate of 300 Kpps. As will be concluded next from Figure 6, the default configuration of budget 300 is huge and not appropriate for network I/O-bound applications such as ITGRecv.

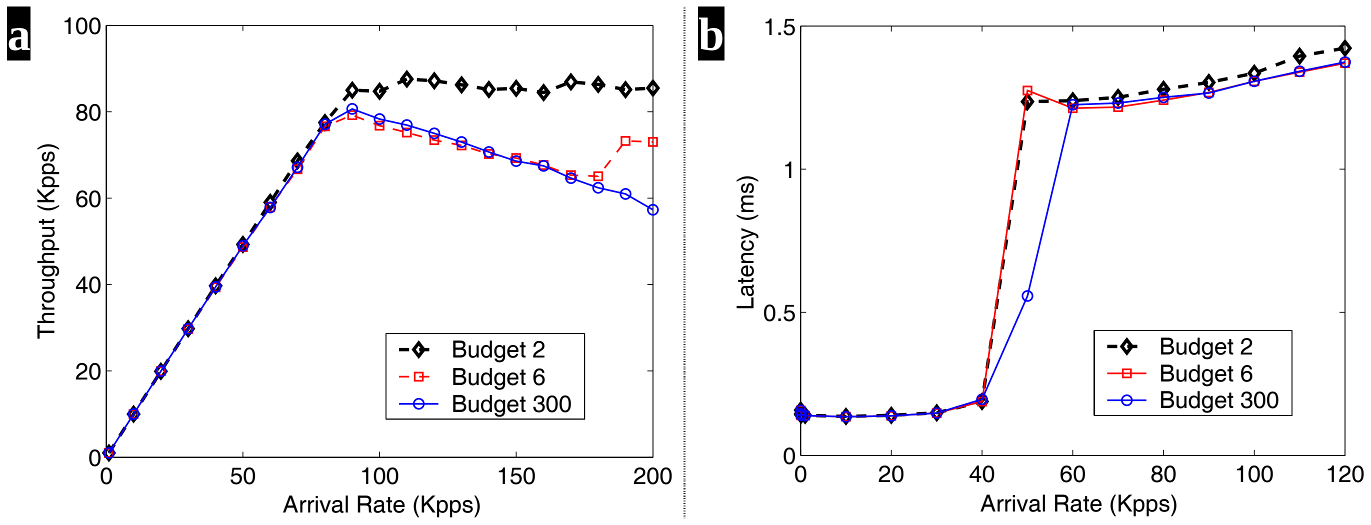

Figure 6. Impact of NAPI Budget on throughput (a), and latency (b)

Figure 6 compares NAPI performance in terms of average throughput and latency for three different values of budget (viz. 2, 6, and 300). The average throughput is recorded by ITGRecv at the receiver. The latency is the average round-trip time recorded by ITGSend at the sender. In measuring the latency, the sender was able to generate a traffic rate of up to 120 Kpps. This is due to the fact that sender processing has doubled as it is not only generating packets but also receiving packets from ITGRecv at the receiver. Figure 6(a) shows that with a budget of 300 and 6, the throughput starts to fall; while a budget of 2 gives a more acceptable throughput. The reason for this is that a smaller budget would give more CPU time for D-ITG to process packets as it will allow *ksoftirqd* to be awakened earlier, i.e., after reaching a maximum of 20 packets are processed in one polling period as opposed to 3000 packets in the default configuration. It is to be noted that at around 170 Kpps the throughput with a budget of 6 starts to improve. This is because *ksoftirqd* gets awakened at this point after processing a maximum of 60 packets in one polling period, and thus allocating more CPU to ITGRecv. Figure 6(b) shows that a budget of 2 results in a relatively higher latency at high rates. Budgets of 6 and 300 give comparable results at low and high rates. With the sender's limited generated rate of up to 120 Kpps, a budget of 6 allowed the kernel to process all the packets queued in DMA ring in one polling period. In conclusion a budget of 2 gives a reasonable and acceptable compromise for performance in terms of throughput and latency for a network I/O-bound process such as ITGRecv. For our hybrid scheme, we will use a budget of 2 for all measurements.

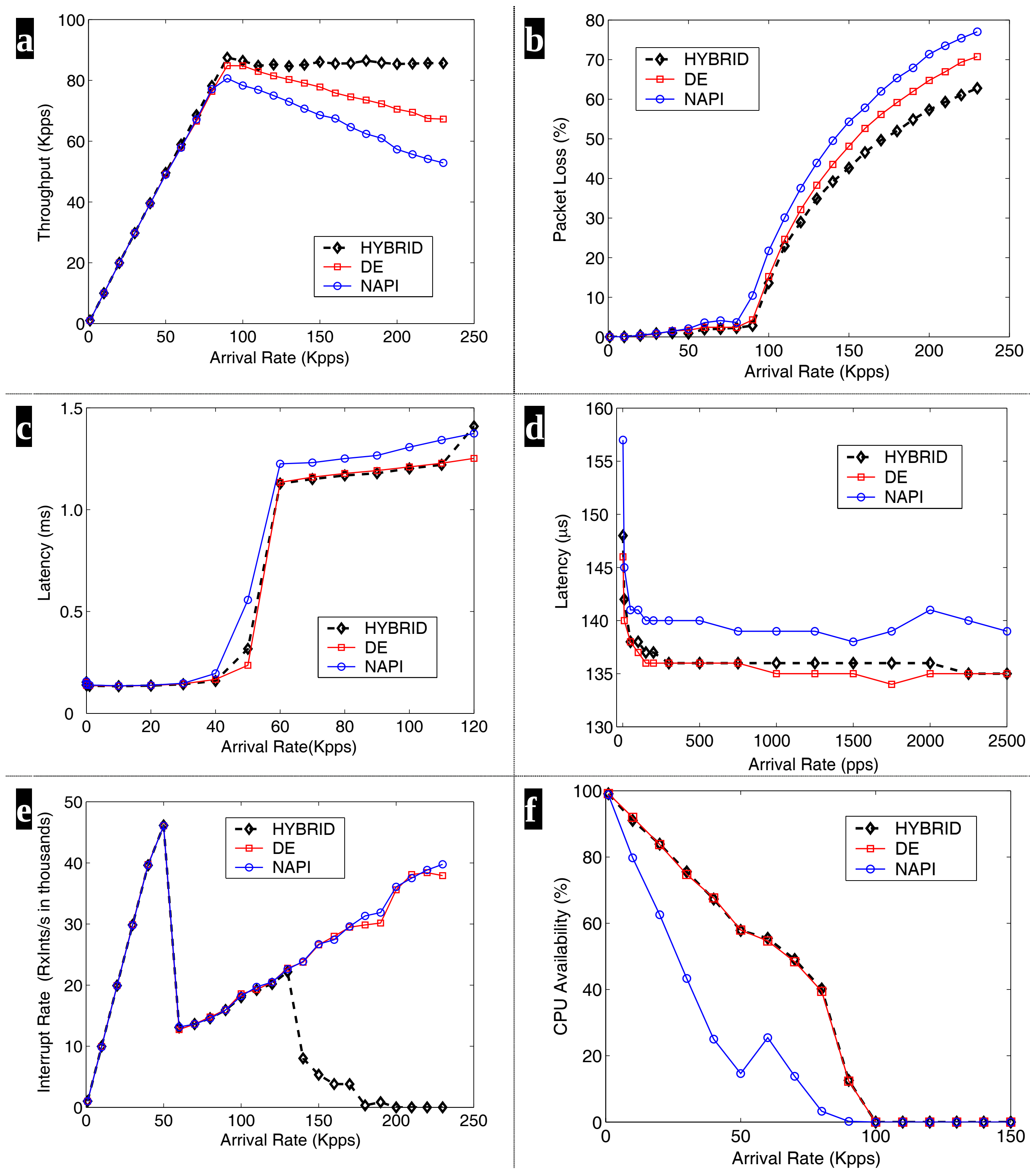

Figure 7. Performance measurements in relation to incoming traffic rate

Next, we study and compare experimental performance results of the schemes of Hybrid, DE, and NAPI. Performance is reported as shown in Figure 7 for a number of important metrics which include throughput, packet loss, latency, interrupt rate, and CPU availability. In terms of throughput and packet loss (exhibited in Figure 7(a) and (b)), Hybrid outperforms both NAPI and DE. At heavy load, Hybrid gives more acceptable and sustainable throughput; whereas the throughput of NAPI and DE starts to degrade as traffic load increases. This is primarily due to the fact that Hybrid, after reaching *cliff2* (which is set to 115 Kpps), switches to NAPI with a budget of 2, and thereby giving ITGRecv adequate CPU time to process packets. In DE and NAPI, the residual CPU time for ITGRecv starts to diminish gradually with higher traffic rate. This shows that adjusting the budget is critical. It is also depicted that DE gives more throughput and less packet loss than NAPI. As noted in Section 4.1, the main reason for this is that DE runs at interrupt level and incurs far less overhead than NAPI.

Figure 7(c) exhibits the average round-trip latency recorded by ITGSend with respect to the generated traffic rate. The figure shows that Hybrid and DE outperform NAPI because NAPI incurs more overhead. Also both Hybrid and DE give comparable results up to a rate of around 115 Kpps (which is *cliff2*). Hybrid latency approaches that of NAPI beyond 115 Kpps as Hybrid switches to operate in NAPI. Figure 7(d), which is a zoom-in of Figure 7(c) at low traffic rate, shows that at severely low rate of less than 200 pps, the latency for all schemes are relatively large, with NAPI exhibiting the largest latency. The reason for this is that an overhead (from interrupt or softirq scheduling) is incurred separately for almost every incoming packet. As the arrival rate increases, the incurred overhead is aggregated for multiple packets. At very low rate, NAPI exhibits the most overhead as noted in Section 4.1. Lastly, it is observed in both figures that the overhead involved in estimating traffic rate in Hybrid did not introduce a noticeable additional delay when Hybrid operates in DE of up to a rate of 115 Kpps. This was expected as the implementation of rate estimation was performed every 8 ms and its code uses shift operations to avoid CPU expensive operations of multiplication and division.

Figure 7(e) compares the three schemes in terms of mitigating the interrupt rate. It is observed that under a rate below 50 Kpps, the interrupt rate increases linearly in relation with the arrival rate for all three schemes. Shortly after that the interrupt rate drops significantly, and then it starts increasing again. This is in line with the expected behavior. Below 50 Kpps, the interrupt rate is low and host is able to finish interrupt handling and processing of packets before the next interrupt. The period of disabling and enabling RxInts (i.e., interrupt masking period) for all of the three schemes finishes before the occurrence of next interrupt. As incoming rate increases beyond 50 Kpps, multiple packet arrivals or interrupts occur within this masking period, and thus showing a significant drop of interrupt rate shortly after 50 Kpps. After 60 Kpps the interrupt rate starts increasing again but very slowly with respect to the arrival rate. There is still interrupt masking occurring;

however, the masking period is not stretching considerably with higher rate. The CPU still has the power to handle and process packets relatively quickly. For Hybrid, at around 130 Kpps, the interrupt rate gradually drops to zero. This is due to the fact that masking period for Hybrid stretches considerably with such a high rate, and therefore forces Hybrid to operate in NAPI with the limited budget of 2. Under such high rate, NAPI with budget of 2 would not be able to exhaust all packets in one polling period and never re-enable RxInt. At severely high rate beyond 235 Kpps, it is expected that the interrupt rate in both NAPI and DE would decrease as the masking period starts stretching more in both. As NAPI polling is not exhaustive, it is expected that the decrease of interrupt rate under NAPI will occur more rapidly than DE.

The CPU availability is highly affected by the interrupt rate. This is clearly demonstrated in Figure 7(f) where it shows a comparison of the three schemes in terms of the percentage of residual CPU power or bandwidth left after processing packets by both Kernel and ITGRecv. As shown, NAPI results in the least residual CPU bandwidth. This is expected as noted in Section 4.1 that NAPI requires the most overhead. It is also shown that curves of DE and Hybrid are comparable with no noticeable impact due to rate estimation. It is observed that at around incoming rate of 50-70 Kpps, the CPU availability does not increase linearly (and in fact is more in NAPI at 60 Kpps than 50Kpps). The reason for this is that the corresponding interrupt rate at rate 50 Kpps (as shown in Figure 7(e)) falls considerably, and therefore results in less overhead. The figure shows that CPU availability at 100 Kpps approaches zero for all schemes. At this point the CPU is fully consumed by both kernel processing as well as ITGRecv.

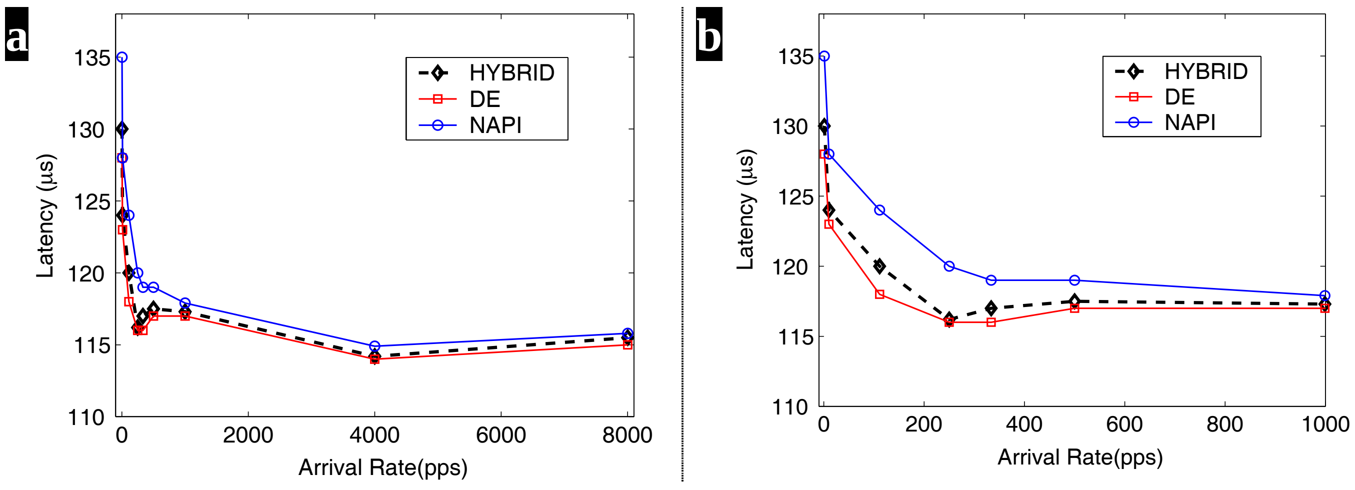

Figure 8. Average latency reported by "ping" utility

Finally, we investigate the performance of the three schemes with respect to processing received packets only at the kernel level with no involvement of user processes. We use "ping" utility. At the sender "ping" issues ICMP echo request packets which get processed by ICMP protocol at the kernel level and then transmitted back to the sender. The same experimental setup was used as shown in Figure 5. The sender machine was configured to issue "ping" with different arrival rates for a duration of 30 seconds for each trial. The results shown in Figure 8 are the average of three trials. We excluded outliers particularly those observed at the beginning of the flow due to ARP caching. We were able to obtain a maximum generate traffic of around 8 Kpps with "ping –f" (a.k.a. ping flooding). We were not able to generate traffic rate with fine granularity. This was a shortcoming of "ping" utility. Figure 8 confirms that DE and Hybrid outperform the default NAPI. Figure 8(b) is a zoom-in version of Figure 8(a). As was the case in Figure 7(d), Hybrid exhibits slightly more delay than DE. This is due to the overhead introduced by rate estimation in Hybrid. Additionally, the shape of three curves of Figure 8 are very similar to those of Figure 7(d), showing high latency at extremely low packet rate.

7. Conclusion

We presented and discussed major changes required to implement a hybrid interrupt-handling scheme in the latest version of Linux kernel 2.6.15. We also identified and addressed important implementation issues related to adjusting the NAPI budget and also determining the cliff or saturation point. It was demonstrated that the default configurations of NAPI were not suitable for network I/O applications. We proved experimentally that the Hybrid scheme can improve the performance of network I/O applications under low and high traffic rate. We measured and compared the performance of DE, NAPI, and Hybrid schemes in terms of throughput, packet loss, latency, and CPU availability. The Hybrid scheme clearly shows noticeable performance gain for general-purpose network desktops or servers running network I/O-bound applications as that of ITGRecv. Such gain can also be achieved for today's general-purpose servers running critical and popular network I/O applications such as web, IRC, database transactions, networked data acquisition, network intrusion detection and prevention, deep-packet analysis, packet logging and monitoring, etc. Also Hybrid scheme has the potential of improving the performance of general-purpose servers configured as NAT/firewalls or routers. This was demonstrated briefly using "ping" utility, but requires more thorough testing. To accomplish this, we plan to evaluate experimentally the performance of Hybrid for hosts configured for IP-forwarding. The measurements will be conducted using IXIA traffic generator which is a more powerful and precise traffic generator and analyzer. We also plan to extend the implementation of Hybrid and evaluate its performance for Linux hosts with a quad-core processor and multiple network interfaces.

Acknowledgements

We acknowledge the support of King Fahd University of Petroleum and Minerals in completion of this work. This work has been funded under Project #INT-383. We are also very thankful to Robert Olsson and Jamal Salim Hadi for their valuable replies and comments to some of the issues faced throughout this work.

References

[1] K. Ramakrishnan, “Performance Consideration in Designing Network Interfaces,” *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, February 1993, pp. 203-219.

[2] J. Mogul, and K. Ramakrishnan, “Eliminating Receive Livelock in an Interrupt-Driven Kernel,” *ACM Trans. Computer Systems*, vol. 15, no. 3, August 1997, pp. 217-252.

[3] Salah, K., El-Badawi, K., and Haidari, F., “*Performance Analysis and Comparison of Interrupt-Handling Schemes in Gigabit Networks*,” International Journal of Computer Communications”, Elsevier Science, Vol. 30(17) (2007), pp. 3425-3441.

[4] K. Salah “To Coalesce or Not to Coalesce”, *International Journal of Electronics and Communications (AEU)*, vol. 61, no. 4, 2007, pp. 215-225.

[5] C. Dovrolis, B. Thayer, and P. Ramanathan, “HIP: Hybrid Interrupt-Polling for the Network Interface,” *ACM Operating Systems Reviews*, vol. 35, October 2001, pp. 50-60.

[6] J. H. Salim, “Beyond Softnet,” Proceedings of the 5th Annual Linux Showcase and Conference, November 2001, pp 165-172

[7] L. Deri, “Improving Passive Packet Capture: Beyond Device Polling,” Proceedings of the 4th International System Administration and Network Engineering Conference, Amsterdam, September 2004.

[8] O. Maquelin, G. R. Gao, H. J. Hum, K. G. Theobalk, and X. Tian, “Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling,” Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, 1996, pp. 178-188.

[9] X. Chang, J. Muppala, P. Zou, and X. Li, “A Robust Device Hybrid Scheme to Improve System Performance in Gigabit Ethernet Networks”, Proceedings of the 32nd IEEE Conference on Local Computer Networks, Dublin, Ireland, October 15-18, 2007, pp. 444-451.

[10] X. Chang, J. Muppala, W. Kong, P. Zou, X. Li, and Z. Zheng, “A Queue-based Adaptive Polling Scheme to Improve System Performance in Gigabit Ethernet Networks”, Proceedings of the 26th IEEE International Conference on Performance, Computing, and Communications Conference (IPCCC 2007), New Orleans, Louisiana, April 11-13, 2007, pp. 117-124.

[11] K. Salah and F. Qahtan, “Experimental Performance Evaluation of a Hybrid Packet Reception Scheme for Linux Networking Subsystem,” Accepted for publications in the 5th IEEE International Conference on Innovations in Information Technology, 2008, Al Ain, UAE, pp. 1-5.

[12] K. Salah, F. Haidari, A. Bahjat, and A. Mana, “Implementation and Experimental Evaluation of a Simple Packet Rate Estimator”, International Journal of Electronics and Communications (AEU), Elsevier Science, In Press.

[13] D. Bovet and M. Cesati, “Understanding the Linux Kernel,” O’Riley Press, 3rd Edition, November 2005.

[14] “3Com® 10/100/1000 PCI-X Server Network Interface Card,” Available from ftp://ftp.3com.com/pub/nic/3c996/linux-8.2.18.zip

[15] J. Salim, “When NAPI Comes to Town,” Proceedings of Linux 2005 Conference, Swansea, U.K., August 2005.

[16] B. Moore, T. Slabach, and L. Schaelicke, “Profiling Interrupt Handler Performance through Kernel Instrumentation,” Proceedings of the IEEE 21st International Conference on Computer Design, Los Alamitos, California, October 13-15, 2003, pp. 156-163.

[17] R. Hughes-Jones, S. Dallison, G. Fairey, P. Clarke and I. Bridge, “Performance Measurements on Gigabit Ethernet NICs and Server Quality Motherboards”, Proceedings of the 1st International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003), Geneva, Switzerland, February 2003.

[18] K. Salah, “Modeling and Analysis of Application Throughput in Gigabit Networks,” International Journal of Computers and Their Applications, *International Journal of Computers and Their Applications*, ISCA Publication, Vol. 12(1) (2005), pp. 44-55.

[19] S. Zander, D. Kennedy, and G. Armitage, “KUTE – A High Performance Kernel-based UDP Traffic Generator,” CAIA Technical Report 050118A, January 2005. Available from http://caia.swin.edu.au/genius/tools/kute/

[20] R. Olsson, “Pktgen the Linux Packet Generator,” Proceedings of Linux Symposium, Ottawa, Canada, 2005.

[21] D. Emma, A. Pescape, and G. Ventre, “D-ITG, Distributed Internet Traffic Generator”, Available from http://www.grid.unina.it/software/ITG

[22] C. Murta and M. Jonack, “Evaluating Livelock Control Mechanism in a Gigabit Network,” Proceedings of 15th IEEE Computer Communications and Networks (ICCCN 2006), Arlington, Virginia, October 2006, pp. 40-45.

[23] F. Haidari., “Impact of Bursty Traffic on the Performance of Popular Interrupt Handling Schemes for Gigabit-Network Hosts,” M.S. Thesis, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, May 2007.