Efficient real-time Linux interface for PCI devices: A study on hardening a Network Intrusion Detection System

Amitava Biswas
Concordia University
Montreal, Canada
amitavabiswas@ieee.org

Purnendu Sinha
Philips Research
Bangalore, India
purnendu.sinha@philips.com

Abstract

Traditional software network interfaces in Linux do not deliver satisfactory real-time performance. Hence alternative efficient real-time interfaces are required in network monitoring, distributed systems, real-time networking and remote data acquisition applications. Designing such a software network interface is not trivial. A PC based software network intrusion detection application is studied as an example. Poor throughput and real-time performance of traditional interfaces or their enhanced versions can cause packet skipping and other non-obvious synchronization related failures, which may make the detector ineffective. The effectiveness of the detector can be enhanced by improving its packet capturing and dispatching interface. We achieve this by using an efficient real-time software interface for a PCI Ethernet card. This paper describes the design and implementation details of this interface and its deployment for Linux based network intrusion detection sensors. The nuances of the system design for high speed packet capturing are discussed and the advantages of the proposed design are demonstrated. This mechanism outperforms existing packet capturing solutions - NAPI, PFRING and Linux kernel under heavy network load in terms of higher load bearing capacity, packet capturing capacity and superior real-time behavior.

Keywords

High bandwidth packet capture, high speed network monitoring, Linux

1. Introduction

Real-time networking, distributed systems, remote data acquisition/logging and control applications require real-time software interfaces for devices in addition to real-time operating system (RTOS) and suitable hardware. Real-time interface for Ethernet PCI network card is especially interesting because it has extensive use in these applications. The implementation of these interfaces are non-trivial and specific to operating systems (OS), however the design principle is transferable. Software network interface (NI) designing for open source "Linux like" or "Linux based" platforms is interesting because these platforms are quite popular in commercial, research and development projects. At present these platforms use traditional NIs. Only a few such open source real-time kernels [1] are available, and these platforms generally offer NIs that are fashioned after traditional Linux ones [2]. The traditional software NI, which constitute the network card driver, the kernel network stack and the socket mechanism, is inefficient and have poor performance. It consumes higher CPU resources, introduces higher packet delivery time, causes higher incidence of packet drops and delivers lower bandwidth capacity [3,4].

A large volume of research work [4,5,6,7] is available, that focus on improving the efficiency and capacity of traditional Unix/Linux network interfaces. However these did not address all the multiple associated problems behind these inefficiencies, and did not identify the causal relationship between real-time behavior of the underlying platform and the NI performance. Some earlier work by the authors [3,8,9] had introduced a design for an efficient real-time NI and demonstrated its benefits in a Network Intrusion Detection System (NIDS). However

authors had not discussed the details of the proposed interface, which would have been useful to real-time network application and network security system developers.

Whereas, this paper illustrates the detailed architecture of the proposed interface and its implementation for two Linux based OS options: Redhat 8 (custom low latency 2.4.18 kernel) and vanilla Linux 2.4.24 with RTAI 3.1, a hard real-time co-kernel. It demonstrates how a modified driver along with user space libraries can efficiently capture packets and carry out packet processing instead of using the traditional NI based on Linux driver, kernel network stack and socket mechanism. The proposed mechanism can be reused with any other PCI bas mastering data acquisition cards or devices. This work was particularly motivated by the fact that a distributed network intrusion detection system (NIDS) may be hardened against high bandwidth network attacks by using this NI at strategic points. Therefore, this paper adopts a case study on a distributed NIDS to demonstrate how the proposed interface can be applied to gain significant advantages. Detailed discussions on the implementation aspects highlight the nuances of system design for wire speed packet capturing. The operation of the proposed interface is also explained, so that the design principles can be transferred to other platforms and peripherals. At the end, NIDS performance related works are discussed for the sake of completeness.

To expose the limitations of the existing packet capturing solutions we carried out all the experimental work on modest hardware: PII 333Mhz CPU with 100Mbps networks. The conclusions drawn based on observation on this hardware and network combination remains valid for 4Ghz CPU with 1Gbps networks, as the ratio of CPU to network speed are in same order for both cases (3:1 vs. 4:1).

2. Motivation: Vulnerability of NIDS

Effectiveness of NIDS has become key in protecting data network against hacker activity, especially when the intruders employ sophisticated attacks with more computing resources at their disposal. Hackers may attack, disable or bypass the NIDS before attacking the protected resources [10,11]. A software NIDS often have limited performance which makes it vulnerable to certain kinds of attacks. However a PC based software NIDS is preferred over a FPGA/hardware NIDS, as it can implement complex, dynamic rules to suit rapidly changing, complex and network specific attack patterns. But, a practical software NIDS running on a Ghz CPU system can only match 100Mbps traffic rate [12,13], because it implements few hundreds of computation intensive pattern matching and event co-relation rules. Such a NIDS with insufficient capacity skips packets [14,15] and may manifest inadequate detection during heavy traffic [16,17]. Thus its detection will deteriorate when loaded beyond its packet rate capacity [18]. A resourceful hacker may overload such NIDS with a flood of smoke screen packets to sneak past the real attack packets, which the NIDS may skip.

In addition to the possible inefficiencies in the NIDS application and in the underlying general purpose commodity PC hardware, inefficient NI which is part of the operating systems (OS), is one significant bottleneck factor. This inefficiency is not generally addressed directly, instead more powerful hardware, multiprocessor or distributed systems are deployed to get more capacity [12,15,19 to 22]. For load sharing and efficiency, the incoming packet traffic may be distributed to an array of specialized network intrusion detection sensors (Fig. 1) based on protocol, flow or some other criteria [12,15,20,21]. Two separate network segments and interfaces, one for receiving, the other for sending, may be used wherever required to improve the throughput. Even such distributed systems have limitations, they have certain components which face full incoming packet load before it is distributed. The traffic splitter analyzes and arbitrates the load dispatching, thus it faces the full load and may become the bottleneck. The sensors also have to efficiently capture at high packet rates and leave enough resources for the detection. In addition to these requirements, the distributed NIDS components require to satisfy some real-time constraints to detect very complex co-ordinated attacks. Instead of the sensors, the centralized alarm analyzer/corelator, which takes alarm

feeds (say as, UDP/IP packets) from different sensors can only suspect a complex attack after carrying out more knowledgeable event correlation (Fig. 1).

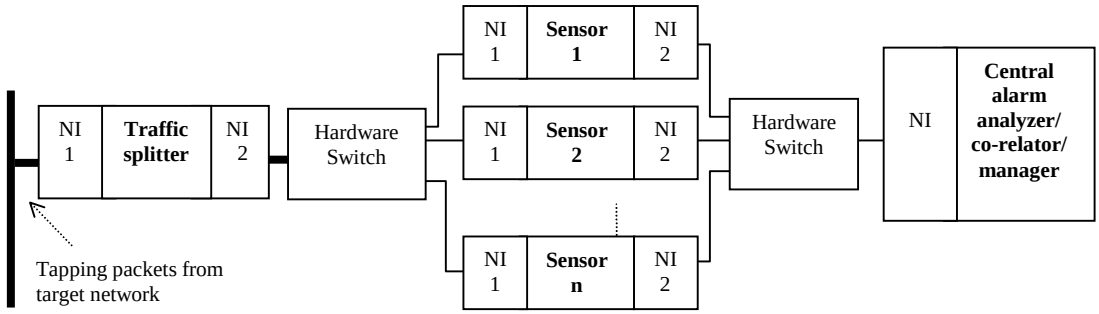


Fig. 1: Distributed NIDS deployment

To carry out effective correlation, all these temporally related alarms from different sensors should arrive at the centralized corelator within a limited time span window. This is explained by the following attacks situation (Fig. 2).

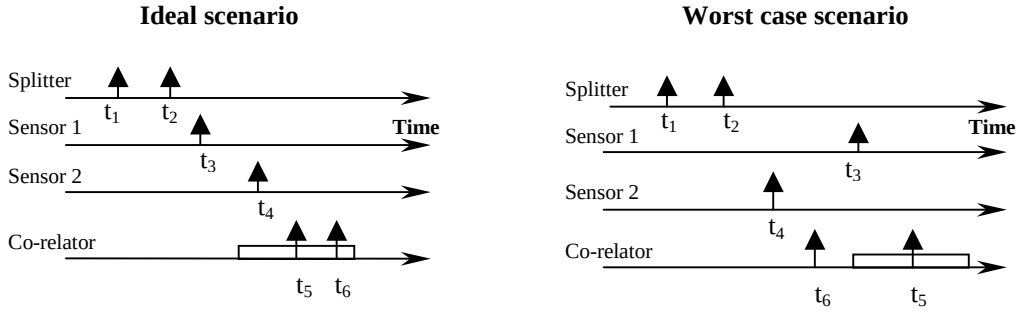


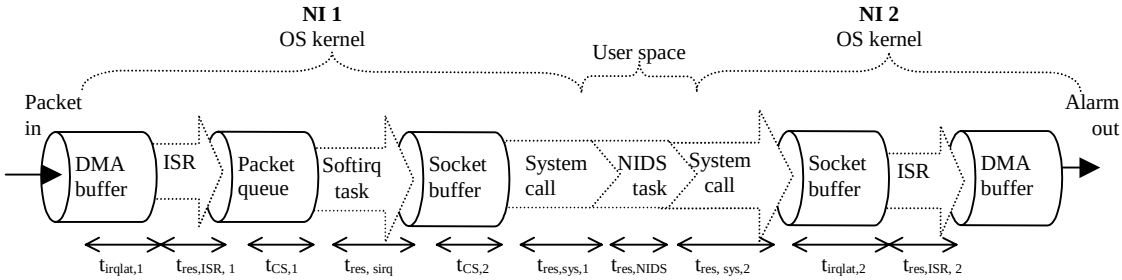
Fig. 2: Failed detection due to timing failure

In a staged attack, the attacker sends first type of attack packet, which may arrive at the splitter at time t_1, whereas the second type of attack packet may arrive at time t_2. The splitter may send the first attack packet to sensor 1 and the second one to sensor 2, where they arrive at time t_3, t_4 and the corresponding alarms arrive at the corelator at t_5 and t_6. The alarms can be correlated and the staged attack can be detected if $|t_6-t_5|$ is bounded within the correlating time span window t_w, i.e. if $|t_6-t_5| < t_w$. This is guaranteed if both $(t_5- t_1)$ and $(t_6- t_2)$ are bounded within t_w. This requires that the splitter, the individual sensors and the corelator have bounded response times. If this is not satisfied then there is a certain probability that the detection will fail in some cases. It is possible to increase the detection failure probability by selectively loading some sensors. It is well known that the response time of a Linux system deteriorates with the incident network traffic (or interrupt) load [23]. A hacker might send a flood of first type of attack packets to load sensor 1, and send a single second type of attack packet which is detected by sensor 2. Sensor 1 will introduce a large time delay in t_5, whereas sensor 2 will not delay t_6. As a result $|t_6-t_5|$ can be greater than t_w, hence the corelator can fail to detect the staged attack.

The alarm corelator can not allow a wider time window t_w to accommodate the time jitter introduced by the sensors, as a larger t_w increases the computation, reduces the throughput of the alarm corelator [9] and thus increases its packet skipping vulnerability. This system constraint demands that the individual sensors and the splitter should complete their tasks within bounded time. Inefficient NIs add non-deterministic delay in the packet delivery latency and the response time in each NIDS component (Fig. 2). This may jeopardize detection of sophisticated attack patterns. Any Linux based components using traditional NIs are likely to suffer from these performance limitations because Linux is not an RTOS and these traditional NIs are inefficient. Next section presents how the performance limitations of traditional NIs depends on poor real-time behavior of the underlying OS and in addition to other factors.

3. Analysis of network interface performance limitations

There are two distinct performance requirements for the NIs. The NI should be efficient in terms of lower CPU and memory utilization, and they should have bounded packet delivery task response times [9]. An efficient NI may have lower average system response time, but it may still manifest a few cases of very high transient response times. By making the system efficient do not reduce these large transient response times. The only way to bound these times is to adopt a real-time OS or task scheduler. The traditional Linux NIs do not satisfy these two requirements. Even the enhanced NIs proposed by other researchers still suffer from performance limitations even at moderate packet arrival rates [3]. Fig. 3 presents the task model for the sensor with two traditional Linux interfaces. The arrow blocks signify distinct task threads. The user space NIDS task and the system calls are in the same task thread. The system response is the sum total of all individual task responses. The buffers act as an interim storage space between each pair of "producer" and "consumer" tasks. The producer task deliver data to the buffer and consumer task picks it up. This allows these two tasks to execute independent of each other to a certain extent. However buffers may overflow if the producer and consumer tasks executions are not balanced. Performance limitations of these traditional and enhanced NIs are primarily due to: (1) large average packet receiving task response times; (2) large average context switching times; (3) buffer overflows due to execution imbalance of producer-consumer task pairs; (4) large jitter in the context switching times; and (5) large jitter in their response times of the receiving task threads. In [3] authors have shown how the response time jitter causes packet skipping due to buffer overflow, high packet delivery latencies, lower throughput and limited system capacity.



$$t_{res} = (t_{irqlat,sw,1} + t_{res,ISR,1} + t_{CS,1} + t_{res,sirq} + t_{CS,2} + t_{res,sys,1} + t_{res,NIDS} + t_{res,sys,2} + t_{irqlat,sw,2} + t_{res,ISR,2})$$

Fig. 3: Task model of the sensor with two network interfaces

Some factors which are behind high receiving task response times, are:

- Significant execution time components of $t_{irqlat,1}$, $t_{res,ISR,1}$ and $t_{CS,1}$ that are involved in interrupt servicing which are collectively known as “interrupt servicing overhead”. Usage of expensive kernel locks, data copy or memory allocation operation to materialize zero copy in the interrupt servicing routing (ISR) task, are some known causes behind high execution time component of $t_{res,ISR,1}$. Similar issues are present at the transmitting interface, NI2, which effects the execution time components of $t_{irqlat,2}$, and $t_{res,ISR,2}$.
- Significant protocol processing time in the kernel are due to excessive layering and associated function call overheads, some of which may be redundant for most packet capturing or UDP processing. This affects execution time component of softirq task, $t_{res,sirq}$.
- Expensive data copy operation in the socket layer, when the packet payload is transferred between kernel and user space memory during the socket read/write system calls, this affects execution time component of $t_{res,sys,1}$ and $t_{res,sys,2}$.
- Similarly excessive function and system call overheads associated with reading/writing a socket from user space also affect execution time component of $t_{res,sys,1}$ and $t_{res,sys,2}$.

High interrupt handler and scheduler latencies affects the interrupt latency time components $t_{irqlat,1}$, $t_{irqlat,2}$, the context switching times $t_{CS,1}$, and the user space task latencies $t_{CS,2}$. Execution imbalances in the receiving tasks occur in Linux because ISR has highest relative priority, software interrupt (softirq) tasks have medium priority and user space task has the lowest priority, hence these static priority assignments cause “receive livelock” or buffer overflows. In Linux, this problem is addressed by limiting the number of instances of producer task executions by limiting the size of the packet queue. But this only delays the onset of this problem but does not completely solve it.

High jitter in response times of the packet receiving ISR, kernel, user space task threads and the context switching times are caused by:

- Preemption of the receiving tasks by a new interrupt. Linux interrupt handler preepmpts any task, even a high priority ISR if another high priority interrupt arrives, to run a new ISR to service the most current interrupt. In presence of high interrupt rate, the ISRs may be nested which can lead to fairly high response jitter in ISR or any other task thread.
- Presence of long critical section paths in Linux. These critical sections are execution paths when all interrupts are kept disabled. Fairly large critical sections of over 200 msec are reported for non-preemptable 2.4.x Linux kernels [24]. A few of such critical sections may still remain in 2.6.x preemptable kernels. Some users have complained of poor latencies in 2.6 kernels under some situations, authors have also noticed significant long latencies in 2.6.x based Fedora core 3 kernels, when observed over long duration under heavy interrupt loads. These long critical sections may add significant jitter to the interrupt latency time component $t_{irqlat.}$ Thus these critical sections adversely affect the OS clock timer and all other software kernel timers, all of which are driven by PIT interrupt in Linux. The critical sections may also affect any other receiving task response times directly, if such critical section is executed within the ISR which preempts the current packet receiving task.

Some solutions have been proposed by other researchers which partially address some of the above mentioned problems. An analysis of those can be found in [3]. Alternative interface mechanisms like NAPI polling driver [25] and PFRING interface [14] suffer from buffer overflow, data copy, memory allocation, redundant kernel processing, context switching related inefficiencies [3]. nCap [26] is an improvement over PFRING, but it still implements two task threads, one ISR, and a user space task in the receiving side, therefore vulnerable to receive livelock problem at a higher traffic rate. Moreover its interrupt based operation introduce high jitters in the system response, therefore nCap is expected to suffer from high packet delivery latencies at high packet rates, similar to PFRING. Real-time OS or kernels like RTAI [1] can address the response jitter related limitations. Reference [8] presents the relevant discussions on this aspect. RTAI based RTnet real-time stack [28] can bound the network processing response time but it still incorporate many inefficiencies that the traditional Linux NI suffers from. It had partially tightened the Linux kernel stack but have not completely removed all inefficiencies. More importantly RTnet can not be used in a generic application where the transmitting node is unknown and thus can not be bound to the receiver by the RTnet protocol. The design presented in the subsequent sections address most of the remaining problems that were not addressed by these solutions.

4. Design overview and rational

Fig. 4 presents how the proposed real-time "DMA ring" interface is deployed to improve the Network Intrusion Detector Sensor throughput. DMA ring on network interface card 1 (NIC1) captures the incoming packets from traffic splitter and makes it available to the NIDS application. The NIDS application transmits alarm packets, if any, through the other DMA ring via NIC2. There can be another traditional socket based interface for sensor management which is not a load bearing component. For superior throughput the detection rules should be

stored in memory which can be updated via the management interface. This scheme can eliminate expensive run-time system calls in the sensor.

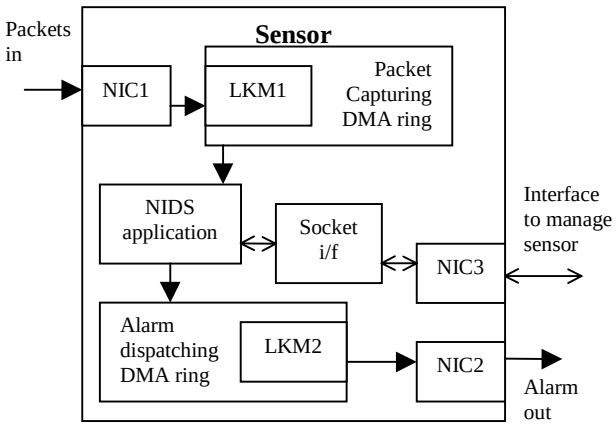


Fig. 4: Sensor with DMA ring

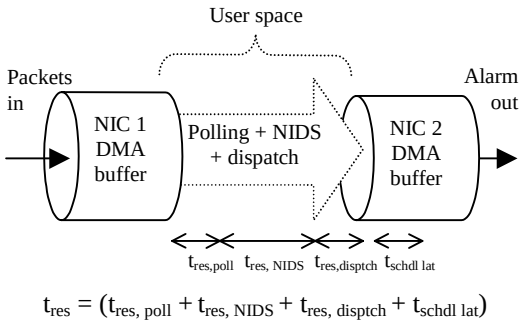


Fig. 5: Proposed sensor task model

The task model for the sensor with the proposed proposed interfaces is presented by Fig. 5. The model is simpler, it consists of a single task thread and only two buffers, one for receiving the packets, the other for sending out the alarm. The NIC firmware task places the packet in the DMA buffer without involving the host CPU. The user space task which run on the host CPU, periodically polls the DMA buffer, picks up the packets and analyzes them. If there is an alarm to be sent, the same user task constructs the alarm packet and places it in the transmitting DMA buffer. The transmitting network card periodically polls this DMA buffer and on finding a packet, it sends it. There is no separate task thread in the transmitting portion. For sake of simplicity now onwards we only discuss the receiving side of the architecture.

In this architecture we have reduced the average response of the packet capturing task by employing the following tactics:

- Simple hybrid interrupt polling mechanism with fixed low frequency polling is employed. Polling is invoked by a hardware based periodic timer.
- Switching between interrupt and polling modes based on the expected packet arrival rates. If the expected packet arrival is larger than a threshold then operation is switched to polling mode, if it is less, then interrupt mode.
- A single user space high priority task thread is employed to carry out the packet receiving, packet analysis and detection task.
- A shared staging area is employed for all packet processing work. Given the system's worst case response jitter, a right size for the staging area is chosen so that it does not overflow. A big common staging area avoids the need for data copy or buffer transfer (explicit zero copy) operations across layers/domains and the need for memory allocation in real-time to replenish the DMA buffer.

The task response jitter is reduced by the following strategies:

- By choosing a low fixed polling rate the interrupt load on the system was limited even at high packet rates. Low interrupt load results lower system response jitters [23].
- An OS is employed which either provides real-time support for user space task or have low average context switch and low task response jitter within the operational range.

The hybrid polling mechanism ensures that, for lower packet rates, the receiver works asynchronously with interrupts and manifests low packet delivery and detection latency. At

high packet rate, the synchronous polling mechanism avoids CPU resource wastage due to high interrupt service overheads. The reclaimed CPU time is utilized for useful packet analysis and detection activities. Fixed polling period avoids the need to program the timer in real-time, hence CPU resources are also conserved. The timer needs to be set only once when the polling is started. High resolution, sub millisecond level software timers are not available in Linux. So a hardware timer is required to get microsecond level periodicity, to serve the 100Mbps or 1Gbps networks. Using a hardware timer instead of Linux kernel's software timer also makes the polling engine less vulnerable to high kernel scheduling latencies and jitters. The hardware timer delivers events to the OS kernel by interrupt mechanism. Low polling rate achieves two purposes. A low polling rate causes a lower interrupt load, hence the OS can maintain its soft real-time response behavior at lower interrupt rates. In addition to this, infrequent polling limits the effect of polling overhead. The timer ISR delivers the timer interrupt event to the poll engine and invokes the polling cycle. Each polling invocation has an associated overhead due to interrupt servicing and context switching between timer ISR and polling engine task. With a lower polling cycle rate multiple packets can be serviced in a single poll cycle. This amortizes the polling overhead over many packets and minimizes the effect of polling overhead.

Using a single thread obviates the need for three things: the interim buffers; context switching times and task balancing between multiple tasks. Single task thread and single staging area/ buffer has simple dynamics and therefore it is amenable to scaling and tuning unlike Linux, NAPI and PFRING. Under high network load, the user space thread performs all the required operations - polling for packets and detection.

This monolithic architecture allows a single staging area for all packet processing and analysis operations. De-layering also reduces the need for interfaces between the layers and the associated function call overheads. A single staging area would require that the DMA buffer be shared with user space and be made the staging area. The DMA buffer should be big enough to contain the effect of task response jitters and avoid buffer overflow. Mapping DMA memory area to user space avoids the data copy operations between NIC layer to kernel and subsequently kernel to user space.

By collapsing all the receiving, processing/detection and transmitting tasks into a single thread, the entire problem is contained to a single tractable problem of a managing a single task response jitter. The problem concerning user space task response jitter is addressed by reducing the interrupt load on the system or by choosing an appropriate OS. Avoiding interrupt based operation minimizes interrupt load in the system which drastically reduces response jitters. However this problem can be best addressed at the OS level, not by a solution at the application level. Therefore an OS with better real-time response is also needed. In case of vanilla Linux 2.4 kernels, very high interrupt latency and kernel-to-user-space context switching times cause DMA buffer overflow at high packet rates. Hence we used Redhat 8 kernel which has lower kernel-to-user-space context switching times. We also ported this to a hard real-time platform concocted with RTAI co-kernel and vanilla Linux. DMA ring not only improves efficiency but also enhances the real-time behavior of the system, especially when a general purpose OS like Redhat is used.

We choose a polling period of 122 µsec (8192 Hz) which is in the order of the packet delivery latency (88 to 110µsec) for the Linux kernel for the given hardware. A higher polling period can be chosen for a higher speed CPU. The polling is clocked by the RTC timer interrupts. Instead of RTC, LAPIC timer can be used. The poll engine computes the average packet rate, which is weighted in favor of most recent rates. The system uses interrupts at low packet rates and switches to polling mode if the average packet rate is beyond a threshold (set to 8.3Khz). By setting an optimum threshold, mode transition can be controlled and system performance can be further optimized for a given platform. A single large DMA buffer of 1024 packets for Redhat 8, or 64 packets for RTAI, is implemented as a circular queue (ring) and is shared between kernel and user space by memory re-mapping (mmap). The polling engine runs as a

high priority real-time task (RT FIFO, priority = 99). The user space and DMA memory is locked in the RAM so that none is swapped out to disk to deteriorate real-time response.

Thus, we have employed additional strategies to avoid the following problems - context switching between ISR, softirq and user space; buffer overflow; lack of execution balance between tasks; higher kernel-to-user-space latency; data copy; memory allocation; high polling or interrupt service overhead and in-deterministic polling period. By having a single buffer, single user space thread, low fixed polling rate and shared staging area on a low latency OS we have addressed most of the problems associated with NAPI and PFRING. Next section presents the implementation for Linux Redhat 8 and RTAI.

5. Architecture and implementation

5.1 Overview

The detailed architecture with all its components is presented in Fig. 6. The whole architecture is packaged in three components - (i) a Loadable Kernel Module (LKM) which would replace Linux kernel's existing network card driver, and (ii) a user space polling engine and (iii) the user space application that analyzes/processes the packet.

All codes that either interacts with hardware directly or exploit any shareable kernel resources are made part of the kernel and packaged in the LKM. These codes are invoked by the user space polling engine via standard open(), mmap() and ioctl() system calls. The open() call initiates the communication mechanism between user space and kernel space (LKM) code, the mmap() call creates a shared memory space between the two domains during setup time. Whereas ioctl() calls are used to exchange control between these two domains during operation. Two parameter can be passed in an ioctl() system call, one can distinctly identify the particular kernel code to call, the second can be the argument. The ioctl counterpart in LKM, implements a switch conditional structure, which switches control to the appropriate case statement code block based on the first parameter. To deploy this interface in a NIDS application, like "snort", which is based on the "libpcap" packet capturing library, the user space polling engine should be implemented within the library, by modifying its two source files "pcap-linux.c" and "pcap-int.h". The polling engine will be initialized when the "pcap_open_live()" library function is called from within the NIDS application, and the polling will begin when the "pcap_read_linux()" function is called. The rest of the packet analysis will take place when the poll engine returns control to the NIDS application through the usual libpcap "callback()" function route.

Loadable Kernel Module : The NIC specific and hardware related codes are kept in the LKM. The LKM takes advantage of the rich set of standard Linux kernel API to set up the NIC hardware and DMA ring. This LKM once developed can be reused with different applications. The LKM included code segments that defines all the standard components for initialization (constructor), exit (destructor), and other Unix file operations. The LKM developed for Linux and RTAI are slightly different. This sub-system includes the code to: (i) define LKM components; (ii) setup the NIC hardware and allocate software resources; (iii) setup the DMA ring; (iv) configure and start the NIC for operation; (v) map DMA ring to the user space; (vi) ioctl function codes that enables the polling timer; (vii) ioctl function code that enables the NIC interrupt; (viii) the interrupt service routine (ISR) for NIC interrupt; and (ix) ISR for polling timer interrupt. The LKM also includes the DMA ring/buffer, which is setup in the kernel memory.

The user space polling engine is generic, simple and portable component that can work with any NIC or host architecture. It includes - (i) code to request mapping of the DMA ring in user space; (ii) the polling engine logic; and (iii) "call back" function.

The existing Linux NIC driver for the chosen network interface card (NIC) hardware was modified to implement the required LKM. To deploy this architecture, the LKM is simply

loaded in memory instead of the existing NIC driver and the user space polling engine is executed from the user space. No other modification in OS or in the hardware is needed.

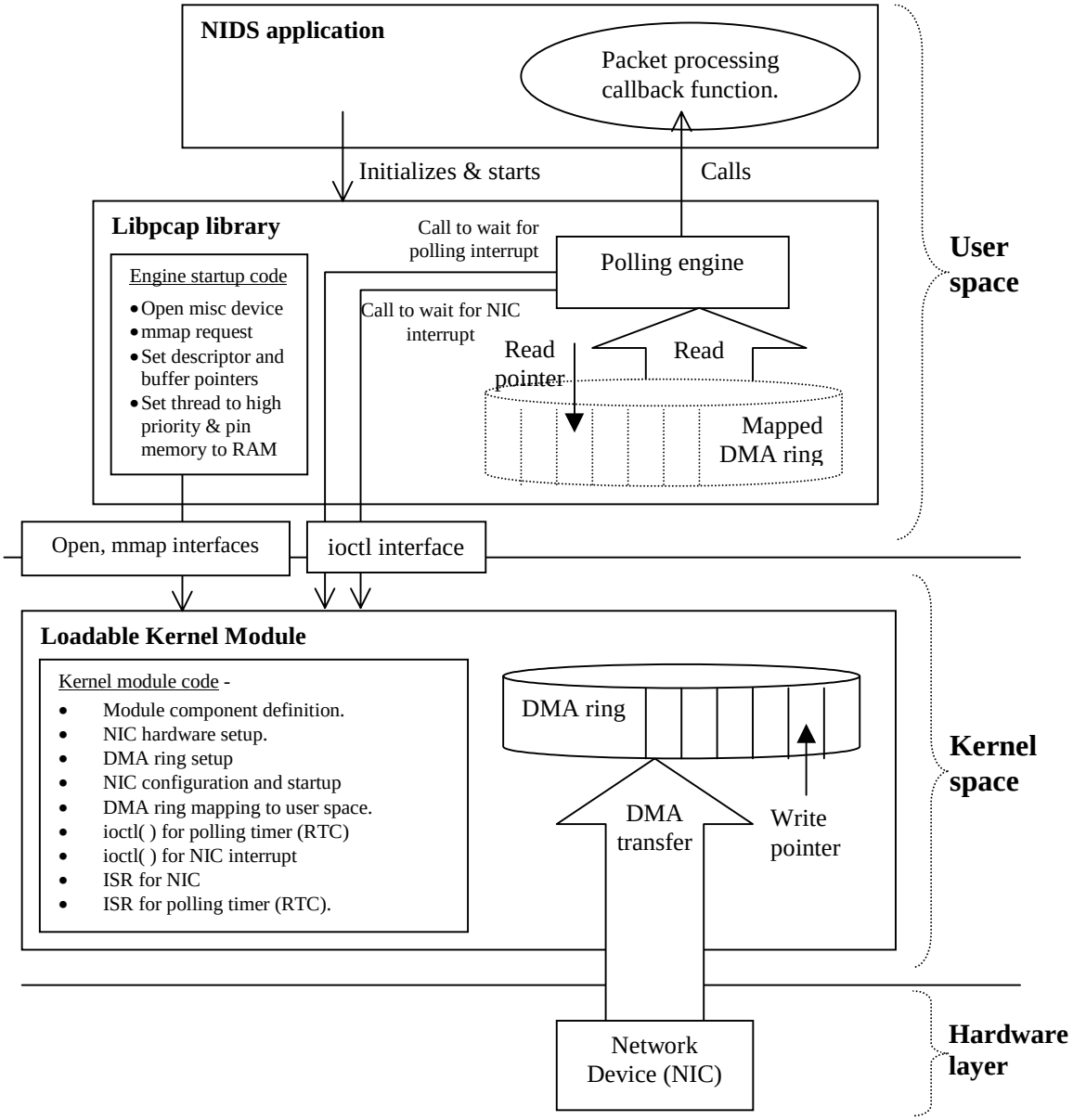


Fig. 6: DMA ring architecture for NIDS application

5.2 Implementation in Linux

All the codes were developed in "C" and compiled by GNU "gcc" compiler (ver 3.2) that came with Redhat 8.

LKM and setup operations: As Linux NIC drivers generally do not provide any mechanism for user defined interfacing with the user space application threads, so a mechanism was added to the driver. This was achieved by declaring/ setting up a miscellaneous device and implementing the device operations in the LKM. The kernel module/ driver can be accessed from user space though this file or device node id of the miscellaneous device. Functions like ioctl, mmap open and release counterparts define the implementations for the file/device operations inside the LKM. The module initialization code registers the "miscdevice" with the unused available major inode number 10 and minor number 240, whereas the exit code de-

registers it. Details about the modifications carried out on the existing NIC driver are presented later. Some of the implementation details are specific to the chosen NIC, which was 3Com905B-TX for our case.

The code that set ups the NIC hardware performs a series of setup tasks to initialize the NIC hardware and allocate software resources to manage the NIC hardware, according to the following sequence:

- The LKM inserts itself, as the driver for the NIC, to a link list maintained by the OS by pci_module_init() kernel API call.
- Wakes up the NIC hardware and assigns an interrupt line (IRQ) and enables the NIC by pci_enable_device().
- Allocate memory resources, i.e. data structures to manage a Ethernet device, by alloc_etherdev().
- Allocates I/O ports for the NIC by request_region() API call.
- Enables NIC's DMA controller and configures the same by pci_set_master() and pci_write_config_byte() calls.
- Sets up the ISRs for NIC and hardware timer interrupts. The periodic polling timer is implemented by Motorola's MC146818 based real time clock (RTC) chip available on Intel x86 motherboards.
- Sets up the data structure in the host memory for constructing the DMA ring. The DMA ring consists of two parts, the descriptor ring and packet buffers (Fig. 7).

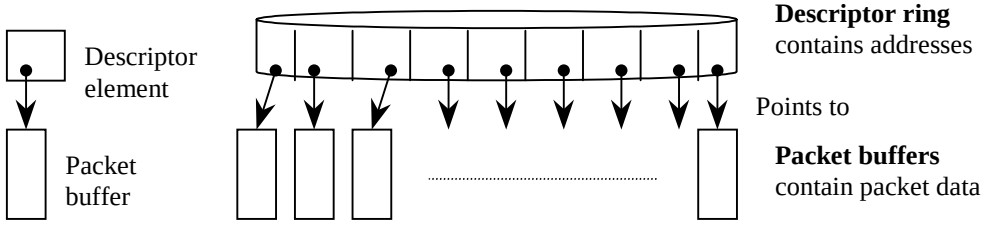


Fig. 7: DMA buffer data structure

The descriptor ring is a chain of descriptor elements. Each descriptor element points to a contiguous memory segment, the packet buffer, which holds the packet data. The NIC places the packet in the packet buffer by DMA transfer, so the packet buffer has associated DMA mapping. The descriptor part holds the addresses and status information about the locations which actually contain the packet data. The descriptor elements provide the target host memory addresses to the NIC so that the NIC can make DMA transfer to those locations. The descriptor ring is constructed as part of NIC hardware initialization task. The packet buffers are allocated later when the NIC is started up for operation. Fig. 8 explains the descriptor details and its functioning. Each descriptor element comprises of four fields. The "next" field is a pointer which points to the next descriptor element. Bit 0-12 of the "status" field indicate the length of the packet in bytes which is transferred by the NIC to the host memory. "Address" field points to the location of the corresponding packet buffer in host memory "DMA region" for which DMA mapping has been generated, and the "length" field stores the length of the packet buffer in host memory. The "next" field of the last descriptor item points to the first descriptor item, so that the end is wrapped around to form a ring. The consecutive descriptor elements are in a contiguous memory segment so that NIC can iterate over them with small memory strides. Smaller memory strides minimize DMA address cycles and the PCI and FSB bus holdup during burst DMA operation. Once the descriptor ring is constructed, and the address of the entry point of the ring is transferred to the "up list pointer register" in NIC's onboard memory. The NIC can get the entry point on the ring, iterate over the

ring, fetch the descriptors from the host memory address, extract the address of the target location in the host memory and make DMA transfers to the target location. This data structure and mode of operation allows the flexibility to choose any DMA ring size. All the addresses stored in this data structure are true physical addresses.

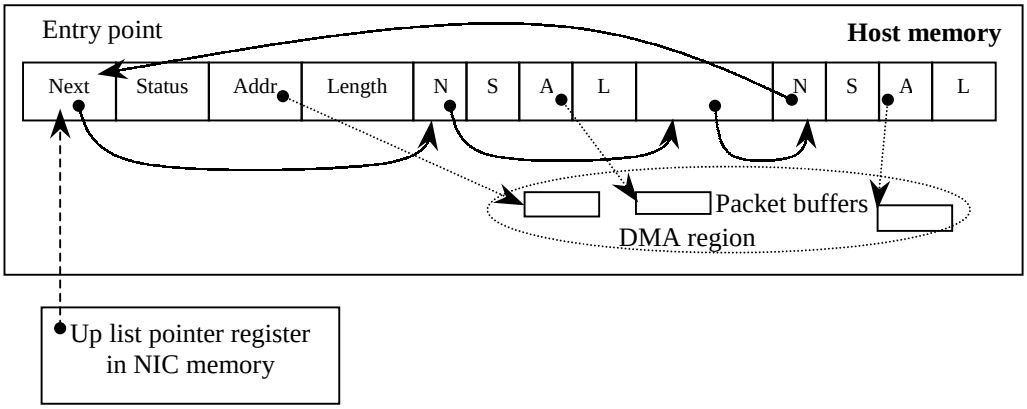


Fig. 8: Data structure for the receive descriptor ring

- The LKM sets up a watchdog timer, configures the media and media access protocol (MAC) parameters for the NIC by programming its EEPROM and reserves resources for the receiver FIFO buffer on the NIC hardware. It also initializes the receiver descriptor ring by resetting all the "status" fields.
- Configures and enables other NIC features like packet check-summing.

The DMA buffer is implemented as a circular queue in form of a ring. This circular DMA buffer is shared between kernel and user space by memory re-mapping, so that once the NIC firmware task places packets in the buffer then the user space polling engine can directly access the data placed in the DMA memory to process it. Bit 13-15 of the "status" field in the descriptor element provides the synchronization between the user space "consumer" thread which picks up the packets from this circular queue and the NIC firmware "producer" task, which is places it. A certain non-zero value in these bits signifies that the corresponding slot in the ring is filled with fresh packet ready for pick up. The NIC firmware task writes this value in these bits after DMA transfer of each packet. The user space "consumer" thread resets these bits after completely processing the corresponding packet. The user space thread operates a read pointer on the circular queue to mark the current packet for pick up and processing. Once the user space gets the entry point on the circular queue it sets up the read pointer and thereafter it simply iterate over the mapped DMA ring.

The packet buffers of DMA ring is setup during the NIC activation. The LKM utilizes a new function, dev_alloc_skb_from_page() to allocate a group of contiguous memory pages and constructs packet buffers from these memory pages. Another new function, dev_kfree_skb_from_pages() dissolves the packet buffer and frees up the memory pages. These two library functions were developed as part of the present work and can be reused to implement LKMs for other NICs. These two functions are analogous to existing Linux kernel API functions, dev_alloc_skb() and dev_kfree_skb(), which are used to allocate and free memory during construction and destruction of the packet buffers, "sk_buff".

When dev_alloc_skb_from_page() is called, it allocates a cluster of contiguous memory pages, puts them in a pool, and then constructs a single packet buffer from the first available page in the pool and returns it. In subsequent function calls, packet buffers are constructed from the pages available in the pool and returned. If the pool gets exhausted then a new cluster of contiguous pages are allocated in the pool. The memory pages were also pinned in the memory by SetPageReserved() Linux kernel API function call, so that the Linux memory

manager do not swap out these pages to the disk. Pinning the DMA memory is required for successful DMA ring operations. A Linux memory page is 4096 bytes long physically contiguous memory segment. Many memory management operations takes place with page level granularity. This function is different from the existing API function - dev_alloc_skb(), which allocates a contiguous memory segment from any available pages, therefore do not ensure that consecutive packet buffers are always placed contiguously within a page. Grouping packet buffers in pages is necessary to map them in the user space and disable their swapping to disk. Mapping and disabling page swap is performed by per page basis, whereas memory is allocated with byte level granularity. So this new memory allocation function was necessary. Putting all packet buffer in contiguous memory may also improve the cache hits and the CPU's translation look ahead buffer (TLB) efficiency. The other new function, dev_kfree_skb_from_pages() dissolves the packet buffer and frees up the page when all packet buffer from that page has been dissolved. It replaces existing "dev_kfree_skb()".

The DMA buffer sharing across kernel to user space border is achieved by a standard Linux (and Unix) mechanism, the mmap() system call. Both kernel and user space threads address memory by virtual memory addresses. The OS and the CPU carries out the virtual to physical memory address translation whenever memory is accessed. Normally the kernel and user space virtual memory addresses are mapped to two exclusive physical memory segments. But for memory sharing, a certain portion of physical memory which has already been mapped as kernel virtual addresses may be remapped as a segment of user space virtual addresses. To share memory, the user space thread makes a request by mmap() system call. In response, the kernel level mmap counterpart re-maps the requested number of pages to the user space by using remap_page_range() kernel API function call. The LKM implements the kernel counterpart of the mmap() function. This function remaps the kernel memory pages to contiguous user space virtual addresses. Fig. 9 presents these mapping relationships.

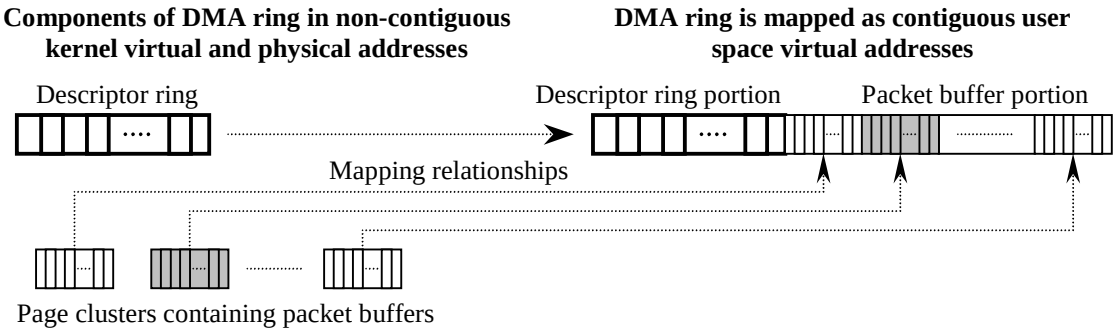


Fig. 9: Mapping relationship between kernel and user space DMA ring components

The polling engine: The polling engine is the heart of the architecture. The flowchart for the polling engine logic is presented in Fig. 10. The top portion of the poll engine logic, above the dotted line, is implemented in the user space, and bottom portion is implemented in the LKM. The portions of the polling engine that directly interact with the hardware or kernel software resources run in the kernel space.

When there is no packet to process, the poll engine blocks itself to yield the CPU, instead of spinning in loops and wasting CPU time. The polling engine relies on the task blocking/ unblocking mechanism provided by the kernel API (refer text box, below). The polling engine runs in an endless loop. To block itself, the user space polling engine thread makes an ioctl() system call. The ioctl function that blocks to wait for the NIC interrupt is called from user space with the first argument defined as an integer constant which corresponds to "wait for NIC interrupt". The ioctl that blocks and wait for timer interrupt is similarly called with its first parameter defined as "wait for timer interrupt". The ioctl counterpart in the LKM blocks the current user space thread in a wait queue by making wait_event_interruptible() call.

Wait queue is a Linux mechanism to block tasks and wake them up in future. To block a task, it is inserted in the wait queue and to unblock it is taken out of the queue. A wait queue is constructed by DECLARE_WAIT_QUEUE_HEAD() kernel API macro call. A current task thread blocks itself by making wait_event_interruptible() Linux kernel API function call. This function blocks the current task thread only when a NIC or timer interrupt event has not yet happened, if an interrupt event has already happened then the current task does not block and the function returns immediately. The boolean variable that keeps track of the interrupt event is set in by the ISR. This variable is explicitly reset in the LKM's ioctl code to re-arm the mechanism, when the wait_event_interruptible() function returns. The blocked task residing in the wait queue is unblocked or waked up by wake_up_interruptible_sync() kernel API function call. A separate task thread, which is currently running, wakes up the blocked task by calling this wake_up_interruptible_sync() function. The waked up task is scheduled to run after the waking task thread exits. Upon waking up, the wait_event_interruptible() function, that was responsible for blocking the task, returns.

The blocked user space thread is woken up either by the periodic RTC timer interrupt or by a NIC interrupt event. The ISRs which run in response to the interrupt events actually wake up the blocked user space thread by wake_up_interruptible_sync() call. The ioctl call returns to the user space when the user space thread is woken up. The polling engine decides whether to block and wait for a NIC interrupt event or to block and wait for the RTC timer interrupt event.

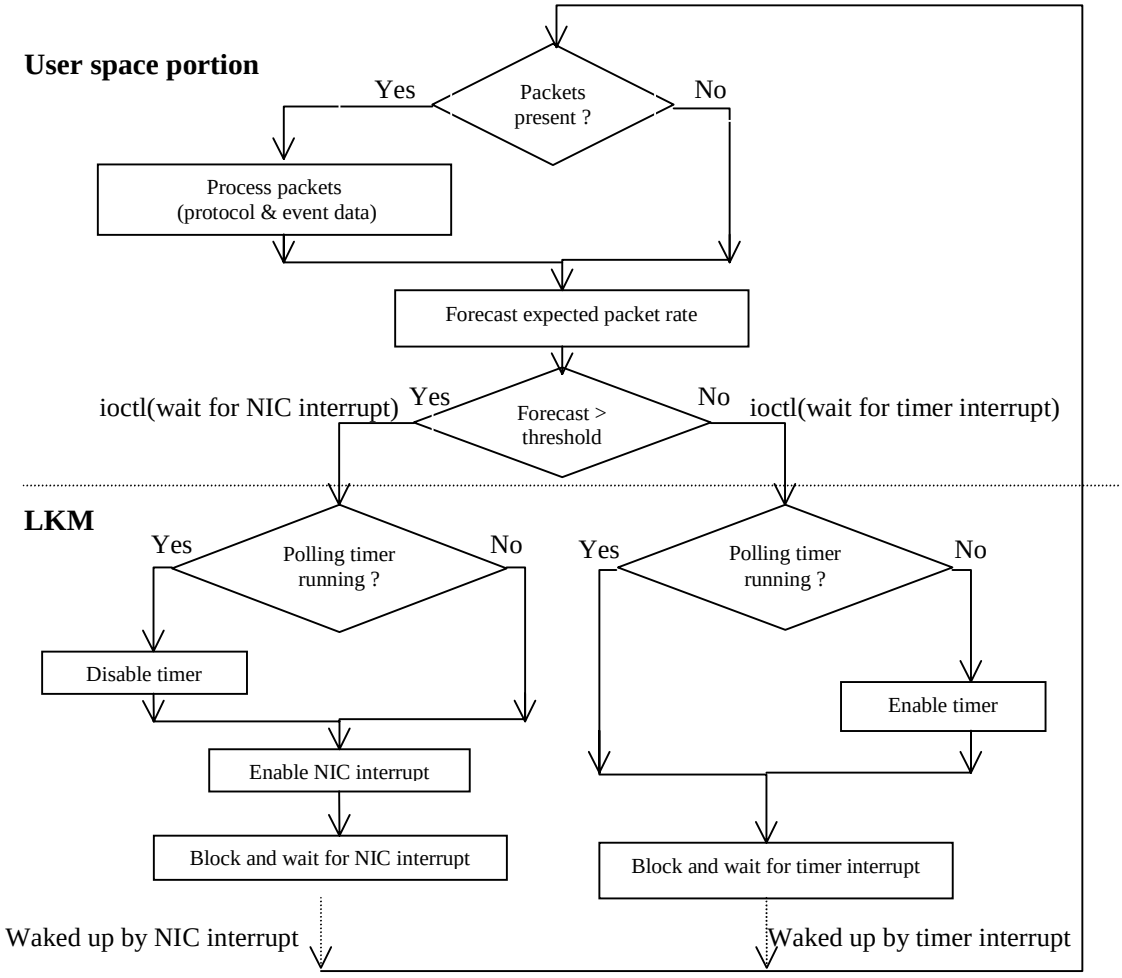


Fig. 10: Polling engine logic

The NIC raises interrupt as soon as it completes DMA transfer of a packet to the host memory. In every ISR cycle the NIC interrupt is disabled. This interrupt is enabled later by

the polling engine thread when it shuts down. The NIC interrupt in the ISR are disabled and enabled by directly programming the NIC hardware I/O ports. Before exiting, the NIC ISR wakes up the user space thread that runs the polling engine.

The periodic polling task is paced by hardware RTC timer interrupts. The RTC timer ISR reads the RTC register and then simply wakes up the user space thread and exits. The RTC register have to be read to keep the periodic timer running and generating interrupts. This is a specific and unwanted feature (for this situation) of Motorola's MC146818 based RTC which is available on every Intel x86 motherboards.

Other than blocking and waking on an event, these ioctl codes also interacts with the NIC and RTC hardware to enable NIC interrupt and the polling timer. The code that enables the NIC interrupt is inside LKM's ioctl implementation for "wait for NIC interrupt". This part of the ioctl implementation also disables the RTC timer before enabling the NIC interrupt (Fig. 10). The NIC interrupt is enabled by directly programming its I/O ports. The code that enables the RTC timer interrupt is part of ioctl implementation for "wait for timer interrupt". The RTC timer is enabled or disabled by programming its I/O ports. More details of these LKM ioctl implementations are explained, in the following paragraphs.

The polling engine logic is presented as pseudo code in Fig. 11 and the LKM ioctl logic is in Fig.12. After the circular queue and the read pointer have been set up, the user space program control enter the poll engine loop (Fig. 11). At the entry point of the loop the DMA buffer is checked for presence of packets, if packets are available then they are processed. NIDS application processing involves examining various portions of the packet. Whereas for other applications, packet processing will mean protocol processing and extraction of data payload. These operations can be carried out by parsing the packet and inserting appropriate pointers.

```
Carry out set up tasks;
Get circular queue entry point;
Set up the read pointer;

While (true) {
   if  (packets_present)  {
      perform packet processing;
     call function for detection tasks;
      increment read pointer;
   }
   else{
         compute arrival rate;
         if (arrival rate > threshold) {
           ioctl (wait for polling timer interrupt);
         }
         else {
              ioctl (wait for NIC interrupt);
         }
}
```

Fig. 11: User space polling engine logic

```
ioctl (command) {
   switch(command) {
       ……..
      case "wait for polling timer interrupt":
              if (timer is not enabled) {
                  enable polling timer;
              }
              sleep in wait queue;
      break;

      case "wait for NIC interrupt":
         if (timer is enabled)
                  disable polling timer;
         enable NIC interrupt;
         sleep in wait queue;
}
```

Fig. 12: LKM ioctl logic

The hybrid poll engine operates in interrupt mode when it expects low packet arrival rates and switches to polling operation when it expects higher packet arrival rates. In every poll cycle, after completion of the event data processing task the expected arrival rate is forecasted. Packet arrival rate is a random variable, so future packet rates cannot determined with certainty, they can only be predicted. This forecasted packet rate is weighed in favor of most recent arrival rates. If the forecasted rate is below a certain threshold then the poll engine blocks itself and waits for the NIC interrupt. If it is above the threshold then the polling engine blocks and waits for the hardware timer interrupt which invokes the polling cycle. This threshold is set to be equal to the fixed poll period. To cover a very wide range of packet rate, say, corresponding to 100Mbps to 10 Gbps wire speeds, two or more thresholds and corresponding different polling rates can be implemented. To block and wait for the NIC and

timer interrupts, the poll engine makes corresponding ioctl call to the LKM as described earlier.

The ioctl code in the LKM, that corresponds to "wait for NIC interrupt", disables the timer if it is enabled, then it enables the NIC interrupt and finally it places the current thread in a wait queue to block it (Fig. 12). This thread is woken up by the next NIC ISR execution. On waking up, the waked up thread returns the ioctl function call made by the poll engine and the poll engine continues with the next iteration of its endless loop.

The ioctl code in the LKM, which corresponds to "wait for RTC interrupt", checks whether the hardware timer is enabled or not, and if it is already enabled, then it places the current thread in the wait queue to block it (Fig. 12). If the hardware timer was not enabled, then the ioctl code enables it before blocking the current thread. When the next timer interrupt invokes the timer ISR, the timer ISR wakes up the sleeping thread. This thread then returns the ioctl function call back to the polling engine, so that the polling engine can proceed with the next iteration of the poll loop.

The polling engine may not switch over to polling mode immediately after getting first few rapidly arriving packets. The polling engine observes a sufficient number of packet arrivals, and if it is convinced that the average packet arrival rate is indeed high, only then it switches to polling mode. Similarly the polling engine takes the decision to fall back to interrupt based operation only after some time has elapsed from the instant the packets arrival slows down. This inertia is implemented in the polling engine as a part of the algorithm which computes the average arrival rate. Mode switching is quite expensive and this behavior avoids frequent switching between modes due to overall system jitter. This strategy conserves the CPU.

The polling engine startup tasks involves: opening the miscellaneous device; making a mmap() request to the LKM to map the DMA buffer in the user space; setting the descriptor and buffer pointers; setting the current user space thread to high priority; pinning the current process memory to RAM by "mlockall()" call; and finally starting the polling engine.

5.3 Forecasting packet arrival rate

In a hybrid interrupt-polling architecture, sharp and correct mode transition is the key to superior performance. A sharp mode transition means that the architecture should switch modes within a narrow packet rate band. Sharp mode transition is achieved by threshold based mode switching. Correct mode transition means immunity to noise. Inter-packet period is a random variable. Time to time, a group of packets may arrive in a closely packed bunch even though the average and most likely inter-packet period may be large. This transient phenomena might misguide the system into frequent mode switching, thus this phenomena is considered as noise. Task response jitters in the receiving host also contribute to this noise. Frequent mode switching wastes CPU resources hence it is undesirable. The system should be immune to packet rate transients but should recognize a more permanent increase in packet rate. Only if sufficient number of packets have arrived with low inter-packet period then the system should quickly switch to polling mode. The packet rate forecasting mechanism manifests all these desirable system behaviors. The subsequent paragraph presents the implementation details of the forecasting mechanism.

Packet arrival rate can be represented by its reciprocal, the inter-packet period. All comparisons and computations in the system are carried out with this inter-packet period representation. Measurement of time between two consecutive packet arrival events gives the inter-packet period. It is impossible to determine the exact moment of a packet arrival in a practical system which suffers from event delivery latencies or which sometimes operate in polling mode. Therefore in lieu of determining the exact moment of packet arrival, the time when a packet is detected in DMA buffer is noted and utilized for estimating inter-packet period. Intel Pentium CPUs provide a register which gives the CPU clock cycle count. Time period between two intervals is estimated by computing the difference between the register

readings corresponding to beginning and end of the interval and then dividing the difference by the CPU clock frequency.

A polling or packet processing task cycle is either initiated by NIC interrupt or a timer interrupt. Packets may be detected and processed in a processing cycle or may not be detected in the current cycle. A processing cycle in which packets are detected and processed is an "active" cycle. More than one packets may be detected and processed in an active cycle. The time period between completion of the last "active" processing cycle and the current "active" one gives an estimate of the total inter-packet period for a group of packets which are detected and processed in the current cycle. So the most recent average inter-packet time can be estimated by dividing this measured time interval by the number of packets processed in the current active cycle. As more than one packets can be processed in the current cycle so a moving average effect is implicit in this computation. This is explained by Fig. 13 and embodied in the following equation (Eqn. 1).

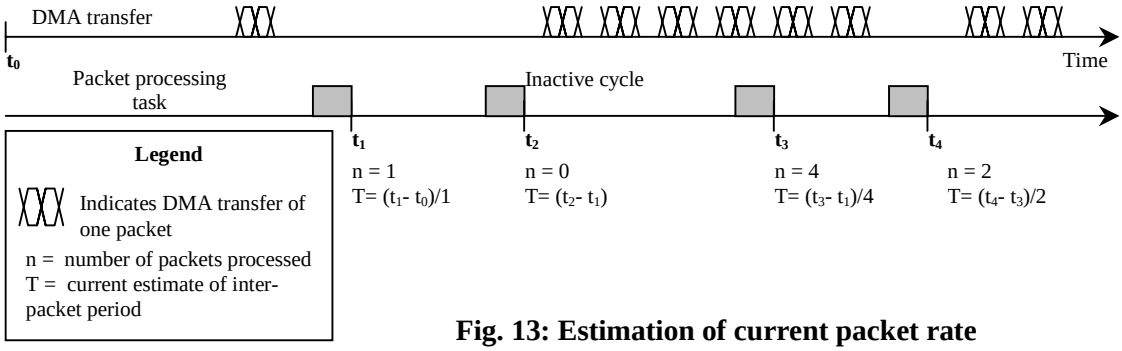


Fig. 13: Estimation of current packet rate

The most current inter-packet period "T_{curr}" is given by -

$$T_{curr} = \frac{(C_{curr} - C_{prev})}{s_{CPU} * n} \quad \text{when } n > 0$$

$$= \frac{(C_{curr} - C_{prev})}{s_{CPU}} \quad \text{when } n = 0$$

.....Eqn.1

where -

C_{curr} is the current CPU clock count

C_{prev} is the CPU clock count of the previous active processing cycle

s_{CPU} is the CPU speed

"n" is the number of packets processed in the current cycle

The C_{curr} and C_{prev} terms represents the time instants in terms of CPU clock cycle count. The most recent packet arrival rates give better estimate about the arrival rate of immediate future. So the packet rate forecast is biased to most recent arrival rate, which is estimated based on sufficient number of inter-packet period observations. If the recent inter-packet period estimate is based on detection of a sufficiently larger number of packets in the current active cycle, then the most recent inter-packet period estimate itself can be considered as a good forecast. But if the number of packets processed in the current active cycle is small so that they do not form a good sample size, then the weighted average of most recent inter-packet period and the average of past inter-packet periods forms an alternative forecast. These computations are expressed by the following equations.

The predicted inter-period rate T_{pred} is given by a weighted average expression -

$$T_{pred} = \alpha * T_{curr} + (1 - \alpha) T_{prev} \quad \text{.....Eqn. 2}$$

where T_{prev} is the inter-packet period forecasted in the previous cycle

The term, T_{prev}, carries information about the historical packet arrival rates. This term has been computed by moving average method. Thus the prediction of expected inter-packet period is based upon a combination of weighted average and moving average methods. The weight "α" of the weighted average expression can be varied depending on "n", the number of packets processed in the current active cycle. This variable weight scheme implements noise rejection and correct mode switching behavior.

The weight "α" is defined as -

$$\begin{aligned} \alpha &= 1 && \text{when } n > n_2 \\ &= \alpha_2 && \text{when } n_2 > n > n_1 \\ &= \alpha_1 && \text{when } n_1 > n \end{aligned} \qquad \text{.....Eqn 3}$$

where $\alpha_1 < \alpha_2 < 1$, α_1 and α_2 are fractional constants and n_2, n_1 integer constants

Due to jitter in task response times and packet arrival rates, "n", the number of packets processed is a random variable. When more packet arrive between two active cycles, the sample size is larger, then more confidence can be placed on the most current inter-period estimate, that, it represents the current inter-packet period. The level of this confidence may be increased if the most current inter-packet period is based on larger number of packets, therefore a larger value of "α", i.e. α_2 may be chosen. This scheme manifests an inertial behavior to avoid frequent mode switching. The inertial behavior can be fine tuned by choosing different combinations of α_1, α_2, n_2 and n_1. These are settable parameters. Fig. 14 presents part of the polling engine code that predicts the packet arrival rate.

```
While (true) {
   if  (packets present)  do packet analysis and detection tasks;
    else{
        get current CPU clock count;
        time elapsed since last active cycle = (current clock count- previous cycle clock count)/CPU speed ;
        if (packets processed > 0)
             current packet period = time elapsed since last active cycle / packets processed;
           else
              current packet period = time elapsed since last active cycle;
           if (packets processed < n1) alpha = alpha1;
            else if (packets processed < n2) alpha = alpha2;
                      else alpha = 1;
           predicted packet period = alpha * current packet period + (1-alpha) previous packet period;
           previous packet period = predicted packet period;
           if (predicted period < threshold)  ioctl (wait for polling timer interrupt);
            else  ioctl (wait for NIC interrupt);
        }
}
```

Fig. 14: Pseudo code for forecasting packet arrival rate

5.4 Settable parameters

The implementation allows several compile time settable parameters in the LKM: page cluster size during page allocation for DMA ring; packet buffer size, that defines the maximum packet size that can be received; and size of DMA ring. The run time settable parameter in the user space drivers are: polling rate; packet rate threshold at which the system would switch its mode of operation; and the prediction algorithm parameters - α_1, α_2 , n_2 and n_1.

A larger page cluster size will allocate larger clusters of contiguous pages which will increase the extent of contiguousness in the DMA buffer. The IP packet buffer sizes (1500 and 9000 Bytes) do not align with Linux memory page (4096 Bytes) boundaries. So some memory is wasted if too small cluster size is chosen. To reduce this memory wastage a bigger cluster is

preferable. On the other hand bigger clusters are difficult to allocate, DMA ring allocation failures would be likely with very big cluster sizes. The memory page contiguousness may have other favorable or unfavorable effects in the memory management system, which may affect the performance of the system as whole. Study of these effects is beyond the scope of this paper, but this settable parameters provides flexibility for such study and optimization.

Packet buffer size setting determines the size of the biggest packet that can be received. Maximum size requirement for fast 100 Mbps Ethernet is 1500 Bytes, for gigabit Ethernet it is 9000 Bytes. This settable parameter allows customization of the architecture for such applications. NIDS application will need the maximum packet buffer size. Provisioning bigger packets means reserving and pinning larger kernel memory, which might adversely affect the system performance unless a larger memory and corresponding support in kernel is available.

A larger DMA ring is necessary to mask adverse effect of jitters in task response and context switching times if a Linux kernel without real-time support is used. A larger buffer reduces the likelihood of buffer overflow. A larger buffer also allocates a larger memory which may adversely affect the system performance beyond a certain level. The settable parameter provides flexibility to set the optimum buffer size for a given OS kernel and hardware.

The polling rate determines the worst case packet delivery latency and the CPU utilization. For a general purpose OS like Linux it also indirectly affects the task response time and polling period jitters. Higher polling rate improves the average packet delivery latency performance to certain extent, because DMA buffer is checked for packets more frequently, hence packet idle time is lower. However higher polling rate increases the CPU utilization sharply due to increased timer interrupt servicing and polling overheads. Due to unavailability of CPU resources, packet receiving tasks are delayed which shows up as higher jitter in packet delivery latency. So for a given CPU speed, a tradeoff has to be made during setting the polling rate. A settable polling rate allows the required flexibility to study these effects and arrive at the optimum figure for a given OS kernel and hardware. The threshold at which the polling engine makes a transition from interrupt based operation to polling mode and back, is generally kept same as the polling rate, to get a smoother transition between operation modes. A combination of higher value of - α_1 , α_2 , n_1 and n_2 yields superior responsiveness and yet manifests better immunity to transient noise. We determined the optimal values based on some quick simulation and empirical tests.

5.5 Start up and run time operations

A device node for the miscellaneous device has to be created from the user shell by a "mknod" command with major number 10 and minor number 240. Instead of the Linux NIC driver the LKM is loaded in the memory by "insmod". After this the LKM waits for the user space driver to start up. The NIDS application initializes the polling engine at startup. Once started, the polling engine examines the DMA ring and upon finding it empty it makes an "wait for NIC interrupt" type ioctl call to the LKM. The corresponding ioctl call blocks in the LKM and the current user process (the polling engine task) thread is put to sleep in the wait queue. The system is now ready to receive packets, it will wake up whenever a packet arrives. A timing diagram associated with run time operation is presented in Fig. 15.

When the first packet arrives (Fig. 15), the NIC transfers this packet to the DMA ring and raises an interrupt. The NIC ISR disables the NIC interrupt and wakes up the blocked "wait for NIC interrupt" type ioctl. The user space thread is waked up and ioctl call returns to user space. If no more packets arrive by the time the user space thread finishes processing this single packet, the user space thread computes that the inter-packet period is higher than the threshold so it makes a "wait for NIC interrupt" type ioctl call. The ioctl counterpart in LKM enables the NIC interrupt and then blocks the current user space thread till the next NIC interrupt. The next NIC interrupt wakes up the user space thread and the ioctl call returns to user space. In the mean time more packets may arrive and may be transferred to the DMA ring

by the NIC, but no more interrupts will be raised. The waked up polling task finds these packets and process them. If there are no pending packets in the mapped DMA ring, the polling engine computes the expected inter-packet period. Due to the inertia the polling engine might still decide not to switch mode and still call the "wait for NIC interrupt" type ioctl which enables the NIC interrupt. So there might be another NIC interrupt arriving leading to another interrupt driven processing cycle.

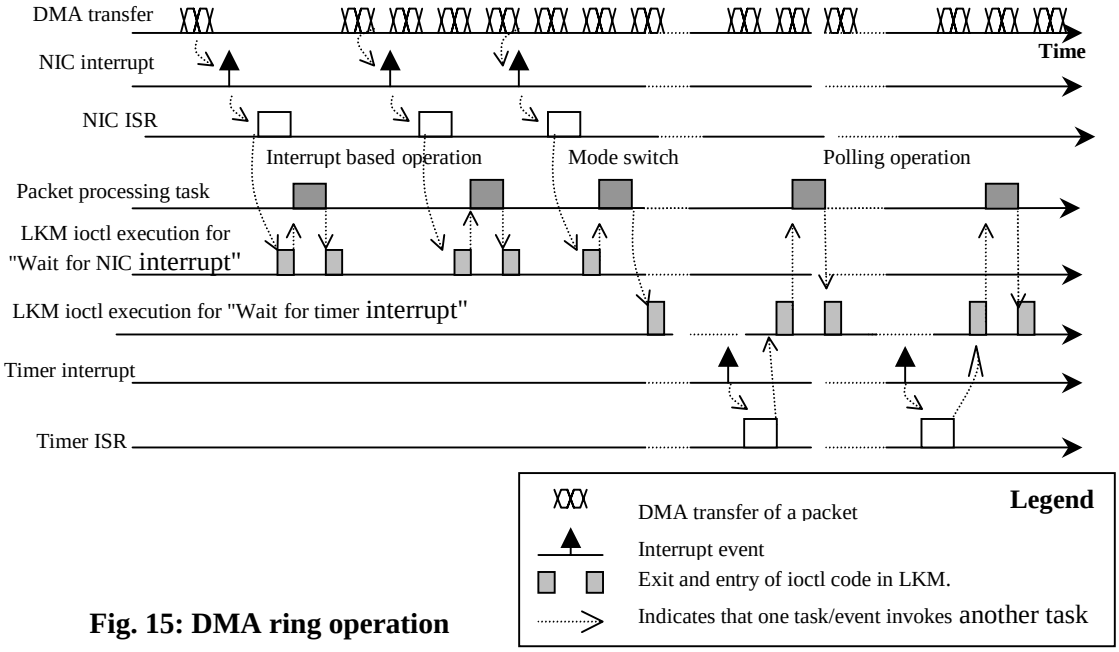


Fig. 15: DMA ring operation

If sufficient number of packets have arrived rapidly and if the expected inter-packet period is lower than the inter-packet threshold, then the polling engine decides that this time it has to switch mode and calls the ioctl of type "wait for polling timer interrupt". The corresponding ioctl code in the LKM first checks whether a polling timer is already running or not. As the polling timer has not been activated earlier, therefore it starts the periodic polling timer. Once the periodic polling timer is started the ioctl blocks itself in the same waiting queue as before.

Now onwards the NIC interrupts are not raised though packets may keep on arriving in the DMA ring. Periodically timer interrupts arrive, and the timer ISR wakes up the sleeping user process and the polling engine. Upon waking up the polling engine process the packets available in the DMA ring. As the packets continue arriving at high rate, so the polling engine decides to continue in polling mode and calls "wait for polling timer interrupt" type ioctl.

If the arrival rate is high then the polling operation is continued in the same manner as described above, else, the polling engine makes a "wait for NIC interrupt" type ioctl call to the LKM to initiate interrupt based operation (not shown in Fig. 15). The corresponding LKM ioctl code stops the polling timer, enables the NIC interrupt and then blocks in the same wait queue till the NIC raises another interrupt.

5.6 Implementation choices and rationale

Some aspects of the architecture can be implemented by alternative means. The most important implementation decision is the choice of OS. Other aspects are: choice of hardware timer; method to disable and enable interrupt; memory sharing mechanism; task blocking and unblocking mechanism; choice of algorithm to predict packet arrival rate and decide switching between interrupt and polling modes; placement of application specific processing code; and measurement of time. Implementation methods, which improved performance, reduced development effort or improved portability, were chosen.

This architecture has limited vulnerability from high interrupt latency and kernel-to-user-space context switching times. These latencies will delay the user space polling task, which clears the DMA buffer. Bigger DMA ring size may only retard the buffer overflow to higher packet rates, but it will not solve the problem. None of the vanilla 2.4 kernels are suitable due to their high task response jitters. Other than Redhat 8 and RTAI-vanilla Linux combination the probable OS candidates are: Redhat 8 (custom 2.4.18) Linux kernel, 2.6 preemptable kernels, Linux based RTOS which support hard real-time in user space, like PSDD with RTLinux (FSMLabs) and LynxOS (Lynux works). 2.6 kernels are claimed to have bounded jitters [23], but 2.6 based Fedora core 3 kernel unexpectedly failed to work satisfactorily, due to high task response jitters. RTLinux and LynxOS have their own proprietary commercial kernels and therefore were too constraining, so those were not explored. Next few paragraphs illustrate why Redhat 8 and RTAI-LXRT are suitable.

Redhat 8 kernel have smaller task response jitters within the given operation range therefore could be used with the proposed architecture. This was primarily due to two reasons: the system wide interrupt rate was low even during high packet arrival rate; and the architecture reclaimed CPU resources so that the adverse impact of CPU overload on response jitter was lower. System wide periodic interrupt rate was low because RTC (8.2Khz) and PIT timer (512Hz) interrupt rates were the only two sources. The normal background aperiodic interrupts due to hard disk, AGP display card, mouse and keyboard activity were never high enough to cause any problem. The jitter behavior of Redhat 8 Linux was quantified to ensure the performance of the proposed architecture operating on it. Jitter in the polling RTC timer period in user space was noted at very high packet rate, (148 kpps) (Fig. 16).

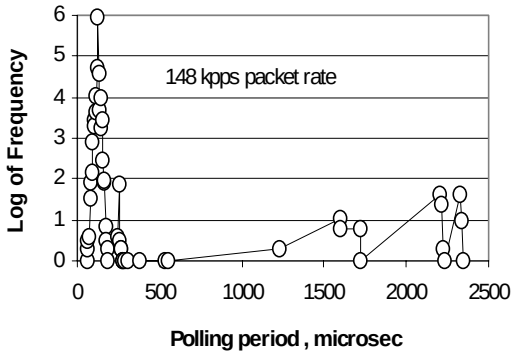


Fig. 16: Redhat 8 timer jitter
(122 microsec period, on PIII 333Mhz)

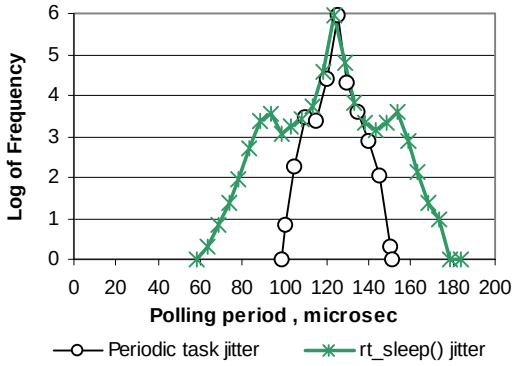


Fig. 17: LXRT-RTAI timer jitter
(122 microsec period, on PIII 333Mhz)

The RTC polling timer periodicity was 122 microseconds, the observed worst case period was 2345 microseconds. The maximum theoretical packet arrival rate possible on a 100 Mbps network for 64 Byte packets is $100*10^6/(64*8)$ = 195 kpps. The corresponding inter-packet arrival rate is 1/195 ≈ 5 microsecond. Number of packets that can accumulate in the DMA buffer due to worst case jitter is 2345/5 = 469 packets. So a DMA buffer size of 1024 is sufficient to contain the effects of this jitter. A DMA ring size of 2048 (safety factor of 4) allowed receiving of over four billion packets without any loss over an 8 hour continuous operation at 148 kpps packet rate with Redhat 8. The VGA display power saving features was kept on, which caused large jitters during power shutdown-wake-up cycle. Redhat 8's success in this stress test further confirmed its suitability of. Whereas 2.6 kernel based Fedora 3 failed.

Similar experiments with two different kind of timers with user space tasks on RTAI-LXRT yielded the better results (Fig. 17). These two timer options are discussed in details later. The timer period was set at 122 microsecond, whereas the observed worst case period was 185 microsecond. Under such situation the maximum number packets that can accumulate in the DMA buffer is 185/5 = 37. So with LXRT, a DMA buffer size of 64 was found sufficient.

Other than RTC, other hardware timers are also available, like LAPIC timer, available in every CPU and the IO-APIC timer in P3 and P4 architectures. Programming these APIC timers requires more effort hence for validation purpose RTC timer was used which is comparatively simple to program. Some NICs have programmable timers, but their reliable operation is doubtful, especially when the NIC gets loaded with high packet arrival rates.

Interrupt masking sti(), cli() operations in the CPU were not used to disable-enable interrupts. Because masking in the CPU would allow the interrupts to enter the programmable interrupt controller (PIC) system and strain it. Interrupt disable-enabling at the source hardware is better generic approach, because if multiprocessor hardware is used because there would be no contention about which CPU's interrupt has to be masked. The network card interrupt can also be enabled or disabled by programming the LAPIC in case of uniprocessor system or IO-APIC in multiprocessor system. But correctly programming these APICs requires more effort than programming NIC I/O ports. However programming the APIC to disable-enable interrupt makes the implementation more generic across different NICs, but it is also difficult to make the code portable across different types of motherboards (without redundant codes that determine the PIC organization). This is because interrupt is routed differently in different motherboards. Some boards use LAPIC, some use IO-APIC to route interrupts, sometimes some interrupts are not routed through APIC but through 8259 based legacy PICs. Interrupt disabling-enabling in 8259 based PIC chips are slow operations, the response is slower than 2 microsecond, whereas programming the NIC employs two I/O operations, a read and a write, which takes a total time around 1 microsecond to complete.

Instead of using ioctl system call and wait-queue mechanism to block and unblock task, poll() function call to the LKM device node could have been made. A poll() system call on a device blocks if there is no data available to read in the device memory, and the poll() call returns when some data is available. But this poll() mechanism was not chosen as it deteriorates performance [3]. Poll() mechanism involves signaling to wake up tasks, which is complex and consumes CPU cycles.

A modified form of the algorithm presented in [5] could have been used instead of the implemented algorithm. However that algorithm did not favor more recent estimates to predict the packet arrival rate. Therefore that algorithm was not chosen. Standard Linux/Unix system call, gettimeofday() and kernel API call do_ gettimeofday() are not used to measure time because these are inaccurate, they do not have sub-microsecond granularity and consumes lot of CPU cycles. On the other hand CPU clock cycle counter provides nano-second level time measurement precision and consumes only a few CPU cycles. Older Intel CPUs (P3, P2) are 32 bit ones, so instead of full 64 bits, only lower 32 bits of the clock cycle counter value was used in the calculation to avoid expensive 64 bit maths.

5.7 Implementation in RTAI-LXRT

About RTAI-LXRT: The proposed interface was also implemented for RTAI-LXRT 3.1 with vanilla Linux 2.4.24 for uni-processor Intel systems (Fig. 18). RTAI or "Real-Time Application Interface" is not a full-fledged RTOS. RTAI co-kernel just provides real-time services in any vanilla Linux platform [1]. The RTAI API is essentially meant for the kernel level tasks. With RTAI, any task that has to run in hard real-time has to be a kernel task that is coded as loadable kernel module (LKM). LXRT is a separate module which sits on top of RTAI and vanilla Linux to expose the RTAI services in user space (Fig. 18). With LXRT in place, a user space task can also enjoy hard real-time privileges. LXRT is employed because the polling engine and data processing tasks can be deployed as a Linux process and still be run as a hard real-time task. With LXRT, the hard real-time task does not need to be segregated from the rest of the application and put in the kernel as a LKM. This hard real-time user space task is not a Linux user space task in strict sense, it is a LXRT task which simple enjoys the Linux user space memory protection privileges. It is not scheduled by Linux scheduler but handled by RTAI scheduler in conjunction with LXRT. The code is developed,

compiled and loaded in memory from Linux as a Linux process. Once loaded, the task is migrated from Linux to LXRT domain by making a special LXRT API function call (Fig. 18). This scheme has an apparent limitation. Once the task has been migrated and it cannot call any function which leads to a Linux system call and yet maintain its hard real-time status. If it ever makes such a Linux system call then the task migrates back to Linux's soft real-time domain. This essentially means that the hard real-time LXRT task has to do without Linux system calls or functions which lead to a system call. This is a fair tradeoff, because even in Linux real-time tasks, one has to avoid system calls as they lead to poor performance. In any case the proposed "DMA ring" architecture is designed to operate without making any system calls during runtime.

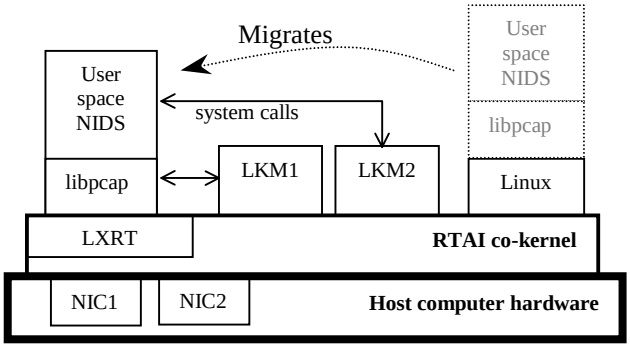


Fig. 18: System architecture with RTAI-LXRT

Specific changes required in user space code for porting to LXRT: The LKM and user space components are similar to those of Linux Redhat 8 implementation, with only few implementation differences. In the user space component the "wait for NIC interrupt" and "wait for RTC interrupt" type ioctl() system calls cannot be used, they are replaced by two LXRT API calls to achieve the block and wait for NIC interrupt or timed wakeup events. There could be many choices among the LXRT API that could achieve this same purpose, however not all lead to superior performance, this aspect is discussed in next sub-section. The following additions are made in the setup portion of the code. After the Linux thread priority is changed to the highest value and memory is pinned to RAM by "mlockall()" call, a LXRT task is created by a LXRT API function call "rt_task_init()". The RTAI clock timer mode is set to periodic by calling "rt_set_periodic_mode()", and then it is started with "start_rt_timer()" call. The current task is migrated from Linux domain to RTAI-LXRT domain by calling "rt_make_hard_real_time()". Ancillary "libpcap" functionalities which depend on Linux system calls can not be supported with LXRT hard real-time task.

Specific changes required in LKM for porting to LXRT: In the LKM, instead of using the Linux ISR, ISR for RTAI have to be used for NIC and RTC timer interrupts. These basic ISR codes for NIC and RTC interrupts remains the same as in Redhat 8, only registration procedure and their prototypes are different. Instead of registering the ISRs with Linux, they are registered with RTAI kernel by calling the "rt_request_global_irq()" RTAI API function. After registering the ISRs, they are enabled in the PIC hardware by calling "rt_startup_irq()" and "rt_enable_irq()" RTAI API functions for each ISR registered. In each ISR code, the interrupt have to be acknowledged by calling "rt_ack_irq()" RTAI API function in addition to normal acknowledging procedure for the NIC and RTC hardware. This "rt_ack_irq()" has to be called at the exit of the ISR. The NIC and RTC interrupts are only to be handled by the RTAI domain therefore these interrupts are not released to Linux. Interrupts are not passed on to Linux domain unless an explicit 'rt_pend_linux_irq()" RTAI API function call is made. These ISRs wake up the blocked LXRT. There are few implementation choices available to achieve this, but not all of them yield good performance.

5.8 Implementation choices in LXRT and rationale

Pacing the poll engine: The polling engine task runs as a periodic task when polling is enabled, but it runs as an aperiodic task when running in interrupt mode. The polling engine task needs to frequently switch back and forth between periodic and aperiodic modes very quickly (Fig. 19). To service 100 Mbps network, this switching has to complete within few microseconds, a larger switching time will increase the task response time, will require bigger DMA buffer and consume CPU time.

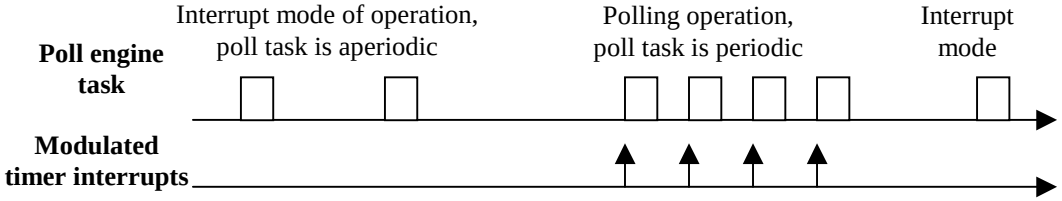


Fig. 19: Poll engine task switching

Four alternatives were considered to implement this mechanism. In LXRT, a task can be defined as periodic by calling the "rt_task_make_periodic()" LXRT API function. Once the task is made periodic it will run with the predefined periodicity. No mechanism could be found to stop and start this periodic task at will. No API could be found that can turn the periodic task back to aperiodic mode. However theoretically this task can be stopped if the timer itself can be modulated to start and stop at will (Fig. 19). As the timer drives the periodic task, so it is hopped that this strategy can indirectly control the task. Modulating the timer can be achieved by two ways: either by explicitly starting and stopping the periodic timer or by using a single shot timer. LXRT API functions are available to start/stop a periodic timer or to employ a single shot timer instead of a periodic one.

The RTAI kernel for uniprocessors is timed by 8254 chip based programmable interrupt timer (PIT). Programming this PIT as a single shot timer is costly, it involves an overhead of 15 to 20% of the time period [27], therefore running the timer as monoshot mode was not explored. An alternative second option is explicitly starting and stopping the timer by "start_rt_timer()" and "stop_rt_timer()" LXRT API calls. But stopping this timer caused preemption of the current task for 10 milliseconds, which is unacceptable. A probable explanation for this phenomenon is presented here. The RTAI co-kernel for uniprocessors is clocked by PIT, which is shared between RTAI and Linux. When the RTAI native kernel timer is set for 122 microsecond periodicity, the RTAI programs the PIT with 122 microsecond. RTAI receives these 122 microsecond interrupts and release them to Linux at 100 Hz rate (approximately). When the RTAI native kernel timer is stopped, the current task thread is preempted and the PIT is programmed back to 100HZ mode. The current task is again waked up at the next scheduling point which happens at the next PIT interrupt event, only after 10 millisecond. For RTAI co-kernels for SMP architectures, the native kernel timers are implemented by LAPIC timers, hence in case of RTAI for multiprocessors this phenomenon may not manifest and perhaps the periodic timer may be modulated. This requires empirical verification.

Due to the above mentioned phenomenon, the third option based on "rt_sleep()" LXRT API function was chosen. In this scheme the task is not defined as periodic, the timer is run in periodic mode with periodicity of 122 microsecond. When the polling task needs to block and wait for the next timer period, it calls the "rt_sleep()" function with a sleep time of 122 microsecond. So after 122 microsecond the polling task wakes up again to execute the next polling cycle. This sleep function blocks the current thread until the sleep time is elapsed, the logic is similar to Unix/Linux sleep mechanism. But "rt_sleep()" has far lower jitter than Unix/Linux sleep due to its real-time nature. This "rt_sleep()" mechanism yielded a little more jitter than the LXRT periodic task jitter, which is the best case (Fig. 17). Though "rt_sleep()" mechanism introduce higher jitters, but still it had to be chosen rather than declaring the polling task as periodic. This allowed the switching of the polling task between periodic and

aperiodic modes at will. The fourth option is to employ the RTC timer interrupts to pace the polling engine. DMA ring operation with RTC timer based polling was also implemented on LXRT to compare its performance against the "rt_sleep()" mechanism. For higher packet rates, during polling mode of operation, rt_sleep() with PIT timer operation consumes less CPU resources compared to RTC timer (Fig. 20). However as the PIT timer is always running, due to additional interrupt servicing overheads the rt_sleep() mechanism consumes higher CPU resources even when the system operates in interrupt mode. RTC timer options have similar response jitter profile as rt_sleep() option as presented in Fig. 17.

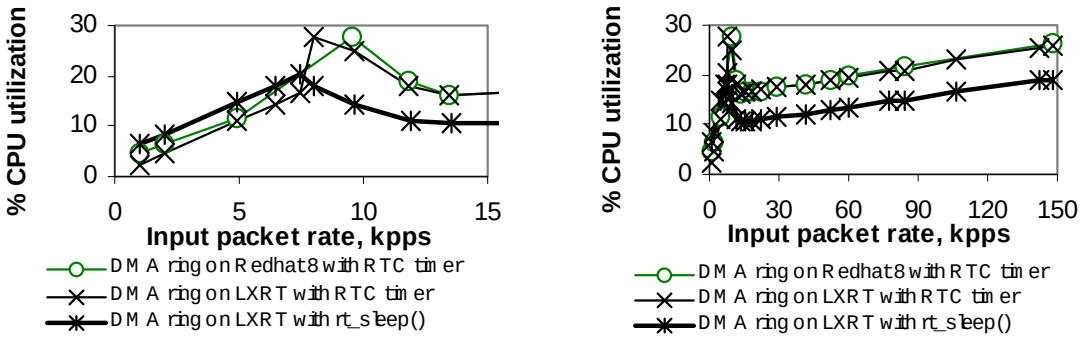


Fig. 20: CPU utilization comparisons at low and high packet rates

Waking up poll engine on NIC interrupt event: The NIC interrupt ISR needs to wakeup the blocked LXRT task. The blocking and waking up of LXRT task can be achieved by two alternate mechanisms. LXRT provides a semaphore, known as RTFIFO semaphore, which can be shared between the user space and RTAI kernel space. A task can take this semaphore only if it is free, if the semaphore has already been taken by another task, then the requesting task blocks till the requested semaphore is freed up by the other task. The requesting task unblocks when the semaphore is available. This binary semaphore is created with an initial value "0", which means semaphore has unavailable status. The LXRT polling task requests for the semaphore and blocks itself by making a "rtf_sem_wait()" LXRT API call. When an NIC interrupt arrives the ISR makes a "rtf_sem_post()" call to release the semaphore, i.e. change the semaphore value to "1". As a result the LXRT task wakes up.

An alternate way to achieve this same blocking waking up operation is to use task suspend-resume mechanism. In this scheme the LXRT task suspends itself by making "rt_task_suspend()" LXRT API call with its own task id as the argument. When a NIC interrupt arrives, the ISR resumes the blocked LXRT task by calling "rt_task_resume()" RTAI API function. The task id is registered with the LXRT so that the same task id is available in both kernel and user space.

Usage of RTFIFO semaphore was suggested in the RTAI documentation code examples, this scheme works fine as long as the interrupt rate is low, however it fails under high operation cycle rate above 6.5 kHz for the given PII 333Mhz hardware. Therefore the suspend-resume scheme was used in the present work. The RTC timer interrupt ISR also wakes up the LXRT task by similar suspend-resume mechanism.

5.9 Modifications carried out in the existing NIC driver

The existing Linux NIC driver was modified to implement the required LKM for Redhat 8 and this LKM was then ported to RTAI-LXRT. Most Linux PCI NIC drivers follow a similar pattern in code organization and operations, so the modifications can be localized within few specific well defined areas in the driver code and the modifications steps can be defined. The existing code for PCI resource, network media and NIC hardware management was retained as it is. These portions embody the NIC hardware specific knowledge. No modifications are carried within these codes. These required modifications are generic and applicable to most PCI NICs. Modification of the existing driver allows reuse of the existing open source code.

This avoids the need to know the NIC hardware specifics and develop the code to manage the NIC hardware from scratch.

Modifications for Redhat 8 implementation: A few new code segments are added in the existing module structure which are marked by "new". The code for the following operations were added in the following twelve areas:

(i) Data structure declarations: All additional data structures required to implement the LKM are added.

(ii) Module initialization code: Access to RTC timer ports are setup along with the ISR for the RTC timer interrupt. The miscellaneous device is also registered here.

(iii) Module exit and clean up code: RTC ports and interrupt are released and miscellaneous device is de-registered.

(iv) Device setup: Allocate memory for the miscellaneous device data-structure, compute memory requirement for entire descriptor ring in term of whole memory pages, reserve and pin memory pages for descriptor ring, and create the descriptor ring.

(v) Device initialization: Allocate memory for packet buffers in contiguous memory segments, pin those memory pages, setup/map these packet buffers for DMA transfer.

(vi) Device shutdown: Unmap DMA region, unpin and free the memory pages allocated to packet buffers.

(vii) Miscellaneous device "open" implementation (new): Increments the device usage counter.

(viii) Miscellaneous device "mmap" implementation (new): Maps the memory pages hosting descriptor ring and the packet buffers to user space.

(ix) Miscellaneous device "close" implementation (new): Decrements the device usage counter.

(x) Miscellaneous device "ioctl" implementation (new): The two ioctl functions corresponding to "wait for NIC interrupt" and "wait for RTC interrupt" are implemented.

(xi) Interrupt sub-routine for NIC: Generally packet receiving is implemented as a function which is called from the ISR. This single function call in the ISR code is replaced by two or three statements to disable the NIC interrupt and to wake up the blocked user space thread.

(xii) Interrupt sub-routine for RTC timer (new): RTC timer register is read to enable it for next interrupt (RTC specific feature) and blocked user space thread is woken up.

The more intricate modifications like packet buffer allocation and freeing operations were packaged inside the two library functions - dev_alloc_skb_from_page() and dev_kfree_skb_from_pages().

Modifications for LXRT implementation: The NIC driver developed for the DMA ring architecture for Redhat 8 was ported to work with RTAI-LXRT by making the following minor modifications, in addition to those enumerated above:

(i) Module initialization code: Instead of setting up the NIC and RTC timer interrupts for Linux, these ISRs are setup for RTAI by calling RTAI API function "rt_request_global_irq()" instead of the usual Linux kernel API function "request_irq()".

(ii) Module exit and clean up code: NIC and RTC timer interrupts are released from RTAI by calling " rt_free_global_irq()".

(iii) Interrupt sub-routine for NIC: The prototype of the existing Linux ISR is simply changed to the prototype for RTAI ISR, the internal code remains the same.

(iv) Interrupt sub-routine for RTC timer (new): Only the prototype is changed from Linux ISR format to RTAI ISR format, internal code remains as it is.

6. Performance evaluation

We compared performances of the proposed architecture [8,9] with the existing solutions - Linux (Redhat 8, custom low latency 2.4.18 kernel with 512HZ), NAPI (as in Redhat 8) and PFRING (patched 2.4.24 vanilla kernel with 100HZ), against five criteria - (i) load bearing capacity, in terms of maximum number of NIDS rules tolerated by each architecture without suffering any packet loss for the highest packet rates possible on 100Mbps, (ii) packet loss percentage (system throughput) for a given number of NIDS rules to execute, (iii) packet rate capacity in terms of maximum tolerable packet rate with no packet loss for a given number of rules to execute, (iv) detection latency for a given number of NIDS rules to execute and (v) worst case memory requirement. Detection latency is the time between the packet arrival event and the completion of the analysis/ detection task. Smaller memory footprint is desirable to get benefits of a limited cache size. All comparisons are presented for small packet sizes (64 Bytes) because small packets that create most stress to the system. Comparison for larger packet sizes can be found in [3,8]. We present the comparison against criteria (ii) to (v) for 160 detection rules, which involves pattern matching. This specific number of rules were chosen because it causes 0% loss for DMA ring. Packet loss of all other architectures is compared against the base case (0% in DMA ring). Higher load bearing and packet rate capacity, lower packet loss, detection latency and memory requirements are desirable features in a packet capturing architecture. All the loss and latency figures are computed based on observing at least 10^6 packets. More details about the setup can be found in [3,8].

Fig. 21 compares the load bearing capacity of DMA ring against Linux, NAPI, PFRING, PFRING with NAPI for different packet rates. These envelopes give the maximum number of NIDS detection rules that each architectures can bear without loosing any packets for a given packet rate. DMA ring shows higher load bearing capacity at higher packet rates compared to all other architectures.

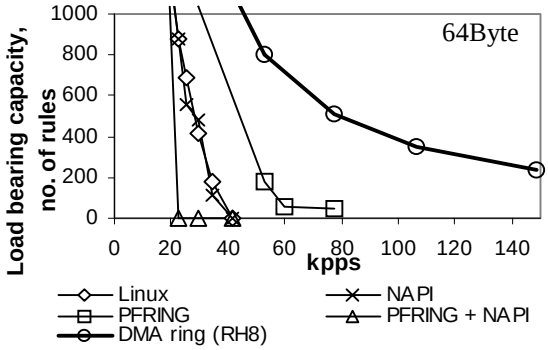


Fig. 21: Load bearing capacity

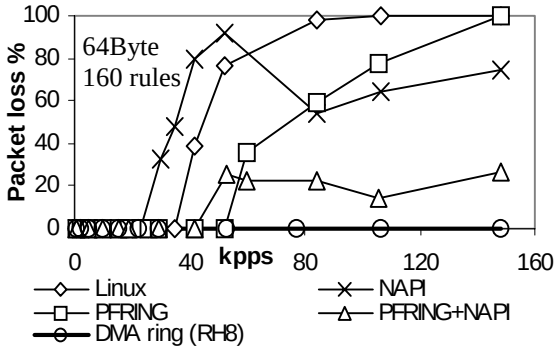


Fig. 22: Packet loss behavior

Fig. 22 compare the packet loss behavior of DMA ring against those of the others. System throughput can be derived from it. When the average throughout the entire range is considered (regardless of the packet loss), then the architectures can be ranked in the following order (best to worst) - DMA ring, PFRING with NAPI, PFRING, NAPI, Linux. These architectures are ranked in terms of the maximum tolerable packet rates for no loss (Table I).

Fig. 23 compares the worst case latencies in log scale for various architectures at different packet rates. Both, at higher and lower packet rates, DMA ring appears to be the best, except for a narrow band between 8kpps and 37kpps, where DMA ring on Redhat scores second position. Except DMA ring all other architectures introduce severe jitter in the detection

latency, whereas DMA ring manifests a low latency value. By introducing high jitter, these other architecture deteriorates the load bearing capacity at small packets sizes (Table I), so only DMA ring can be used when real-time, mission critical packet capturing is involved.

Table I: Maximum Tolerable No Loss Packet Rate

(160 rules, 64Byte packet)

Rank	Architecture	No loss capacity
1	DMA ring (best)	148 kpps
2	PFRING	52 kpps
3	PFRING + NAPI	18.5 kpps
4	Linux	35 kpps
5	NAPI (worst)	23 kpps

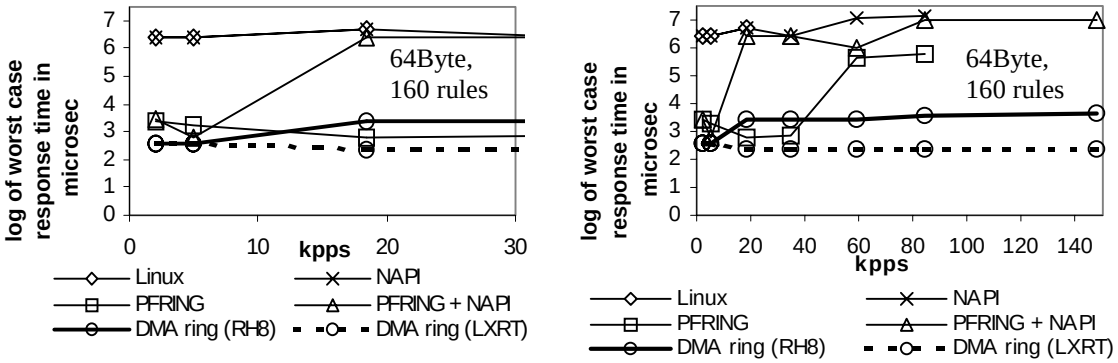


Fig. 23: Worst case detection latency at low and high packet rates

Table II ranks the architectures on basis of worst case memory utilization criteria, when these architectures are provisioned to capture and transmit a maximum size of 1500 Byte IP packets. DMA ring uses significantly less memory than PFRING architectures. Memory requirement of DMA ring on Redhat 8 is higher than NAPI and Linux because DMA ring requires a large DMA buffer of size 1024. However high memory requirement is not intrinsic to DMA ring and it can be reduced by implementing it on real-time platform like RTAI-LXRT.

Table II: Worst Case Memory Requirement

(for 1500Byte packet)

Rank	Architecture	Worst case memory requirement in Bytes
1	DMA ring (LXRT) (best)	98304
2	NAPI	524288
3	Linux	1150976
4	DMA ring (RH8)	1572864
5	PFRING + NAPI	6532608
6	PFRING (worst)	6536064

DMA ring is found to be overall superior in terms of higher load bearing capacity and packet rate capacity; lower packet loss, detection latency and memory utilization. This demonstrates the advantage of using - lower polling rate, single user space task thread to avoid context switching, and shared staging area.

7. Advantages of the proposed interface

DMA ring architecture reduces both per packet and per byte packet processing cost component, thus it yields superior performance at small and large packet sizes, unlike [6,7].

DMA ring is a scalable software solution that does not require any modifications in the network adapter, host hardware, or in the OS kernel. It does not depend on any low availability, difficult to administer custom patch for the OS kernel. DMA ring can work on either Redhat 8 or any RTAI-LXRT vanilla Linux combination. The performance of the single thread, single buffer architecture scales linearly with CPU speed. In its present form it can work on any uniprocessor or multiprocessor Intel Pentium hardware. With a single minor modification it can be ported to work with other non-Intel CPUs that supports either Redhat 8 or RTAI-Linux. The timing measurement based on Pentium clock cycle counter (TSC) is the only CPU dependent function that needs to be modified. It can work with any PCI NIC with commodity features. The modifications carried out on the NIC driver to obtain the LKM component are well defined and can be applied to any generic Linux NIC driver. The intricate modifications are packaged into two functions, which are similar to existing Linux kernel APIs. Other NIC drivers can be modified using these functions.

DMA ring is a non-adaptive hybrid interrupt-polling mechanism which is partly implemented in the user space and partly in the NIC driver. It involves only a single task thread, thus it reduces context switching in the real-time critical path. The differentiating features of DMA ring are - low fixed polling rate (polling overheads amortized over many packets), lower context switching frequency, no border crossing for data, no copy, no memory allocation operation, simplified computation to decide operation mode (polling or interrupt), no need for task balancing and simple buffer overflow management (moderate increase of DMA buffer size solves the problem).

8. Related works and discussions

Discussions on related research about improving network interfaces [3,4] and their evaluation can be found in [3]. So performance related works on NIDS are only discussed here. Much work on improving performance of NIDS is based on applying efficient pattern matching algorithms or signature detecting approaches. Some NIDS vendors claim that protocol analysis improves both efficiency and effectiveness [29]. Appropriate signature keyword selection by neural networks and automatic rule clustering have been shown to improve NIDS effectiveness [18,30]. A combination of static pattern matching along with dynamic checking for protocol analysis have been also claimed to improve effectiveness [16]. Faster string matching algorithm can also improve performance [11,31]. Interestingly [32] had pointed out that along with improving string matching algorithm, custom NIC drivers may improve "snort's" throughput, but unfortunately the idea of custom NIC was discouraged on grounds that it would reduce the portability of snort. We disagree with this position due to following reasons: (i) Linux NIC drivers can be modified by applying a patch, (ii) making patches for couple of Linux drivers for popular NIC hardware are moderately trivial if the guideline and the library functions, developed by us are used, as presented in this paper. Installing and maintaining a NIDS itself demands quite some expert time, patching or modifying the existing NIC drivers does not require much additional effort. The additional driver modification effort is worth the performance gain if it can avoid distributed NIDS deployment and the associated complexities. Implementing detection, traffic splitting or preprocessing functionalites in the NIC firmware itself may reduce the computation load in the host CPU and improve the throughput. NIC based NIDS has been shown to be feasible [33]. The NIC may implement onboard traffic splitting logic, it can receive inbound traffic and then route the outbound traffic to specific sensors without involving the host CPU. This strategy may boost the throughput of a host based traffic splitter. Simple traffic splitting rules can be implemented on the NIC firmware, whereas the software running on host CPU can implement more complex rules. Favorable effects of cache memory and adverse effects of deep CPU pipelines on sensor performance have been discussed in [34] and [20]. Reference [12,15,20,21] addressed distributed deployment of NIDS to tackle high traffic volume. Reference [22] addressed the same bottleneck problem by suggesting multithreaded NIDS architecture on multiprocessor hardware. Reference [35] also suggested use of multiprocessor hardware on improving packet

capturing, however this work did not demonstrate the realizable benefits by presenting any implementation. This work is largely theoretical and had not examined the key issues of practical packet capturing and multi threaded architecture to sufficient depth. None of these works addressed or attempted to solve the root cause of this bottleneck, i.e. limitation of the OS and its device interface. On the other hand our work identified the causal factors behind this bottleneck and proposed a viable solution to improve the performance at the component level, in the traffic splitter and in the detection sensors.

Reference [34] compared performance of NIDS applications on a variety of hardware which provides some insights to choose the right hardware. Reference [36] compared performance of a NIDS on a variety of OS. On the other hand [31] compared performance and execution behavior of different content matching NIDS rule sets.

9. Conclusion

In this paper, we presented a case study on Network Intrusion Detector System (NIDS) to demonstrate the importance of an efficient real-time software interface. This study articulated how a NIDS can be hardened against high bandwidth network attacks by adopting a real-time interface for packet capturing and alarm dispatching. We briefly discussed the key causal factors behind the packet capturing bottleneck in Linux and the existing packet capturing solutions: NAPI and PFRING. The details of the design, implementation, deployment and operation of the proposed interface for open source Linux and RTAI were presented. We discussed most of the relevant nuances of a system design and implementation for wire speed packet capturing. Some performance comparisons were presented to demonstrate how this mechanism may obviate the need for additional or more powerful hardware for monitoring moderate packet rates. Finally, we discussed related works on NIDS performance improvement.

References

[1] RTAI Official website. https://www.rtai.org/

[2] Hard Real-Time Networking for Real-Time Linux. http://www.rts.uni-hannover.de/rtnet/

[3] A. Biswas, P Sinha, “A high performance packet capturing support for alarm management systems”, Proce. of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems, 2005.

[4] P. Wang, and Z. Liu, "Operating system support for high performance networking, a survey", http://www.cs.iupui.edu/~zliu/doc/os_survey.pdf

[5] B. Vanderpool, and Z. Smith, "A linux implementation of HIP," Project report, University of Wisconsin, MD, http://web.demigod.org/~zak/documents/college/ece750-report/pdf, 1998.

[6] M.N. Thadani, and Y.A. Khalidi, "An efficient zero-copy I/O framework for Linux," Sun Microsystems Laboratories Inc., CA http://research.sun.com/techrep/1995/smli_tr-95-39.pdf. 1995.

[7] B. Murphy, S. Zeadally, and C.J. Adams, "An analysis of process and memory models to support high-speed networking in a UNIX environment," Proc. of the USENIX 1996 Annual Technical Conference. San Diego, CA, 1996.

[8] A. Biswas, P Sinha, “A high performance platform for intrusion detection sensors,” Proce. of the 4th IASTED International Conference on Communications, Internet and Information Technology, Cambridge, MA, 2005.

[9] A. Biswas, P Sinha, “On improving performance of Network Intrusion Detection Systems by efficient packet capturing,” Proce. of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada, 2006.

[10] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," Proce. of 7th USENIX Security Symposium, San Antonio, Texas, 1998. http://www.ece.cmu.edu/~adrian/731-sp04/readings/paxson99-bro.pdf

[11] C.J. Coit, S. Staniford, J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of Snort," Proce. of DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01. Vol. 1, pp. 367 - 373, June 2001.

[12] C. Kruegel, F. Valeur, G. Vigna and R. Kemmerer, "Stateful intrusion detection for high-speed networks," Proce. of 2002 IEEE Symposium on Security and Privacy, pp. 285- 293, 2002.

[13] G. Judge, A Verus,"FPGA Architecture Ups Intrusion Detection Performance," Sep 03, 2003, http://www.commsdesign.com/printableArticle/?articleID=16502099

[14] L. Deri, "Improving passive packate capture : beyond device polling", http://luca.ntop.org/Ring.pdf, 2004.

[15] L. Schaelicke, K. Wheeler and C. Freeland, "SPANIDS: A scalable network intrusion detection load balancer," Proce. of the 2nd Conf. on Computing frontiers, Ischia, Italy, pp. 315 - 322, May 2005.

[16] R. Sekar, Y. Guang, S. Verma, T. Shanbhag, "A high-performance network intrusion detection system", Proce. of the 6th ACM conference on Computer and communications security, Nov. 1999.
http://dbvis.fmi.uni-konstanz.de/members/panse/seminar_ws0203/pdf/sekar99highperformance.pdf

[17] S.A Yemini, S. Kliger, E. Mozes, Y. Yemini and D. Ohsie, "High speed and robust event correlation," Communications Magazine, IEEE, Vol.34(5), pp. 82 - 90, May 1996.

[18] C. Kruegel and T. Toth, "Automatic Rule Clustering for improved, signature-based Intrusion Detection," tech. report, Distributed Systems Group, Technical Univ. Vienna, Austria.
konstanz.de/members/panse/seminar_ws0203/pdf/sekar99highperformance.pdf

[19] A.P. Foong, T.R. Huff, H.H. Hum, J.P. Patwardhan, G.J. Regnier, "TCP performance re-visited".
http://www.cs.duke.edu/~jaidev/papers/ispass03.pdf.

[20] I. Charitakis, K. Anagnostakis, and E. Markatos, "An active traffic splitter architecture for intrusion detection," Proce. of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, pp. 238 - 241, Oct. 2003.

[21] N. Desai, "Optimizing NIDS Performance," at http://online.securityfocus.com/infocus/1589

[22] B.Haagdorens, T. Vermeiren and M. Goossens, "Improving the performance of Signature based Network Intrusion Detection Sensors by Multi-threading," http://dasan.sejong.ac.kr/~wisa04/ppt/5A1.ppt

[23] P. Laurich, "A comparison of hard real-time Linux alternatives." http://linuxdevices.com/articles/AT3479098230.html.

[24] C. Williams, Linux scheduler latency, 2002.
http://www.linuxdevices.com/articles/AT8906594941.html

[25] Salim, J. H. and Olsson, R., "Beyond softnet", 5th Annual Linux Showcase & Conference, Oakland, CA, pp. 165-172, 2001.

[26] L. Deri, "nCap: Wire-speed Packet Capture and Transmission," Proce. of the 3rd IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON), Nice, France, 2005.

[27] E. Bianchi, L. Dozio, P. Mantegazza, "A Hard Real Time support for LINUX."
http://www.aero.polimi.it/~rtai/documentation/reference/rtai_man.pdf

[28] J. Kiszka, "Real-Time Ethernet on Top of RTAI," www.rts.uni-hannover.de/rtnet/download/RTAI-Meeting04_RTnet.pdf .

[29] Network ICE, "Protocol Analysis vs. Pattern Matching in Network and Host Intrusion Detection Systems," http://www.seclib.com/seclib/ids.general/Protocol_Analysis_vs_Pattern.pdf

[30] R. Lippmann and S. Cunningham, "Improving intrusion detection performance using keyword selection and neural networks," Computer Networks, vol. 34, pp. 594-603, 2000.

[31] S. Antonatos, K.G. Anagnostakis, E.P. Markatos, and M. Polychronakis, "Performance analysis of content matching intrusion detection systems," in Proce. of 2004 International Symposium on Applications and the Internet, pp 208 - 215, 2004.

[32] N. Desai, "Increasing Performance in High Speed NIDS, A look at Snort's Internals,"
http://www.linuxsecurity.com/resource_files/intrusion_detection/Increasing_Performance_in_High_Speed_NIDS.pdf

[33] M. Otey, R. Noronha, G. Li, S. Parthasarathy and D. K. Panda, "NIC-based intrusion detection: A feasibility study," http://dmrl.cse.ohio-state.edu/papers/ICDM02-ws.pdf, November 27, 2002 .

[34] L. Schaelicke, T. Slabach, B. Moore, & C. Freeland, "Characterizing the performance of network intrusion detection sensors," Proc. of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID), Pittsburgh, PA, pp. 155-172, 2003.

[35] G. Varenni, M. Baldi, L. Degioanni, and F. Risso, "Optimizing packet capture on symmetric multiprocessing machines", Proc. 15th Symposium on Computer Architecture and High Performance Computing, São Paulo, Brazil , pp. 108 - 115, 2003.

[36] F. Risso, and L. Degioanni, "An architecture for high performance network analysis," in Proce. of Sixth IEEE Symposium on Computers and Communications, pp. 686 - 693, July 2001.