A High Performance Real-time Packet Capturing Architecture for Network Management Systems

Amitava Biswas

A Thesis

in

The Department

of

Electrical and Computer Engineering.

Presented in Partial Fulfillment to the Requirements

For the Degree of Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

June 2005

© Amitava Biswas, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-10233-0
Our file Notre référence
ISBN: 0-494-10233-0

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

A High Performance Real-time Packet Capturing Architecture for Network Management Systems

Amitava Biswas

In a network management system, software sensors (agents) collect system information and notify the central control system utilizing UDP/IP packets. When the network suffers a failure, the packet receiver encounters a flood of messages from a large number of agents. In such a situation, the receiver has to capture all these messages and process them in real-time. The thesis addresses this problem about how to design and develop a receiver that can capture large volume of packets while consuming minimum CPU resources. Linux network stack, contemporary related research work and some high capacity packet capturing solutions like NAPI and PFRING were studied to expose their limitations. Issues that limit the performance of these architectures have been identified and addressed in this thesis. Based on this analysis, a few design principles have been identified which can be applied to design and implement a high capacity, efficient packet capturing solution. An architecture, "DMA ring", that embodies these principles has been designed and implemented with commodity hardware and software components. The performance of this architecture has been tested on Redhat 8, a general purpose OS and RTAI 3.1 on Linux 2.4.24, a hard real-time platform. Its performance was compared with that of Linux, NAPI and PFRING. The results obtained from the experimental studies demonstrate that DMA ring outperforms these existing solutions. This study has also

demonstrated that a user space network processing solution will consume less CPU resources and may be quite effective under heavy network load, if appropriately designed.

Acknowledgement

This is my opportunity to express my gratitudes to all those who have made this work possible. My sincere gratitude to my advisor Dr. Purnendu Sinha, faculty of Electrical and Computer Engineering, for his support, advice, suggestions and encouragement.

My gratitudes are to my parents for their encouragement and blessings. Finally again, but always the most, my sincere gratitude to my beloved wife ***Mousumi*** for her domestic support which kept me alive, and for valuable suggestions, comments, and editing of this thesis.

Contents

		Page no.
	List of Figures	xi
	List of Tables	xv
	List of Nomenclature	xvi
1.	**Introduction**	**1**
	1.1 Background	1
	1.2 Data acquisition problem: system requirements	2
	1.2.1 Remote data collection by IP networks	2
	1.2.2 Packet capturing problem: Requirements of system monitoring applications	3
	1.3 Objectives, scope and contributions of the thesis	5
	1.4 Thesis organization	7
2.	**Analysis of Packet Capturing Problem**	**9**
	2.1 Packet loss in IP networks : analysis and causal factors	9
	2.1.1 Locations of packet loss	9
	2.1.2 Factors behind packet loss	11
	2.2 Need for high performance receiver	15
	2.3 Receiver performance parameters and architecture attributes	16
	2.3.1 Definition of performance	16
	2.3.2 Performance improvement approaches	17
	2.3.3 Architecture parameters that affect performance	18
	2.4 Performance improvement: opportunities, constraints and design approaches	22
	2.4.1 Improvement opportunities	22
	2.4.2 Constraints	24
	2.4.3 Solution approach	25
3.	**Limitations of Linux Network Processing Architecture**	**27**
	3.1 Linux network processing architecture	27
	3.1.1 System components	27

3.1.2 Packet receiving operation... 28

3.1.3 System task model 31

3.2 Receiver performance limitations and causal factors 32

3.2.1 Analysis of the system response and its jitter 32

3.2.2 Problems behind lower system throughput 39

3.3 Solution requirements 45

3.3.1 Task balancing is a challenge 45

3.3.2 Increasing buffer sizes is not solution 45

3.4 Summary... 47

4. Contemporary Solutions **50**

4.1 Operating system improvements 50

4.1.1 Avoiding data copy operations 50

4.1.2 Minimizing scheduling latencies 53

4.1.3 Managing border crossing costs 53

4.2 Real-time support for Linux 54

4.2.1 Managing jitters in interrupt latency and ISR response 54

4.2.2 Improving response of real-time packet receiving tasks 59

4.3 Miscellaneous schemes 62

4.3.1 Avoiding interrupt service overhead 62

4.3.2 Adopting efficient protocol processing 64

4.3.3 Reducing jitters due to hardware factors 64

4.4 Alternative packet processing architectures 65

4.4.1 NAPI.. 65

4.4.2 PFRING 72

4.5 Summary... 77

5. Design Principles and Proposed Architecture.................. **81**

5.1 Design principles and rationale 81

5.2 System requirements.. 85

5.3 Proposed architecture 85

5.3.1 Task model 85

5.3.2 Overview of implementation on Linux 87
5.3.3 Implementation details 90
5.3.4 Forecasting packet arrival rate 101
5.3.5 Settable parameters 106
5.3.6 Start up and run time operations 108
5.4 Implementation choices and rationale 111
5.5 Implementation in LXRT 116
5.5.1 About LXRT 116
5.5.2 Specific changes required for porting to LXRT............................ 118
5.5.3 Implementation choices in LXRT and rationale 119
5.6 Modifications carried out in the existing NIC driver 122
5.6.1 Modifications for Redhat 8 implementation 123
5.6.2 Modifications for LXRT implementation 125
5.7 Performance analysis and limitations 125
5.8 Summary 127

6. Performance Evaluation: Measurement Techniques, Instrumentation and Experimental Setup **128**

6.1 Evaluation criteria and rationale 128
6.2 Variables to measure, required instrumentation and test methodology 131
6.2.1 Detection and measurement of packet loss and system throughput............................ 132
6.2.2 Measurement of CPU utilization 134
6.2.3 Measurement of packet delivery latency............................ 135
6.2.4 Ascertaining memory requirement 147
6.2.5 Assessing maximum available CPU resources and robustness... 147
6.3 Test setups 148
6.4 Operating range 152
6.5 Summery............................ 152

7. Performance Evaluation Results and Comparison **154**

7.1 Normalization 154

7.2 Performance profiles and comparison 155

7.2.1 Packet loss and system throughput profile 155

7.2.2 CPU utilization profile 159

7.2.3 Packet delivery latency profile 161

7.2.4 Memory requirement profile 170

7.2.5 Robustness of DMA ring architecture on Redhat 8.................. 170

7.3 Contribution of the design and implementation strategies 172

7.3.1 Benefit of polling.. 172

7.3.2 Benefit of lower context switching frequency, integrated protocol processing, efficient border crossing, and no memory allocation 178

7.3.3 Benefit of shared staging area....................................... 180

7.3.4 Summary 181

7.4 Limitations of DMA ring on Redhat 8............................ 182

7.5 Performance of DMA ring on real-time platform 183

7.5.1 CPU utilization profile 183

7.5.2 Packet delivery latency profile 185

7.5.3 Memory requirement profile 188

7.5.4 Robustness 189

7.6 Comparison of overheads in LXRT and Redhat 8................ 189

7.7 Summary.. 190

8. Related Work and Contributions **194**

8.1 Related works covering aspects of network receiving performance.. 194

8.1.1 Approaches to tackle interrupt servicing overheads 194

8.1.2 Optimized network processing 196

8.1.3 User space network access 197

8.1.4 Enhanced network interface cards 198

8.1.5 Real-time networking stack 201

8.2 Differences and advantages of proposed DMA ring architecture... 204

8.3 Future research directions and insights for further improvements.. 205

8.4 Conclusion and discussions... 206

9. References **208**

List of Figures

Figure no.	Title	Page no.
1.1	A centralized Network Management System………………………….	3
2.1	Possible locations of packet loss……………………………………….	10
2.2	The queuing model of the receiver……………………………………..	11
2.3	Backlog packet queue in the receiver buffer…………………………..	14
2.4	Interrupt driven asynchronous architecture……………………………	18
2.5	Batch processing, synchronous architecture…………………………..	20
2.6	System organization……………………………………………………	24
3.1	Components for network receiving…………………………………….	27
3.2	Packet receiving operation in Linux……………………………………	29
3.3	2.4, 2.6 Linux architecture (3 tasks, 1 copy, 1 memory allocation)…….	31
3.4	Event processing response time………………………………………..	32
3.5	NIC ISR response jitter due to nested ISR execution…………………...	34
3.6	Hardware system organization…………………………………………	38
3.7	Receive livelock in Linux kernel………………………………………..	40
3.8	Packet loss measurement instrumentation for Linux……………………	44
3.9	Causes behind poor packet receive performance of Linux……………...	49
4.1	Performance limitations of zero copy implementations………………...	51
4.2	Shielding real-time interrupts by co-kernel……………………………..	56
4.3	Shielding real-time interrupts by Adeos layer………………………….	57
4.4	NAPI (2 tasks, 1 copy, 1 memory allocation)…………………………..	65
4.5	Interrupt to packet ratio for NAPI at different packet rates…………….	68
4.6	Livelock phenomena in NAPI…………………………………………..	68

4.7	Packet loss measurement instrumentation for NAPI........................	69
4.8	PFRING (2 tasks, 1 copy, 1 allocation).....................................	72
4.9	PFRING with NAPI (2 tasks, 1 copy, 1 allocation)........................	72
5.1	Task model of the proposed architecture (DMA ring) (1 task, 0 copy)...	86
5.2	Proposed DMA ring architecture..	88
5.3	DMA ring data structure..	91
5.4	Data structure for the receive descriptor ring...............................	92
5.5	Poll engine logic..	97
5.6	User space polling engine logic...	99
5.7	LKM ioctl logic..	99
5.8	Estimation of current packet rate...	103
5.9	Pseudo code for forecasting packet arrival rate............................	105
5.10	DMA ring operation...	109
5.11	Jitter of timer periodicity in Redhat 8.......................................	112
5.12	Jitter of timer periodicity in LXRT..	113
5.13	System architecture with RTAI-LXRT......................................	116
5.14	Poll engine task switching..	119
5.15	LXRT timer jitter comparison...	121
5.16	DMA ring operation at high packet rate....................................	125
6.1	Packet delivery latency measurement in interrupt based architectures...	136
6.2	Placement of probes in interrupt based architectures......................	138
6.3	Instrumentation architecture for interrupt latency and aggregate kernel time sample collection..	141
6.4	Probability density function for packet arrival event with respect to system time frame...	143
6.5	Instrumentation architecture for polling period sample collection.........	146

6.6	Hardware setup for the performance tests	148
7.1	Packet loss performance profile	155
7.2	System throughput performance profile	157
7.3	CPU utilization profile	159
7.4	CPU utilization profile at lower packet rates	159
7.5	Frequency distribution of interrupt latency for Linux	163
7.6	Aggregate kernel time frequency distribution for Linux	164
7.7	Frequency distribution of poll period for NAPI	165
7.8	Frequency distribution of the aggregate kernel time for NAPI	165
7.9	Interrupt latency frequency distribution	166
7.10	Frequency distribution of the aggregate kernel time for PFRING	167
7.11	Frequency distribution of the aggregate kernel time for PFRING with NAPI	168
7.12	Frequency distribution of poll period for DMA ring	169
7.13	Benefit of polling on CPU utilization	173
7.14	Sharp mode transition in DMA ring: Interrupt to packet ratio	173
7.15	Mode transition in DMA ring and NAPI: A comparison	174
7.16	CPU utilization vs. polling rate	176
7.17	CPU utilization vs. packet rate	176
7.18	CPU utilization profile for different operation modes	179
7.19	Effect of packet size on CPU utilization for Linux, PFRING and DMA ring: Benefit of shared staging area	181
7.20	Relative contribution of various performance improvement strategies	182
7.21	CPU utilization profile of DMA ring architectures	183
7.22	CPU utilization profile of DMA ring architectures at lower packet rates	184
7.23	Frequency distribution of packet delivery latency in LXRT	186

7.24 Frequency distribution of DMA ring poll period in Redhat 8 and LXRT 187

7.25 Frequency distribution of polling period paced by PIT and RTC timer in LXRT.. 188

8.1 RTnet network stack (3 tasks, 1 copy, 1 memory allocation)............... 202

List of Tables

Table no.	Title	Page no.
3.1	Factors behind packet processing response for Ultrix...........	39
3.2	Effect of DMA buffer size on Linux throughput................	44
4.1	Effect of DMA buffer size on NAPI throughput................	70
4.2	Effect of DMA buffer size on PFRING throughput............	75
4.3	Solutions and approaches available for Linux..................	79
4.4	Solutions that can be deployed under wide variety of circumstances..	80
7.1	CPU utilization normalization chart..............................	154
7.2	No loss capacity profile...	156
7.3	Packet loss comparison...	158
7.4	Packet delivery latency profile...................................	161
7.5	Memory requirement profile......................................	170
7.6	Maximum usable CPU resources for DMA ring on Redhat 8...	171
7.7	Robustness of DMA ring on Redhat 8............................	171
7.8	Packets processed per polling cycle..............................	176
7.9	Packet delivery latency profile for DMA ring architecture......	185
7.10	Memory utilization of DMA ring on LXRT and Redhat 8 compared with other low memory architectures.................	189
7.11	Superior performance of DMA ring: Comparison with best of class solutions..	191

List of Nomenclature

ACPI: Advanced Configuration and Power Interface. It specifies how operating system can manage power distribution to various hardware devices.

AGP: Accelerated Graphics Port. A standard for enhanced PC graphics display.

API: Application Programming Interface.

APIC: Advanced Programmable Interrupt Controller. This is a new PIC system architecture, different than legacy 8259 chip based PIC.

APM: Advanced Power Management. A Microsoft, Intel defined software interface between hardware-specific (BIOS) power management software and the operating system power management driver. Replaced by ACPI in resent systems.

ARP: Address Resolution Protocol.

ATM: Asynchronous Transfer Mode. A high speed network protocol.

CPU: Central Processing Unit.

CS: Context Switch.

DCN: Data Collection Network.

DMA: Direct Memory Access.

FIDDI: Fiber Distributed Data Interface. A set of ANSI and OSI standards.

FIFO First In First out. A real-time task scheduling policy.

FSB: Front Side Bus. Also known as System Bus or Memory Bus.

GPOS: General Purpose Operating System.

ICMP: Internet Control Message Protocol. It is an extension to the Internet Protocol. It allows for the generation of error messages, test packets, and informational messages related to IP.

IO-APIC I/O Advanced Programmable Interrupt Controller. In addition to LAPIC in each CPU, the SMP hardware organization implements a common APIC system to route interrupts to different CPUs. This is known as IO-APIC or I/O APIC, etc.

IP: Internet Protocol.

IRQ: Means interrupt request. It is signifies the hardware interrupt signal that is send by a PC peripheral to the host hardware.

ISR: Interrupt Service Routine.

LAPIC: Local Advanced Programmable Interrupt Controller. Recent Intel and other CPU chips have this onboard APIC known as LAPIC.

LKM: Loadable Kernel Module. A kernel stay resident component that can be loaded from Linux console.

LXRT: It is an extension of RTAI which adds user space real-time support to RTAI on Linux.

MAC: Media Access Control.

NAPI: New API. A new Linux kernel API used in network interface device drives to hand over packets to the Linux kernel network stack. This also refers to the scheme that implements polling to receive packets. This scheme requires a special NIC driver.

NIC: Network Interface Card.

NIDS: Network Intrusion Detection System.

NMS: Network Management System.

OS: Operating System.

PCI: Peripheral Component Interconnect. It is a I/O bus standard to interface host PC to various I/O hardware. Bus clock speed is 33/66 Mhz, bus is capable of 32 bit memory access.

PCI-X: Peripheral Component Interconnect - Express. An enhanced version of PCI which works with higher (66/133/266/533 Mhz) clock speed and has 64 bit memory address capability.

PFRING: An architecture which implements a low level packet capturing socket.

PIC: Programmable Interrupt Controller.

PIT: Programmable Interval Timer. A 8254 chip based timer that drives the OS clock in PCs.

POSIX: Portable Operating System Interface. A generalized set of standards derived from UNIX operating system.

RT: Real-time.

TCP: Transmission Control Protocol.

RTOS: Real-time Operating System.

RR: Round Robin. A real-time task scheduling policy.

RTAI: Real-time Application Interface. A co-kernel based architecture that adds real-time support to Linux.

RT FIFO: A real-time first in first out task scheduling policy under POSIX standard.

RT RR: A real-time round robin task scheduling policy under POSIX standard.

SMP: Symmetric Multi Processing. Defines an multi processor architecture for PC and server hardware.

SMT: Simultaneous Multi Threading.

SNMP: Simple Network Management Protocol.

SONET: Synchronous Optical Network.

TLB: Translation Look-aside Buffer. A special cache in the processor which holds the mapping between virtual and physical memory addresses.

TSC: Time Stamp Counter. A mechanism to obtain very high resolution event time information in terms of CPU clock cycle counts in the Intel CPUs.

UDP: User Datagram Protocol.

VGA: Video Graphic Adapter. Also denotes a standard for PC graphics display adapter.

XT-PIC: Signifies interrupt routing through 8259 chip based legacy PIC system.

Chapter 1: Introduction

1.1 Background

Real-time event data acquisition is a challenging problem in network management, scientific applications, industrial controls and system monitoring domain. High performance systems are needed to successfully collect and process large amount of data within short bounded time. Real-time data about the target system needs to be collected to manage or control it. Multiple sensors are often deployed in the system to collect this data. Sensor data can be collected either by synchronous polling or by event driven asynchronous method. Polling involves poll request communication traffic and higher data collection latency. The central system has to wait for the sensors to respond to each poll request made, so the data collection latency is aggregate of all the individual sensor latencies plus the additional poll request communication latencies. Multiple threads to overlap polling and response handling, can bring in only limited improvements in latency. Therefore, for real-time data collection, asynchronous, event driven strategy is preferred over synchronous poll method for its lower data communication traffic and data collection latency. In such systems, the sensors notify the central control system in case there is deviation in the target system's state. The control system analyzes and process the received notifications send by the sensors and takes control action in real-time.

Despite its inherent efficiency, the event driven data collection strategy has a potential drawback. When the target system rapidly changes its state, many sensors simultaneously send a flood of notifications to the central system. For complex systems like ship, turbine, particle accelerators and communication networks, the incoming event and aggregate data rate is quite high, though data content in individual events from each sensor source may be small [1,2,3,4]. The task load of handling all these events in real-time under such

circumstances may be quite high. For an asynchronous real-time data collection application, the key challenge is to engineer a system that can accept very high event and data rate in real-time. This multiple data source (sensor) and single sink (data aggregators and processors) scenario demands a data collection system, which has enough throughput capacity to handle high event and data rate. Collection and processing of events and data have to be completed within a bounded time to allow tighter control loops and to realize better control system response.

1.2 Data acquisition problem: system requirements

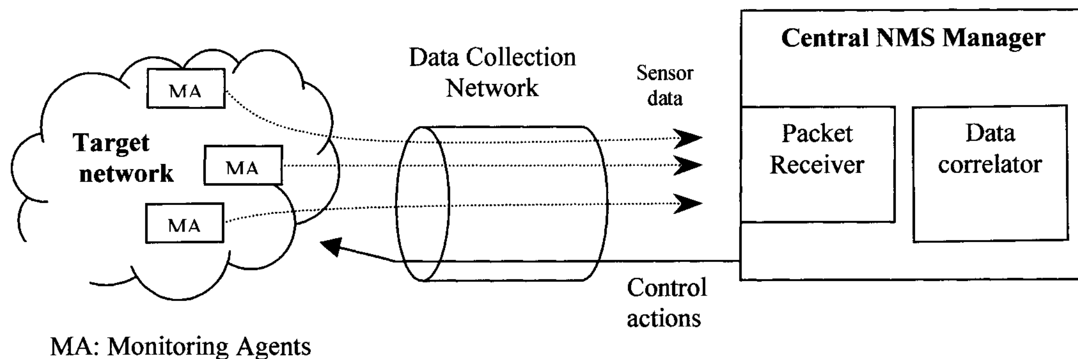
1.2.1 Remote data collection by IP networks

Standard IP based data networks are often employed for remote data collection, as IP packets are routable across network segments over large geographical distance. IP networks has been successfully used as event data collection pipe in particle accelerator, radio telescope, network management and network security (packet sniffing, network intrusion and attack detection) applications [5,6,7,8,9,10]. There are two transport protocol choices - TCP and UDP, for IP data collection networks (DCN). For real-time applications, UDP is chosen over TCP to carry the notifications because UDP has low transmit latency as it does not need connection setup time as in TCP. UDP achieves better real time transmit-receive response, higher throughput and better relative signal to noise ratio compared to TCP under similar packet loss scenarios [6,11]. Additional application level protocol may be used over UDP depending on the application. However such UDP/IP scheme pose its own challenges in regards to real-time data collection.

1.2.2 Packet capturing problem: Requirements of system monitoring applications

A typical example of such application is a centralized Network Management System (NMS) for managing a target network (Fig. 1.1).

Fig. 1.1: A centralized Network Management System



Although a centralized architecture has associated scalability limitations [12] but it is still preferred over distributed or hierarchical NMS when an automated real-time fault management system is deployed. All fault event or alarm data have to be gathered at a central place to correlate alarms, process them to detect/predict faults, perform root cause analysis and localize faults across network domains using artificial intelligence or other statistical techniques [13,14]. The target network is monitored by agents (software sensors) which send notifications or alarm messages to the central network manager application. Monitoring agents send SNMP traps (alarm messages) over UDP/IP through this data collection network. Information useful to predict faults are found in the alarm messages that arrive shortly (<1 sec) before the fault events happen [14]. The prediction/detection, diagnostic, fault localization processing are computationally intensive and takes time [14,15]. With lower packet receiving latency, more time is available for these computation intensive operations. Therefore the alarm notification receiving should be performed as fast possible [15], preferably within few milliseconds

or less, and more CPU time should be left aside for these computation intensive processing. Therefore packet receiving and delivery mechanism should cause minimum CPU utilization.

Network fault events occur in bursts. A single causal fault in a network resource may cause faults in related resources, symptoms may be amplified by various protocol mechanisms and faults may propagate among related resources across domain boundaries [16]. Hence a single causal fault event may generate an avalanche of fault events within a short interval, this phenomena manifests itself as a strong temporal correlation between fault events [17]. Therefore under serious target network failure condition, the network management system will receive a flood of alarm message packets from several network monitoring agents (sensors) [4]. There are at least two dimensions in this problem scenario. This situation pose a scalability problem at the centralized NMS as lot of packets have to be captured and processed within a short duration. Secondly, these alarm packets may be lost in the data collection network before it is delivered for analysis [6,18]. Too many packets arriving simultaneously at a network segment leads to network congestion. Studies have shown that congestion leads to packet losses [18,19]. On the other hand if the packet receiver of the NMS can not cope up with the incoming packet rate then packets also get dropped [20,21]. Such packet drops in the data collection network or at the packet receiver causes information loss, which manifests as noise in the system [6]. This noise deteriorates the precision of fault prediction, detection or localization. Thus ideally all packets have to be captured without any loss to reliably analyze, detect, identify or predict network faults. Efficient real-time algorithms can address the scalability problem at the processing level. But an efficient packet capturing solution is required to capture all packets from a network segment, extract all data and deliver them to this efficient real-time algorithm.

There are other applications, which require similar efficient packet capturing mechanisms. Network intrusion detection system (NIDS) is one, which needs a mechanism to capture packets arriving at high rates while utilizing minimum CPU resources. Network monitoring sensors taps all the packets flowing through the network segment, scan the packet payload data, applies several pattern detection rules to detect anomalous packet content or behavior which may suggest network security breach, intrusion or attacks [8,9,10]. The sensor is the likely bottleneck in a NIDS. The packet receiving and detection processing in the sensor has to be done at a very high rate, which should match the maximum traffic rate of the monitored network segment. The detection processing is computation intensive because the number of pattern detection rules required for acceptable level of accuracy may be quite high. More detection rules implies better accuracy, however more rules means less throughput or possibility of more packet loss for given sensor architecture. Again packet loss may degrade the accuracy of the sensor [8]. Hence packet capturing in the sensor should be completed in the shortest possible time leaving aside more CPU resources for detection tasks to improve the sensor throughput. Mobile and handheld devices with low power processors, which have to handle high speed network connections for real-time multimedia applications also require similar packet capturing or network I/O processing solution.

1.3 Objectives, scope and contributions of the thesis

The primary objective of this work was to design and implement a high performance packet capturing solution suitable for NMS and NIDS applications. Design and implementation of a suitable architecture on a low power uniprocessor system was the primary focus. Under certain cases, such packet capturing solution can be also applied in the network devices to increase their bottleneck capacity and reduce congestion. The

other purpose was to identify all the issues that limit the packet capturing performance in a receiver system and explore all available solutions.

Available literatures [20,21] indicate that the packet capturing performance is limited by the receiver system capacity. A receiver system is composed of various components like network interface card (NIC), the host hardware, NIC driver, operating system, and the network protocol stack. This thesis analyzed the problem of packet capturing in Linux Operating System and identified the issues that limit the capacity of the NIC driver, Linux network stack, existing Linux based packet capturing solutions - NAPI and PFRING. Linux was chosen for a variety of reasons: its importance in research community; its open source status; its constantly improving nature; and for its commercial potential. Linux is the fastest growing operating system especially in the embedded and mobile system domain [22,23]. Many network routers run on Linux OS. Linux supports a wide variety of network interface cards, hardware devices and platforms, it has proven track record for stability, reliability and performance and provides a rich set of OS services compared to other general purpose or specialized OS or RTOS like QNX, VxWorks, etc. Moreover real-time support for Linux have started emerging, a variety of alternate options exists today, which can support the real-time applications developed for Linux. In some cases only a minimum effort may be required to port these applications from Linux to real-time versions of Linux. A real-time platform has distinct advantages when real-time data acquisition is concerned. Versions of Linux, suitable for low power processors, embedded, handheld and mobile systems are also getting available. Hence it is worthwhile to explore this problem on this OS. To increase the conspicuity of the problems, all explorations were carried out on a modest PII 333Mhz system, but conclusions that were drawn based on the observations are still valid for a high speed (Ghz) hardware.

Based on the analysis of Linux, NAPI and PFRING, a set of performance enhancing design strategies for a soft and hard real-time packet capturing system were proposed. These design concepts were applied to engineer an efficient hybrid interrupt-polling packet capturing architecture. This architecture was implemented for Redhat 8 Linux (2.4.18 kernel), a general purpose operating system (GPOS) and for RTAI 3.1 on Linux 2.4.24, which is a hard real-time platform. Redhat 8 was chosen because it has lower scheduler latencies compared to vanilla Linux kernels.

The performance of this proposed architecture was measured and was compared against NAPI, PFRING and Linux's network stack. The performance measurements demonstrated the superior performance of the engineered architecture over the Contemporary Solutions. The proposed architecture is packaged in two components - a modified NIC driver and an user space polling driver component. This user space driver is generic can be used with any network interface hardware and may be also used with real-time data acquisition cards. A user space network driver is unconventional, but it demonstrated its advantage over kernel space drivers. The proposed solution depends on commodity PCI network interface card (NIC) features and not tied to any specific NIC. An available NIC driver was modified to implement the solution. The key modifications were packaged into two functions, which are very similar to two analogous existing Linux kernel API functions. Other NIC drivers can be modified using these two functions with minimum effort.

1.4 Thesis organization

This thesis is organized as follows:

Chapter 2, *Analysis of Packet Capturing Problem* provides theoretical understanding of the problem. The location of bottleneck in a data collection network is identified and abstract models of different forms of receiver architectures are introduced. The

relationship between different architectural parameters and performance elements are analyzed and presented. This chapter also discuses the opportunities and hurdles for performance improvements and possible approaches to implement a packet capturing solution.

Chapter 3, *Limitations of Linux Network Processing Architecture* documents the anatomy of the Linux network stack and identifies specific underlying factors that limits its packet receiving performance.

Chapter 4, *Contemporary Solutions* classifies various existing ideas, solutions and software components that address some of the underlying limitations of Linux. Advantages and limitations of these available artifacts are deliberated to identify the prospective candidates that can be adopted.

Chapter 5, *Design Principles and Proposed Architecture* propose specific design and implementation principles to conceive a high performance packet capturing employing available software artifacts. A high performance architecture, which is founded on these concepts, is presented. The rational behind the design, implementation aspects and operation of this architecture is also discussed.

Chapter 6, *Performance Evaluation: Measurement Techniques, Instrumentation and Experimental Setup* describes the instrumentation techniques and performance test setups employed to appraise and compare performance of the proposed architectures and various other packet capturing solutions.

Chapter 7, *Performance Evaluation Results and Comparison* collates the observations and summarizes the findings from the test data.

Chapter 8, *Related Work and Contributions* discuss some related works and clarifies the contributions and limitations of the present work in background of these existing works.

Chapter 2: Analysis of Packet Capturing Problem

In a large centralized NMS, the Data Collection Network (DCN) which collects data from the sensors, is generally implemented on a WAN/MAN or a LAN with multiple network segments. It consists of multiple active and passive components like routers, switches, bridges, concentrators, media etc. Packet loss under network congestion takes place due to bottlenecks in these network components. The following sections identify the network components that are most likely to cause packet losses under congestion and ascertain the underlying factors that cause bottlenecks in these components. Abstract models of different forms of packet receiving architecture and their attributes that define their performance are also discussed.

2.1 Packet loss in IP networks : analysis and causal factors

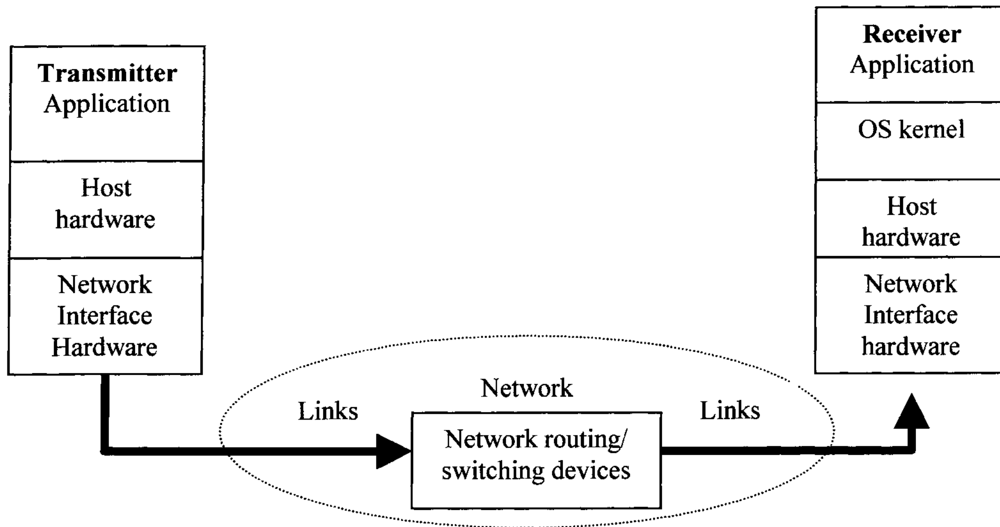
Small packets (size < 500Bytes) have small wire transit times, hence they have lower inter packet periods and manifest highest possible packet rates. As packet loss due to network congestion and receiver limitations are associated with high packet rates, so it is the small packet that pose the problem. Most of the subsequent discussions in this thesis are focused on the small packet scenario. To address the packet loss problem, it is imperative to locate the packet loss hotspots in the network and identify the causal factors behind them.

2.1.1 Locations of packet loss

UDP, the real-time alarm message transport in a NMS, is often found to be unreliable as it drops packets. Actually, packet loss in the underlying IP protocol layer is the reason behind UDP packet drops. This happens because UDP has no mechanism (lacks flow control) to provide reliability against IP packet loss, unlike TCP. Packet losses in the network can happen anywhere in the transmitter network interface hardware,

transmission links, the routing/ switching devices, receiver network interface hardware or at the receiver OS kernel (Fig. 2.1).

Fig. 2.1: Possible locations of packet loss



Packet losses in data link network interface and host hardware layers are negligible, bit error rates are in the order of 10^{-14} [24]. For this given bit error rate the packet loss rates will be in order of 10^{-10} or less for IP networks, because a packet may consist of 10^4 bits, and error in any one bit will cause packet discarding at the receiver. Packet losses are predominantly due to receiver host limitations and packet drops at the routing devices. Though IP packet loss obeys Poisson statistics over finer time granularity (i.e. loss instances are independent to each other) [6], but often these losses occur in bursts over larger time range [18,19] due to congestion in the network. UDP losses have similar behavior [18,19]. UDP flow rate is determined by the transmitter speed. Even if the receiver's throughput cannot match the transmitter's send rate, the UDP receiver does not provide feedback to the transmitter to reduce the transmission rate. Thus the receiver gets overwhelmed either by a single powerful, or by multiple low power senders [20,21]. For a given UDP transmitter-receiver pair, UDP packet loss starts happening beyond a certain packet rate (no loss capacity) and after that it increases proportionally with packet rate

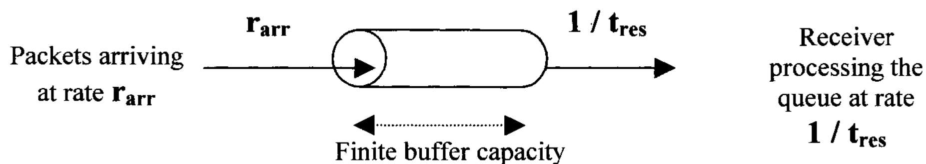
[20]. Such behavior strongly suggests that sender overflows the receiver's buffer before the receiver's host processor can accept and clear away all the data from the buffer. Similar opinion has also been voiced by other researchers [20,21].

2.1.2 Factors behind packet loss

Receiver may loose packets if the number of packets accumulating in the buffer builds up monotonically and consistently leading to receiver buffer overflow. Even though there may not be consistent packet accumulation, but still packets may be lost due to transient overflow in the buffer. Consistent packet build up happens if the receiver's packet processing capacity is not high enough or its processing response time is not low enough to match the packet arrival rate. On the other hand transient overflows happen due to transient delays in receiver's packet processing task invocation and/or completion. Transient overflows may occur even if the receiver may have sufficient capacity to handle a certain packet arrival rate, however likelihood of such transient overflow generally increase with the packet arrival rate. In the next few paragraphs examine both transient and non transient aspects of this dynamics.

The model below (Fig. 2.2, Eqn. 2.1 to 2.4) explains the empirical relationship, at an aggregate level, between the receiver behavior - packet loss, output packet rate, CPU utilization, and the causal factors - high packet processing response time and low CPU speed.

Fig. 2.2 : The queuing model of the receiver



For a given uniprocessor system with a finite receive buffer size (Fig. 2.2), the packet loss, r_{loss} (a fraction), depends on the packet arrival rate, r_{arr} (packets per second), and receiver's processing ability, which is represented by system response time, t_{res} (seconds) (Eqn. 2.1).

$$\text{Packet Loss} \quad r_{loss} = \left(1 - \frac{1}{t_{res} * r_{arr}}\right) \quad \text{when} \quad r_{arr} \geq \frac{1}{t_{res}}$$
$$= 0 \quad \text{when} \quad r_{arr} < \frac{1}{t_{res}} \qquad \text{.......Eqn. 2.1.}$$

Higher processing ability results in lower system response time. When the arrival rate is greater than the maximum processing rate possible, the buffer overflows, leading to packet loss. The maximum processing rate is determined by how fast (t_{res}) the system completes processing of each packet.

The receiver output rate, r_{out}, (packet per second) can be derived from packet loss and incoming packet rate (Eqn.2.2).

$$\text{Output packet rate} \quad r_{out} = r_{loss} * r_{arr} \quad \text{when} \quad r_{arr} \geq \frac{1}{t_{res}}$$
$$= r_{arr} \quad \text{when} \quad r_{arr} < \frac{1}{t_{res}} \qquad \text{.......Eqn.2.2.}$$

The receiver's CPU utilization, η_{CPU} (a fraction) depends on the receiving task response and packet arrival rate (Eqn. 2.3).

$$\text{CPU utilization} \quad \eta_{CPU} = t_{res} * r_{arr} + \eta_{BG} \quad \text{when} \quad r_{arr} \leq \frac{1}{t_{res}}$$
$$= 1 \quad \text{when} \quad r_{arr} < \frac{1}{t_{res}} \qquad \text{.......Eqn. 2.3.}$$

where η_{BG} is the CPU utilization due to other background tasks.

When the arrival rate, r_{arr}, is higher than the system's capacity, the system has no slack CPU time to process additional packets, CPU utilization hits 100% and the buffer overflows.

Response time of the receiving task depends on receiver's CPU power (Eqn. 2.4) represented by its speed (Mhz).

$$\text{Response time} \quad t_{res} = \frac{k_{arch}}{s_{CPU}} \qquad \text{...........Eqn. 2.4.}$$

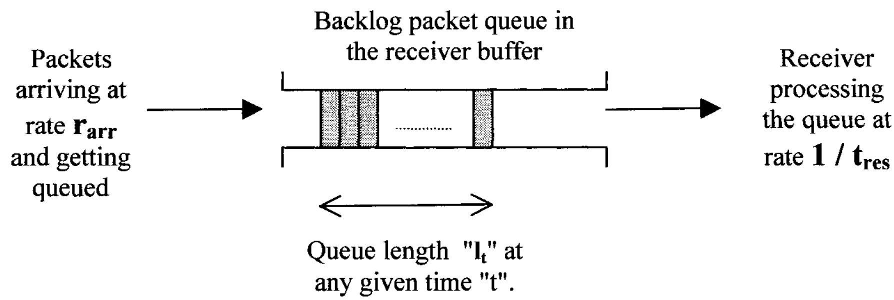
where "k_{arch}" is a constant for a given packet receiving architecture, s_{CPU} is the CPU speed in Mhz

Lower response time improves packet loss and CPU utilization performance of the receiver, this can be achieved by improving the architecture (which lowers the k_{arch}).

Transient overflow in receiver buffer may happen if the packet processing task is delayed and packets are not cleared within certain time. This transient delay in invocation and/ or completion of packet processing task is represented by the jitter in its response time.

Therefore along with all these (Eqn. 2.1 to 2.4), the variance or "jitter" of the task response time, t_{res}, also determines the system's packet loss performance. Even for a small average task response figure, a large "jitter" in the task response will deteriorate the system performance. Packets arriving at the receiver get queued up in the receiver buffer (Fig. 2.3). Even though the average receiver processing rate, $1/t_{res}$ may match the arrival rate, r_{arr}, but still there would be a finite queue length greater than one due to jitter in the response time, t_{res}. The upper bound of the system performances, i.e. the potential capacity of the system is determined by these equations (Eqn. 2.1 to 2.4) whereas the system jitter behavior defines the actual performance, which is far worse.

Fig. 2.3: Backlog packet queue in the receiver buffer



The jitter in the response time may be represented as an error in the response time, "ε", which manifests as a random variable when observed across time. For any given time "t" the queue length of the backlog packet queue, "l_t" is the number of backlog packets. This random variable "l_t" is given as -

$$l_t = (t_{res} + \varepsilon) * r_{arr} \qquad \text{................. Eqn. 2.5.}$$

When the receiver buffer has a size greater than one, it contains the transient effect of jitter to some extent. It averages the random effect of the processing task response jitter over certain period of time. But if this jitter is too high it's effect cannot be contained by the finite receiver buffer. The receiver buffer will overflow when the queue length, "l_t" shoots up above the buffer size. When the queue fills up and subsequent packets have to be dropped. A larger buffer lowers the likelihood of buffer overflow. However, for a given finite buffer size, the receiver buffer may sometimes overflow if "ε" is not restricted. This means that if worst case task response is not bounded, the receiver buffer may overflow resulting in packet loss.

The jitter in task response will significantly limit the capacity of the receiver system. If the task response jitter is high, the buffer can overflow even though the average task response may be quite low and may match the packet arrival rate, r_{arr}. The system may be

correctly designed on basis of the average response, but the realizable no loss capacity will be much lower due to this jitter. Thus along with improving the average task response figures, it is also important to address their jitter figures. The task response jitter can be minimized by employing a real-time platform at the receiver. It is also evident from this relationship that likely hood of buffer overflow increases with higher incoming packet rate.

2.2 Need for high performance receiver

Network routing devices also drop packets in case of congestion due to buffer overflow [18,24]. Thus UDP's unreliability problem is the general manifestation of device buffer overflow problem due to receiver capacity limitation. A high performance receiver that can receive all the packets without dropping them even at high worst case packet rates, is a solution for packet loss or network congestion problem.

Some NICs and switching/routing devices offer "hardware flow control" feature. The host computer can notify the NIC about its overloaded condition, in response the NIC will notify the network switching/routing device to reduce its send rate. Though this hardware level flow control may appear to reduce the packet arrival rate and packet loss at the receiver, but it may increase congestion and packet loss at the intermediate sender in a multi-hop network. So this cannot be a viable solution for a WAN or multiple segment LAN with multiple intermediate senders.

On the other hand, a solution, which can improve the no loss packet capturing capacity of the receiver, can be sometimes applied on the switching/routing devices to augment the performance of the data collection network. Therefore, the key issue is how to engineer receiver systems that can receive high packet rates without any packet loss and suffer minimum packet loss at higher packet rates. Ideally the receiver system should deliver the received packets in minimum time and keep as much CPU resource as possible for useful

event data processing. Conserving CPU resources is all the more important when CPU resources are scarce, especially in embedded systems and network devices.

2.3 Receiver performance parameters and architecture attributes

2.3.1 Definition of performance

Given the application requirement, the performance of a receiver system can be defined as a combination of four basic key elements -

- No loss capacity: The maximum tolerable packet rate, at which there is no packet loss.
- Packet loss percentage at higher packet rates: Loss as a percentage of total packets that has arrived.
- Packet delivery latency: Time taken to deliver the packet to the application.
- CPU utilization: CPU resources consumed by packet capturing tasks.

High no loss capacity, low packet loss percentage, low packet delivery latency and low CPU utilization are desirable features in a packet capturing system. The CPU resources that are left over from the packet capturing tasks are available for the event data processing application. As this application should get maximum CPU resources possible, so the CPU utilization figure is a measure of CPU wastage from the application viewpoint. For all practical purposes, delivery latency is same as the system response time in an uniprocessor system. The no loss capacity is represented in (kilo) packets per second (kpps or pps), packet loss is represented as a percentage of the total packets send, packet delivery latency is represented in micro seconds (μ sec) and CPU utilization is represented as a percentage of the total CPU resources (time) available. The upper bound of the no loss capacity is defined by the Eqn. 2.1 and 2.3. But the actual no loss capacity

is much lower than that and it is determined by the system response jitter. Similarly the actual packet loss percentage is also defined by the jitter behavior. The actual loss is higher than the figure obtained from Eqn. 2.1. The average packet delivery latency is given by Eqn. 2.4, however the actual values will be largely determined by the system response jitter "ε". By definition, CPU utilization is computed over longer time interval, thus it is not so much affected by jitter, hence the actual average is still given by Eqn. 2.3. A higher system response jitter means worse actual performance figures for no loss capacity, packet loss percentage and packet delivery latency.

2.3.2 Performance improvement approaches

System performance can be improved by two ways. Upper bound of the performance elements can be raised by an appropriate choice of architecture. Raising the upper bound figures improves the actual figures. On the other hand, limiting the system response jitter improves the actual performance figures themselves even if the upper bound figures are not raised. Preferably, both improvement approaches should be pursued.

In general, these four performance elements are not always positively co-related, though it may appear so in the first glance from Eqn. 2.1 to 2.4. Only for certain event driven architectures, these elements are positively co-related. In such cases, any design improvement that reduces the delivery latency will also reduce CPU utilization, limit packet loss and improve throughput without any need to make design tradeoffs. However for alternate architectures, which are not event driven, each of these four elements have to be individually improved by appropriately choosing proper architectural parameters. In some cases a design feature might improve a performance element, but may worsen another element, so design tradeoffs may be involved. The next sub-section discuss how architectural parameters play an important role in defining these performance elements for various architectural forms.

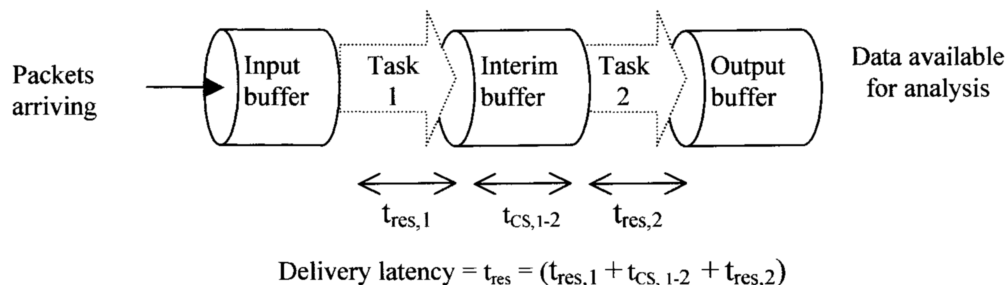
2.3.3 Architecture parameters that affect performance

By appropriate choice of architectural form and attributes it is possible to improve the receiver's performance elements individually and strike a balance between them. Next few paragraphs present the effect of the architecture's attributes on performance elements.

The packet receive processing may be carried out in different layers of the protocol stack by multiple task threads, like task 1 and 2 (Fig. 2.4). Henceforth task threads are termed as "tasks". There are various alternate possibilities to schedule these tasks. Scheduling of these tasks may be synchronized either with the packet arrival events or with the system clock. The architectures whose tasks are synchronized to the system clock are termed as synchronous architectures and the ones whose tasks are driven by packet arrival interrupt events are labeled as asynchronous architectures. The interim buffer between these tasks holds the semi-processed packets.

Asynchronous architectures: For interrupt driven architectures the packet processing tasks are triggered by packet arrival events, i.e. interrupts from the network interface card. Task 1 is invoked by the packet arrival event, task 2 is invoked by task 1 (Fig 2.4).

Fig. 2.4: Interrupt driven asynchronous architecture



If the task 1 and 2 are scheduled to execute alternately, then the interim buffer is not required, however it is shown here to generalize the analysis. The packet delivery latency is aggregate of the response of the tasks, $t_{res,1}$ and $t_{res,2}$, plus the time required to switch

between the tasks (context switching time $t_{CS,1-2}$) and some background scheduler and OS overhead, $t_{res,BG}$ as in Eqn.2.6.

$$\text{Packet delivery latency} = t_{res} = (t_{res,1} + t_{res,2} + t_{CS,1-2}) + t_{res,BG} \quad \text{.......Eqn. 2.6.}$$

The packet loss percentage is governed by Eqn. 2.1.

No loss capacity is the packet arrival rate at which the CPU utilization hits 100%, this is given by -

$$\text{No loss capacity} = r_{arr,100\%} = \frac{s_{CPU}(1-\eta_{BG})}{k_{arch}} \quad \text{........... Eqn. 2.7.}$$

The CPU utilization for a given arrival rate, r_{arr} is given by -

$$\left.\begin{aligned} \eta_{CPU} &= (t_{res,1} + t_{CS,1-2} + t_{res,2}) * r_{arr} + \eta_{BG} && \text{when } r_{arr} \leq \frac{1}{t_{res}} \\ &= 1 && \text{when } r_{arr} \geq \frac{1}{t_{res}} \end{aligned}\right\} \quad \text{.......Eqn. 2.8.}$$

The jitter in the task response is given by -

$$\varepsilon = \varepsilon_{res,1} + \varepsilon_{CS,1-2} + \varepsilon_{res,2} + \varepsilon_{res,BG} \quad \text{................. Eqn. 2.9.}$$

Where $\varepsilon_{res,1}$ $\varepsilon_{res,2}$, $\varepsilon_{CS,1-2}$ and $\varepsilon_{res,BG}$ are the jitter in response time of task 1, 2 and the context switch time.

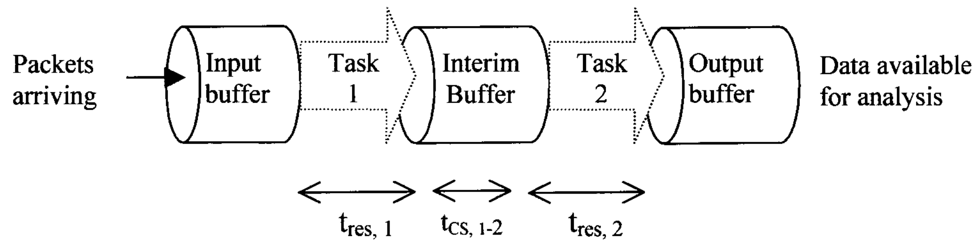
This jitter adversely affects the packet delivery latency performance. Eqn. 2.5 governs the likelihood of the input buffer overflow for this architecture (Fig 2.4).

Any engineering effort to reduce task response and context switch times will improve all the four performance elements, because of the simple inter-relationship between them

(Eqn. 2.1, 2.6 to 2.8). In addition to this form with interleaved task scheduling, an interrupt driven architecture with batch scheduling is also possible but it is not discussed here.

Synchronous architectures: Alternate architectural forms are possible where tasks invocation do not depend on the packet arrival event. Tasks maintain synchronism among themselves and with the system clock.

Fig. 2.5: Batch processing, synchronous architecture



For such architectures, as a design choice, task 1 and task 2 may not be scheduled alternately in an interleaved fashion, but a "n" tasks of type 1 may be scheduled together in a batch followed by a batch of "n" tasks of type 2 (Fig 2.5). The interim buffer should have sufficient capacity to hold at least "n" packets. In that case, an interim buffer is required between these two task layers to hold the semi-processed data. For this arrangement the average CPU utilization is given by -

$$\left.\begin{aligned} \eta_{CPU} &= (t_{res,1} + \frac{t_{CS,1-2}}{n} + t_{res,2}) * r_{arr} + \eta_{BG} && \text{when} \quad r_{arr} \leq \frac{1}{t_{res}} \\ &= 1 && \text{when} \quad r_{arr} \geq \frac{1}{t_{res}} \end{aligned}\right\} \quad \text{....Eqn. 2.10.}$$

The context switching time $t_{CS,1-2}$ is substantial, so the CPU utilization can be reduced significantly by simply choosing a higher value of "n", which amortizes the context

switching overhead over many packets. This also improves the packet loss and throughput capacity. However a higher value of "n" will deteriorate the worst-case packet deliver latency, which is given by -

$$= n * (t_{res,1} + t_{res,2}) + t_{CS,1-2} + t_{res,BG} \quad \text{...........Eqn. 2.11.}$$

The jitter in the task response is governed by Eqn. 2.9 which adversely affects the packet delivery latency performance. Eqn. 2.5 also governs the likelihood of the input buffer overflow for this architecture (Fig. 2.5). In addition to this form with batch scheduling, a polling architecture with interleaved scheduling is also possible but it is not discussed here.

For both architectures, the interim buffer can also overflow if the scheduler does not ensure execution balance between the "producer" and "consumer" tasks. Task 1 that performs the first stage of protocol processing is a "producer". Whereas, task 2, that receives the interim data is a "consumer". To avoid piling up of data anywhere in the network stack, both task 1 and 2 tasks should be balanced (1:1 execution ratio) against each other. If the producer tasks have higher task priority, then in case of rapid arrival of packets and under CPU resource constraint only "producer" tasks will run and "consumer" tasks will starve. Thus the producer will pile up data causing interim buffer overflow and loss of subsequent packets, i.e. no more packets will be stored. If "consumer" tasks have higher execution priority then buffer overflow problem can be avoided. The higher priority consumer task tries to execute but it blocks if there are no packets in the interim buffer. Once the consumer blocks then the low priority producer task gets a chance to execute and place packets in the interim buffer. Once there is packets in the interim buffer the high priority producer becomes "runnable" and takes the

CPU to process/consume the packet. Thus this priority scheme enforces interleaved task scheduling which ensures 1:1 execution balance.

Thus, along with architectural form, the scheduling policy and task priorities also govern the performance of the architectures. Though there may be favorable or adverse cross effects for every design choice made, but it is possible to improve these four performance elements. Therefore each of these four performance parameters have to be concurrently addressed in the design. There can be several other design approaches to improve receiver performance. Simply increasing the CPU speed is the most costly option and not an elegant solution. Reducing or completely eliminating the context switching time, reducing its impact on CPU utilization in case it cannot be completely eliminated, reducing task response times by re-designing/ simplifying the tasks are some alternative approaches to improve these four performance parameters.

2.4 Performance improvement: opportunities, constraints and design approaches

2.4.1 Improvement opportunities

There is enough opportunity to improve the packet receiving performance in existing commodity systems. The host hardware generally have sufficient capacity to handle high packet rates. For example a PII 333Mhz system is sufficient for a 100Mbps network. The two key bottlenecks in host hardware are memory and I/O bus. The throughput of the DMA transfer is around 118 MByte per sec over PCI I/O bus [25,26,27,28], whereas the memory bandwidth is the FSB clock speed multiplied by the data bus width, which is 66 Mhz * 4 Byte = 264 MBytes per sec for a PII 333 Mhz CPU. At least two memory accesses are required for any data processing task, one access to read the raw data, another access to write the processed data back in the memory. So the throughput of the data processing task running on this hardware is memory bandwidth divided by two, which is 264/2 = 132 Mbytes per sec which is larger than 118Mbytes per sec. Therefore

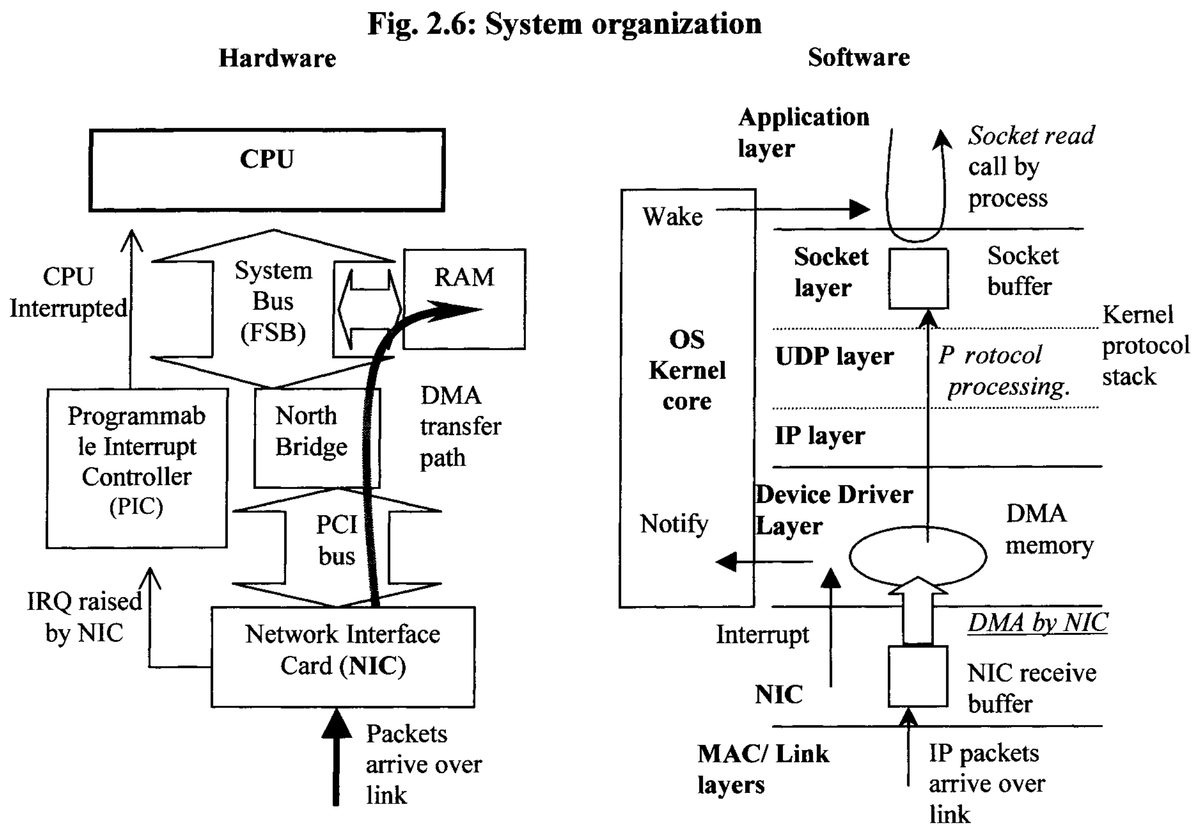
the limiting throughput in the host is 118Mbytes per sec. For a 100Mbps Ethernet NIC, the line speed is 100/ 8 = 12.5Mbytes per sec which is fraction of the host CPU data path throughput. This shows that the host hardware's data path has enough throughput capacity to handle this data rate. This also means that it is the inefficiency of the software components that limit the host's receiving capacity. An inefficient software will increase the frequency of memory access from the minimum requirement of two to a higher value "n", hence the data processing throughput drops from 118 to (264 / n) Mbytes per sec (n >2).

Theoretically it is possible to deliver packets within few microseconds, but even the fastest system take around 100 microseconds or more to deliver. For the two most used transport protocols used, TCP and UDP, the receive side processing can be completed by few instructions. TCP processing involves around 30 RISC instructions, IP processing requires 25 and demux (delivery to correct socket) operation needs 10 instructions [29]. For simpler stateless UDP/IP protocol combination, the necessary processing could be completed by 16 "C" statements. For a 333Mhz PII processor, execution of these 16 statements takes only 0.746 microsecond without benefit of cache. DMA transfer invalidates the cache locations of the received packets so cache hits are not possible for memory access to these locations. On the other hand, receive processing for UDP/IP in Linux kernel (Redhat 8) on the same processor takes 25 to 100 microseconds, depending on the size of packet payload. Even on a faster 2.2Ghz dual Itanium (IA64) server with 622Mhz dual memory bus and 10Gbps 133 Mhz PCI-X Ethernet card, the total packet delivery latency (hardware + software) was 100 microsecond for a 1500 byte packet [30]. With regards to receive side packet delivery latency, this large gap between what is possible (fraction of microsecond) and what is realized (100 microseconds) presents the opportunity for performance improvement.

2.4.2 Constraints

However several constraints exists in the design space which restricts realization of such promise. Hardware and operating system (OS) organization in these machines have been matured over the years keeping in mind the performance requirements of several services in addition to receiving network data (Fig. 2.6, more details in Chapter 3).

Fig. 2.6: System organization



The choice of organization of PCI I/O bus, the bridge between I/O bus and the system bus (FSB), hardware interrupt delivery mechanism (PIC), software interrupt handling mechanism in the OS kernel, interrupt service routines (ISR) in the network device driver, protocol stack in OS kernel, socket read and payload delivery semantics, process management in OS, is responsible behind this apparent gap between the possibility and the realized performance. These system components are not optimized for only network data receiving, many of them are optimized for other operations, some are designed to

reduce development and life cycle costs by component standardization, modularization and layering.

Over and above these, real-time performance limitations of OS and hardware platform may be significant hurdles towards realization of superior performance. A general purpose OS (GPOS) may not guarantee the task completion within bounded time. Such OS may result high task response jitters under heavy system load which may lead to buffer overflows. Some hardware may have similar real-time performance limitations. These platforms may severely restrict complete realization of a solution's true capacity and full achievement of the improvement potential.

2.4.3 Solution approach

It may be possible to engineer an optimum system from scratch, which can reach the packet receive performance limits and yet satisfy the bare minimum performance requirements for other services. But, cost of developing, deploying and life cycle management for such a radically engineered system will be prohibitively large. On the other hand, there exists enough scope to re-engineer key software system components within the present hardware and software system organization to reclaim enough receive performance. An appropriately conservative design approach to leverage existing OS and hardware components makes such improvements deployable at low costs with the available commodity off the shelf hardware and software. Therefore packaging and modularizing the improvements within as few components as possible is also critical as the performance gain itself.

By designing within the existing stable architectural framework, it is easier to avoid deteriorating performance of other services during an attempt to improve receive performance. By conforming to existing OS framework, much design efforts can be avoided which would have been otherwise required to guarantee bare minimum

performance of other necessary services. With such approach, it is possible to choose the best from the available alternative components and OS kernels. This approach also brings in the collateral advantage of having options to choose and seamlessly employ contemporary and future OS developments that delivers direct or indirect benefits to the receiving performance. Fortunately new solutions are available which address many limitations of the standard platforms.

In the present work, such re-engineering possibilities in specific key software components are explored, which avoids modifying the OS kernel or any hardware components. Such discreet component re-engineering approach saved design efforts but yielded significant performance gains. A mix of developed, modified and readily available off the shelf hardware and software components were employed to engineer a high performance packet capturing solution. The NIC driver was modified and an additional component in the user space was adopted. This avoided patching and re-compilation of the OS kernel or the costly hardware, firmware development cycle. The modification of the driver was limited to very specific well defined zones in the NIC driver. Even these zones are fairly generic in a NIC driver. In addition to this, packaging more intricate modifications are packaged into two functions which are analogous to existing kernel API functions. This reduced the cost of design comprehension and modification.

Chapter 3: Limitations of Linux Network Processing Architecture

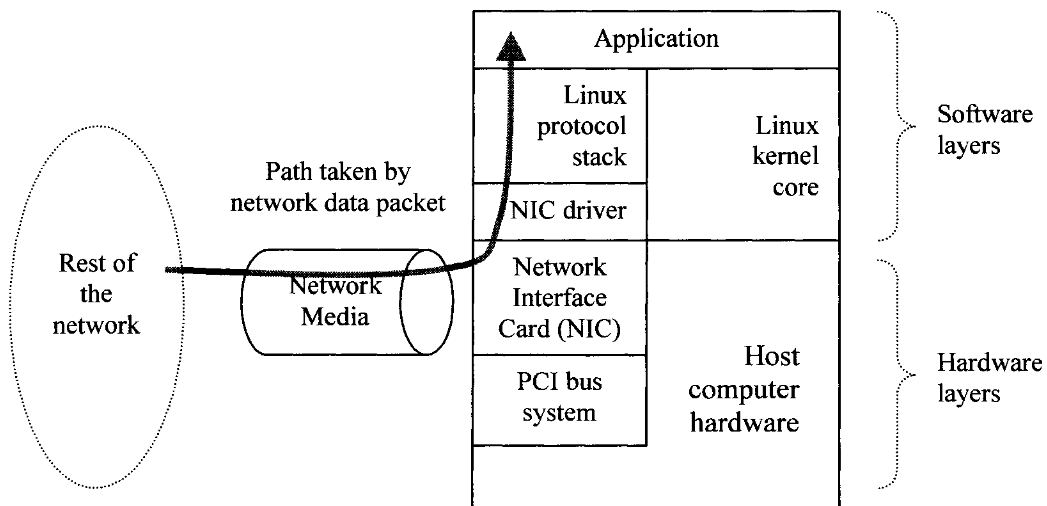
Many underlying platform and implementation specific factors determine the architectural attributes, which in turn defines the receiver performance. To identify these underlying factors for the chosen platform, i.e. Linux, its packet receiving architecture needs to be studied. The following sections present the anatomy of the Linux network stack and its packet receiving operation. Based on the analysis of the architecture and its operation, specific factors that undermine its packet receiving performance are identified.

3.1 Linux network processing architecture

3.1.1 System components

Receiving data packets over the IP network requires a few hardware and software components: a network interface card (NIC) hardware with embedded firmware; the host computer hardware; the Linux OS (kernel core & protocol stack) and the network card driver software; and finally the application software which consumes the data packets and carries out the event data processing (Fig. 3.1).

Fig. 3.1: Components for network receiving



The NIC receives the packets from the rest of the network over the data link, which can either be Ethernet, FDDI, ATM etc. The network media (CAT 5 copper or optical fiber) physically connects the NIC to the rest of the network. The NIC implements the data link layer of the ISO/OSI framework. The NIC sits on the PCI or PCI-X slot of the host hardware and communicates with the host CPU over the PCI/PCI-X bus. The NIC driver initializes and manages the NIC hardware, it also receives the data packets from the NIC and delivers them to the Linux protocol stack for network processing. The protocol stack, which handles the protocol processing, is part of the Linux OS. The Linux OS and the driver software run on the host computer hardware. The application, siting on top of the Linux, receives packet data from the network stack and then carries out the application specific event data processing.

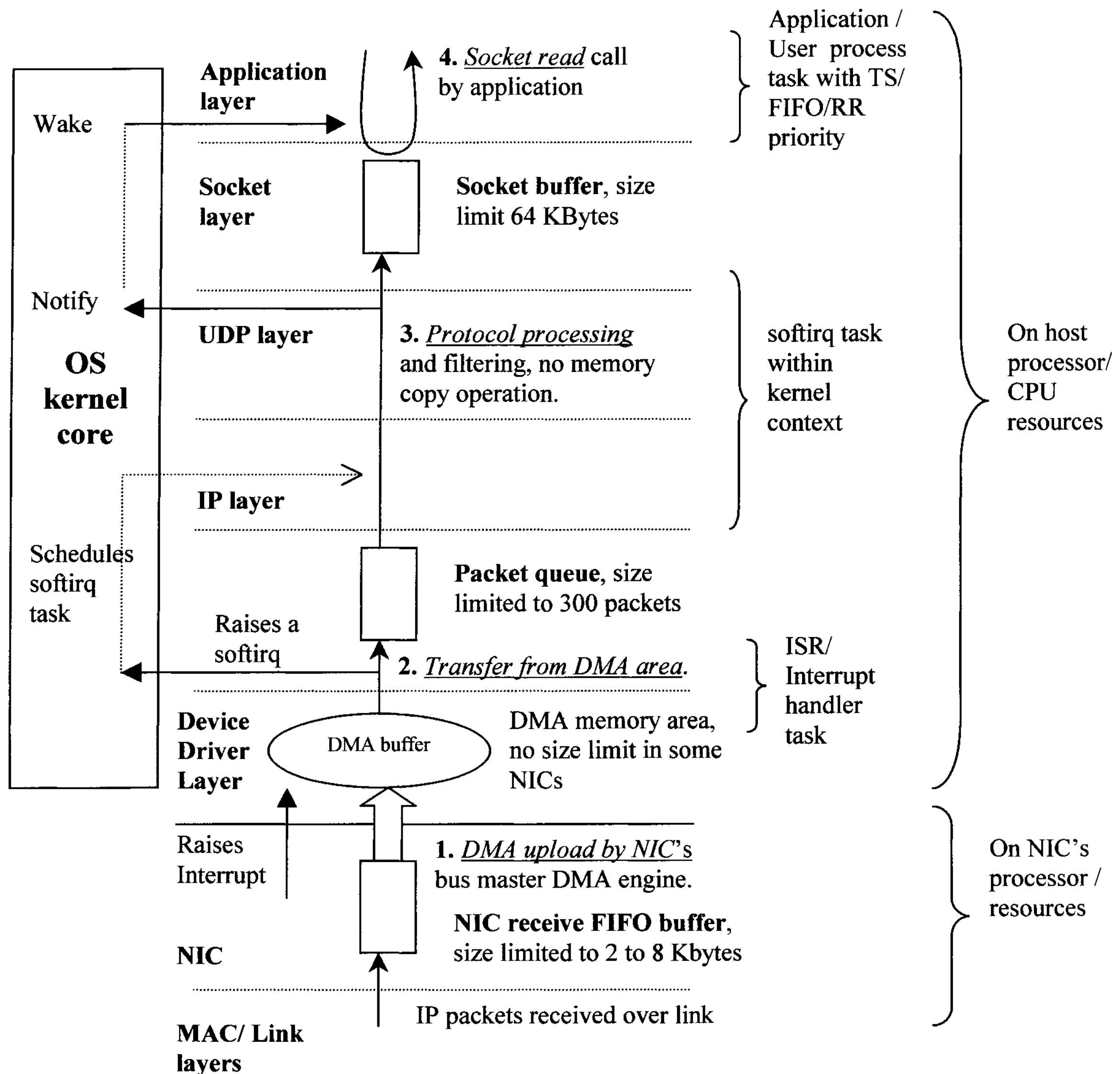
3.1.2 Packet receiving operation

Network stack is implemented as pile of protocol processing layers. These layers implement ARP, ICMP, UDP, IP protocol processing operations. The software architecture for network processing in Linux, is derived from the Unix model. The network stack delivers data packet to the application through a socket mechanism. The simplified packet receiving path for Linux for later 2.4 and 2.6 kernels is presented in Fig. 3.2 (next page). The receiving operation is described in the following paragraph.

The application runs as a user space (Linux process) task, this task tries to read from the socket buffer and blocks if there is no packet to read. After the packet arrives in the network interface card's FIFO receive buffer, the PCI bus master network interface card (NIC) transfers the packet by direct memory access (DMA) to the kernel DMA memory (operation no. 1, Fig. 3.2) and interrupts the host processor. Along with making the DMA transfer the NIC's DMA engine invalidates the cache location corresponding to the

memory address where it transferred the packet. This ensures that the host CPU does not process the stale cached data instead of fresh packet.

Fig. 3.2: Packet receiving operation in Linux



As a response to the interrupt, the host processor runs an interrupt service routine (ISR). This ISR task moves the packet from the DMA memory to a packet queue implemented in regular kernel memory (operation no. 2, Fig. 3.2). This data movement is performed either by data copy operation for small packets or by passing memory address (pointers)

for large packets. When packets are transferred by passing pointers, new memory buffers known as "packet buffers" are allocated to receive new packets. As a part of the allocation process, the DMA address of these packet buffers are computed ("DMA mapping") and these DMA addresses are placed in the descriptor address locations from where the NIC DMA engine can access them. NIC access these descriptor address locations to ascertain the addresses of the packet buffers where it can make the DMA transfers.

In 2.4/2.6 Linux kernels, after transferring the packet, the ISR raises a softirq, this softirq task is scheduled to run immediately after the ISR task completes (returns). The softirq task picks up the packet from the packet queue and performs the protocol processing to extract the data payload (operation 3, Fig. 3.2). The protocol processing and packet filtering operation alone involves more than eight levels of complex nested function calls and couple of "hook" mechanisms, implemented by function pointers.

After all these processing and filtering operations, the softirq task inserts the processed packet in a queue (socket buffer) and notifies the scheduler to wakeup the blocked user process. The packet is copied to user space memory as the blocked socket read function call returns (operation no. 4, Fig. 3.2). The packet buffers, whose data has already been transferred to the user space, are dissolved and the memory is freed. The socket read system call from the user space is quite complex and involves more than eight nested function calls. After waking up, the user space task processes the event data.

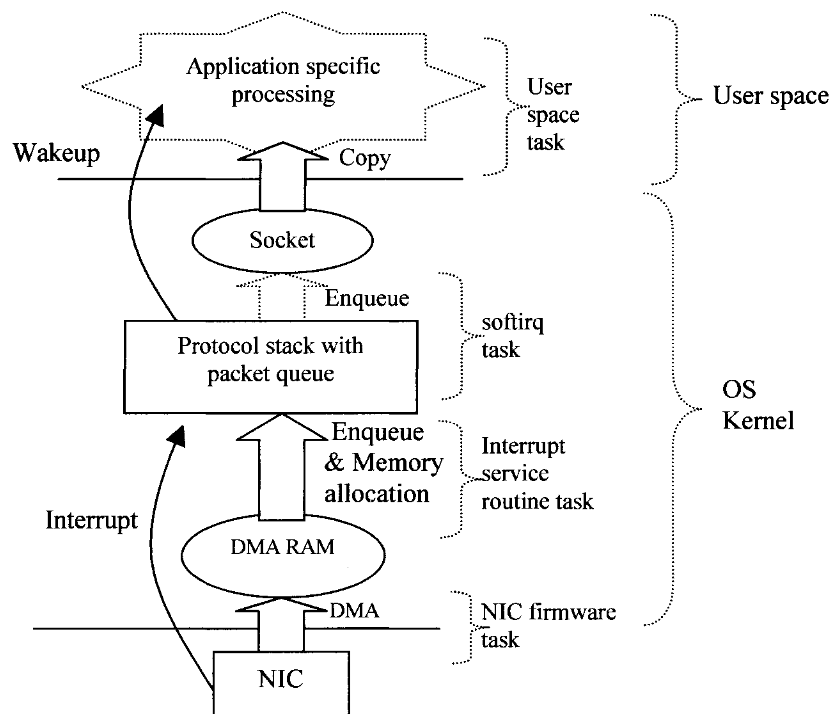
The size of the NIC receive FIFO buffer depends on the NIC, it may range from 2 KB to 8KB. The size of the DMA buffer depends on the NIC driver code. Typically it is provisioned to hold 32 to 128 packets. The default packet queue size is 300 packets and the default socket buffer size is 64KB. The ISR task has the highest priority, the softirq task has higher priority than any user space task even though the user space task may

have highest POSIX priority like RT FIFO/RR priority = 99. Under this priority scheme, under very high packet arrival rate, many ISRs may execute even before a single softirq task may get the opportunity to complete, likewise under moderately high packet rates many softirq tasks may execute even before the user task may get the chance to start. Therefore this scheme may pile up packets in the packet queue or the socket buffer and may overflow them due to heavy incoming network traffic.

3.1.3 System task model

The receiving operation tasks are presented in abstract form in Fig. 3.3.

Fig. 3.3: 2.4, 2.6 Linux architecture (3 tasks, 1 copy, 1 memory allocation)



The Linux network architecture comprises of three layers - NIC driver, protocol processing and socket layer. The receiving mechanism employs three task threads, which run on the host CPU in three different contexts - ISR, softirq (kernel) and user space. It

uses three buffers - DMA, packet queue and socket buffer. The operation involves one memory allocation during packet transfer from DMA buffer to packet queue and one copy during data transfer from kernel to user memory. The firmware task and DMA operation, do not utilize host CPU time so these are not counted in our analysis.

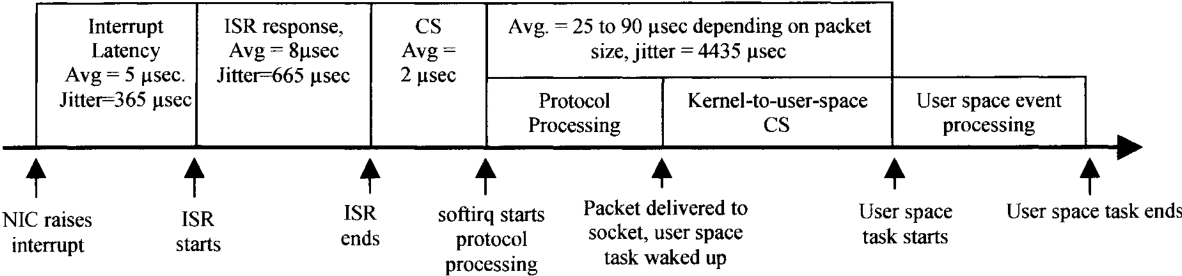
3.2 Receiver performance limitations and causal factors

Its network receiving architecture is not suitable for processing high event rates. UDP packet losses are common in Linux at high packet rates [20,21]. There are two reasons behind packet losses - high average task response time and high task response jitter, both of which can independently cause packet losses. There are substantial inefficiencies in Linux network processing architecture, which result higher average task response times and manifest as lower average throughput. On the other hand Linux being a non real-time operating system (RTOS) do not guarantee the bounded behavior of these task response times which also results packet losses (Eqn. 2.5). Next few sub-sections present an analysis of the task response times and identify the causal factors behind their high average and jitter figures.

3.2.1 Analysis of the system response and its jitter

The anatomy of the receiver response time for each packet is presented in Fig. 3.4.

Fig. 3.4: Event processing response time
(Redhat 8, PII 333Mhz 3C905B-TX, 64B UDP/IP)



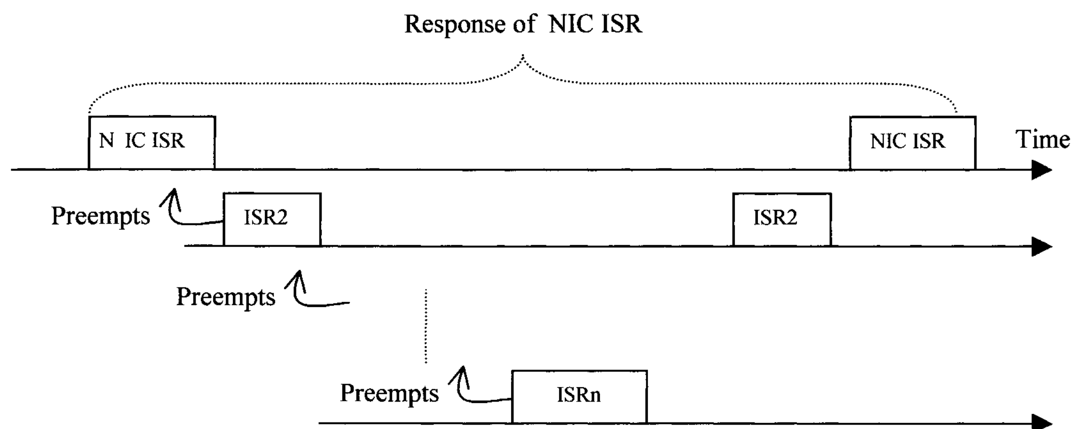
The receiver response time is the aggregate of the "packet delivery latency" of the kernel and the event data processing response time of the application task running in the user space. The "packet delivery latency" is the time taken by the kernel to deliver the packet data to the user space application. It is aggregate of: the interrupt latency, interrupt service routine (ISR) response, context switching time between ISR and protocol processing softirq task, the protocol processing response, and the context switching time between protocol processing task and the user space application task. The "jitters" figures are actual observations which represent the order of latencies under heavy interrupt load (under high packet rates) for a Dell PII 333Mhz, machine with 3C905B-TX (3Com) NIC for 64 byte UDP /IP packet receiving on Redhat 8 Linux. Redhat 8 runs on a customized 2.4.18 kernel. The "Avg." indicates the average response time figures for the same.

The interrupt latency is the time between, when an interrupt is raised and when its ISR start executing. It is the aggregate of the time taken by the programmable interrupt controller (PIC) to dispatch the interrupt to the CPU, the response of the CPU hardware to accept this interrupt and the response of the OS kernel interrupt dispatcher to start the interrupt service routine (ISR). This may be also viewed as a context switching time between the NIC firmware task which transfers the packets to DMA buffer and the ISR which picks up the packets. The order of this response is around 5 micro seconds or more. Generally, the jitter is order of 15 micro seconds, which is primarily due to the host hardware. However some long "critical section" path in Linux kernel might disable the interrupts and execute for a longer time, which may cause very high interrupt latency. Interrupts are disabled for this critical section execution time and are not reported to the CPU till the interrupts are enabled. In 2.6 preemptable kernels many of these long critical sections have been broken down to multiple smaller paths. Critical sections of 50 microseconds have been reported for 2.6 kernels [32]. A few interrupt latency jitters in order of 250 to 365 microseconds was observed in Redhat 8 (custom 2.4.18) on PII

333Mhz Dell desktop. Interrupt latency jitter increases with interrupt load (interrupt rate) on the system [32].

The ISR execution time is the time taken by the ISR task to move packets from DMA buffer to the packet queue. For example, this time is around 8 microseconds for the "3c59x" NIC driver, that came along with Redhat 8. The ISR code is packaged as a part of NIC driver. High ISR response jitters up to 665 microseconds on a PII 333Mhz with Redhat 8 was observed. These high jitters happen because the NIC ISR may be preempted by other types of ISRs, for example, a hard disk ISR invoked in response to hard disk interrupts. Multiple levels of nested ISR execution may be possible under heavy network load and heavy disk activity. Such nesting of ISRs stretches the response times of NIC ISR (Fig. 3.5).

Fig. 3.5: NIC ISR response jitter due to nested ISR execution



The NIC ISR may be preempted by another interrupt which invokes ISR2 (Fig. 3.5). ISR2 may be again preempted by another type of interrupt, and so on. On the return path the remaining portions of the ISRs execute in the reverse order, as they are unstacked. This sort of ISR nesting stretches the ISR response time.

Hard disk/VGA/AGP ISRs may preempt NIC ISR. The hard disk/VGA/AGP ISRs may take significant amount of time to set up hardware resources for bulk DMA transfers before they yield the CPU back to the preempted NIC ISR.

The context switching (CS) time between the ISR and the corresponding packet processing softirq task is due to the kernel scheduler latency and response times of other high priority pending softirq/ kernel tasks which may execute before the softirq can get the CPU. When the ISR yields CPU back to the scheduler, the scheduler checks for pending softirqs, their order of priorities, and runs other pending high priority softirq tasks before it initiates the pending softirq task raised by the NIC ISR. Generally only NIC operation involves frequent softirq tasks, so execution of other high priority softirqs are not very frequent. The average context switching time between NIC ISR and the network softirq is around 1 to 2 micro second, though there may be jitters due to execution of other high priority softirq tasks in some cases. Jitter in the total time spent in ISR execution and context switching increases with interrupt load (interrupt rate) on the system [32].

The protocol processing time is the time needed by the softirq task to execute all the protocol processing steps on a packet. It may have high jitters as it may be preempted by an ISR anytime. The same causal factors, i.e. heavy network traffic, hard disk, VGA/AGP or other DMA operations are behind high jitter in the protocol processing task response.

The kernel to user space context switching time between the protocol processing softirq task and the blocked user process task is primarily due to scheduler latency and response time of all ISRs, other pending high priority kernel/ user space tasks and softirqs. Even the highest priority (RT RR/FIFO priority = 99) user space tasks have lower priority than softirq tasks. All pending softirq tasks have to complete before any user task can run.

This context switching time is also known as "user space latency". This time has severe jitters due to low kernel system clock rate (HZ value) and due to the lack of fine grained kernel preemptability. The HZ value is in order of only 100Hz to 1000Hz. 2.4 Linux kernels are not preemptable and for vanilla 2.4 kernels the HZ value is 100. The main source tree maintained by Linus Torvalds and stored in www.kernel.org are termed as vanilla kernels. 2.6 kernels are preemptable with high granularity. Vanilla 2.6 has a HZ value of 1000. Redhat 8 has a HZ value of 512. A 2.4 Linux scheduler has to wait till the next system clock tick or wait for another task (high priority kernel/ user task or softirq or ISR) to voluntary yield the CPU. Once CPU is available the blocked user task can be scheduled. On the other hand a 2.6 Linux scheduler waits for the next system clock tick or the next preemption point. Both these two events happens earlier than voluntary CPU yielding, thus a 2.6 kernel is observed to manifest lower user space latency than vanilla 2.4 kernels in general [32]. However no guarantees can be given about which kernels - 2.4 or 2.6 have lower worst case latencies, because all the paths in Linux kernels and device drivers that cause these large worst case latencies are yet to be identified and improved upon in later 2.6 versions. These long paths exist since earlier versions of 2.4 Linux.

Worst case user space latency jitter in order of 300 milli second or more has been reported for Redhat 8 at high interrupt load [33]. For vanilla 2.4 kernels the average scheduler latency is in order of 10 to 50 milliseconds where as worst case latency can be greater than 200 milli second [34]. For all Linux kernels, likelihood of higher latencies sharply increases with interrupt load [32]. Higher interrupt rates can be due to higher network load or computations which generate hard disk activities or asynchronous interrupt driven data acquisition over I/O bus (PCI/ ISA).

The aggregate kernel time is the combined response time for the network softirq and the scheduler latency. The average value of this time is in order of 25 to 90 microsecond,

depending on the packet payload size for Redhat 8 on PII 333Mhz Dell desktop. Processing bigger packets take more time due to data copy operations involved. These data copy execution times increase proportionally with packet payload sizes. For a 64 byte packet, a jitter of around 4435 microsecond was observed on Redhat 8 for the combined protocol processing softirq response and user space latency. Therefore the time taken to deliver packets to the user space, can be anything between 40 microseconds to over 5000 microseconds from the instant it was DMA transferred by the NIC.

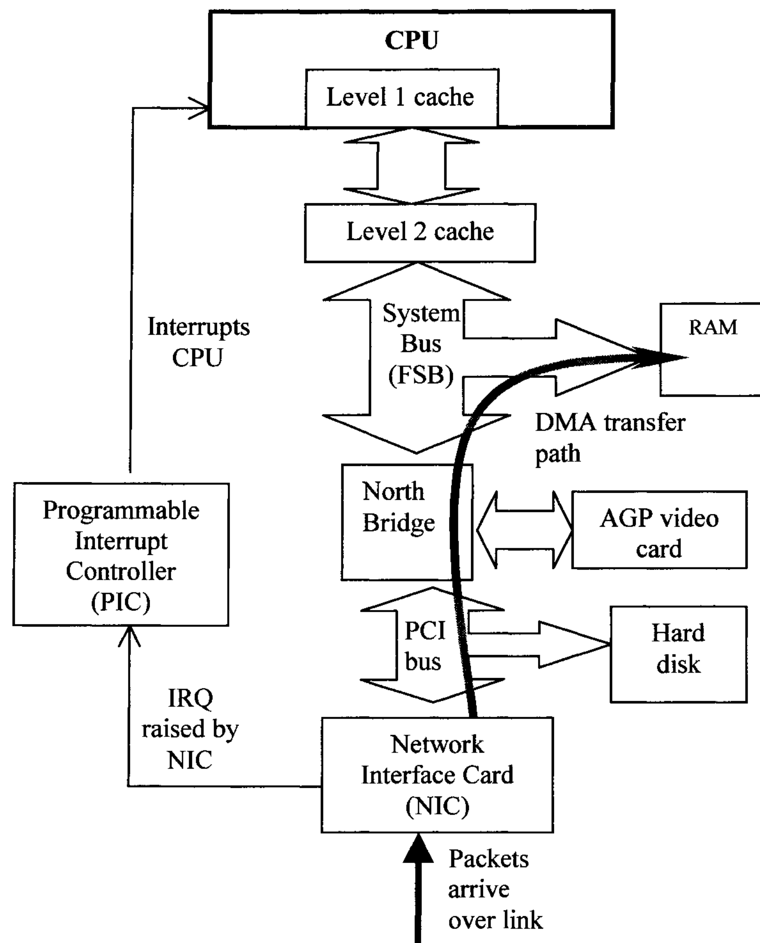
The user space event data processing execution times depends on the nature of the application and it is independent of the Linux kernel behavior, though its response time will suffer from the non real-time nature of the Linux OS. This happens because even the highest priority real-time user space task may be frequently preempted by the ISRs and softirq tasks when the system is loaded by high interrupt rates, due to heavy disk or network I/O. Interrupt based I/O and non real-time nature of the Linux are the factors behind such system response jitters.

In addition to the above pure software driven factors, hardware actions may also cause response jitters. VGA/AGP video card and hard disk operations in the background may also introduce jitters in ISR, softirq, context switching and user space task response times. These bus-mastering devices operate independently and their operations are not synchronized with the host CPU. These operations cause bulk data traffic over PCI and FSB bus by DMA operation (Fig. 3.6, next page) which are notorious for locking the I/O and system (FSB) bus and hogging time.

Bulk DMA transfer operations lock the PCI and FSB bus away from other CPU transactions till a number of PCI DMA cycles are completed. Setting up a PCI DMA transfer takes time, so there is a tendency to do as much transfer as possible (and result maximum PCI and FSB bus lock up) in a single setup cycle to amortize the set up cost

over many data byte transfers. Even though DMA transfer does not involve CPU time directly, but bulk DMA transfers do take time on the FSB, which may stall the CPU. CPU may have to wait for the access to FSB for memory operations in case of cache misses. Though the north bridge chip have internal buffers (4 level deep) for FSB, PCI and AGP ports and are designed to reduce waiting times due to FSB and PCI bus contentions, but still PCI DMA operations can cause CPU stalls under heavy processing loads. Memory access may increase by 50 % and task execution/response time increase 300 % by these PCI DMA operations [35].

Fig. 3.6: Hardware system organization



Energy savings features of the CPU and other hardware may also introduce response jitters. CPU frequency step down, power management (ACPI, APM) in BIOS etc.

introduces uncertainty in execution times. Some newer, low cost motherboards have poor real-time behavior.

3.2.2 Problems behind lower system throughput

High task execution times are the primary causes behind the high average task response times and lower system throughput. For Unix OS variants the average packet receiving response times have two components - fixed component (per packet cost) and data dependent or "data touching" component, which is proportional to the data payload size (per byte cost) [41]. Table 3.1 presents the approximate contributions of different layers and operations in the network processing response time for small packets (< 200 Byte) in Ultrix OS on a DEC Station 5000/200 system with 100Mbps FDDI [36].

Table 3.1: Factors behind packet processing response for Ultrix

Layers	Time spent	% of total
NIC Driver layer/ Interrupt servicing	57 μsec.	21 %
UDP/IP Protocol Layers	102 μsec.	39 %
Socket layer (kernel-to-user-space border crossing)	104 μsec.	40 %
Total	**263 μsec.**	**100 %**

(Derived from the data in the original paper)

Operations	% of total
Data copy	11 %
Checksum	9.2 %
Protocol processing and error check.	35.4 %
Operating system	11.1 %
Memory allocation.	16.6 %
Other	16.6 %
Total	**100 %**

Some of these operations in Table 3.1 can take place in multiple layers, for example data copy operation takes place in NIC driver and socket layer. Ultrix's network processing architecture is similar to that of Linux's, so the corresponding figures for Linux are expected to be similar. The estimated latency for NIC driver, protocol and socket layer

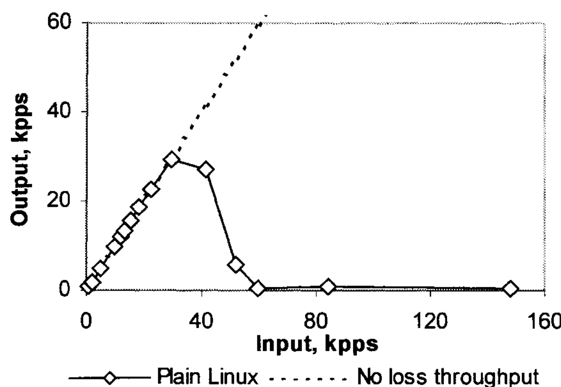
execution are : 21%, 39% and 40% of the total. Around 20 % time is spent in data touching operations like data copy and checksum, which constitutes per byte costs.

The issues that limit the network processing throughput [38] are -

- High interrupt servicing overhead : This is a per packet processing cost. This is different than interrupt latency, though the OS interrupt dispatcher execution time constitutes a part of both interrupt latency and interrupt servicing overhead. This overhead includes the time consumed in context switching, storing retrieving process states, memory cache and TLB purging due to process preemptions.

  For high packet arrival rates the host processor is overwhelmed by constantly servicing the interrupts, having no time for performing any other useful operations on the arrived packets, as result system throughput drops. Significantly high interrupt service overhead is one cause behind the "receive livelock" phenomena [39] which even troubles the fastest Ghz processors. Interrupt servicing may claim upto 30% of the total CPU time for a 2.4Ghz P4 CPU [40]. Due to livelock phenomena, for a PII 333Mhz system with Redhat 8 (Fig. 3.7), the throughput starts dropping if the input packet rate is increased beyond 30 kilo packets per seconds (kpps).

Fig. 3.7: Receive livelock in Linux kernel.
(Redhat 8, PII 333Mhz with 3C905B-TX)



Beyond 60 kpps the kernel is too overloaded and starves the user space task completely and as result the throughput drops to near zero. Interrupts also deteriorate performance by increasing cache thrashing and cache misses.

- Data copy and memory allocation costs: Due to high aggregate incoming data rate, the copying of data between DMA area to regular kernel memory and subsequent copy and delivery from kernel to user space, takes significant CPU time. These data copy operations affect the NIC's interrupt service routine response and kernel's socket layer response times. Around 11 % of the total processing time may be spend in data copy operations during network processing (Table 1). The checksum operation is taken care of by the contemporary NICs so it is not an issue anymore. The memory bandwidth being the main hardware bottleneck, it limits the network processing throughput and allows less time for event data processing in the user space [40]. Thus, this factor remains the biggest contributor to per byte processing cost.

  Instead of copying data, when some NIC drivers, pass memory address (pointers) to transfer packets from NIC DMA region to the packet queue new memory buffers ("packet buffers") are allocated and DMA mapped to receive new packets (section 3.1.2). Provisioning new memory buffers is another costly operation, which consumes significant amount of CPU time, around 16.6 % of total (Table 3.1).

- Redundant protocol processing in network stack in the kernel : Linux being a general purpose operating system, it has lot more protocol processing steps than needed for real-time data acquisition. So for our application many of these protocol processing steps are unnecessary and can be done away with. Even the structured layering of network stacks may be avoided to save on execution time [38]. These factors primarily contribute to the per packet cost and may constitute around 39 % of the

total network processing time (Table 3.1). Therefore, avoiding many of these layers can reclaim significant amount of CPU time.

- Kernel-to-user-space border crossing costs : The task that consumes the arriving event data usually runs as a process in the user space. The kernel carries out the protocol processing and delivers the packet to this user process. Therefore packet receiving and delivery involves two border crossings - a border crossing for data and a border crossing for the control. Border crossing for data involves a data copy from kernel to user space, whereas border crossing for control involves a system call from user space, a context switch and the return path for the system call made. The response time for all these operations together can be significant and can be as high as 40 % of the total processing time (Table 3.1). The border crossing response time contribute to the per packet cost.

- Buffer overflow problem: Packet losses occur when packets are dropped due to buffer overflow. Buffer overflow may occur due to variety of reasons. Three primary causes behind buffer overflow are - (i) the lack of execution balance between "producer" and "consumer" tasks, (ii) the jitter in "consumer" task response and (iii) jitter in the context switching time between the "producer" and "consumer" task

  There are three "producer-consumer" pairs - softirq-user space tasks, ISR-softirq tasks and NIC firmware-ISR tasks. Between these "producer-consumer" pair, there is DMA, packet queue or socket buffer to hold the semi-processed packet / data. In case of the ISR-softirq "producer-consumer" pair, the ISR has higher task priority over softirq tasks. So the finite sized packet queue can overflow under heavy packet arrival load.

  Data bits arriving at the NIC are streamed into NIC's onboard FIFO buffer. Once all the bits of a packet have successfully arrived and if space is available in the DMA

buffer, then these packets are immediately DMA transferred from NIC's FIFO buffer to the host's DMA buffer (Fig. 3.2). If no DMA buffer space is available then NIC cannot offload its FIFO buffer, as a result the FIFO buffer can get filled up. Once NIC's FIFO buffer gets filled up, the NIC drops bits of subsequent packets. As DMA buffer overflow cause the NIC's FIFO buffer overflow, so this packet dropping phenomenon is simply termed as DMA buffer overflow in subsequent discussions.

NIC firmware task is not within the control of developers, so NIC firmware task and ISR execution cannot be balanced. Jitter in ISR response may cause the DMA buffer to overflow. Higher interrupt latency jitter will delay the invocation of ISR consumer task, this might result in DMA buffer overflow if its size is small.

Similarly, jitter in user space task response may cause socket buffer overflows as the socket buffer has a limited size of 64 Kbytes by default. In addition to this, jitter in kernel-to-user-space context switching times may delay in starting user space task, as a result the socket buffer may also overflow. Such overflow may start happening at lower packet rates, even before the softirq task may start starving the user space task.

In a given receiver for small sized packets, the order of overflow precedence is: packet queue and DMA buffer. This means the packet queue overflows at lower packet rate before the socket buffer can overflow and DMA buffer overflows at a much higher packet rate. Socket buffer overflow may not be observed at all. For smaller packets of 64Bytes, the packet queue overflows before the socket buffer capacity is breached, this is because 300* 64 Byte capacity of packet queue is less than 64 KByte capacity of the socket buffer. For very large packet sizes the socket buffer can overflow before the packet buffer overflows. In that case, the order of overflow precedence is: socket buffer, packet queue and DMA buffer.

Packet counters were placed in the Redhat 8 packet receive path to estimate the location and extent of packet loss (Fig. 3.8, next page).

Packet counter placed in the ISR measured the packets received in the DMA buffer and counter placed in the user space task measured the packets received at the socket buffer. All packets that reach DMA buffer enters the kernel and all packets that reach socket buffer enter user space. The packet loss patterns are presented below, which demonstrate the buffer overflow phenomena (Table 3.2, next page).

Fig 3.8: Packet loss measurement instrumentation for Linux

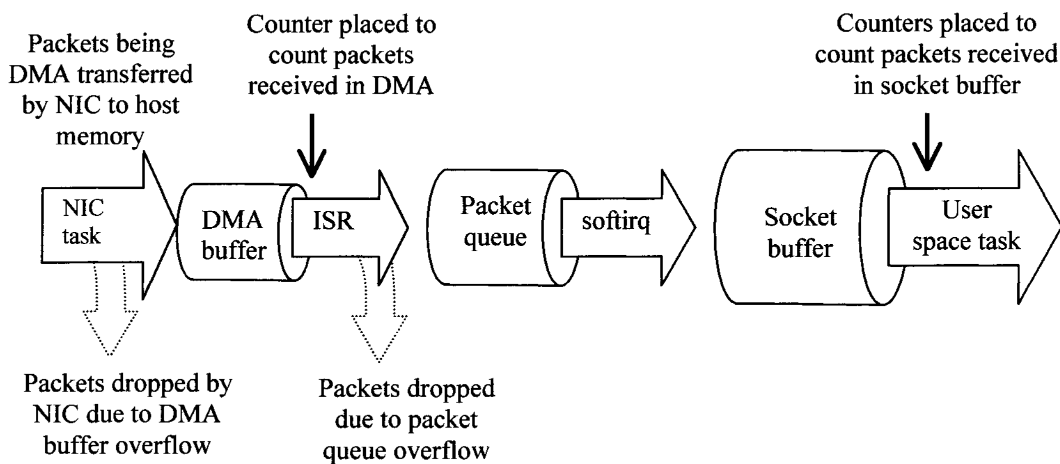


Table 3.2: Effect of DMA buffer size on Linux throughput

DMA buffer size	**Packets rate** 200,000 64Byte packets (100 %)	**Packet received in DMA buffer** (% of 200,000)	**Packet received in socket buffer** (% of 200,000)
32 (Default)	143 kpps	2779 (1.4%)	499 (0.25%)
	147 kpps	376 (0.19%)	376 (0.19%)
128	143 kpps	200,000 (100%)	903 (0.45%)
	147 kpps	1089 (0.55%)	671 (0.34%)
512	143 kpps	200,000 (100%)	763 (0.38%)
	147 kpps	52,790 (26%)	742 (0.37%)

From the above table it is evident that when the DMA buffer size is 32, the default size in 3Com905B-TX NIC driver, then the DMA buffer overflows at a packet rate of 143 kpps. Only 2779 packets out of 200,000 are stored in DMA buffer, rest are discarded by the NIC, and out of 2779 packets only 499 reach the user space, rest are discarded in kernel due to packet queue overflow. If the DMA buffer size is increased to 128 or 512, dramatic improvements in packet availability is observed in DMA buffer. For 143 kpps, a DMA buffer size of 128 is sufficient to store all incoming packets (100%). A size of 512 improves the packet availability form 0.19% to 26% for a packet rate of 147 kpps. A bigger DMA buffer also improves the packet availability in the user space. However due to limited packet queue size not all packets can reach user space for high arrival rates. Packet loss due to DMA buffer overflow increases with packet rate for all DMA buffer sizes.

3.3 Solution requirements

3.3.1 Task balancing is a challenge

The solution space, which can address the previously mentioned problems is constricted, not all common sense approaches will work. Balancing the three "producer-consumer" pairs to improve the overall throughput is difficult. For a given CPU and architecture it may be possible to balance these tasks by tuning the buffer sizes, but such optimization is likely to fail for a different CPU speed as all the latencies and their jitters involved may not scale down proportionately with CPU speed.

3.3.2 Increasing buffer sizes is not solution

DMA and socket buffers can be over provisioned, their sizes can be increased to get a lower packet loss, but increasing packet queue size is not a viable option. Increasing packet queue size actually deteriorates performance by aggravating the livelock

phenomena. A larger packet queue will allow more packets to enter the kernel, this will increase the kernel task load and starve the user space task. In such situation, the socket buffer will not be cleared and it will overflow to result more packet losses than earlier.

To gain a small improvement in the packet loss performance the DMA and socket buffers may require disproportionately large over provisioning. Such large over provisioning of kernel memory have multiple adverse performance implications. All the memory used in a real-time system have to be pinned down in the RAM so that they are not swapped out to the hard disk by the Linux memory manager. DMA buffer memory also needs to be pinned down so that NIC can successfully make DMA transfers. Reserving very large pinned down kernel memory for the buffers may force eviction of other virtual addresses to disk swap space. Though kernel space memory also remain pinned so kernel tasks will not suffer in general, but parts of other OS related tasks, for example, X terminal operation may be carried out in process context which may use non pinned memory. Nevertheless, there is limit on how much total memory can be pinned without slowing down the system by too many "page faults".

Pinning large memory segments raises frequency of "page faults". "Page faults" exceptions arise when a memory segment has been swapped out to the disk and it is not available in the RAM when demanded. In case of such exceptions, the memory is brought back from the disk swap space on to the RAM. Such "page faults" introduce task response jitters and the disk I/O interrupts further amplifies this jitter in the system response. Such system activity also increases cache thrashing.

With large buffer memory, the virtual memory manager will have to make large memory strides for any operation, which increases cache misses. Present hardware architectures allows augmenting of RAM but Level 1 and Level 2 cache cannot be increased at will. Thus even though more RAM may be added but without proportionate increase in cache

sizes the performance will deteriorate when large memory is utilized. Thus over provisioning the system by simply increasing the buffer sizes is very expensive and not an elegant solution.

A large kernel to user space context switching time jitter is the cause behind the socket buffer overflow, a large socket buffer can only mask this problem. If the root cause, i.e. high context switching time jitter, is not addressed then this will anyway cause large jitters in packet delivery latency. Thus this problem will still hit the other performance elements, if not the packet loss performance.

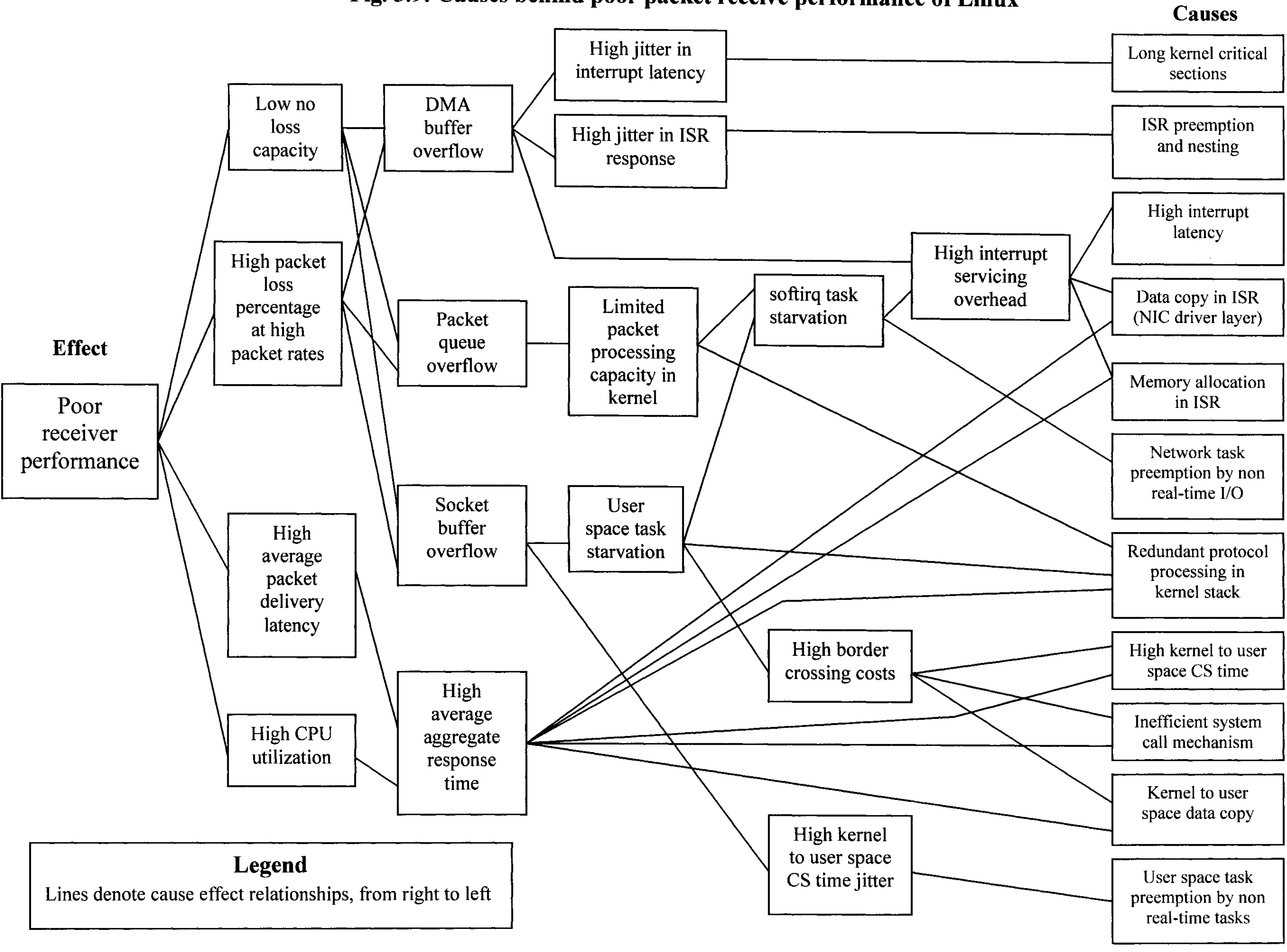
An approach, which will improve the efficiency of the packet receiving operations by minimizing both: the average response time and the response time jitter, will be a real solution. However this approach has to fit well with the rest of the Linux architecture mechanisms and bring a balanced improvements in all the four system performance elements.

3.4 Summary

Poor packet receiving performance of Linux is primarily due to its design. Linux is designed to provide multiple services to multiple users fairly. Therefore the interrupt and packet receiving mechanisms are shared resources placed inside the kernel. To have a faster interrupt service routine, the packet receiving task is split into two parts - a quick portion to be handled as an ISR and a time consuming portion to be handled as softirq task. Whereas the application specific event data processing task had to be implemented as the user space task thread. Linux did not guarantee resources required to complete packet receiving tasks in bounded time or their scheduling sequence. This multiple thread scheme without real-time task scheduling gave rise to jittery task response, context switching times and the need for multiple interim buffers between the threads.

Fig. 3.9 summarizes the causes behind poor packet receive performance in Linux. The leaves in the right most side represent Linux's limitations. The causes behind high packet delivery latency jitters is not included explicitly because those causal factors are anyway exposed from analysis of the other three performance elements.

Fig. 3.9: Causes behind poor packet receive performance of Linux



Chapter 4: Contemporary Solutions

Some of the limitations of Linux that undermine the performance of its packet receiving architecture, have been addressed as general OS problems. Real-time support for Linux provides features that can address the response jitter problems. In addition to these, some solutions have been proposed to improve the packet receiving performance in Linux. Thus the solution space that address these problems consists of four segments - (i)improvements in Linux as general purpose operating system (GPOS), (ii) real-time support for Linux, (iii) some miscellaneous performance improvement schemes that do not fall in earlier two categories and (iv) complete solutions to receive packets arriving at high rate. But even these approaches have limitations, and do not address all the problems identified in the previous chapters. Next few sections discuss these artifacts along with their limitations. These were considered relevant for packet capturing in a NMS /NIIDS application. Some of these suitable artifacts, that can be utilized to design a high performance packet receiving architecture, are selected. There are other related works which cannot be categorized as solutions in their present forms, these are discussed later in Chapter 8.

4.1 Operating system improvements

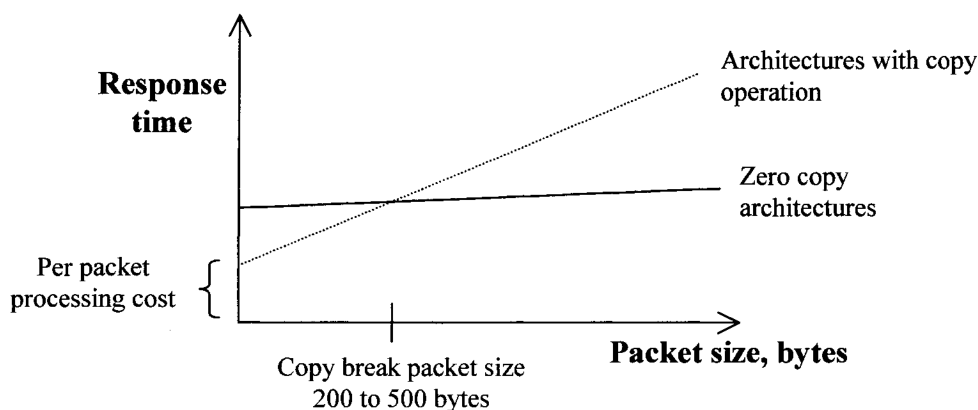
4.1.1 Avoiding data copy operations

Data copy reduction has been addressed in Unix variants by using shared memory between different network stack layers within kernel and between kernel and user space. This is known as "zero copy" approach in the literature [38,42,43,44,45,46]. Instead of memory copy only the address of memory locations are exchanged between kernel layers and kernel to user space (pointer passing).

Superior performance of zero copy TCP processing on Sun SPARC and other platforms has been demonstrated [45,46]. By page re-mapping memory is shared between kernel and user space and the kernel to user space. Linux uses similar shared memory between kernel and user space in some of its device drivers which involve bulk data I/O. These works primarily improved performance of very large packets (>1500B ATM packets) but they did not address the per packet costs which hit the performance of small packets.

The available literatures demonstrated benefits of "zero-copy" approaches in reducing CPU utilization and improving throughput at higher packet payloads. However performance improvements are visible only at higher payload sizes and some implementations [42,43,44] deteriorated performance at lower packet sizes (Fig. 4.1).

Fig. 4.1: Performance limitations of zero copy implementations



Unix and Linux packet processing has both per packet and per byte processing costs, therefore the response time locus slopes upwards as packet size increases [42]. These zero copy implementations [42,43,44] raised the fixed per packet cost component of the response time. But per byte cost component is low for zero copy architectures, therefore its response time locus is a flatter line. Zero copy mechanism only made sense when the packet size was beyond a threshold, the "copy break packet size", which is between 200 and 500 Bytes for various implementations and platforms [42,43,44, 3c59x Linux driver].

Due to this reason, NIC drivers copied smaller packets from DMA buffer to the kernel memory and made zero copy transfers when the packet size was greater than the "copy break size".

Poor performance for lower packet sizes is not intrinsic to zero copy mechanism, but it resulted from the manner the zero copy scheme was implemented. For each zero copy transfer cycle, full sized (1500 byte for IP) memory buffers are allocated in real-time to receive subsequent packets. Allocating memory for maximum packet sizes requires some fixed amount of time, which offsets the benefits received from avoiding copy mechanism. At lower packet sizes the benefit of zero copy is smaller whereas cost of full packet sized memory allocation are incurred. Therefore the net benefit realizable from zero copy is negative at smaller packet sizes. This limitation can be avoided if a large circular buffer is constructed at set up time, and if this buffer is allowed to be the staging area for the protocol processing layers. A large circular buffer will avoid the need for buffer transfers across various layers, and hence avert memory allocation in real-time.

Apparently an alternate solution to avoid memory allocation is to recycle packet buffers. In this scheme, the packet buffers will not dissolved in the socket layer after transferring the data to user space, but they will be returned to a empty packet buffer pool. Whenever a packet buffer is required in the NIC layer, it is taken out of the pool instead of allocating afresh. A real-time network stack named RTnet implemented this logic [47]. In fact, a bigger circular buffer actually clubs both the DMA buffer and the pool in a common structure.

4.1.2 Minimizing scheduling latencies

Some popular low scheduling latency solutions are - preemptable kernels, O(1) schedulers, low latency kernel patches and higher operating system clock rate (HZ value). Many of these have been implemented in 2.6 kernels.

Kernel preemption is achieved by two approaches [48]. In the first approach, preemption logic is placed with specific kernel locks, kernel get preempted every time the lock is released. As these locks are used throughout the kernel, so preemption points gets automatically deployed at various points in the kernel. In the second one, explicit preemption points are manually added at various places in the kernel which correspond to long kernel execution paths. The location of these long paths are often determined based on experimental observations on Linux's real-time behavior. Though these approaches improved the average and median latencies, but these approaches failed to remove all the worst case latencies due to long critical section paths. Even after implementing these approaches, worst case latencies greater than 500 microsecond still remained in the 2.6 kernels for Intel Celeron 650Mhz CPU [32]. For the same CPU, vanilla 2.4 Linux kernel resulted 4446 microsecond preemption latency.

4.1.3 Managing border crossing costs

Response times of system calls made from user space can be reduced if CPU registers are utilized in the system call mechanism. In 2.6 kernels such mechanisms reduced the system call response time by a factor of 2 for Intel P4 CPUs [49]. Border crossing involves system calls, therefore the border crossing costs can be lowered by employing 2.6 kernels.

4.2 Real-time support for Linux

4.2.1 Managing jitters in interrupt latency and ISR response

High interrupt latency jitters are caused due to interventions from Linux kernel's critical sections, which disable the interrupts when they execute. These critical sections are associated with specific kernel locks. In the process of making Linux kernel preemptable in 2.6, developers re-engineered these big granular locks to smaller ones. But there is no guarantee that these "lock breaking" approaches exhaustively removed all the long critical section paths. Still some difficult to find long critical sections may exist in 2.6 preemptable kernels, which may manifest occasionally.

ISR response jitters due to nested ISR execution phenomena may be reduced by appropriate interrupt prioritization scheme. A prioritization scheme will shield a high priority ISR from being preempted by a lower priority ISR. Normally Linux does not implement multiple levels of interrupt prioritization. Only two priority levels, "fast" and "slow", are offered by Linux. These levels are defined for an interrupt when its ISR is setup with or without SA_INTERRUPT flag. Fast ISRs are executed with all interrupts disabled, whereas slow ISRs execute with interrupts enabled. Any new interrupt with a "fast" ISR can preempt a currently running "slow" ISR. But a fast ISR may not preempt another fast ISR, whereas slow ISRs may only preempt other slow ISRs. System timer (e.g. PIT) and hard disk interrupts are handled by fast ISRs. NIC, some floppy disk and VGA ISRs are defined as slow type. Some ISRs that preempt NIC ISR, can take longer time to execute therefore a NIC ISR is vulnerable to response jitters. This two level prioritization scheme is implemented at the software level by masking interrupts at the CPU. The order of precedence for external interrupt (IRQ0-15) reporting, defined by the legacy 8259 chip based PIC hardware organization is a different mechanism. This IRQ reporting precedence only defines which interrupt request will be first reported to the

CPU when multiple IRQs are triggered simultaneously at the PIC. Generally the order of precedence are IRQ 0,1,8,9,10,11,12,13,14,15,3,4,5,6,7.

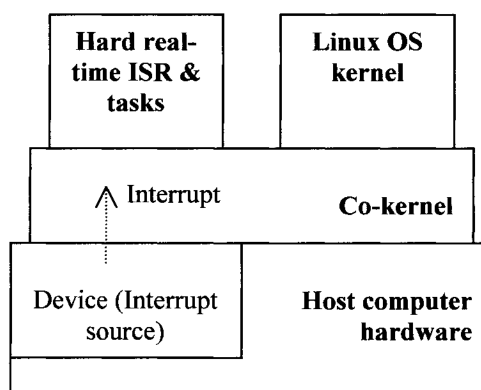
This two level interrupt priority scheme of Linux, is not enough to avoid NIC interrupt latency and ISR response jitters. System timer interrupts (e.g PIT) should have higher priority, NIC interrupt should have a priority level lower than system timer devices. Hard disk, video card and other I/O device interrupts should have a priority lower than that of the NIC. Thus this scheme will require at least three or more levels of priorities. The PIC hardware can be programmed to implement multiple levels of priority for all the interrupts. This prioritization scheme can be implemented over and above PIC's interrupt request precedence and should replace Linux's two level interrupt priority system. With such interrupt prioritization, only a high priority interrupt's ISR can preempt a low priority interrupt's ISR. This protects the high priority interrupt's ISR from preemption by a low priority ISR. The interrupts involved in real-time tasks can be granted higher priority to decrease their vulnerability from preemption and improve their response.

A solution, "real-time interrupt (RTirq) patch" [50,51] implements such interrupt prioritization by programming IO-APIC chip and provides kernel locks with multiple priorities and usage rules. But this solution has limitations. This requires kernel re-compilation, which is a complex process. This solution is packaged as a kernel patch for vanilla Linux 2.4.23 and 2.6.2 to 2.6.5 kernels. Applicability of the available patches is limited to certain versions of Linux kernels, not all versions of 2.4 and 2.6 vanilla kernels can be patched. Many kernel compilation options do not compile with this patch. This means, users may have to forego many useful kernel features to use this solution in its present state. For example, these patches cannot be complied for multiprocessor SMP systems. This solution is yet to be verified to work successfully on a wide variety of motherboards. This solution is hardware specific, it leverages the IO-APIC chip features and interrupt hardware organization. Patches are not yet available for wide variety of

CPU, as covered by Linux. Only Intel x86, AMD Athlon, etc. have been reported to work on a limited set of motherboards having a certain kinds of interrupt hardware organization and motherboard chipsets. Finally, the patch for 2.4.23 vanilla kernel only supports legacy interrupt system organization, where IRQ0-15 are type XT-PIC, i.e. the interrupt type that are not routed through IO-APIC. Interrupts that are routed through IO-APIC are not covered. However this solution do have some advantages, the average and worst case interrupt latency performance of this solution is claimed to be better than other co-kernel (e.g. Adeos) and threaded ISR solutions. "Real-time interrupt's" response time approaches the hardware response times for a given hardware as claimed in one email from its developer. This solution enforces prioritization even in cases when multiple interrupts arrive at the PIC simultaneously and due to the PIC organization, a low priority interrupt may arrive at the CPU before a high priority interrupt.

The problem concerning un-deterministic interrupt latency and ISR jitters have also been addressed by other, hardware independent approaches. RTLinux is one such approach that implements a co-kernel layer sitting between Linux and the hardware (Fig. 4.2) [48].

Fig. 4.2: Shielding real-time interrupts by co-kernel

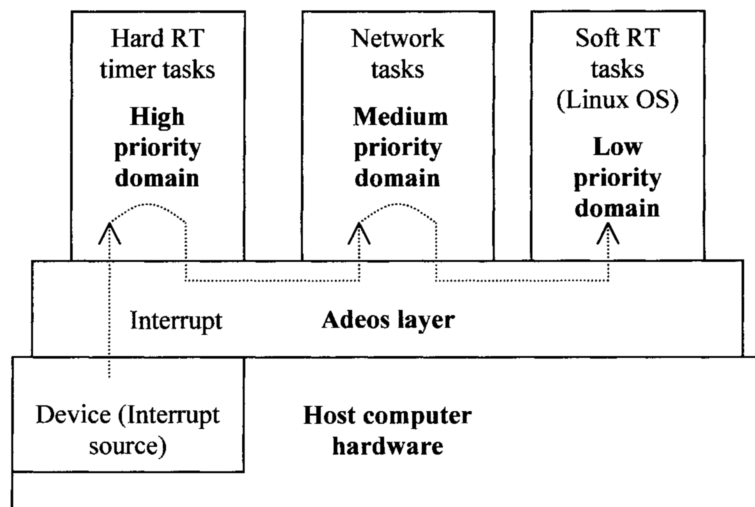


The co-kernel hosts the hard real-time ISRs and tasks that need shielding from Linux's intervention, outside Linux domain. The co-kernel also hosts Linux. The "software PIC"

in the co-kernel layer determines dispatch order of interrupts to the real-time task and Linux, irrespective of Linux's critical sections and interrupt enable/disable status. The software PIC implements its own interrupt priority rules. Thus the real-time task/ISR is protected from Linux's interventions. The co-kernel is implemented to be portable across different hardware. RTAI based on Linux is another similar co-kernel approach which utilizes Adeos [48]. The limitations of co-kernel approaches are discussed later under section 4.2.2.

Other than RTAI, Adeos nanokernel is another distinct hardware independent approach (Fig. 4.3), which can shield the real-time interrupts from Linux kernel's critical sections. By shielding from Linux's interventions, the jitters in interrupt latencies and ISR response times for high priority interrupts can be bounded.

Fig 4.3: Shielding real-time interrupts by Adeos layer



Adeos layer sits between client domains and the hardware [52]. Linux can be one of the client domains, real-time ISR, tasks, schedulers or co-kernels can be other domains. Unlike RTLinux and RTAI co-kernels, Adeos does not include a real-time scheduler. The interrupt priority is also realized by a different mechanism. Adeos layer intercepts any

hardware manipulations made by the client domains. On the other hand, it intercepts the events raised by the hardware. When an interrupt is raised, the Adeos layer intercepts it and passes it to the first high priority domain. This domain executes whatever tasks it intends to execute as a response to this interrupt and then it notifies the Adeos layer. The Adeos layer then passes this interrupt to the second high priority domain and so on. A domain may terminate the interrupt at its level, so that the interrupt is not passed to next lower priority domains, or it may choose to ignore the interrupt, so that Adeos may quickly send the interrupt to the next one. Each domain may have its own interrupt handlers or ISRs. A domain may choose to serialize the interrupt handling, it may stall the interrupts at its level till the handling of the current interrupt is completed. A lower priority domain may be preempted by a higher priority domain if an interrupt is reported at the higher priority domain.

Any hard real-time task, like network processing which should be shielded from Linux's critical sections and interrupt preemptions, can be placed in a higher priority domain compared to Linux. The hard real-time timer tasks can be the highest priority domain. Linux may have its own soft real-time system timer tasks in addition to these hard real-time timers. The hard real-time timer task domain may ignore NIC or other interrupts, which are passed over to the network processing and Linux domain. The network processing domain may terminate the NIC interrupt at its level, whereas ignores other I/O interrupts. When a network processing task is in progress, the network processing domain may stall a newly arrived I/O interrupt at its level, and it will only release that interrupt to Linux when the current network processing task is over. Thus effectively the client domains can acquire distinct multiple level of priorities in regards to interrupt handling to manifest bounded task latencies.

Adeos approach has some disadvantages. The average interrupt latency is in the order of 30 microseconds even though the jitter is low. This figure of 30 microsecond is on the

higher side. Adeos only manages ISR response time jitters, not interrupt latency jitters. Adeos may not guarantee that a high priority ISR will start executing first when multiple interrupts occur simultaneously. The PIC organization may define a different interrupt reporting precedence. Thus when multiple interrupts are raised simultaneously at the PIC, a low priority interrupt, as defined by Adeos, may supercede a high Adeos priority interrupt, for e.g. IRQ9 will be reported earlier than IRQ5, even though IRQ5 should have higher effective priority in Adeos. In that case Adeos will run the ISR corresponding to IRQ9 first while the ISR of IRQ5 has to wait. However the interrupt latency of IRQ5 in such cases is bounded by the execution time of ISR for IRQ9. The execution times of ISRs are typically in the order of 10 microseconds, so the order of interrupt latency jitter is not much, at least one order smaller than the 250 microsecond interrupt latency jitter in case of Linux.

Priorities to ISRs can also be granted by threaded ISR approach as in TimeSys's Linux variant [48]. ISRs are dispatched as schedulable threads, therefore ISRs can have multiple levels of priorities and even lower priorities than hard real-time tasks. So high priority ISRs and hard real-time tasks can be protected from preemption by low priority ISRs. Mutex can serialize execution of these threaded ISRs wherever needed. However thread dispatch time will add up to the interrupt latency and thus the performance of this approach can never match the performance of "real-time interrupt (RTirq) patch". TimeSys Linux has its own proprietary kernel.

4.2.2 Improving response of real-time packet receiving tasks

Though Linux is not a RTOS, but attempts have been made to bound the worst case response times of tasks within Linux. Section 4.1.2 and 4.1.3 presented some approaches that also improve response of Linux domain tasks. But these approaches only reduce average response times but do not bound their jitters. Other than these, real-time variants

of Linux have emerged and real-time service support that can work along with Linux, have been proposed.

As Linux kernel does not have a real-time scheduler, so a dual or co-kernel approach has been proposed (Fig. 4.2, previous sub-section). In addition to the Linux kernel, another kernel with a real-time scheduler runs in such architectures. This real-time scheduler runs the whole Linux kernel as an idle task, only when no hard real-time task is runnable. Linux scheduler sitting within the Linux kernel schedules Linux domain tasks as it would normally do. Tasks within Linux domain maintain their relative priorities within themselves. The Linux kernel is modified to accommodate the co-kernel. The co-kernel may be loaded as an "Loadable Kernel Modules" (LKM) during startup from Linux environment. Real-time ISR and task codes are packaged and loaded as LKMs. The hard real-time tasks are run by the co-kernel's real-time scheduler, outside Linux domain and beyond the scope of Linux scheduler. The real-time co- kernel does not provide other extensive services as Linux, so it has a small foot print. These are also known as "micro" or "nano" kernels. RTLinux and RTAI are examples of such dual kernel approach.

Management of Linux's kernel preemptability and long critical section paths are not exhaustive, and they are sensitive to operating system evolution and driver performance. In the process of evolution, new long critical section path may be introduced inadvertently. Moreover device drivers are not part of core kernel, and not so well written drivers may be deployed which may introduce long critical section paths. So it is quite difficult to manage the real-time performance of Linux with those approaches as presented in section 4.1.2. In such scenario the co-kernel and Adeos approach have a distinct practical advantage. These approaches guarantee real-time system behavior with Linux irrespective of Linux's intrinsic non real-time behavior.

However all co-kernel approaches have practical limitations. The hard real-time tasks and ISRs have to run outside Linux domain. Run time system function call to Linux cannot be made from hard real-time tasks. In general, run time process related services like memory protections etc., from Linux is not be available. These put severe constraints in the design space. To develop a real-time application, hard real-time and soft or non real-time tasks need to be clearly identified and segregated. The non real-time components can be developed to run under Linux, whereas hard real-time components have to be developed without utilizing Linux system APIs. This segregation becomes challenging for a complex application where lots of interaction between real-time and non real-time tasks are expected. Inter-task communications between real-time and non real-time tasks across Linux and co-kernel domain borders are quite complex and severely limits the performance of such design. Exhaustive support for inter-task communication and interactions may not be always available for the co-kernel used. Memory protection may not be available for the hard real-time tasks, so designers have to take precautions and manage memory access themselves.

However some of these limitations are addressed by approaches like LXRT for RTAI [53]. LXRT-RTAI supports hard real-time tasks in Linux user space. This means the hard real-time tasks can enjoy the memory protection offered by Linux and can be run as a user space process from within Linux domain. A more extensive real-time POSIX support is available which ease development. However RTAI documentation warns that performance of LXRT-RTAI in terms of lower CPU utilization may not be so good [54]. PSDD on RTLinux (FSMLabs) and LynxOS (Lynux works) are other possible Linux based options which support hard real-time tasks in user space [55,56].

Real-time response of Linux tasks in multiprocessor systems can be improved by dedicating specific CPUs for real-time tasks, while shielding them from interference from non real-time activities like asynchronous I/O related ISRs and kernel critical sections

[33]. Concurrent Computer Corporation achieved worst case real-time response in order of 107 milliseconds in their shielded CPU RedHawk Linux (based on 2.4.21 kernel) compared to Redhat 8's worst case response of 323 milliseconds. This approach has limitations, it requires multiprocessor systems and a modified kernel, which may not be as cheap as Redhat or freeware vanilla Linux kernels and low latency patches.

All the real-time support for Linux may be computation intensive, which means determinism in task response time may be obtained at the cost of getting a much higher average response time. However this tradeoff is worthwhile to avoid buffer overflows at high packet rates under high CPU utilization scenarios.

4.3 Miscellaneous schemes

4.3.1 Avoiding interrupt service overhead

The interrupt overhead issue have been addressed in past by different approaches. Some NICs provide interrupt mitigation. The NIC issues a single interrupt for a group of packets instead for every packet received. This reduces frequency of interrupts and the interrupt overheads. Interrupt mitigation feature is not offered by all NICs. Moreover the NICs with this features do not adaptively mitigate interrupts based on CPU load situation. Mitigation will start only when packet rate is above a threshold. A NIC which can provide enough mitigation for a high speed CPU with no other system load, may fail to be effective for a slower CPU or for a CPU which has other task loads, unless an adaptive intelligence is executed in the host (packaged in the NIC driver). Moreover this approach will result in higher average packet delivery latencies.

Synchronous polling based operation instead of asynchronous interrupts has been proposed to tackle high interrupt loads at higher packet rates. In polling mode, the CPU checks the DMA buffer or NIC's shared memory frequently, if there are available

packets, then it processes them. With this strategy the interrupts and the associated overheads are avoided, thus reducing CPU utilization. However such savings come at a cost. Arrival of a packet may not detected till the next poll cycle, and thus its processing may be delayed by the poll period time in worst case. Polling increases the worst case packet delivery latency and may increase average latency.

As a solution to this limitation, a "hybrid interrupt-polling" scheme, have been proposed, implemented and studied for 10 Mbps networks [57]. The hybrid receiver operates in interrupt mode at lower packet rates but switches to polling mode at higher packet rates. In general, the decision when to switch, can be either decided adaptively at run time or can be pre-determined. Run time adaptive algorithms to decide when to switch between synchronous and asynchronous modes have been implemented and observed [57]. In this approach, the polling period is also adjusted based on the recent packet rate trends, adjusting polling period takes time as timer have to be reprogrammed every time. Therefore this approach may actually realize lower CPU time saving if the timer reprogramming is significantly expensive. The periodicity of packets on a 100Mbps network may be as high as 6 microseconds for smaller packet sizes (< 64 bytes). Software kernel timers in Linux do not have microsecond level resolution due to high scheduler latency, therefore they are not suitable for 100Mbps network operation. Using a hardware timer to pace the polling at the rate similar to the expected packet rate, is out of question. At present there is no other known general and tractable event dispatch mechanism in Linux other than interrupt mechanism. So hardware timer events can be only dispatched to the software domain by interrupts. These timer events will cause as much interrupt servicing overheads as the NIC interrupts themselves, so this cannot be a solution. However it will be interesting problem to study how CPU based LAPIC timer or TSC events may be dispatched smoothly to the software domain without causing much turbulence as the normal interrupt mechanism.

Significant portion of interrupt overhead is due to the interrupt latency. The PIC hardware performance is behind the interrupt latency. Interrupt latency does not scale down proportionally with CPU speed. Therefore, for a slower CPU, the polling overhead may be higher than the interrupt latency overhead. In such cases polling may not be beneficial to reduce the interrupt overheads.

4.3.2 Adopting efficient protocol processing

Some literatures argue the benefits of monolithic architecture for integrated and simplified protocol processing to improve the response time [38,58]. Run time flexibility associated with layering may be lost in such de-layering approaches. Ability to insert various packet filters and additional protocol stacks at run time becomes difficult. However for pure data acquisition applications such flexibility is generally not required. Therefore instead of Linux kernel's protocol processing stack, a simplified custom protocol processing stage may be employed to save significant CPU time.

4.3.3 Reducing jitters due to hardware factors

DMA bus mastering controllers of individual devices can be programmed to reduce the bus locking times. All DMA controllers provides programmable interface to achieve that. This approach requires knowledge of device hardware to carry out modifications in all device drivers. This may not be a very practical approach. The lack of device hardware documentation may limit implementation of this approach in practice. Energy savings features (APM, ACPI) can be turned off in the BIOS and during kernel compilation to improve the hardware response times. New motherboards, which are notorious for their un-deterministic response, may be avoided. Lists of suitable hardware are often available with the RTOS vendors. Choice of motherboards may severely restrict exploitation of fastest Ghz processors in real-time applications.

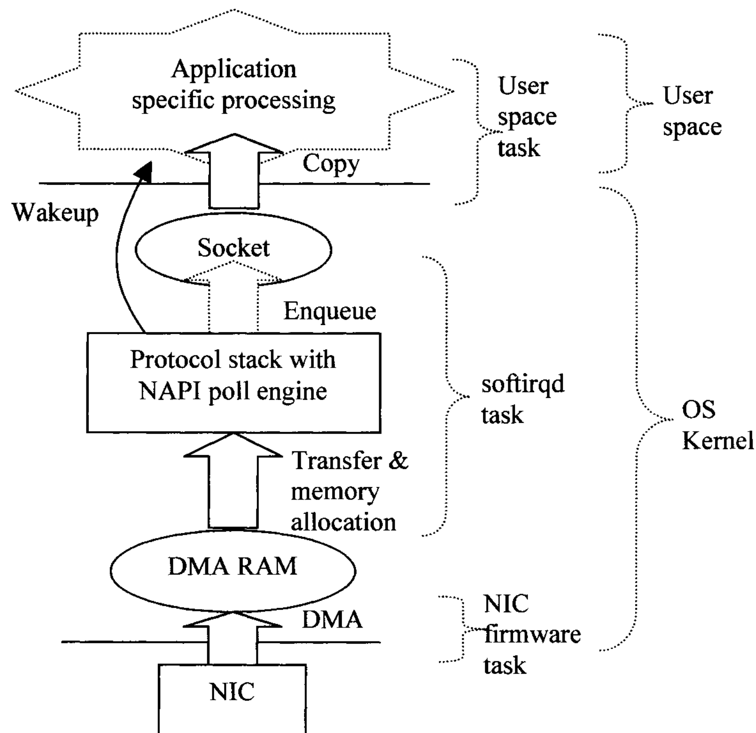
4.4 Alternative packet processing architectures

To allow packet processing in case of high packet arrival rate a few solutions have been proposed after release of 2.4 kernels. Most notable among them are NAPI and PFRING. These are discussed in subsequent sub-sections.

4.4.1 NAPI

New API (NAPI) utilizes polling based approach in high packet arrival situation [59]. NAPI is part of the vanilla Linux kernel from version 2.4.20 onwards. It was also back ported to version 2.4.18 to be included in Redhat 8. NAPI mechanism (Fig. 4.4) was designed to avoid interrupt overheads, to reduce CPU utilization and leave more time for user space event data processing task.

Fig. 4.4: NAPI
(2 tasks, 1 copy, 1 memory allocation)



At high packet arrival rates, the NAPI engine polls the DMA region, picks up the packets and performs protocol processing at regular intervals. Instead of the softirq task as in Linux, the NAPI poll engine task processes the packets through the kernel network stack. The NAPI poll engine task finally delivers the processed packets to the socket buffer and wakes up the waiting user space "consumer" task. At high packet rates the NAPI poll engine runs as a Linux kernel thread (ksoftirqd) which has lower priority than any high priority real-time user space "consumer" task. Under high packet arrival rate the NAPI mechanism uses two task threads which run on host CPU in two different contexts - ksoftirqd (kernel) and user space. It employs two buffers - DMA region buffer and socket buffer. The mechanism also involves one memory allocation during packet transfer from DMA buffer to packet queue and one copy during data transfer from kernel to user memory.

In NAPI, the NIC driver is different. As the first packet arrives and raises an interrupt, the first instance of ISR disables the interrupt and invokes the NAPI poll engine. NAPI poll engine runs as a softirq task to perform the polling, after the ISR returns. During execution of this softirq task, new packets might arrive though they do not raise any interrupts as interrupt has been disabled by the first ISR. The NAPI version of the NIC driver implements a function to access DMA memory and pick up packets. Kernel's NAPI poll engine calls this function to pick up and process packets. If all packets have been processed and no packets are pending, then the NAPI poll engine enables interrupt and shuts itself down. There is a limit (budget) on maximum number of packets that can be processed by the softirq task in a single NAPI poll cycle. With this budget and the time limit, it is intended that the poll engine should not run for a long time and starve the user space task under high packet arrival rates. If there are still some pending packets, then another softirq request is raised by the current softirq task. This next round of softirq processing is supposed to handle the remaining packets.

The softirq logic is designed in such a way that if too many softirq requests are raised within a short time, then instead of invoking another softirq, the softirq tasks are offloaded to be handled by a low priority "softirqd" thread by lazy invocation strategy. Therefore at high packet rate situation, the NAPI poll engine runs as softirqd thread. The low priority softirqd kernel task runs when CPU has no other higher priority user space or kernel tasks to execute. Thus effectively, the "producer" NAPI poll engine task can remain balanced with any high priority real-time user space task. Therefore, if a high priority real-time user space task runs in a tight loop to clear away the socket buffer, then the socket buffer overflow can be avoided. As NAPI uses one less task compared to plain Linux kernel network processing, so the ISR to softirq context switching time is avoided. The packet queue is absent, so managing its overflow is not an issue. Without interrupts, interrupt overheads are avoided.

But NAPI has limitations, it does not address the issue of time lost due to memory allocation, data copy, unnecessary protocol processing and context switching when the packets cross the kernel-to-user-space border. NAPI does not address issues of jitter in interrupt latency and kernel-to-user-space context switch. The NAPI NIC drivers are modified form of original Linux NIC drivers, so they still have the inefficiencies associated with run time memory allocations.

Over and above the above mentioned limitations, NAPI was also observed to have a very basic limitation quite similar to plain Linux network processing. NAPI was designed to avoid livelock arising out of interrupt based operations, but an experimental study verified that NAPI still suffers from livelock problem. Fig. 4.5, next page, presents the interrupt coalescing/ mitigation effect of NAPI. The ratio between number of interrupts raised and packets received is low when NAPI polling operates. NAPI polling starts at a packet rate of 22 kilo packets per second (kpps) when this ratio starts dropping from 1.0 (Fig. 4.5).

Fig. 4.5: Interrupt to packet ratio for NAPI at different packet rates

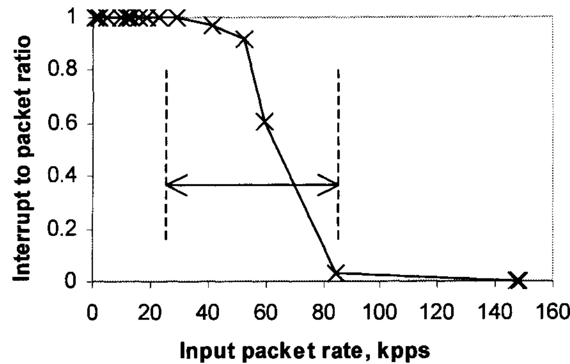
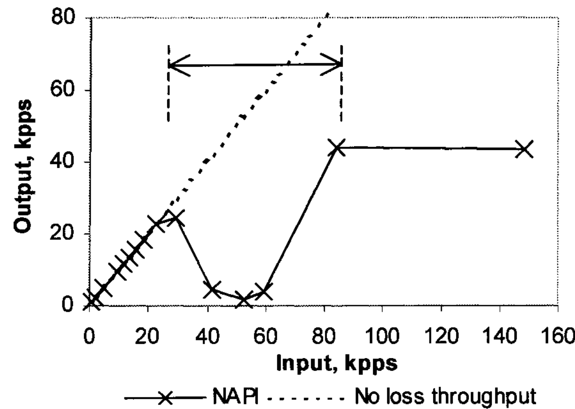


Fig. 4.6: Livelock phenomena in NAPI



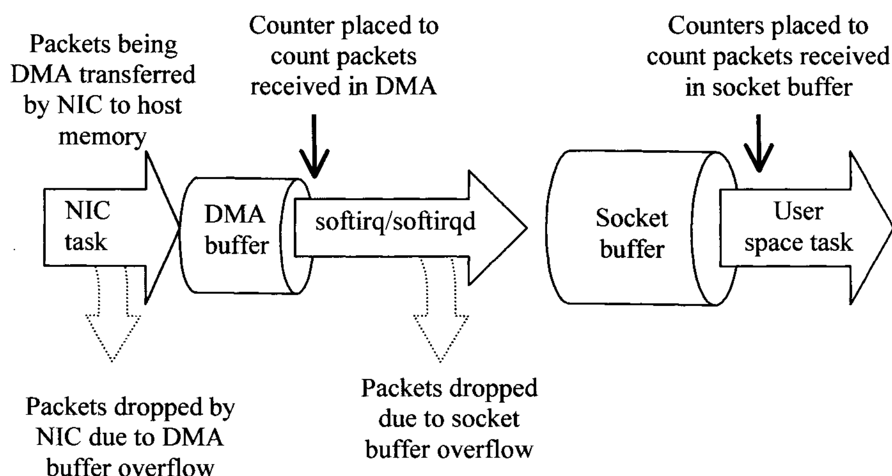
It is evident from Fig. 4.6, that livelock phenomena commence at packet rate of 30 kpps therefore the output packet rate starts dropping. Only when the packet rate is greater than 60 kpps, the interrupt rates drops significantly and packet output rate starts improving. NAPI polling only kicks in at a very high packet rate, whereas, livelock phenomena commence at a much lower packet rate for a slower CPU. If NAPI polling operation was made to commence at lower interrupt rate or if it had the intelligence to monitor the CPU load and decide when to start the polling operation then this problem could have been avoided. This same limitation also manifests a wider transition band (22 to 85 kpps),

when NAPI makes transition from interrupt driven operation to sustained polling operation.

The limitation lies in the design of the NAPI logic, "softirq" and "softirqd" tasks. NAPI polling happens too fast and as enough packets do not arrive within the small poll period so polling is not sustained. Performance loss takes place due to frequent startup and shutdown of the poll engine because each switchover between interrupt and polling modes involves high penalty due to interrupt enable-disable, softirq and softird invocation overheads. The throughput does not improve above a certain level even when NAPI polling fully operates above a packet rate of 84 Kpps (Fig. 4.6). This is due to inefficiencies associated with memory allocation in the NIC driver layer, unnecessary protocol processing in kernel, data copy and context switching in socket layer. Due to these limitations, NAPI cannot be an effective solution to the livelock problem.

To study and expose these problems, packet losses were measured at various locations inside the NAPI architecture. Fig. 4.7 presents the instrumentation employed to measure the extent and location of packet losses.

Fig. 4.7: Packet loss measurement instrumentation for NAPI



The packet losses measured are presented in Table 4.1.

Table 4.1: Effect of DMA buffer size on NAPI throughput

DMA buffer size	Packets rate. 200,000 64Byte packets (100 %)	Packet received in DMA buffer. (% of 200,000)	Packet received in user space. (% of 200,000)
128 (Default)	84 kpps	183,563 (92 %)	105,199 (52 %)
	147 kpps	104,539 (52%)	63,477 (32%)
512	84 kpps	200,000 (100%)	24,848 (12%)
	147 kpps	177,040 (89%)	3993 (2%)

In NAPI, at high packet rates, both DMA and socket buffers overflow (Table 4.1). For default DMA buffer size of 128, at 84 kpps only 92% of packets are available in DMA buffer, rest of the packets are discarded by the NIC. Packet loss due to DMA buffer overflow also increases with packet rate. For a DMA buffer size of 128, the packet availability in DMA buffer drops from 92% to 52% as the arrival rate increase from 84 kpps to 147 kpps. Socket buffer overflows are also evident at packet rates 84 kpps and higher. For default DMA buffer size of 128, out of 183,563 packets available in the DMA buffer, only 105,199 reached the socket buffer and user space, rest were discarded by the kernel due to socket buffer overflow. The loss increased at higher packet rates. For DMA buffer size of 512, 12 % packets reached socket buffer and user space at 84 kpps, while only 2 % reached at 147 kpps.

Increasing DMA buffer size to avoid packet loss at NIC, is not a viable option. A bigger DMA buffer improves the packet availability in DMA buffer, but have significant unfavorable impact on the overall system throughput. Increasing the DMA buffer size from default 128 to 512 completely avoids the packet loss arising from DMA buffer overflow for an arrival rate of 84 kpps. The packet availability in DMA buffer improves from 92 % to 100% in this case. Though a larger DMA buffer size allows more packets to

enter kernel region, but less packets actually reach to the user space. For 84 kpps arrival rate, an increase of DMA buffer size from 128 to 512 allows 100% packets to enter the kernel but packet arrival at user space falls from 52% to 12%. Similar phenomenon is also observed at 147 kpps. This happens because the CPU gets overloaded by more protocol processing tasks at higher arrival rates and starves the user space task which is supposed to clear away the socket buffer. Other than this, the 2.6 Linux kernel may also suffer from route cache overflows if too many packets enter kernel within a short time [60]. Therefore even though a smaller DMA buffer can overflow, a smaller default DMA size for NAPI is well justified. It serves the same gate keeping function as the packet queue in case of plain Linux. Smaller DMA buffer size actually improves system throughput by limiting the intensity of livelock phenomena.

This demonstrates that there is limited latitude to completely avoid packet loss by simply increasing the DMA buffer size. Linux allows limited latitude to increase the socket buffer size, but that may not be enough to completely check socket buffer overflow. Large socket buffer only masks the symptom but does not address the root cause, i.e. high jitter in the kernel to user space context switching time. Provisioning a very large kernel memory has its costs and such masking approach does not reduce high packet delivery latency jitter caused by the context switching time jitter. Consequently a different approach is needed to avoid packet loss.

4.4.2 PFRING

One recent high packet rate capturing solution "PFRING" [39] for vanilla Linux 2.4 and 2.6 kernels addressed some limitations of the Linux and NAPI, but not all of them. The decision to use polling based approach instead of interrupts is left to the NIC driver implementation, therefore PFRING may be used without or with NAPI (Fig. 4.8 and 4.9).

Fig. 4.8: PFRING (2 tasks, 1 copy, 1 allocation)

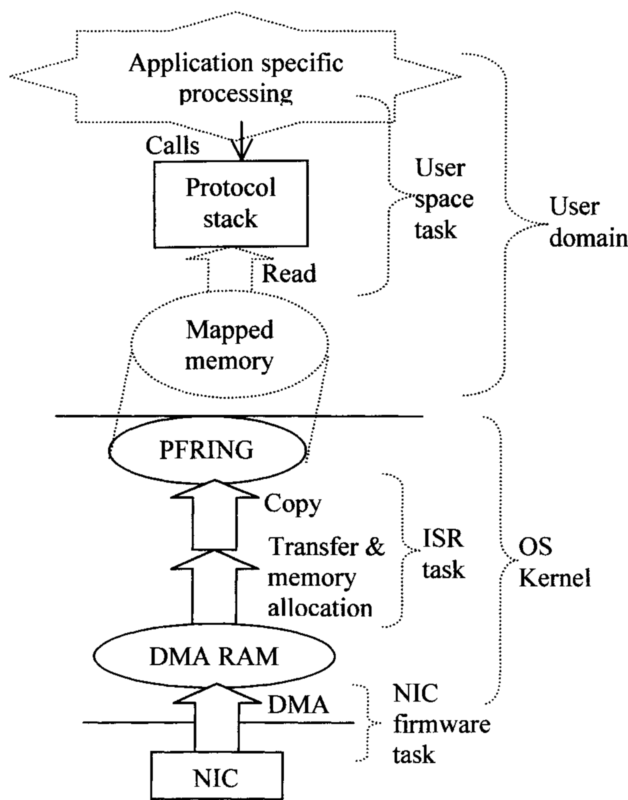
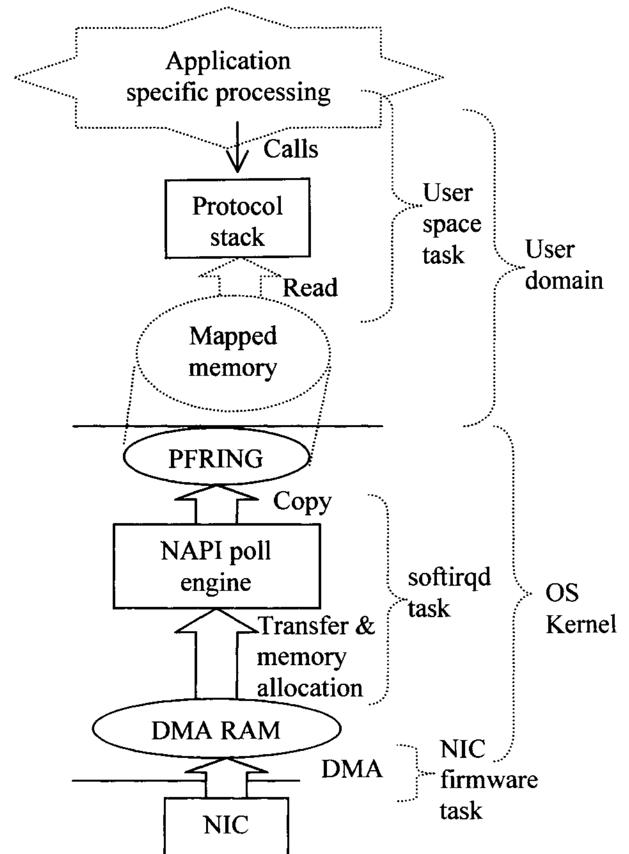


Fig. 4.9: PFRING with NAPI (2 tasks, 1 copy, 1 allocation)



PFRING avoids the packet queue and protocol processing in the kernel network stack and employs a bigger socket buffer (allows 4096 packets). In case of PFRING without NAPI, the ISR task moves the packets from the DMA region to the large socket buffer called PFRING. With NAPI, softirq or softirqd task does this job. Based on the application requirement the protocol processing may be implemented in the user space. Under high

arrival rate the PFRING approach without NAPI mechanism (Fig. 4.8) employs two task threads which run on host CPU in two different contexts - ISR (kernel) and user space. The PFRING architecture with NAPI (Fig. 4.9) also employs two task threads from two contexts - softirqd and user space. Both architectures employs two buffers - DMA buffer and PFRING socket buffer. Both the mechanisms involve one copy and one memory allocation operation.

In PFRING without NAPI, the NIC driver is same as in case of plain Linux network processing. The Linux kernel is patched to incorporate the PFRING mechanism. When a packet arrives, after moving the packet from DMA buffer to PFRING socket buffer, the ISR task wakes up the blocked user space task. No protocol processing is carried out in the kernel. The PFRING buffer memory is mapped to the user space by Linux's mmap mechanism. This memory sharing between kernel and user space avoids the need for an additional data copy operation when data cross the kernel to user space border.

For PFRING with NAPI, the same NAPI NIC driver is employed. The NAPI poll task transfers a batch of pending packets from DMA buffer to PFRING socket buffer and then wakes up the user space task at the end of each poll cycle. The NAPI mechanism in PFRING architecture operates in the same way as it would in case of NAPI alone. Similar to PFRING, this architecture avoids kernel protocol processing and data copy between kernel and user space.

PFRING solves only some problems associated with Linux and NAPI. Unlike Linux, PFRING architectures do not employ packet queue, thus avoids its overflow. Unlike Linux and NAPI, PFRING architectures save time by avoiding redundant protocol processing.

The PFRING architectures employed "real-time interrupt (RTirq) patch" to manage the interrupt latency jitter [39]. It is not clear how this RTirq patch can contribute in any

general hardware, because if the NIC interrupt is routed through legacy 8259 chip based PIC, there is no way that the NIC interrupt can be guaranteed distinct high real-time priority compared to all other interrupts. If all interrupts IRQ 0 to 15 are routed through legacy PIC system then they cannot be assigned distinct priorities [50]. RTirq patch for 2.4.23 Linux was able to assign distinct high priority to only LAPIC interrupt so far in most systems. Generally LAPIC timer interrupt is not routed through legacy PIC system. Perhaps the developer of PFRING employed a hardware where all interrupts were routed through IO-APIC, therefore he got some benefit under that special case. Nevertheless as the developer of PFRING suggested using RTirq in their publication, it was similarly employed in the experimental setup.

Though PFRING architectures perform better than NAPI and Linux [38], but they have limitation. In PFRING architectures, the two tasks and two buffers will have associated task balancing and overflow issues along with data copy, memory allocation, task context switching and border crossing inefficiencies. PFRING without NAPI will suffer from ISR to user space task context switching times in real-time. Whereas, PFRING with NAPI will suffer from high NAPI polling overheads. As a result this architecture may manifest livelock phenomenon similar to NAPI at lower and medium packet rates. However severity of the livelock in PFRING with NAPI will be less than in NAPI alone. In addition to this, there is performance loss due to softirq/ softirqd to user space context switching times in case of PFRING with NAPI. For PFRING the context switch is between ISR to user space task. PFRING architectures did not address the problem of high kernel to user space context switching time jitter. The effects of this jitter is only partially masked by the bigger socket buffer (4096 entries). The large socket buffer only decreases the likelihood of overflow event, but the high context switch time jitter still manifests as a very high packet delivery latency jitter.

In PFRING architectures DMA and PFRING socket buffers remain vulnerable. At high packet arrival rates, under CPU constraints both the DMA buffer and PFRING socket buffer overflows. This limitation was verified by analyzing the packet loss data (Table 4) which was gathered by using similar instrumentation, as presented in Fig. 4.7, in the previous section.

Table 4.2: Effect of DMA buffer size on PFRING throughput

DMA buffer size	**Packet rate.** 200,000 64Byte packets (100 %)	**Packet received in DMA buffer.** (% of 200,000)		**Packet received in user space.** (% of 200,000)	
		Without NAPI	*With NAPI*	*Without NAPI*	*With NAPI*
32	84 kpps	200,000 (100%)	-	118,439 (59%)	-
	143 kpps	436 (0.22%)	-	436 (0.22%)	-
128	84 kpps	200,000 (100%)	194,096 (97%)	122,983 (61%)	186,982 (93%)
	143 kpps	2092 (1%)	196,514 (98%)	2092 (1%)	196,514 (98%)
512	84 kpps	200,000 (100%)	198,846 (99%)	111,355 (55%)	163,611 (82%)
	143 kpps	200,000 (100%)	198,859 (99%)	24,535 (12%)	198,859 (99%)

With a DMA buffer size of 32, without NAPI, at 84 kpps, all the packets (100%) sent are available in DMA buffer, however only 59% packets reach the user space. The socket buffer is the only buffer after DMA buffer, which is overflowing. Similar socket buffer overflow happens for all sizes of DMA buffer without NAPI for an arrival rate of 84 kpps. It is suspected that this overflow is primarily due to combined effect of higher jitter in kernel-to-user-space context switch times, high interrupt overhead and lack of balance between the ISR and user space task. The ISR producer task has higher priority and so it starves out the user space consumer task under high interrupt load. As a result, the user space consumer tasks are not initiated early enough to clear away the socket buffer.

NAPI certainly reduces the socket buffer overflow. With NAPI, whatever packets are stored DMA buffer, most of them are successfully reach the socket buffer. For a DMA buffer size of 512, 198,859 packets are available in DMA buffer, the same amount successfully reach the user space via socket buffer. The transferring efficiency is different at lower arrival rates. NAPI polling consumes less CPU resources than plain kernel, therefore NAPI leaves aside more CPU time for the user space task to run and successfully pull out more packets from the socket buffer. However NAPI could not entirely prevent socket buffer overflow. For DMA buffer size of 128, at an arrival rate of 84 kpps, out of 194,096 packet available in the DMA buffer, only 186,982 reaches user space.

DMA memory buffer overflow at NIC driver level is not completely addressed in the PFRING approach. Smaller DMA buffers (sizes of 32, 128) overflows at higher arrival rates (143 kpps) for both with and without NAPI. However NAPI seemed to reduce DMA buffer overflow in some cases. For DMA buffer size of 128, for packet rate of 143 kpps, the availability of packets in DMA buffer increases drastically from 1% to 98% due to NAPI. In other cases the packet availability in DMA buffer with NAPI was slightly less than 100%, however this may not be due to unfavorable effects of NAPI on DMA buffer overflow in those cases.

By simply increasing DMA buffer size, packet loss cannot be avoided in PFRING architectures. DMA buffer overflow cannot be completely avoided even with a bigger DMA buffer size. Without NAPI, the packet availability at DMA buffer is only 1% for DMA buffer size of 128, at a packet rate of 143 kpps. Similarly with NAPI, the packet availability at DMA buffer is less than 100% for DMA buffer size of 512 and 128, at a packet rate of 84 and 143 kpps.

Allocating more kernel memory for a larger (>4096) PFRING socket buffer for full sized IP packets (1500 bytes) may have degrading effects on the system (section 3.3.2). So it may not be prudent to simply increase PFRING socket buffer size to avoid packet loss. Moreover this approach do not reduce the packet delivery latency. This means, a different solution is needed to avoid or minimize packet losses.

4.5 Summary

Table 4.3 summarizes the requirements and the solutions offered / claimed. From this table it is evident that there are no available solutions for buffer overflow and low performance due to memory allocation operations in NIC ISR in Linux. Not all solutions that are offered or claimed can be deployed for packet receiving due to their inherent limitations. Table 4.4 presents the subset of these solutions that are deployable under wide variety of circumstances.

Interrupt mitigation feature is not found in all NICs so cannot be deployed. Only a modified and simplified form of "hybrid interrupt-polling" approach can be deployed for high speed network on present Linux and hardware platforms. Adaptable polling period cannot be implemented. NAPI and PFRING have poor performance under CPU resource constraints or low power CPU, so they cannot be deployed in NMS, NIDS, mobile or embedded systems.

"Real-time interrupt (RTirq) patch" [50,51] cannot be deployed for its limitations (section 4.2.1). Real-time Linux support like RTAI which is available for a wide variety of Linux kernels, is fit for a wide variety of circumstances. Choice of Linux kernels is not constrained by RTAI. Possibly, RTAI can be made to work along with many other kernel patches. RTAI also supports user space hard real-time tasks. As RTAI is a mature and evolving freeware, so a large user community, technical discussion and support forums exist, unlike commercial endeavor like RTLinux. RTAI includes Adeos in later versions.

Lower scheduling latency benefits of 2.6 kernels can only be obtained if 2.6 kernels are chosen. It may not be possible to modify all the device drivers, to tune their DMA controllers, as that is effort intensive task.

It is to be noted that there are no readily available or deployable solutions in Linux for the following problems: High protocol processing time, memory allocation in ISR, DMA buffer, packet queue and socket buffer overflow.

Table 4.3: Solutions and approaches available for Linux

Problems/ Requirements	Solutions offered				
	Improvements in Linux OS	Real-time support for Linux/ Linux based RTOS	Misc. solutions	Packet receiving solutions	
				NAPI	PFRING
High jitter in user space event data processing task.		RTAI-LXRT, PSDD-RTLinux, LynxOS			
High border crossing cost	2.6 kernels with efficient system call mechanism.				
High kernel to user space CS time / scheduling latency	2.6 preemptable kernel with O(1) scheduler, low latency patches and higher system clock rate (HZ =1000)				
Kernel to user space data copy	Memory sharing by Linux's mmap( )				
Socket buffer overflow	No solution available				
High protocol processing time			Avoid kernel stack, use alternate simplified stack.		Possible to use alternate stack
Jitter in protocol processing		RTLinux, RTAI, RedHawk, TimeSys Linux			
Packet queue overflow				√	√
Memory allocation in NIC ISR	No solution available				
High ISR response jitter	Do	Do			
DMA buffer overflow	No solution available				
High interrupt service overhead			Interrupt mitigation. Hybrid interrupt polling.	√	
High interrupt latency jitter	Real-time interrupt patch.	Adeos. RT Co-kernels (RTLinux, RTAI) Shielded CPU (RedHawk), Threaded ISR (TimeSys Linux)			
Jitter due to hardware			DMA controller tuning, disable ACPI, APM		

Table 4.4: Solutions that can be deployed under wide variety of circumstances

Problems/ Requirements	Deployable solutions			
	Improvements in Linux OS	**Real-time support for Linux/ Linux based RTOS**	**Misc. solutions**	**Packet receiving solutions**
High jitter in user space event data processing task.		RTAI-LXRT		Offered solutions have limitations and cannot be deployed for high packet rates under CPU resource constraint situation.
High border crossing cost	2.6 kernels with efficient system call mechanism.			
High kernel to user space CS time / scheduling latency	2.6 preemptable kernel with O(1) scheduler, low latency patches and higher system clock rate (HZ =1000)			
Kernel-user space data copy	Memory sharing by Linux's mmap( )			
Socket buffer overflow	No solution available			
High protocol processing time	No ready solution available			
Jitter in protocol processing		RTAI		
Packet queue overflow	No deployable solution available			
Memory allocation in NIC ISR	No ready solution available			
High ISR response jitter		Do		
DMA buffer overflow	No solution available			
High interrupt service overhead			Modified form of Hybrid interrupt polling	
High interrupt latency jitter		Adeos. RTAI		
Jitter due to hardware			Disable ACPI/APM	

Chapter 5: Design Principles and Proposed Architecture

Linux was designed to provide multiple services fairly to multiple users. This design objective indirectly led to a poor performing packet receiving architecture in Linux. Fairness among services and servicing multiple users are not design priorities in NMS or NIDS applications. Without such constraining requirements, it is possible to design a high performance packet receiving solution within the present Linux OS framework. Many of the existing components, solutions and approaches can be reused to minimize the design effort. The next section presents some design principles for a high performance architecture. The subsequent sections present the design, implementation and operation of a high performance packet receiving architecture which is based on these design principles.

5.1 Design principles and rationale

Performance of a packet capturing system can be improved: by raising the upper bounds of the four performance elements, (the potential capacity); and by improving their realizations (actual figures) by limiting the system response jitter (section 2.3.1). The potential capacity of the system can be improved: by choosing an appropriate architectural form; by adopting a suitable task scheduling policy and by reengineering certain performance hotspots. Whereas, system response jitter can be reduced by limiting the interrupt load on the OS and/or by employing a soft or hard real-time platform. Based on our study of Linux and other available solutions, the following principles were identified that materialize these performance improvement approaches. These are as follows:

- Employ simple hybrid interrupt polling mechanism with fixed low frequency polling. Drive the polling period by hardware periodic timer.

- Switch between interrupt and polling modes based on the expected packet arrival rates. If the expected packet arrival is larger than a threshold then switch to polling mode, if it is less, then switch to interrupt mode.
- Employ a single task thread to carry out the packet receiving and event processing operation. If the NMS or NIDS application reside in the user space, then employ a real-time high priority user space thread to carry out these tasks.
- Employ a minimal integrated protocol processing instead of kernel's layered protocol stack.
- Employ a shared staging area for all packet processing work. Given the system's response jitter, choose a right size for the staging area so that it does not overflow. A big common staging area will avoid the need for data copy or buffer transfer (explicit zero copy) operations across layers/domains and the packet buffer allocation in real-time.
- Employ an operating system which either provides soft or hard real-time support for user space task or have bounded task response jitter within the operational range.

The hybrid polling mechanism will ensure that, for lower packet arrival rates, the receiver will work asynchronously with interrupts and manifest a low packet delivery latency. At high packet arrival rate, the synchronous polling mechanism will avoid CPU resource wastage due to high interrupt service overheads. The reclaimed CPU time can be utilized for useful data processing 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 and other similar OS kernels. 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 slower CPU or a system with CPU resource constraints cannot handle hardware timers events at a rate greater than 40 Khz due to high interrupt service overheads, hence a low timer periodicity has to be adopted. 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. The polling engine may be implemented in the user space. So each polling invocation has an associated overhead due to interrupt servicing and context switching between timer ISR and polling engine task. If the polling frequency is lower than the packet arrival rate, then 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. With polling, the effects of jitters in interrupt latency or ISR are also avoided, however the causal factors behind these jitters may still cause jitters in the user space task.

Using a single thread avoids the need of interim buffer and the context switching times between multiple tasks. This makes the architecture simple and amenable to scaling unlike Linux, NAPI and PFRING. If rest of the application is in user space then this single thread should reside in user space and should also execute the application. Simplified integrated protocol processing operations should be implemented in user space and carried out under this single user space thread. Generally it is preferred that the applications are executed in user space. User space application development is faster and

easier. Furthermore, user space applications allow the whole system to be more secure and stable due to user space memory protection privilege.

Minimal integrated protocol processing avoids redundant protocol processing operations and avoids the layered architecture. De-layering reduces the need for interfaces between the layers and the associated function call overheads. A monolithic architecture also allows a single staging area for all protocol processing operations. A single staging area avoids the need to transfer data across layers either by data copy or by buffer exchange (explicit zero copy) implemented by memory address or pointer copying.

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. Thus this scheme implicitly implements the zero copy mechanism without having to transfer packets and replenish the buffer by allocating memory in real-time.

By collapsing all the network receiving tasks into a single thread, the entire problem is contained to a manageable issue of a single task 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. Some design decisions at the application level improve the real-time behavior of the OS. Avoiding interrupt based operation minimizes interrupt load in the system which drastically reduces response jitters. Minimizing CPU usage also leaves more CPU time to handle transient overloads in a better manner, this also reduce the 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 needed.

5.2 System requirements

A packet capturing system should satisfy the following basic functional requirements:

- High "no packet loss" capacity.
- Low packet loss percentage at packet rates greater than no packet loss capacity.
- Low packet delivery latency.
- Minimum CPU utilization and leave more CPU resources for event data processing tasks in the application.

In addition to these, the packet capturing system should preferably satisfy the following non-functional requirements or goals:

- Robustness.
- Minimum modifications in the OS or the hardware.
- Implementation with commodity, easy availability, off the shelf components.
- Portability across variety of hardware and software platforms.
- Low life cycle (deployment and maintenance) costs.

The following section presents an architecture that satisfies the above requirements.

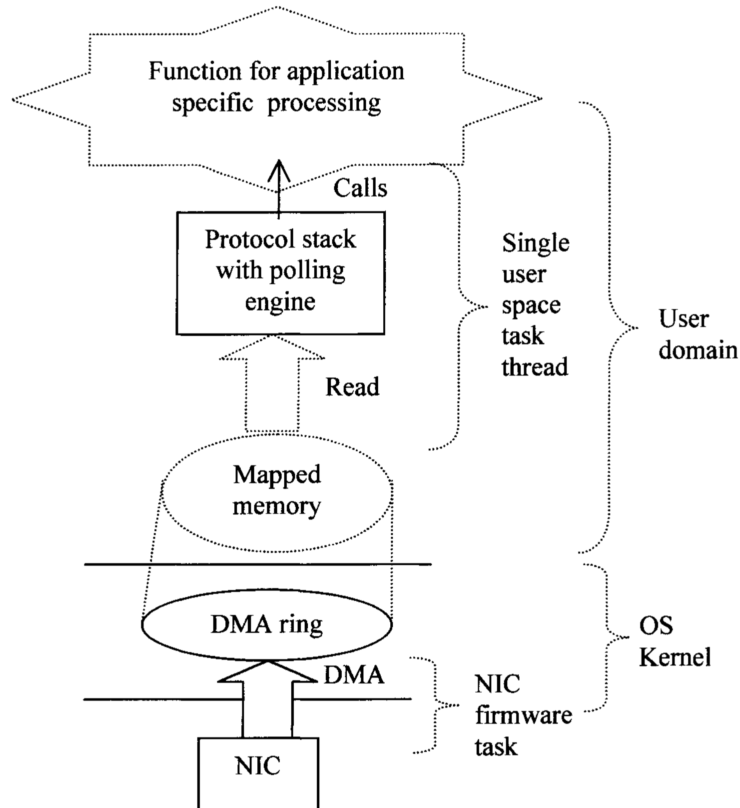
5.3 Proposed architecture

5.3.1 Task model

The task model for the proposed high performance packet receiving architecture, "DMA ring", is presented in Fig. 5.1, next page. This architecture, employs a single buffer - the

DMA buffer. This DMA buffer is actually a large circular queue, thus the architecture and DMA buffer both are called "DMA ring".

Fig 5.1: Task model of the proposed architecture (DMA ring) (1 task, 0 copy)



The NIC firmware task places packets in the queue and the receiving task picks them. The synchronization between the NIC firmware task thread and the receiving task thread is simple. For every packet slot in the DMA ring there is a status flag, a single 32 bit integer which indicates whether the packet slot is empty of filled. The NIC firmware task sets this integer after completing the DMA transfer for the corresponding packet. If the host task thread finds this integer set, then it knows that the corresponding slot is filled. The host task then processes the packet residing in the corresponding packet slot. The firmware task does not place packet in DMA ring if there is no vacant slot, whereas receiving task does not execute if all slots are empty. 32 bit memory access is an atomic

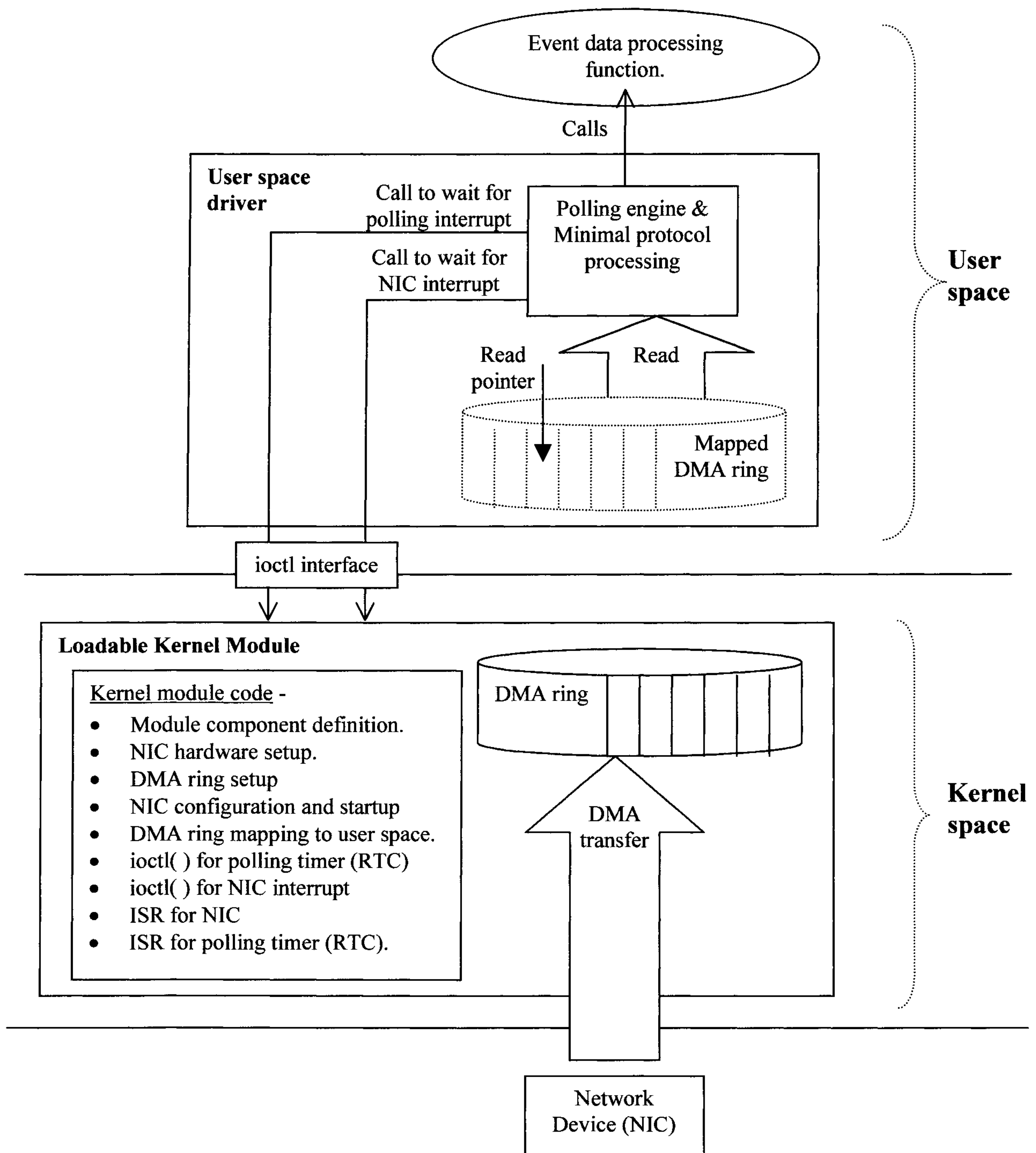
operation in 32 bit hardware and the cache location for this status flag is invalidated whenever it is set by the NIC firmware task. Therefore this mechanism achieves all the synchronization that is needed. This architecture involves no copy operation. The DMA buffer memory is mapped in the process/ user space to share it between NIC, kernel and user space. Under high network load (high packet rate) situations, it utilizes a single task thread that runs on host CPU in the user space context. This single user space thread performs all the operations needed to receive the packets and executes the poll engine. This task thread also runs the application that consumes the event data carried by the packet. The polling engine calls the application function. Using a single domain thread avoids context switching between task threads. This form of architecture reduces both per packet and per byte cost components of the system response time and can address most of the limitations of the Linux, PFRING and NAPI. This architecture only serves a single packet capturing user process, which is sufficient for the NMS or NIDS application.

5.3.2 Overview of implementation on Linux

The detailed architecture with all its components is presented in Fig. 5.2, next page. The whole architecture is packaged in three components - (i) a Loadable Kernel Module (LKM) which would replace Linux kernel's existing NIC driver, (ii) a user space driver and (iii) an event data processing function. The event data processing or application function depends on the specific application. The user space driver interacts with the LKM through the standard ioctl() system call mechanism available in Linux. To conform to the OS framework, all codes that either interacts with hardware directly or with exploit any shareable kernel resources are made part of the kernel and packaged in the LKM. These user defined kernel space function may be invoked from user space via ioctl() system call. Two parameter can be passed in an ioctl() system call. One parameter can be used to distinctly identify which particular kernel function to call, the second parameter

can be the argument for the function called. The ioctl counterpart in LKM, implements a switch conditional structure which tests the first argument and switches control to the appropriate case statement block.

Fig. 5.2: Proposed DMA ring architecture



Each case statement block implements a specific user defined function. Thus ioctl implements an easy to use form of user defined system call mechanism. The hardware or kernel resources that are to be controlled or utilized from user space, are exposed through this ioctl mechanism. Next few paragraphs describe these three components and their functioning.

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 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, which is setup in the kernel memory.

The user space driver is very generic, simple and portable component that can work with any NIC or host architecture. This component calls the application function. It includes - (i) code to request mapping of the DMA ring in user space; (ii) the polling engine; and (iii) the code for the minimal protocol processing embedded within the polling engine.

Event data processing/application function: For efficiency the application specific event data processing function can be implemented as an inline function or as a code embedded within the user space driver. This function can also be hooked to the polling engine by means of function pointers.

The existing Linux NIC driver for the chosen NIC hardware was modified to implement the required LKM. To deploy this architecture, the NIC driver is simply loaded in

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

5.3.3 Implementation details

All the codes were developed in "C" and compiled by GNU compiler ("gcc" version 3.2) that came with Redhat 8. The LKM included code segments that defines all the standard components of the LKM. It implements the initialization (constructor), exit (destructor), and other Unix file operations.

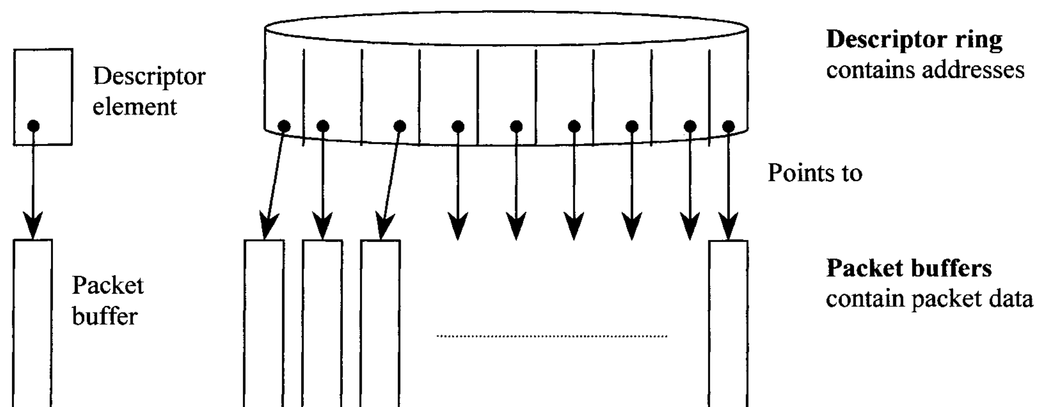
LKM and setup operations

Linux NIC drivers generally do not provide any mechanism to interface with the user space threads, so a mechanism was added to provide user space access to the kernel NIC driver. This was achieved by declaring/ setting up a miscellaneous device and implementing the device operations in the LKM. This miscellaneous device is exposed to the user space as a Unix file with a distinct file node identification. The kernel module/ driver can be accessed from user space though this file or device node identification. Functions like ioctl, mmap open and release counterparts define the implementations for the file/device operations inside the LKM. The ioctl() operations as depicted in section 5.3.2 are included in these implementations. The module initialization code registers the device node identification with the OS, whereas the exit code de-registers it. The LKM is registered as "miscdevice" having major inode number equal to 10 and minor number as 240, which is generally unused and available. Details about the modifications carried out on the existing NIC driver are presented later, in section 5.6.

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:

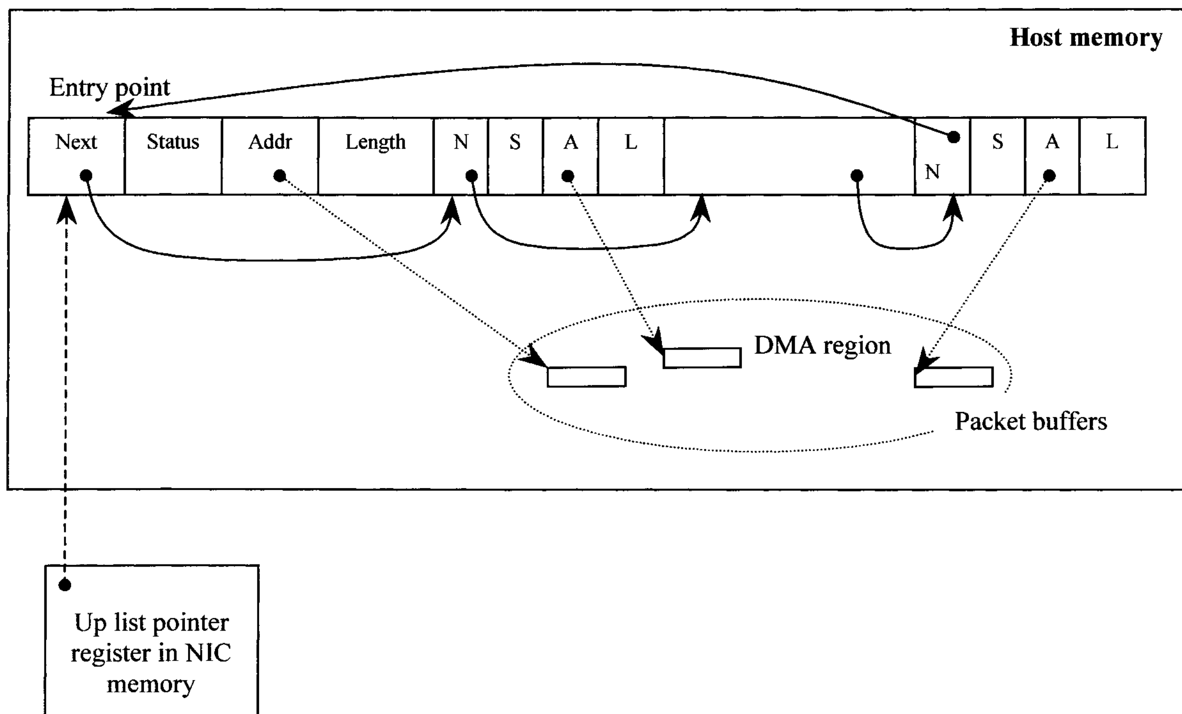
- The LKM inserts itself, as the driver for the NIC, to a link list maintained by the OS. This is accomplished by pci_module_init() kernel API call. Once it is inserted the OS can utilize the driver.
- Wakes up the NIC hardware and assigns an interrupt line (IRQ) and enables the NIC by pci_enable_device() API call.
- Allocate memory resources, i.e. data structures to manage a Ethernet device, by alloc_etherdev() API call.
- 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. 5.3).

Fig. 5.3: DMA ring 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 belongs to the DMA portion of the host memory. 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. 5.4 explains the descriptor details and its functioning.

Fig. 5.4: Data structure for the receive descriptor ring



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, 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 transfers.

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. The addresses of the locations are true physical addresses, not virtual addresses.

- 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 the 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. The user space

"consumer" thread picks up the packets from this circular queue without synchronizing with the NIC firmware "producer" task. 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 increments the read pointer and 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 the LKM 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 this new function 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 <u>any</u> 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.

Very high interrupt latency and high jitter in kernel-to-user-space context switch times cause DMA region buffer overflows in case of vanilla Linux 2.4 kernels. If Redhat 8 is used as the OS, a bigger DMA ring has to be implemented which accommodates at least 1024 packet buffers. Smaller DMA buffers can be used when a Linux based RTOS is used or when hard real-time support is available for Linux.

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.

The polling engine

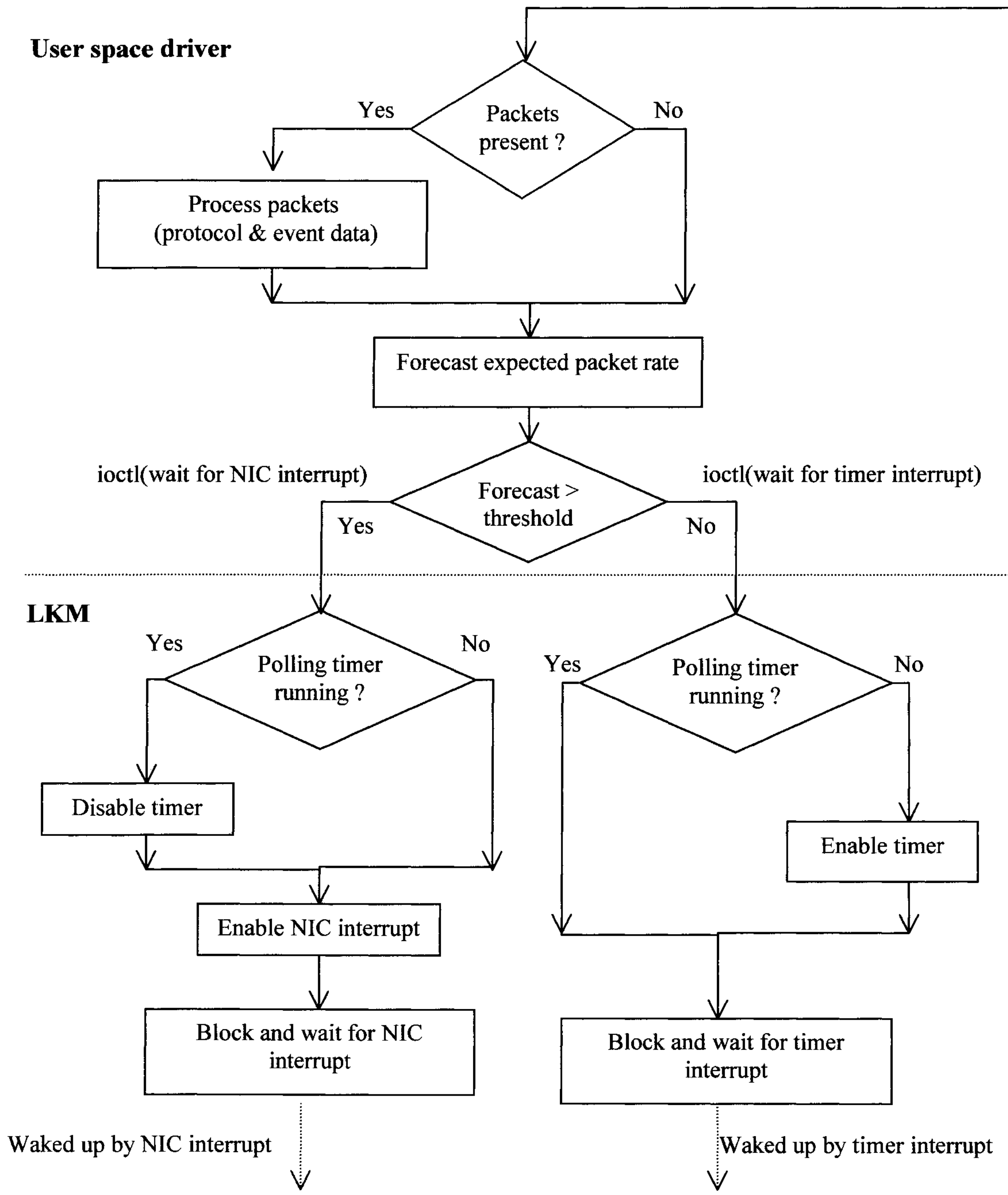
Wait queue is a Linux mechanism to block tasks and wake them up in future. A task is inserted in the wait queue when it is blocked and taken out when unblocked. A wait queue is constructed by DECLARE_WAIT_QUEUE_HEAD() kernel API macro call. A current task thread blocks itself by making wait_event_interruptible() 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 polling engine is heart of the architecture. The flowchart for the polling engine logic is presented in Fig. 5.5, next page. The top portion of the poll engine logic, above the dotted line, is implemented in the user space driver, 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 exploits the kernel's wait queue mechanism to block and wake up tasks. The polling engine runs in an endless loop. To block itself, the user space polling engine thread makes an ioctl() system call. When

these ioctl calls are made from the user space, the ioctl counterparts in the LKM are executed in the same process context under which the ioctl was called. Therefore the user space thread can enter the kernel space by calling an ioctl() function.

Fig. 5.5: Poll engine logic



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 function 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 thread, the user space thread, in a wait queue by making wait_event_interruptible() call.

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.

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. 5.5). 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.

A fixed low value polling frequency is adopted. A polling period can be chosen, which is in same order as the packet delivery latency of Linux architecture on a given hardware. For example, a polling frequency of 8192 Hz or a poll period of 122 microsecond was chosen for PII 333Mhz CPU which has a network processing latency of 40 to 120 microsecond in Linux. The polling engine runs as a high priority real-time user space task (RT FIFO, priority = 99). The process virtual memory is locked in the RAM so that none is swapped out to disk to deteriorate real-time response of this task.

The polling engine logic is presented as pseudo code in Fig. 5.6 and the LKM ioctl logic is in Fig.5.7.

Fig. 5.6: User space polling engine logic

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

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

Fig. 5.7: LKM ioctl 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;
}
```

After the circular queue and the read pointer have been set up, the user space program control enter the poll engine loop (Fig. 5.6). At the entry point of the loop the DMA buffer is checked for presence of packets, if packets are available then they are processed to extract the data payload. After completion of protocol processing, the event data processing function is called to work upon the extracted data. The extraction of payload from packets does not imply removing the data from its present staging area, but insertion of appropriate pointers in the data area of the existing packet buffer structure.

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 but 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 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. 5.7). 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. 5.7). 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.

5.3.4 Forecasting packet arrival rate

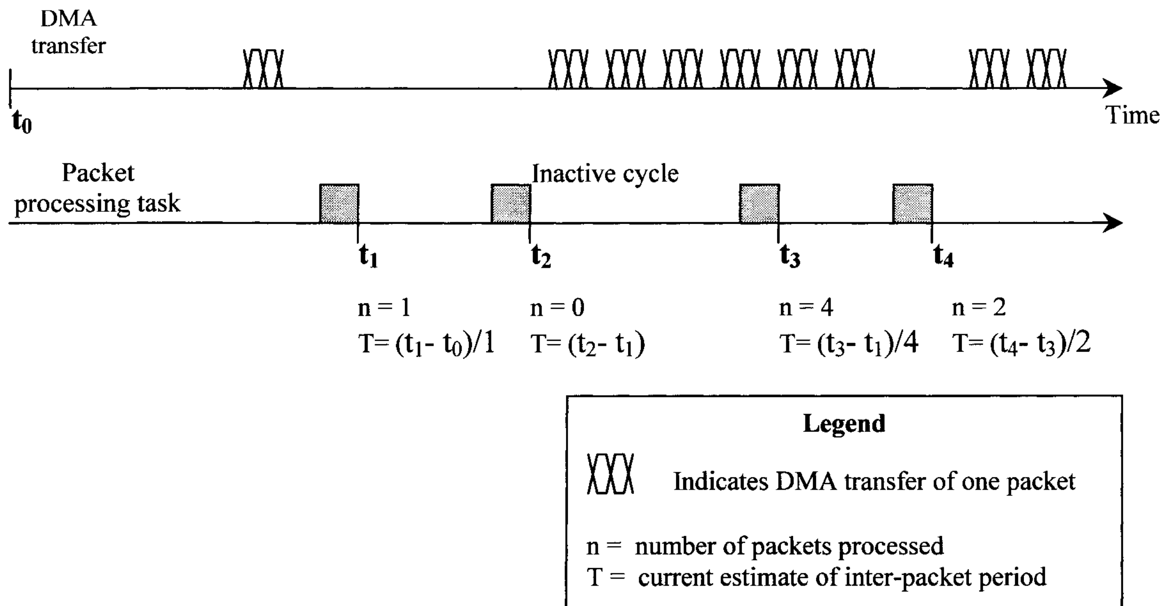
In a hybrid interrupt-polling architecture, sharp and correct mode transition is the key to better performance. A sharp mode transition means that the architecture should switch modes within a narrow packet rate band. Sharp mode transition is achieved by threshold based mode switching. Correct mode transition 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. 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 implements all these desirable system behaviors. The subsequent paragraph presents the implementation details of the forecasting mechanism.

Packet arrival rate can be also 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 counter 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 in the following paragraphs with Fig.5.8.

Fig. 5.8: Estimation of current packet rate



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

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

$$= \frac{(C_{curr} - C_{prev})}{s_{CPU}} \quad \text{when } n = 0 \qquad \text{..........Eqn.5.1}$$

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 packet rate forecast is biased to most recent arrival rate. The most recent rate estimate is based on sufficient number of inter-packet period observations. If sufficient

number of observation is not yet available then the forecast is made based on long term historical trend. Thus the inter-packet period forecast expression contains both - historical inter-packet period and the most recent inter-packet period terms. The rationale behind the estimation method is, that a future inter-packet period has a stronger correlation with the most recent arrival rate and depends weakly on older historical values. Therefore the most recent packet arrival rates give better estimate about the arrival rate of immediate future. 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. 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 -

$$T_{pred} = \alpha * T_{curr} + (1 - \alpha)T_{prev} \quad \text{..............................Eqn 5.2}$$

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 -

$$\begin{aligned} \alpha &= 1 && \text{when } n \geq n_2 \\ &= \alpha_2 && \text{when } n_2 \geq n \geq n_1 \\ &= \alpha_1 && \text{when } n_1 \geq n \end{aligned} \qquad \text{............................Eqn 5.3}$$

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 against frequent mode switching. The inertial behavior can be optimized by choosing different combinations of α_1, α_2, n_2 and n_1. These are settable parameters. Fig. 5.9 presents the polling engine code that predicts the packet arrival rate.

Fig. 5.9: Pseudo code for forecasting packet arrival rate

```
While (true) {
  if (packets present)  do packet and data processing 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);
        }
}
```

5.3.5 Settable parameters

The implementation provides for 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 are more likely with bigger 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 the present work, but this settable parameters provides flexibility for such study and optimization (if relevant).

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. Most of the UDP/IP packet sizes are generally within 600 Bytes for NMS applications or for normal user traffic [36,37]. NIDS application will need the maximum packet buffer size. Provisioning bigger packets means reserving and pinning larger kernel memory, which adversely affects the system performance unless a larger memory support in kernel is available along with larger RAM.

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 GPOS 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. DMA buffer is checked for packets more frequently, so packet idle time is lower. However higher polling rate increases the CPU utilization sharply due to increased timer interrupt servicing and polling overheads. Higher interrupt load due to higher polling rate also adversely affects the real-time performance of the system, it increases jitter in task response and polling period which result higher jitter in packet delivery latency. So for a given CPU speed, a tradeoff has to be made to decide 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. However if required the system may be forced to work either entirely in interrupt or polling mode. A settable threshold allows this flexibility. A very high packet rate threshold forces the system to work in interrupt mode, whereas a very low threshold forces the system to work in polling mode all the time irrespective of the packet arrival rate. Such flexibility is useful to study the system behavior and performance.

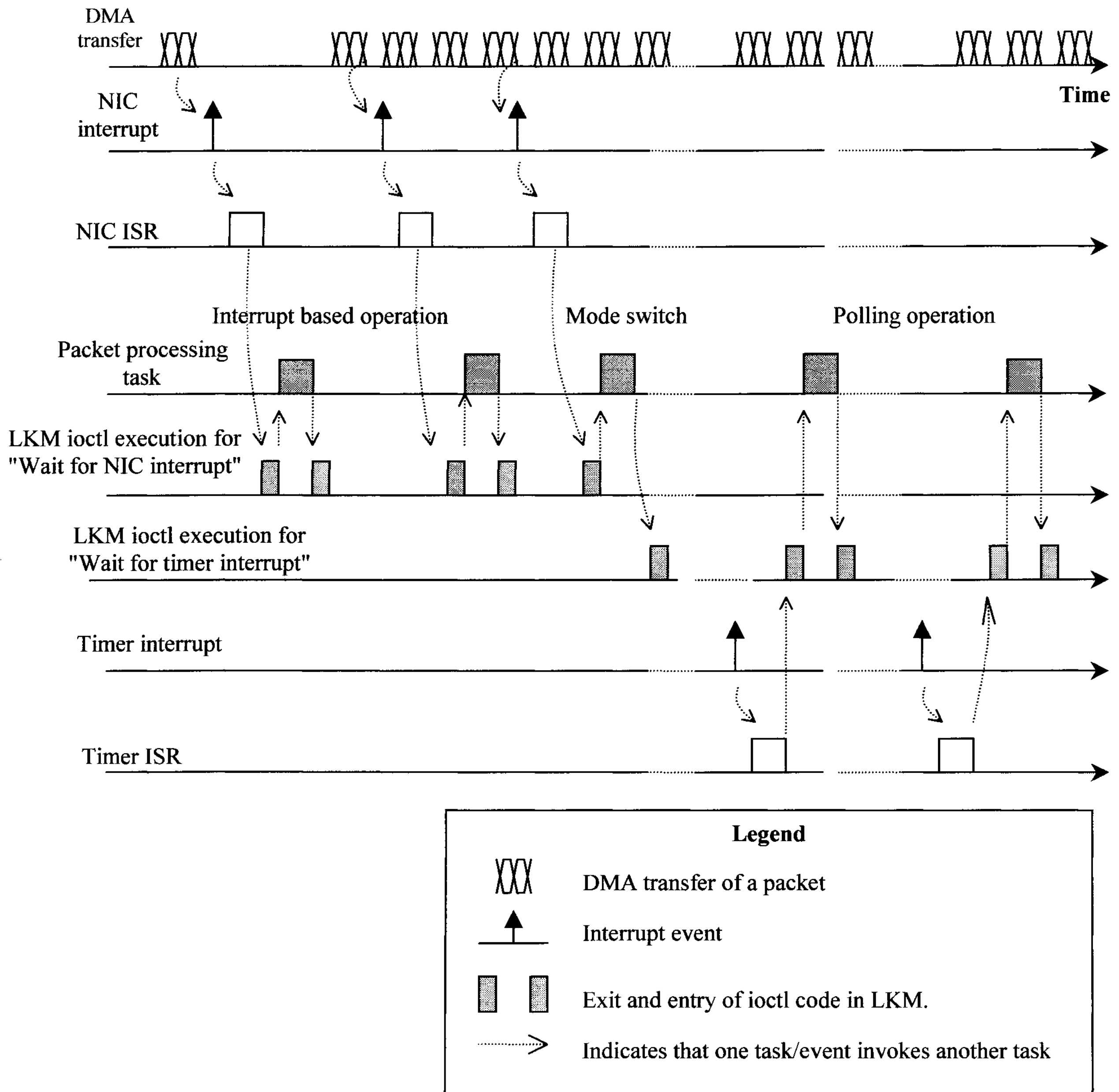
The prediction algorithm parameters - α_1 , α_2 , n_2 and n_1 determine the inertial response behavior of the polling engine during mode switching. Higher α_1 and α_2 values implies stronger bias towards most recent packet arrival rate, therefore makes the system more responsive to changes in arrival rate. It is assumed that packet traffic has stronger auto correlation over a smaller time window. The number of packets processed is a random variable, therefore a higher n_2 and n_1 values will reduce the likelihood of the events when "n" can actually cross these limits n_2 and n_1, and when higher values of α_1 and α_2 can have an impact. These settable parameters provide flexibility to optimize the inertial behavior and system performance. Ascertaining the impact of these parameters on system performance is beyond the scope of the present study.

5.3.6 Start up and run time operations

A device node with a distinct name has to be created from the user shell with a "mknod" command for major number 10 and minor number 240. The LKM registers itself with with the OS by these major and minor numbers. Instead of the Linux NIC driver the LKM is loaded in the memory by "insmod" command. After this the LKM waits for the user space driver to start up and make requests by ioctl function calls. User space driver is started by specifying a distinct device inode as a command line parameter. This device name links the user space driver to the LKM. As soon as the user space driver is loaded, it opens up an ioctl interface to communicate with the LKM using the device inode. After initialization, the user space thread makes a mmap() request to the LKM to map the DMA buffer in the user space. After maping the DMA buffer, the polling engine is 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 a 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. 5.10.

Fig. 5.10: DMA ring operation



When the first packet arrives (Fig. 5.10), the NIC transfers this packet asynchronously 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.

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. 5.10). 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.

5.4 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.

DMA ring is vulnerable to 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. Jitter in vanilla 2.4 kernels are too high therefore none of the vanilla 2.4 kernels are suitable.

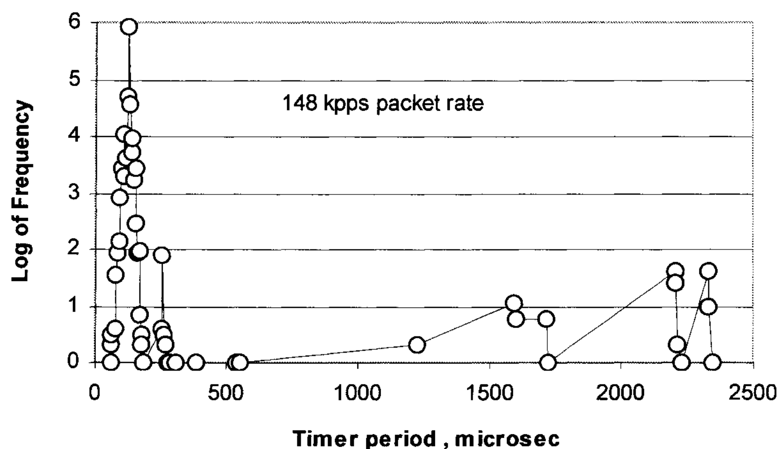
Other probable OS candidates are: Redhat 8 (custom 2.4.18) Linux kernel, 2.6 preemptable kernels, Linux based RTOS which support hard real-time in user space, like PSDD with RTLinux (FSMLabs), LynxOS (Lynux works) and hard user space real-time support for Linux like RTAI-LXRT. 2.6 kernels are claimed to have bounded jitters [32], hence appears to be promising. DMA ring architectures works with 2.6 kernels, but all aspects of the performance have not be explored in details in the present work due to paucity of time. RTLinux and LynxOS have their own proprietary kernels and therefore

were too constraining, so those were not explored. However suitability of Redhat 8 and RTAI-LXRT were explored and are presented in the next few paragraphs.

Redhat 8 kernel have smaller task response jitters within the given operation range therefore could be used with the proposed architecture. Jitter in Redhat 8 was within certain limits because the system was never loaded with high interrupt rates irrespective of the packet rate. RTC and PIT timer interrupts were the only two sources of periodic interrupts. PIT interrupt rate was only 512 Hz, whereas the RTC time interrupt rate was 8192 Hz. The normal background aperiodic interrupts due to hard disk, AGP display card, mouse and keyboard activity was never high enough to cause any problem. The intrinsic efficiency of the DMA ring architecture saved quite an amount of CPU time, which handled any situation of high transient loads quite successfully, if there were any.

The jitter behavior of Redhat 8 Linux was quantified to ensure the performance of the proposed architecture operating on it. Jitter in the timer period in user space can represent the extent of these jitters in a given OS. To measure this the DMA ring architecture was run with a very high packet rate, (148 kpps). Therefore the test scenario was an extreme case of actual production situation. The frequency distribution of observed polling period, for 10^6 samples, on a PII 333Mhz system is presented in Fig. 5.11.

Fig. 5.11: Jitter of timer periodicity in Redhat 8
(RTC timer period 122 microsec, Redhat 8, PIII 333Mhz)

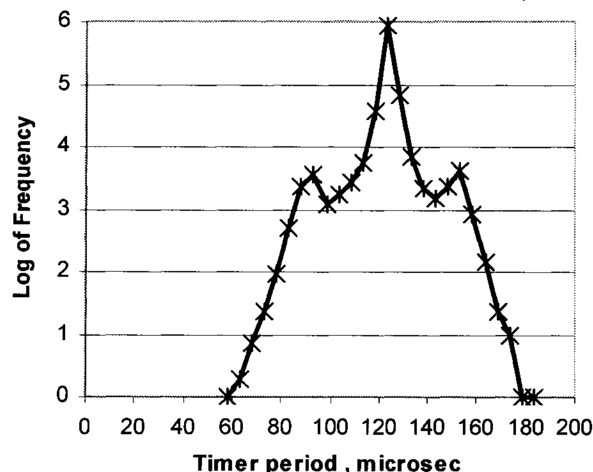


The RTC polling timer frequency was 8192 Khz, thus the periodicity was 122 microseconds. The observed worst case period was 2345 microseconds. Therefore the maximum packet arrival rate possible on a 100 Mbps network for 64 Byte packets is $100*10^6/(64*8)$ = 195 kpps. The corresponding inter-packet arrival 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.

In practice a DMA buffer size of 1024 or 2048 (safety factor of 4) would be used. A DMA ring size of 2048 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 KDE GUI and the power saving features on the motherboard was kept on. This stress test further confirmed the suitability of Redhat 8.

Similar experiments with user space hard real-time periodic task on RTAI-LXRT yielded the better results (Fig. 5.12). 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, which does not overflow a DMA buffer size of 64.

Fig. 5.12: Jitter of timer periodicity in LXRT
(Timer period 122 microsec, RTAI-LXRT with 2.4.24 vanilla kernel, PIII 333Mhz)



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 NIC also have programmable timers, but reliable operations of these timers is doubtful especially when the NIC gets loaded with high packet arrival rates.

Interrupt masking sti(), cli() operations in the CPU were not considered to disable-enable interrupts. Because masking in the CPU would allow the interrupts to enter the PIC system and strain it. Compared to interrupt masking at the CPU, interrupt disable-enabling at the source hardware is better approach when multiprocessor hardware is used because there is no contention about which CPU's interrupt has to be masked. An interrupt can also be enabled or disabled by programming the LAPIC in case of uniprocessor system or IO-APIC in multiprocessor system. LAPIC and IO-APIC routes interrupts in these systems, however programming the APIC correctly requires more effort than programming NIC I/O ports. However using 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. Interrupt disabling-enabling in 8259 based PIC chips are slow operations, response is greater than 2 microsecond, whereas programming the NIC employs two I/O operations, a read and a write, which takes around 1 microsecond to complete.

Instead of re-mapping kernel memory to user space an alternative possible method may be to share the user space memory with the kernel. This possibility has not ascertained due to time limitation.

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. Poll() mechanism involves signaling to wake up tasks, which is complex and consumes CPU cycles. Performance test result, presented in chapter 7, also demonstrated that ioctl system call along with wait-queue yielded better performance than poll() mechanism.

A modified form of the algorithm presented in [57] 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. Another motivation was to employ a new method and see its performance, though study of that algorithm is not within scope of the thesis.

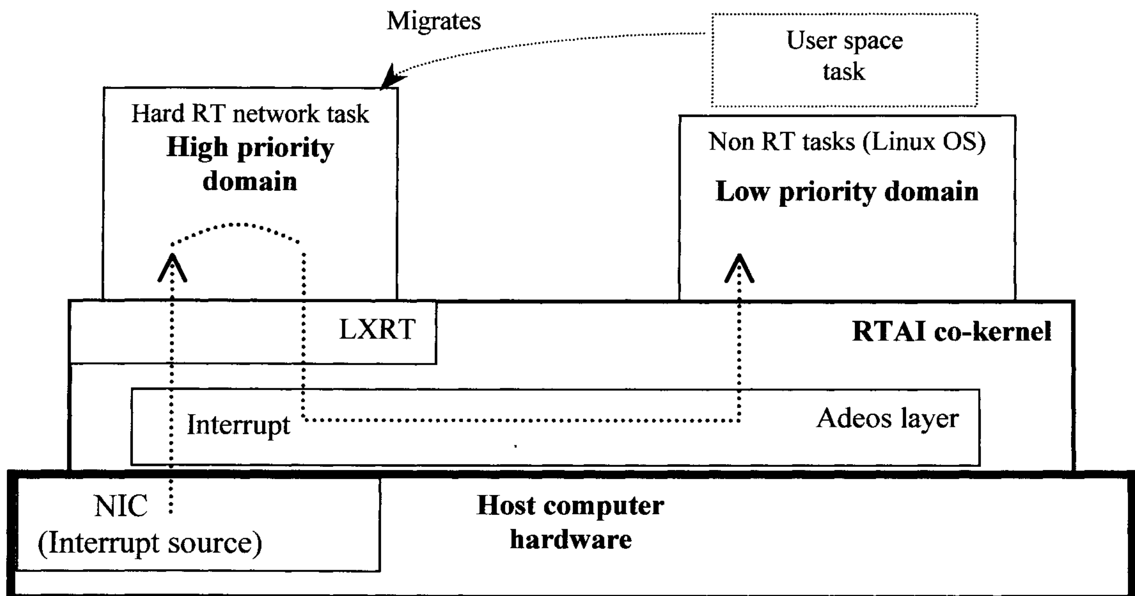
To get superior performance the event data processing function may be embedded within the poll engine loop or defined as an inline function to avoid the function call overheads. Standard Linux/Unix system call, gettimeofday() and kernel API call do_ gettimeofday() are not used to measure time because 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.

5.5 Implementation in LXRT

5.5.1 About LXRT

Implementation of the proposed architecture is better appreciated with some understanding about how LXRT and RTAI operate. RTAI or "Real-Time Application Interface" is not a full-fledged RTOS. RTAI just provides real-time services in any vanilla Linux platform. The RTAI API is essentially meant for the kernel level tasks. In 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 extend the RTAI services in user space (Fig. 5.13). Therefore with LXRT in place, a user space task can also enjoy hard real-time privileges.

Fig. 5.13: System architecture with RTAI-LXRT



LXRT expose most of the RTAI kernel services in user space. The corresponding LXRT API functions have similar name as their RTAI API counterparts, but their prototypes may be different. These services somewhat resembles to POSIX and RTLinux's real-time

APIs. When a real-time task is defined in user space, by a LXRT API function call "rt_task_init()", the LXRT module creates a corresponding real-time task counterpart in the RTAI domain which is scheduled by the real-time RTAI scheduler. This real-time task counterpart executes all the services in RTAI kernel domain on behalf of the user space task. LXRT also provides mechanism to share data structures across kernel-user space border.

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 do 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, this task is migrated from Linux domain to RTAI-LXRT domain. The task execution thread calls a special LXRT API function "rt_make_hard_real_time()" to request this migration. 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 call then the task migrates back to Linux's soft real-time domain. So this essentially means that the hard real-time LXRT task has to do without Linux system calls and 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.

In other Linux based real-time platforms where hard real-time support is not available in user space, the hard real-time tasks had to be separated out from rest of the application,

they have to be developed as kernel tasks and have to be packaged as a LKM. In a complex applications, segregation of tasks and components and managing the interaction between real-time and non real-time components severely constricts and clutters the design space. With user space support like LXRT these hurdles are removed to a great extent. High degree of segregation is not required and LXRT services are available that ease the real-time and non real-time task interactions.

5.5.2 Specific changes required for porting to LXRT

The LKM and user space components are similar to those of 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.

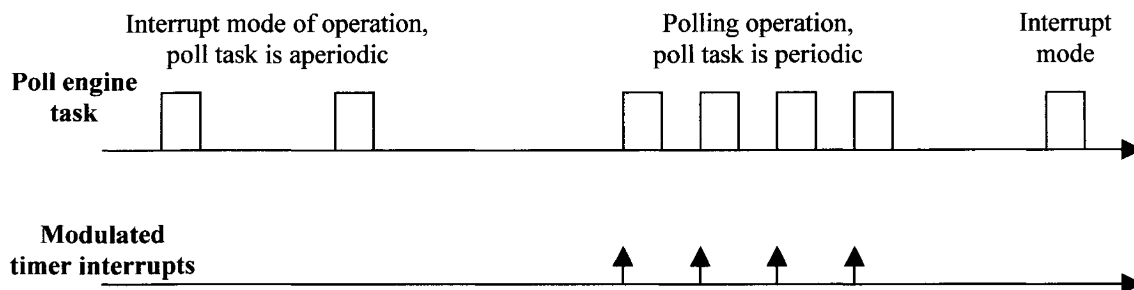
In the LKM, instead of using the Linux ISR, ISR for RTAI have to be used for NIC and RTC timer interrupts. These basic ISR codes for NIC and RTC interrupts remains the same as in Redhat 8, only registration procedure and their prototypes are different. Instead of registering the ISRs with Linux, they 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. Just for information, 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.

5.5.3 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 a 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 without bounded time (Fig. 5.14). To service 100 Mbps network, this switching has to complete within few microseconds, a larger switching time will increase the task response jitter and will require bigger DMA buffer.

Fig. 5.14: 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. 5.14). 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.

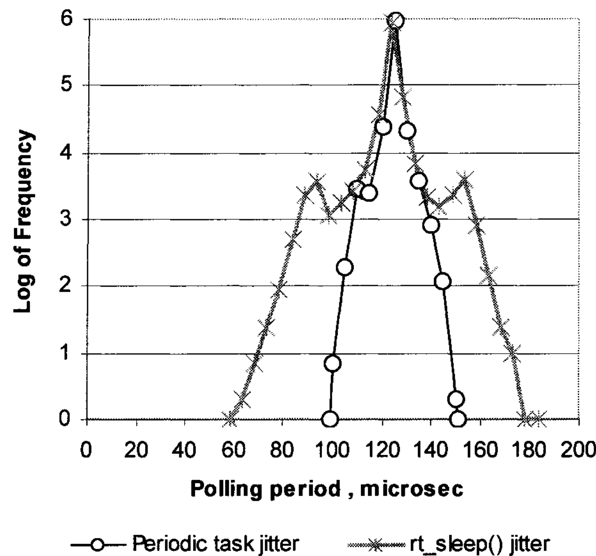
The RTAI kernel for uniprocessors is timed by 8254 chip based programmable interrupt timer (PIT). Programming this PIT as a single shot timer is costly, it involves an overhead of 15 to 20% of the time period [61]. Running the timer as monoshot mode was not explored. Stopping this timer caused preemption of the current task for 10 milliseconds, which is unacceptable. A probable explanation for this phenomenon is presented below.

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 this phenomenon may not manifest in that case and perhaps the periodic timer can be modulated.

Due to this phenomenon, the third option of using "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 (Fig.5.15).

Fig. 5.15: LXRT timer jitter comparison
(Timer period 122 microsec, RTAI-LXRT on PIII 333Mhz)



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 was 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.

<u>Waking up poll engine on NIC interrupt event</u>: 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.

5.6 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. All 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 all PCI NICs. Modification of the existing driver allows reuse of the existing open source code. This avoids the need for knowing the NIC hardware specifics and developing the code to manage the NIC hardware from scratch. This saves substantial design and development effort. Other NIC driver can be similarly modified if these NICs are to be used to implement the proposed architecture.

5.6.1 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().

5.6.2 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:

(i) Module initialization code: Instead of setting up the NIC and RTC timer interrupts for Linux, these ISRs are setup for RTAI.

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

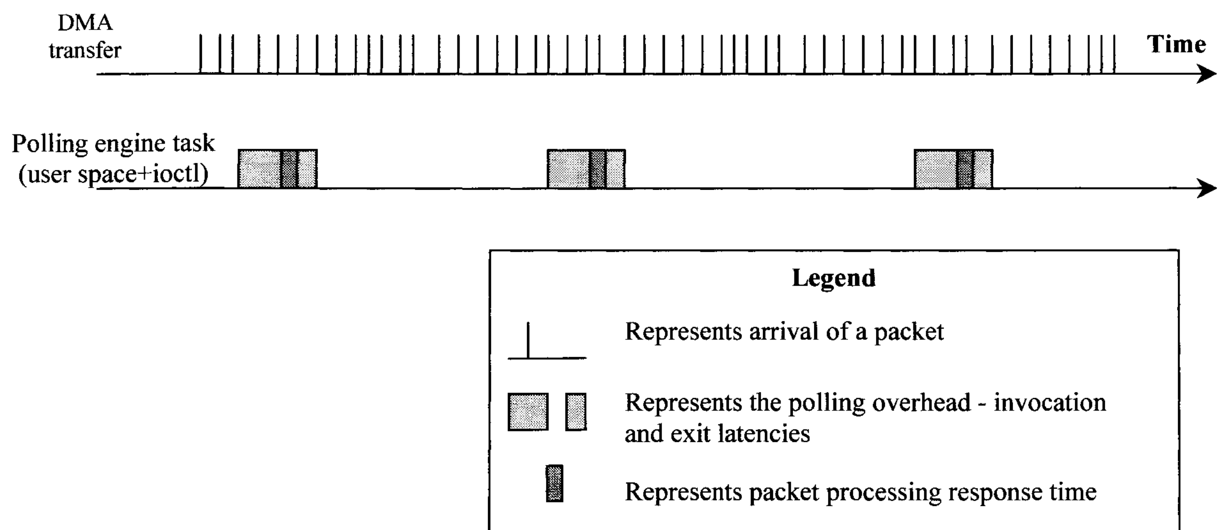
(iii) Interrupt sub-routine for NIC: The prototype of the existing Linux ISR is simple changed to the prototype for RTAI ISR, the internal code remains the same.

(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.

5.7 Performance analysis and limitations

Fig. 5.16 presents the operations of the DMA ring architecture in case of high packet arrival rate.

Fig. 5.16: DMA ring operation at high packet rate



For high packet arrival rates when the DMA ring architecture operates in polling mode, the invocation to a polling cycle and exit from it involves a system call return and a system call invocation respectively for every poll cycle. The time spent in the system call is the polling overhead. The total execution time for DMA ring is aggregate of the time spent in packet processing and the time wasted in polling overheads.

DMA ring architecture has simple form with a single task thread, therefore the CPU utilization is largely governed by Eqn. 2.3, section 2.1.2. For a given packet rate and polling rate the CPU utilization is given by -

$$= t_{POH} * f_{Poll} + t_{PP} * f_{PR} + \eta_{BG} \qquad \text{.................Eqn. 5.4}$$

where - t_{POH} is the polling cycle overhead in seconds, i.e. time required to invoke and shutdown each poll cycle.
f_{Poll} is the polling frequency.
t_{PP} is the time required to process each packet.
f_{PR} is the incoming packet rate.
η_{BG} is CPU utilization due to background tasks.

CPU utilization increase with incoming packet rate. Till the CPU utilization hits 100 %, the architecture can tolerate higher packet rates without any loss. The packet rate which causes 100% CPU utilization is considered as the upper bound of no loss capacity. This is given by -

$$= \frac{1 - \eta_{BG} - t_{POH} * f_{Poll}}{t_{pp}} \qquad \text{............................Eqn. 5.5}$$

The average packet delivery latency is given by the special case of Eqn. 2.6, section 2.3.3, where there is only one task. However the jitter in task response time and context switching time will affect the packet loss and no loss throughput capacity, these would be much worse than those given by Eqn. 2.1, section 2.1.2. Jitters in task response and context switching times deteriorates the no loss capacity and aggravates the packet loss

even the system may have enough capacity. So performance will be poorer in case of Redhat 8 compared to RTAI-LXRT, as Redhat 8 has higher task response jitter. DMA ring architecture has simple form, it has a single task thread, hence the architecture is scalable, i.e. the system dynamics will scale linearly with the CPU and network speed.

5.8 Summary

Unlike NAPI and PFRING, "DMA ring" employs additional strategies to avoid the following problems: data copy operation; memory allocation operation; frequent context switching between ISR, softirq and user space contexts; problem of buffer overflow arising out of smaller buffer size and lack of execution balance between tasks; and high kernel-to-user-space borders crossing costs. It also minimized the effects of high interrupt servicing overheads.

The differentiating features of DMA ring are - low fixed polling rate (polling overheads amortized over many packets), less context switching, minimal protocol processing, 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 (simply increase the DMA buffer size). Appropriate implementation choices are as important as the architectural design to gain superior performance.

Chapter 6: Performance Evaluation: Measurement Techniques, Instrumentation and Experimental Setup

The performance of the proposed and existing architectures have to be studied and compared against specific benchmarks. Performance evaluation and comparison criteria are based on robustness, system resource requirements and the four performance elements previously described (section 2.3.1). To evaluate and compare performances, measurement of certain variables are required. This involves limited amount of non-invasive instrumentation (from kernel's point of view) on the architectures. The next few sections explain the rationale behind the performance evaluation framework, the instrumentation required and the test setups.

6.1 Evaluation criteria and rationale

Only run time performance aspects of the operations are covered by the four basic key performance elements: no loss throughput capacity; packet loss percentage; packet delivery latency; and CPU utilization. The rationale for choosing these four had been discussed in Chapter 2. In addition to these, memory requirements and robustness aspects of the architectures also have to be considered. Some of these architectures reserve significant amount of memory during setup time, which is not available for other purposes. The amount of reserved memory indicates the memory requirement of the architecture.

In the present context, robustness is defined as the ability of the system to perform even in the presence of other non-network tasks. If CPU resources are available then even the heaviest data processing task should successfully complete irrespective of the amount of pending non-network task. The event data processing task in the user space has real-time high priority (RT FIFO priority = 99) in all these architectures, so any other low priority

user space tasks will not compete with this task for CPU resources. Only high priority kernel tasks or ISRs can preempt this user space task and steal CPU resources. In Linux, NAPI and PFRING, the kernel or ISR tasks involved in network processing will get preempted by other non-network ISRs. These other non-network kernel tasks and ISRs will cause higher packet delivery latency jitter and packet loss in worst case.

Computation intensive tasks are generally caused by applications which run as user space task with non-real-time priority. Therefore these do not compete with high real-time priority network processing tasks (user space/ kernel /ISR). High priority background kernel tasks run by OS for system management are comparatively infrequent. The non-networking tasks which can preempt the networking tasks (user space/ kernel /ISR) are mostly due to interrupts involved with asynchronous I/O activity. Interrupt based I/O affects the packet receiving architectures most. Therefore it is important to note the effects of these non-networking asynchronous I/O on the performance of these architectures.

Under robustness criteria, packet loss due to transient non-networking I/O are only considered. This is justified because a packet capturing system is expected to run on a dedicated system, which provides no other services. This system is not expected to have heavy average non-networking I/O activity as in a file server. However any system is expected to have some transient I/O due to hard disk or GUI activities. Availability of spare CPU resources plays an important role in masking these performance degrading effects of transient I/Os. If more spare CPU resources are available then the architectures are in a better position to endure these transient non-networking I/O tasks. Robustness in any architecture may be obtained by leaving spare CPU resources to handle transient overloads. However one would like to allocate as much as CPU resources possible for event data processing tasks rather than leave them spare. So robustness of an architecture has to be ascertained along with CPU availability for event data processing. An

architecture which is intrinsically robust and which does not get affected by non-networking I/O, will allow more CPU resources for event data processing tasks. Therefore in an intrinsically robust architecture a higher fraction of free CPU resources is actually available for event data processing without risk of packet loss due to overload by transient I/O activities. Effect of non-networking transient I/O on packet delivery latency is not considered separately as one measure itself can reasonably represent the robustness aspect.

Transient I/O may be caused due to GUI (mouse, keyboard) and hard disk activities. These transient I/Os demand significant CPU resource, result high hard disk interrupts, DMA, AGP, PCI bus activities and associated bus locking. In a robust architecture these transient I/O due to user activity should not cause significant packet loss. In addition to this, a robust architecture should also allow higher amount of data processing per packet without causing any packet loss.

There are two aspects related to robustness that has to be measured - (i) maximum event data processing load that the system can bear and (ii) robustness of the system against transients at this maximum load. All the free CPU resources as shown by the CPU utilization profile of the DMA ring may not be available for use, some portion of the CPU has to be left aside to buffer the transient overloads. Transient overloads cause task response jitters which can be masked by larger DMA buffer size. So for a given DMA buffer size, how much CPU resources can actually be used for event data processing task without causing packet loss is first ascertained. Then to establish the robustness aspect, transient overloads are created, by GUI and hard disk activities like mouse clicking, console switching, launching programs, browsing /dev and other directories on hard disk in rapid succession, while the system is bombarded by highest possible packet rate. If there is no packet loss then the system is considered to have passed the test. This pass-fail test is carried under two conditions - with minimal event data processing load and with

maximum tolerable event processing load which soak up a significant fraction of the available CPU resources. For a given packet rate, the event data processing load can be either minimal or heavy depending on the amount of data processing carried out per packet. This test establishes how much CPU resources is actually available for the data processing work for a packet receiving architecture.

To sum up, the performances of the existing solutions and the proposed architecture is evaluated against the following seven criteria -

(i) maximum tolerable packet rate with no packet loss (no loss capacity),

(ii) system throughput and packet loss percentage at packet rates higher than no loss capacity,

(iii) CPU utilization at various packet rates,

(iv) packet delivery latency at various packet rates,

(v) memory requirement,

(vi) maximum CPU resources that can be utilized for event data processing without causing any packet loss.

(vii) ability to withstand transient I/O loads and jitters even under large event data processing load.

6.2 Variables to measure, required instrumentation and test methodology

To evaluate an architecture against the seven performance criteria (section 6.1), the following variables have to be measured -

- packet loss
- CPU utilization

- packet delivery latency
- memory requirement
- maximum amount of free CPU resources that can be used for event data processing.

6.2.1 Detection, measurement of packet loss and system throughput

The extent of packet loss is represented as a percentage of total number of packets arriving at the NIC. This percentage is computed for each 1 million packets arrived. A sample size of 1 million packets is big enough to represent a realistic average packet loss behavior. A large sample size also manifests a lower variance in the packet loss observations.

To measure aggregate packet drops, counting number of packets received at the receiver end is sufficient. However to observe the patterns in packet loss, a more elaborate mechanism is employed. To note these patterns, each packet is stamped with an unique sequence number by the packet sender. The sequence numbers of the consecutive packets form a series of monotonically increasing consecutive integers. After receiving the packets, the packet sequence numbers are checked to detect and count the absent sequence numbers. From the series of packet sequence numbers extracted at the receiver end, the bursty behavior of packet drops or packet reordering can be observed. The length and instance of the loss bursts and extent of packet reordering can be computed.

In an ideal case with no packet loss or reordering, the received packets will deliver a series of monotonically increasing consecutive packet sequence numbers. In such cases, the absence of a sequence number in this monotonic series indicates a packet drop. At the receiver end, the sequence number is extracted from the packet data payload and compared with the sequence number extracted from the previous packet. If the difference

in the consecutive sequence number is greater than one, then a drop in packet is acknowledged. The number of packets dropped in this case is estimated as the difference of the two consecutive sequence numbers minus one. The aggregate of all drops gives the total packet drop, whose percentage may be computed for a given amount of packets send.

Reordering of packets may happen when packets travel across multiple network segments through multiple routers over redundant links or when a multi-processor hardware platform is involved in the receiving side. Multi-processor systems may run multiple concurrent threads on multiple CPUs to carry out receiving processing and may deliver packets out of order. However the present work focuses on the performance of software components on a low power uni-processor, hence the tests setups employed only uni-processor platforms and single passive network link without any routing device. The packet loss due to data link or hardware bus errors are negligible ($\sim 10^{-10}$ or less) so for all practical purposes the number of packets arriving at the receiver is equal to the number of packets send and packets suffer no reordering in transit or at the receiver (section 2.1.1). Therefore this method to detect and measure packet loss is valid for the given problem context. This same setup can also be used to study packet reordering in networks.

For all the tests UDP/IP packets were used and UDP data payload carried the packet sequence number. The packet generator, which sends packets, was implemented with a modified "pktgen" module that came with Redhat 8. This module was modified to generate and inject packet sequence numbers in the UDP payload. Newer versions of "pktgen" modules, which are packaged with 2.6 kernels or available from the web were not used, though they incorporated packet sequence number feature. The readily available newer versions of "pktgen" modules consume more CPU resources and may not yield high packet rates. This "pktgen" module can be configured to send a specific number of packets with a specified inter-packet period. The inter-packet period determines the

packet rate for a given CPU and OS platform on which this module runs. It is also possible to configure the module to generate a continuous stream of packets and specify other protocol and packet parameters like packet length, source, destination ports, IP address and MAC identifiers. The packet loss detection and measurement code which executed on the receiver, was implemented in the user space for all cases.

To measure packet loss profile of a given architecture, the receiver system is bombarded with 1 million packets at high packet rates and the packet loss percentage is noted for each packet rate. The maximum packet rate which gives no loss (0%) indicates the maximum tolerable packet rate or no loss capacity of the system (first criteria, section 6.1). The packet loss profile at packet rates higher than the no loss capacity indicates the packet loss behavior of the system. (second criteria, section 6.1). The output packet rate for a given input packet rate can be obtained by multiplying input packet rate with the percentage of packets received. The system throughput profile is obtained by plotting output rate against the input packet rates.

6.2.2 Measurement of CPU utilization

CPU utilization is measured as a percentage of the total available CPU time, averaged (moving average) across a specific time period. Available CPU time is the resource available for the user space tasks, so it is less than total CPU resource/time minus the time taken by kernel tasks. CPU utilization is a random variable but manifests a lower variance if a wider moving average window is chosen. To measure CPU utilization a tool named "cyclesoak", downloaded from the web, was used. This tool runs a lowest priority idle task in the user space, which loops continuously. The system measures the number of loops possible per unit time. This tool is first calibrated by running it under no network load condition with no other user space tasks or non-essential service daemons running. During calibration the tool notes the number of loops possible at no load ("N"). By

running this tool on the same system at the time when an architecture is processing packets, the number of loops under load ("n") is measured. The remaining CPU time under load is available from the ratio - number of loops under load divided by number of loops under no load. The actual CPU utilization by the packet receiver is given by one minus this ratio. A lower "n" means higher CPU resource consumption or utilization by the architecture. The tool displays the system load as a percentage (1-n/N)*100 every few seconds, a parameter which was set to 2 seconds. 2 seconds provides a wide moving average window and yields a smaller variance between CPU utilization observations. Popular Linux/ Unix tools like "top" was not used, as it does not give accurate results as "cyclesoak".

These architectures do not work on the same base OS platform, hence there may be variations in the CPU utilization due to OS factors. Before measuring the CPU utilization of the architectures, the "cyclesoak" tool was first calibrated for the respective platforms, but not against a single common platform. Therefore these CPU utilization figures have to be normalized against a common base to allow comparison. This normalization is carried out against the most efficient platform which allows the highest no load loop count.

6.2.3 Measurement of packet delivery latency

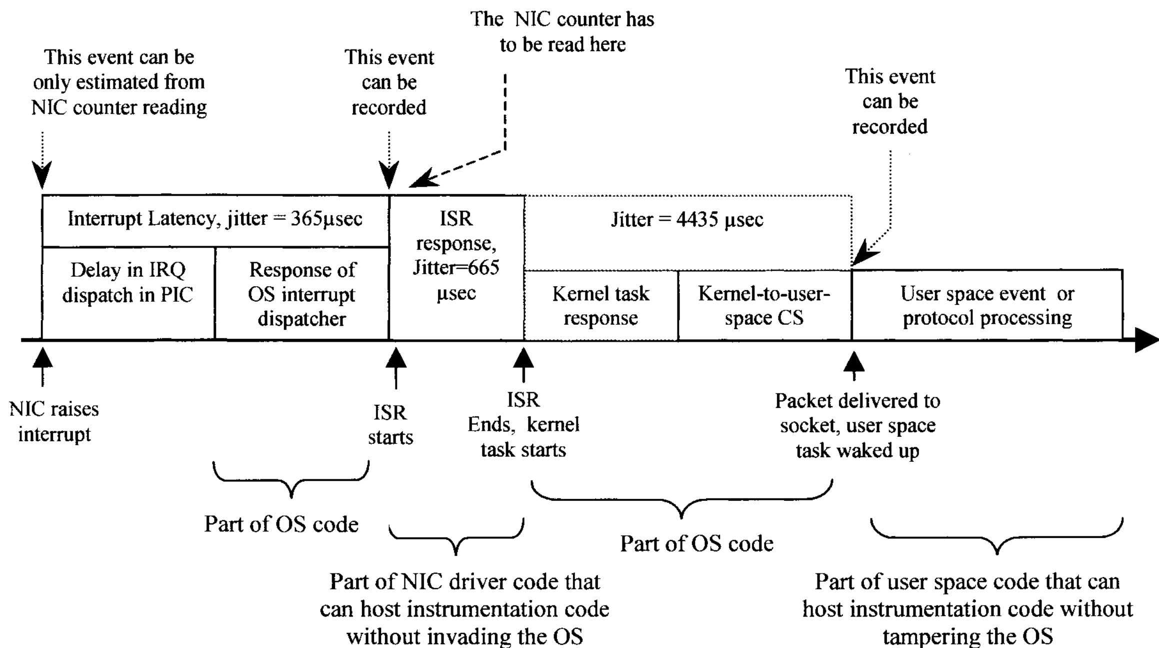
Packet delivery latency for the software architecture is defined as the time period between the instant when a packet is DMA transferred to the host memory and when it is delivered to the user space after necessary protocol processing. It is represented in micro seconds. Packet delivery latency does not include delays in the NIC hardware. Determining this latency accurately and deterministically is a challenge, it can either be measured with a limited degree of accuracy or its statistics can only be estimated. The variance or jitter in the observed latency is significant, so rather than the individual

packet delivery latency values, their statistics is of interests in the present work. The latency statistics are represented as a tuple - the average latency and the 98th percentile for a sample size of 10^6. There are multiple ways to estimates these statistics - from observed samples, by analytical estimation or a combination of both. Experimental packet delivery latency data can be obtained if the packet DMA transfer event can be ascertained deterministically. For polling based packet receiving operations, it is impossible to determine the exact moment when a packet is transferred to the host memory, whereas for interrupt based operations recording the packet arrival time requires special instrumentation.

For Linux, PFRING and interrupt operation

For interrupt driven architectures like Linux and PFRING, the packet delivery latency comprises of delays due to interrupt latency, the ISR response time, the time spent in kernel task execution, if any, and the kernel-to-user space context switching time (Fig. 6.1).

Fig. 6.1: Packet delivery latency measurement in interrupt based architectures
(Redhat 8, PII 333Mhz, 3C905B-TX, 64B UDP/IP)



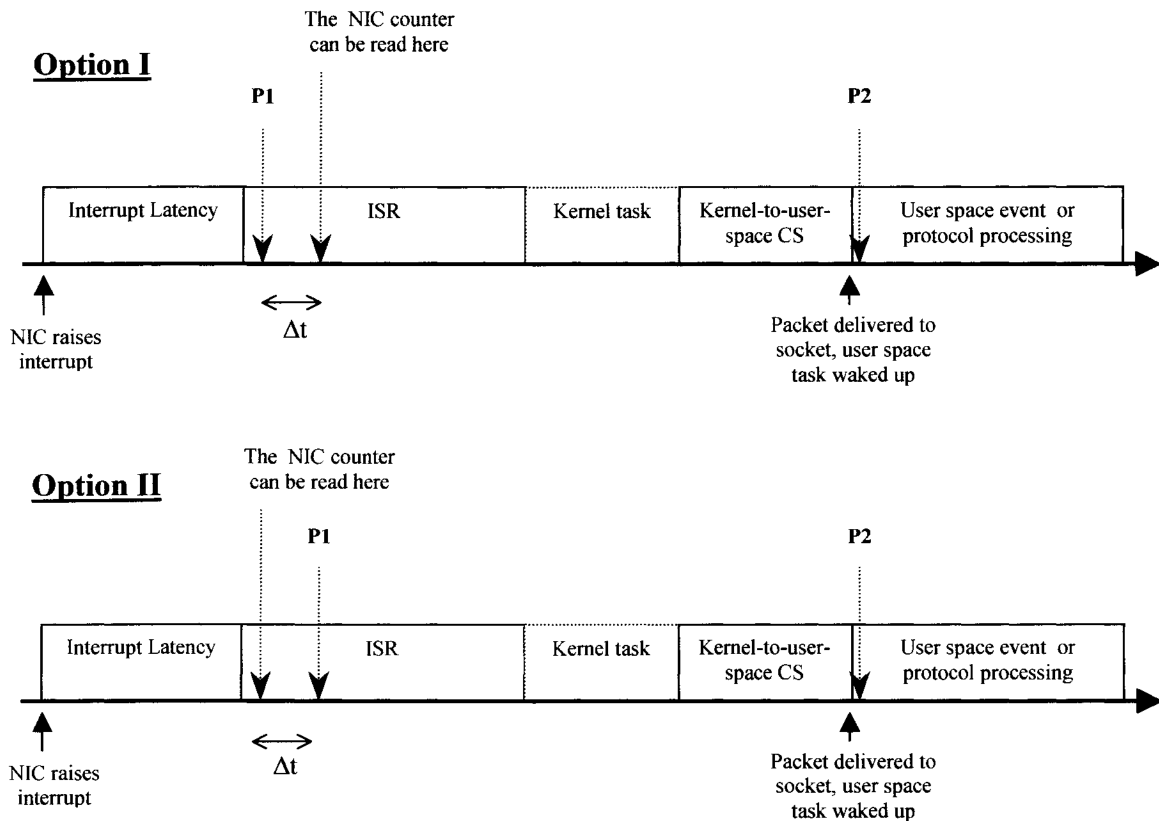
The kernel task in Fig. 6.1 is shown in dotted line because PFRING does not have any kernel task other than ISR, whereas it exists in Linux. The challenge of estimating packet delivery latency lies in measuring the interrupt latency delay component accurately. The interrupt latency comprises of delays due to IRQ dispatch though the PIC hardware and OS's interrupt dispatcher code response time which delays the ISR invocation (Fig. 6.1).

Invasive procedures like inserting instrumentation code in the kernel requires tampering the kernel source code and kernel re-compilation, therefore it is not preferred. The packet delivery to user space event is synchronous and can be determined easily without invasive instrumentation. A simple time recording probe put in the user space gives the time when a packet reaches the user space (Fig. 6.1). On the other hand it is impossible to determine the exact time when a packet is DMA transferred or when the NIC raises the interrupt. Only the instant when an ISR is invoked can be determined fairly accurately with non-invasive instrumentation like putting a time recording probe in the ISR, in the NIC driver code (Fig. 6.1). Invasive instrumentation strategies like putting a time recording probe in the interrupt dispatcher code of the OS can only account the response time of the OS interrupt dispatcher but not the delays in the PIC hardware. Delays due to IRQ dispatch though the PIC hardware can be accounted only if instrumentation support is available in the NIC hardware. Some NICs provide instrumentation support to measure interrupt latency with limited degree of accuracy. In the present work the instrumentation support available in the NIC (3Com's 3C905B-TX) hardware was exploited to measure the entire interrupt latency without any invasive instrumentation.

A 8 bit counter was available in the NIC which starts counting from zero after the NIC raises an interrupt to indicate that DMA transfer of a packet has been completed. The counter increments every 3.2 microsecond. The host can read this counter to ascertain the instant when the DMA transfer was completed. However this approach has two limitations. The time of interrupt event can be determined with an accuracy of ±1.6

microseconds at best. The host has to read the counter before $(2^8-1)*3.2 = 816$ microsecond has elapsed, as the counter is only 8 bit wide. This upper limit of 816 microsecond puts severe limitations on when this counter can be read. Interrupt latencies have worst case jitters in order of 365 microseconds, ISR response jitters are in order of 665 microseconds, whereas user space latencies (kernel execution and kernel to user space context switching time) are in order of 4000 microsecond or more. So this means that the reading of the counter has to be performed at the beginning of the ISR without delaying any further (Fig. 6.2). The code to read the NIC counter can be placed in two ways with respect to the other time recording probe P1 (option I and II, Fig. 6.2). This is discussed in details in the following paragraphs. The kernel task in Fig. 6.2 is marked by dotted line because PFRING does not have any kernel task other than ISR, whereas it exists in Linux.

Fig. 6.2: Placement of probes in interrupt based architectures



Reading the NIC counter at the beginning of ISR does not allow inclusion of ISR, kernel task response times, and the kernel to user space context switching times in the measurement and thus these have to measured separately. The aggregate of these times can be measured by putting two time probes, P1 and P2, as in Fig. 6.2. Probe P1 records the event when the ISR was invoked and probe P2 marks the event when the packet is delivered. The difference of these two recorded times gives the aggregate kernel time, i.e. the aggregate of ISR response, kernel response and kernel to user space context switching times. The packet delivery latency can be computed by adding these two observed time differences - the interrupt latency and the aggregate kernel time.

The placement of NIC counter reader and probe P1 can be done in two ways, option I and II in Fig. 6.2. In option I, the probe P1 is placed before the probe which records the NIC counter value. In option II, P1 is placed after this NIC counter reader. Let Δt be the difference between the time recorded by probe P1 and the time represented by NIC counter. Under scheme, option I, this small time interval Δt is counted twice, it is included once in the interrupt latency measurement, and again in the aggregate kernel time measurement. Whereas the scheme, option II, overlooks it completely, it is neither counted in the interrupt latency measurement nor in the aggregate kernel time measurement. Δt is a small enough interval to ignore in most cases, unless the ISR gets preempted during that interval. There is fair amount of possibility that the NIC ISR may get preempted by another ISR and this time interval Δt may stretch to 600 microsecond or more (experimental observation). For worst case scenario analysis of these real-time receivers it is better to count this interval twice rather than ignoring it completely, so instrumentation option I is chosen.

There is some likely hood that the ISR may be preempted as soon as it starts, and this preemption might happen even before the NIC counter is read. Both events, i.e. an extremely long interrupt latency of 300 microsecond and a long ISR preemption even

before the NIC counter is read, may happen one after another for the same interrupt event. An ISR preemption of over 600 microsecond may be possible. Thus theoretically, the total time of these two events may stretch over 900 microseconds, so such occurrence may cause the NIC counter to overflow. However such occurrence is extremely unlikely, no such occurrence was ever observed during any experiments even under heavy interrupt load over large sample sizes. In worst case the ISR latency plus Δt remained bounded within 665 microsecond (experimental observation), so the NIC counter does not overflow and it can be deployed successfully. So option I is a valid instrumentation method.

The time recording probes, P1 and P2 are simple assembly instruction ("asm" declaration) within the "C" code to record the lower 32 bit of Intel Pentium CPU clock cycle counter value. The time interval (microseconds) between two time recording probes is obtained by dividing the difference of the counter values by the CPU clock speed (Mhz).

This instrumentation scheme generates two sample data sets - one for the interrupt latency, another for the aggregate kernel time. The packet delivery latency is the total of interrupt latency and the aggregate kernel time. Interrupt latency is independent of aggregate kernel time. Therefore the packet delivery latency statistics can be computed from the interrupt latency and aggregate kernel time statistics.

Thus the expected (average) value of the packet delivery latency is given by -

$$= \mu_{IL} + \mu_{resp-CS} \quad \text{..............................(Eqn 6.1)}$$

where -

μ_{IL} is the expected value of the interrupt latency and

$\mu_{resp-CS}$ is the expected value of the aggregate kernel time.

The 98th percentile of the packet delivery latency is given by -

$$= t_{IL,99} + t_{resp-CS,99} \qquad \text{.....................(Eqn 6.2)}$$

where -

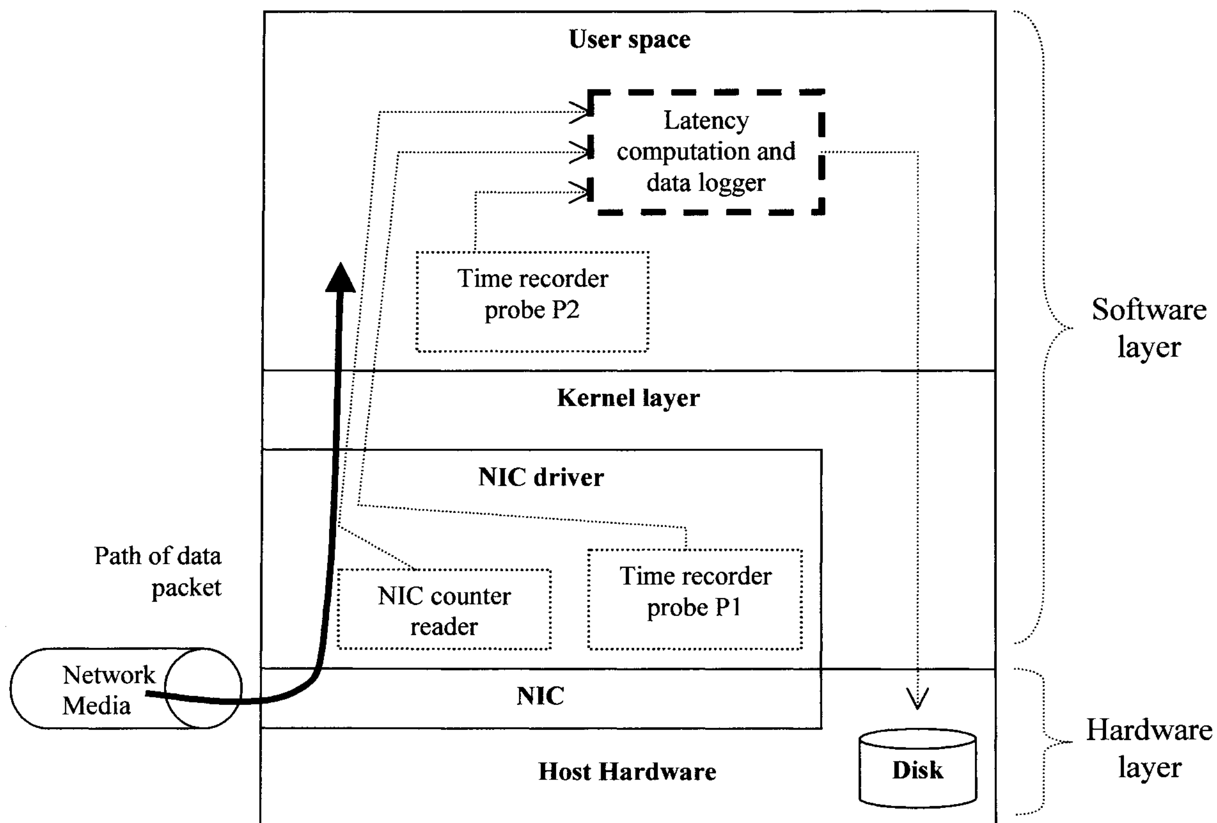
$t_{IL,\,99}$ is the 99th percentile of the interrupt latency and

$t_{resp\text{-}CS,\,99}$ is the 99th percentile of the aggregate kernel time.

The statistics - μ_{IL}, $\mu_{resp\text{-}CS}$, $t_{IL,\,99}$ and $t_{resp\text{-}CS,\,99}$ are computed from observed samples with 1 million sample size. All these statistics were computed from the same sample space of 1 million packet arrivals. Once these statistics are available the statistics of the packet delivery latency can be estimated from these utilizing the Eqn. 6.1 and 6.2.

The instrumentation architecture for Linux and PFRING is presented in Fig. 6.3.

Fig. 6.3: Instrumentation architecture for interrupt latency and aggregate kernel time sample collection



The architecture consists of two class of components - probes and a centralized computational and data logging component. The probes either read the NIC counter or collect event time and supply these information to the centralized component. The probes are embedded in suitable locations which facilitate the collection of the required data. For this case the NIC counter reader and probe P1 that notes the ISR invocation event time are placed at the beginning of NIC interrupt's ISR as mentioned in Fig. 6.2, option I. The probe P2 that records the packet delivery event time is placed in the user space, it executes as soon as the datagram socket "recvfrom()" call returns. The computational component receives the NIC counter value, computes the interrupt latency, receives the time values from P1 and P2, computes the difference of time received from probe P1 and P2, constructs the frequency distribution of these two latency/time variables: interrupt latency, aggregate kernel time, and then logs them in the memory in real-time. Once one million sample have been collected the data logger writes the summary statistics on a disk file. The probe P2 and the centralized component execute on the same user space thread that reads the datagram socket in a tight loop.

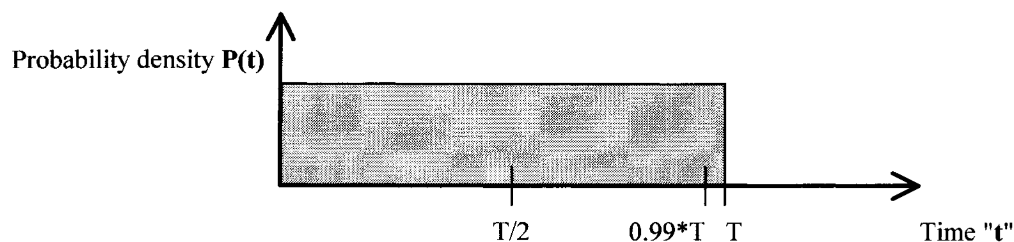
The NIC counter reader and probe P1 placed in the ISR injects their values in the packet's data payload. When the packet reach the user space, the centralized component extract these two values from the packet's data payload. By utilizing the packets to carry the instrumentation data flow across the kernel-to-user space border, the effects of instrumentation on the system was minimized. This scheme consumed a very small fraction of CPU resources and did not affect the measured variables significantly. These are test packets carrying no valuable data, hence this mechanism could be used without any problem. Probe P2 directly reports its value to the central component, as both are within same user space thread.

For DMA ring and other polling operation

For hybrid architectures like DMA ring, the packet delivery latency during the interrupt mode operation is estimated in same manner as in interrupt driven architectures, whereas during polling operation, it is estimated analytically. A practical hybrid system may operate both in interrupt and polling mode for certain range of input packet rates. For those packet rates, the packet delivery latency statistics are estimated by both the methods, and then the highest values from the two sets are accepted as a conservative estimate.

During polling operations the packet delivery latency samples cannot be determined from experimental data. There is no interrupt to mark the packet transfer event. However the statistics of this latency can be analytically estimated based on the observed polling period statistics. Invocation of polling cycle and packet arrival are independent events, therefore with respect to the polling time frame the packet arrival event is random. Polling is synchronous with system reference time frame, but packet arrival event is asynchronous to it. In such scenario, with a system reference time frame, the packet delivery latency is a random variable which is uniformly distributed between "0" and "T", the polling period, with expected value being "T/2" (Fig. 6.4). In worst case, a packet will have to wait for time "T", till the next polling cycle to get processed and delivered to the user space. The 99th percentile for such a uniform distribution is given by 0.99*T.

Fig. 6.4: Probability density function for packet arrival event with respect to system time frame



For practical systems the polling period "T" itself is an independent random variable with significant jitter. So the 98th percentile for packet delivery latency in practical polling system is given by -

$$= 0.99 * (99^{th} \text{ percentile of polling period}) \quad \text{.............(Eqn 6.3)}$$

because these two random variable are independent and (0.99 * 0.99) ≈ 0.98, which corresponds to 98th percentile.

The average/expected packet delivery latency statistics is given by -

$$= \frac{\mu_T}{2} \quad \text{........................(Eqn 6.4)}$$

where μ_T is the expected value of the polling period.

The expected value and 99th percentile of the polling period can be estimated from the observed polling period samples. For a given set of polling period statistics, the expected value and 98th percentile statistics of packet delivery latency can be computed using Eqn. 6.3 and 6.4.

For NAPI and PFRING with NAPI

For both NAPI architectures, the packet delivery latency during the interrupt mode operation is estimated in the same manner as for Linux or PFRING, where as during polling operation it is estimated analytically as done for DMA ring. For packet rates when NAPI architectures operate both in polling and interrupt mode, the packet delivery latency statistics are estimated by both the methods, and then the highest values are accepted.

However as NAPI polling is measured at kernel so the packet delivery latency jitter in the user space is given by the polling period latency jitter plus the aggregate kernel time

jitter. Thus for NAPI polling the expected (average) value of the packet delivery latency is given by -

$$= \frac{\mu_{pollprd}}{2} + \mu_{resp-CS} \qquad \text{..................(Eqn 6.5)}$$

where - $\mu_{pollprd}$ is the expected value of the NAPI poll period measured in kernel and $\mu_{resp-CS}$ is the expected value of the aggregate of kernel kernel time.

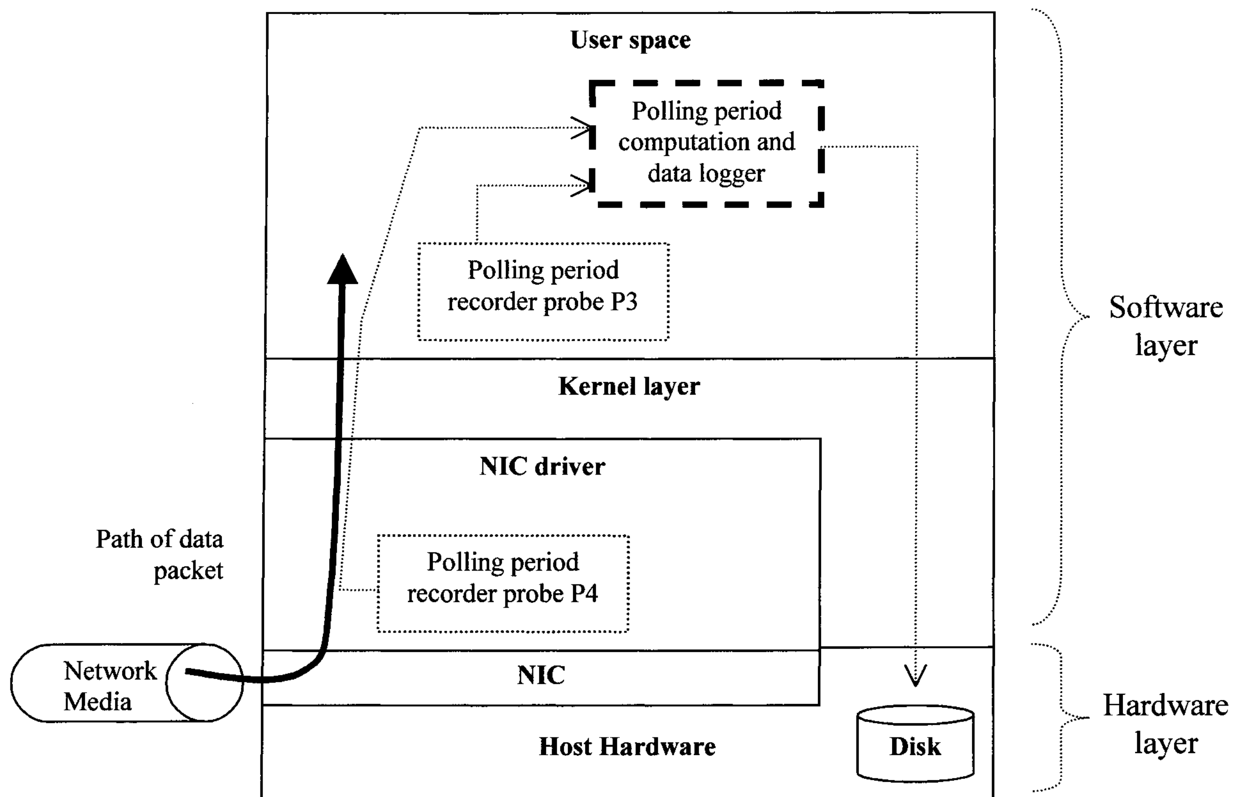
The 98th percentile of the packet delivery latency is approximately given by -

$$= t_{pollprd\ ,99} + t_{resp-CS\ ,99} \qquad \text{......................(Eqn 6.6)}$$

where - $t_{pollprd,\ 99}$ is the 99th percentile of the NAPI poll period and $t_{resp\text{-}CS,\ 99}$ is the 99th percentile of the aggregate kernel time.

The instrumentation architecture for measuring poll period in DMA ring, NAPI and PFRING with NAPI is presented in Fig. 6.5, next page. The probes, P3 and P4, collect the polling cycle invocation event time and report to the central component. The central component computes the time difference between two consecutive polling cycle invocation event to get the polling period sample. After sufficient number of samples has been collected the logger writes the summary statistics to a disk file. Probe P3 marks the packet delivery event at user space in NAPI, PFRING with NAPI and DMA ring. In DMA ring, the difference between two consecutive P3 time measurements gives the DMA ring polling period. For DMA ring architectures the polling engine is run by a user space thread, hence probe P3 is embedded inside the user space polling engine (Fig. 6.5) to record the polling period in the user space. In this case, P3 directly sends data to the central component, as both are within same user space thread.

Fig. 6.5: Instrumentation architecture for polling period sample collection



Probe P4 is only used in NAPI and PFRING with NAPI, the difference between two consecutive P4 time measurements gives the NAPI polling period. For NAPI architectures, the polling engine is run by a kernel thread which invokes the packet handling function implemented in the NIC driver. Therefore the probe P4 is embedded at the beginning of that specific function inside the NIC driver (Fig. 6.5) to note the polling period in the kernel space. The time data from P4 is piggybacked on the packet data payload as in case of packet delivery latency data collection in Fig. 6.3, for similar reasons. For NAPI architectures, the difference of time recorded by P3 and P4 gives the aggregate kernel time when the NAPI polling is sustained.

6.2.4 Ascertaining memory requirement

Some architectures require significant amount of reserved memory to provision large buffers. The NIC driver for all these architectures have very similar codes. So only buffer memory requirements are accounted, run time or stack requirements are not considered. Compared to the small difference in run time or stack memory requirements for various architectures the difference in buffer memory requirements is the dominant differentiating factor when memory requirement is concerned. This justifies the adopted memory accounting policy for comparison purpose.

Accounting of memory requirement do not involve any instrumentation, only the buffer sizes used in the architectures are noted. The memory requirement is computed by multiplying largest packet buffer size with the DMA buffer, packet queue, socket buffer sizes. Largest packet size possible under Fast Ethernet (100Mbps) is 1500 bytes, and for Gigabit Ethernet (1Gbps) is 9000 bytes. For the present problem context, largest fast Ethernet packet size of 1500 Bytes was assumed, and each packet buffer on the DMA buffer in all the architectures were provisioned for 1536 Bytes. The extra 36 byte space was allowed to align the IP payload to 32 bit boundary in memory. DMA buffer is allocated in the setup time, and does not vary with system load. Whereas packet queue and socket buffer is allocated on demand during run time. Kernel packet queue and socket buffer memory requirements were also accounted as applicable. The worst case requirement of kernel packet queue and socket buffer memory is accounted in the present analysis.

6.2.5 Assessing maximum available CPU resources and robustness

To simulate a heavier per packet data processing load, a looping code segment which multiplies two bi-dimensional matrices of double sized floating point numbers, was employed. By varying the loop size a variable event data processing load was simulated.

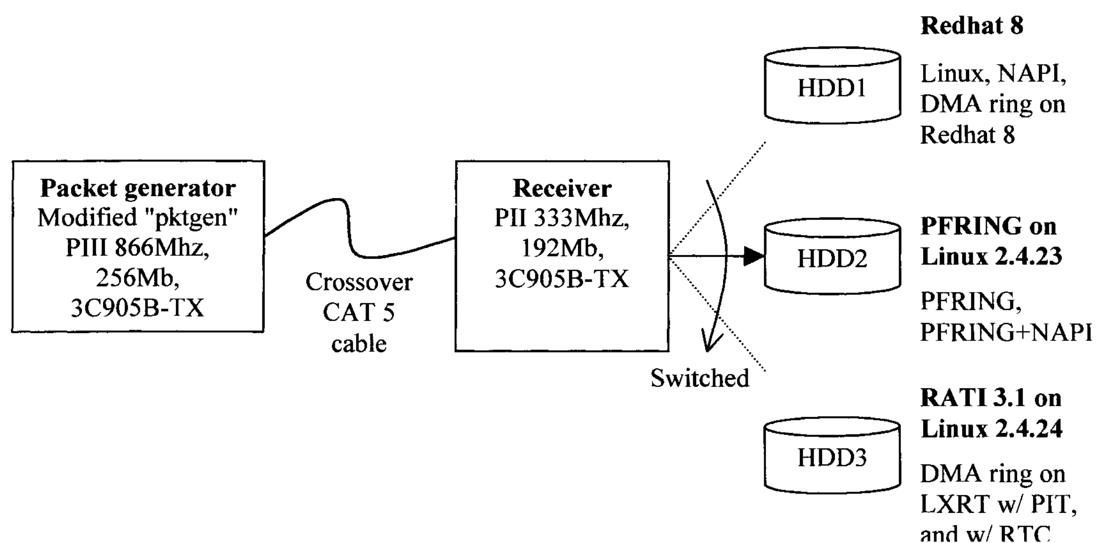
A range of loop sizes were tried out to identify the maximum event data processing load which a given architecture can bear. The CPU utilization for the maximum tolerable load was noted. This maximum load is represented as total or additional CPU utilization figures.

Once the maximum load was identified, then the robustness test was carried out at maximum packet rate for no load and maximum tolerable load conditions. Transient loads were simulated by GUI and hard disk activities. To simulate transient high jitter the video power shutdown feature of the motherboard was employed. When there is no user activity on the system, the system shuts down the video power after some time. When either keyboard or mouse is used the video power and video operation is restored. The restoration of power cause sufficiently large jitter. This test can verify endurance of an architecture against large system response jitters.

6.3 Test setups

To study and compare performances of the architectures, seven receiver test setups and one packet generator setup was arranged. The packet generator and receiver setups were connected with each other by a cross-over CAT 5 cable (Fig. 6.6).

Fig. 6.6: Hardware setup for the performance tests



The CAT 5 cable being a passive component, therefore there is no packet loss or packet re-ordering in transit.

The packet generator is implemented with the modified "pktgen" module (section 6.2.1). The packet generator setup ran on Redhat 8 on a PIII 866 Mhz, 256 MB Compaq desktop with a 3Com905B-TX 100 Mbps NIC. The packet generator was capable of generating 64 Byte UDP packets at the rate of 148,000 packets per second (148 kpps) which consumed 75 % of the line speed. Due to various protocol overheads only 75 % of the line speed is available for data transfers at this low packet sizes (64 bytes). Other researches achieved 1488 kpps with 1 Gbps Ethernet networks for similar small packets [62] with other packet generators. So it seems that the setup packet generator covered the entire possible operating range.

All the seven receiver setups ran on PII 333Mhz 192MB RAM Dell desktop with a 3Com905B-TX 100 Mbps NIC. The same hardware were used for all the seven receivers. The rationale for using these slower CPUs is presented in the next section. The receiver architectures and the OS kernels were loaded on separate hard disks (HDD1 to 3) and these hard disks were switched as required. All the receivers ran along with the same Redhat 8 KDE GUI on their respective OS kernels. Most of the service daemons which normally run a Redhat 8 system were shutdown except a few essentials like "syslog", "xinetd", "autofs", "random" and "network". These daemons are essential for system management, logging and operations. For all receiver setups the same default kernel and network settings were maintained. All the kernel versions were very close to each other - Linux version 2.4.18 (Redhat 8), 2.4.23 (PFRING) and 2.4.24 (RTAI-LXRT). So the differences in CPU utilization were mainly due to differences in system clock rate (HZ value), differences in packet receiving architectures and due to scheduler differences like presence of real-time scheduler (in case of RTAI) or presence of low latency schedulers (in case of Redhat 8). Even these differences were properly accounted for during data

analysis. Redhat 8 KDE GUI provided a base background non-networking transient load on the OS and facilitated interactions with the experimental setups. This base transient loading was maintained to verify the robustness of the architectures for all operational conditions for all tests.

The first receiver setup uses plain BSD datagram socket to receive the arriving UDP packets. A user space POSIX thread with high real-time priority (RT FIFO, priority = 99) runs in a very tight loop to read this datagram socket, to extract the packet sequence number and identify any packet loss. This setup runs on Redhat 8 kernel (customized 2.4.18 Linux kernel with 512 Hz system clock and low latency patches). The NIC driver module ("3C59x") for 3C905B-TX card that came along with Redhat 8 was used. This setup is dubbed as "Linux".

The second setup uses PFRING socket on a patched 2.4.23 Linux kernel. The vanilla 2.4.23 kernel is patched with PFRING and "RTirq" patch as suggested by the PFRING designer [39]. NAPI is not used. A user space posix thread with high real-time priority (RT FIFO priority = 99) runs in a very tight loop to: read the PFRING socket, parse the received packets to get the UDP payload meant for a specific destination IP and port, extract the packet sequence number and detect any loss of packets. The PFRING had 4096 slots. The NIC driver module ("3C59x") for 3C905B-TX card that came along with Redhat 8 was used. This setup is nicknamed as "PFRING".

The third setup runs a NAPI NIC driver on Redhat 8 (modified Linux 2.4.18) kernel. This driver was obtained by patching the Redhat 8 NIC ("3C59x") driver with a patch downloaded from web. The NAPI operation of this patched driver was verified and validated before using it. The same user space code as used in the "Linux" setup, is employed to read the datagram socket. This third setup tests the performance of NAPI architecture. The third setup is termed as "NAPI".

The fourth setup uses the same NAPI NIC driver as in case of "NAPI", and a similar PFRING socket on patched 2.4.23 Linux kernel as in case of "PFRING". The receiving code is the same as used in the "PFRING" setup. Thus this fourth setup implements PFRING with NAPI. The fourth setup is named as "PFRING + NAPI" or "PFRING with NAPI".

The fifth setup uses the proposed solution, DMA ring, on Redhat 8 kernel. A user space posix thread with high real-time priority (RT FIFO priority = 99) runs the poll engine. The received packets are parsed for a specific destination IP, port, sequence number and packet loss was detected. The DMA buffer accommodated 2048 packets. The NIC driver employed by this setup was especially developed to implement the DMA ring architecture. This driver was modified version of the "3c59x" driver which came along with Redhat 8 kernel (section 5.6.1). This setup is titled as "DMA ring on RH8" or "DMA ring on Redhat 8".

The sixth setup uses the proposed solution, DMA ring, on vanilla 2.4.24 Linux kernel (which was patched for RTAI 3.1) along with RTAI version 3.1 with LXRT enabled. A user space POSIX thread with high real-time priority (RT FIFO priority = 99) executed the poll engine. The poll engine was paced by RTAI kernel timer which is driven by the 8254 chip based programmable interval timer (PIT) interrupts. The user space protocol layers was similar to the one implemented for "DMA ring on Redhat 8". The driver used in this setup was ported version of the driver developed for "DMA ring on RH8" setup (section 5.6.2). The DMA buffer size was kept at 64. This setup is titled as "DMA ring on LXRT with PIT (8254) timer".

The seventh setup was similar to the sixth "DMA ring on LXRT with PIT timer" setup in most aspects. The only difference being, instead of employing the PIT based RTAI kernel timer, the polling is paced by Motorola's MC146818 based real time clock (RTC)

interrupts. This setup was used to study the performance difference when an external hardware timer is used instead of RTAI kernel timer. This setup is called as "DMA ring on LXRT with RTC timer".

For profiling packet loss and CPU utilization behavior the above setups were used. To collect data to estimate packet delivery latency, additional instrumentation were employed as discussed in section 6.2.3.

6.4 Operating range

All the studies were carried out on 100Mbps network, with packet size of 64 bytes and 1024 bytes and packet rate up to 148 kpps. For the given slow CPU (333Mhz) these packet rates are considered quite high. Packet rates higher than 148 kpps will require a network line speeds higher than 100Mbps, but that was considered outside the scope of present work.

The CPU speed (333Mhz) is in the same order of network speed (100Mbps). The conclusion drawn about the architectures with PII 333Mhz CPU and 100Mbps network will remain valid for higher speed CPUs operating with a high speed network. These architectures will similarly perform on faster present generation 4Ghz systems with 1Gbps Ethernet. The ratio of 4 Ghz CPU speed to 1 Gbps network speed is in same order as in case of 333Mhz CPU and 100Mbps Ethernet. On the other hand, tests on a slower CPU helped to assess the applicability of these architectures under resource constraints or in embedded systems.

6.5 Summery

To evaluate performance of all the architectures, seven evaluation criteria were employed. This involved measurements of several variables and their statistics. To collect these performance data, several instrumentation and analysis techniques were employed

to minimize the extent of invasion and adverse measurement effects on the target systems. Readily available tools were adopted and adapted to rig up the test setups. The next chapter discusses and analyzes the result obtained from the experimental setups.

Chapter 7: Performance Evaluation Results and Comparison

The performance of the proposed DMA ring architecture is evaluated against specific benchmarks and then compared with those of Linux, NAPI and PFRING. These performance studies demonstrate how the design principles contribute to the superior performance of DMA ring. In addition to performance comparisons, the DMA ring architecture is characterized from performance viewpoint and its limitations are discussed. Limitations of the DMA ring architecture arise from the shortcoming of the underlying general purpose OS (GPOS). As a solution, a real-time platform is suggested instead of a GPOS like Redhat. Performance of DMA ring on a real-time platform, RTAI-LXRT (version 3.1) with vanilla 2.4.24 Linux is compared with the performance of DMA ring on Redhat 8. This comparison establishes that these few remaining performance issues can be taken care off by an underlying real-time platform. The subsequent sections present and discuss the performance results in details.

7.1 Normalization

These architectures run on a variety of OS platforms with different system clock rates (HZ value), therefore the variations of CPU utilization due to OS factors needs to be quantified and accounted for when CPU utilization of different architectures are compared and conclusions are drawn from them. Table 7.1 presents the no load loop counts measured by the "cyclesoak" tool for the three OS utilized (section 6.2.2).

Table 7.1: CPU utilization normalization chart

Platform	No load loops (N)	Difference with max. N
Linux 2.4.24, RTAI ver 3.1 with LXRT	158,818 per sec.	0 %
Redhat 8	155,389 per sec.	+2.2 %
Linux 2.4.23 (with RTirq and PFRING)	156,954 per sec.	+1.2 %

From table 7.1 it is evident that the no load loop count varies by a very small percentage. Therefore the normalization is approximated by simply adding another 2.2 and 1.2 points to the observed CPU utilization percentage figures for Redhat 8 and Linux 2.4.23 respectively. All the CPU utilization percentage figures presented henceforth are normalized figures (normalized against the base - Linux 2.4.24 with RTAI-LXRT).

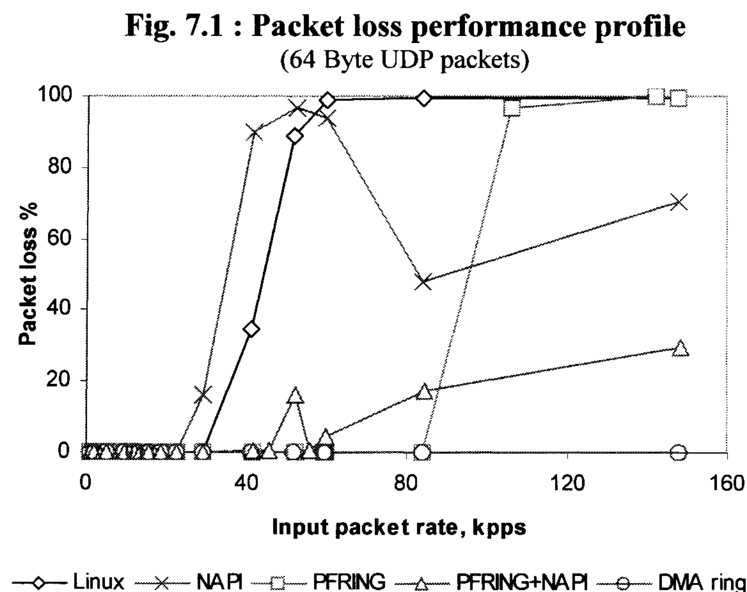
7.2 Performance profiles and comparison

The DMA ring architecture is compared with Linux, NAPI and PFRING against five criteria - (i) maximum tolerable packet rate with no packet loss (no loss capacity), (ii)system throughput and packet loss percentage at packet rates which are higher than the no loss capacity, (iii) CPU utilization at various packet rates, (iv) packet delivery latency at various packet rates and (v) memory requirement.

7.2.1 Packet loss and system throughput profile

Packet losses in all these architectures occur in bursts, and the burst length also increases with packet rate. Both these patterns indicate loss behavior due to buffer overflow. The packet loss behavior of Linux, NAPI, PFRING, PFRING with NAPI and DMA ring at different packet rates for 64 byte UDP packets are presented in Fig. 7.1.

Fig. 7.1 : Packet loss performance profile
(64 Byte UDP packets)



Smaller 64 byte packets allow highest possible packet rates, which causes greater packet loss, hence results with smaller packets are presented to bring out the worst case behavior.

From Fig. 7.1, the maximum tolerable packet rates which results 0% loss can be identified for these architectures. These are tabulated below in Table 7.2.

Table 7.2: No loss capacity profile

Rank	Architecture	No loss capacity
1	DMA ring on Redhat 8	No loss (>148 kpps)
2	PFRING	84 kpps
3	Linux	29 kpps
4	PFRING + NAPI	22 kpps
4	NAPI	22 kpps

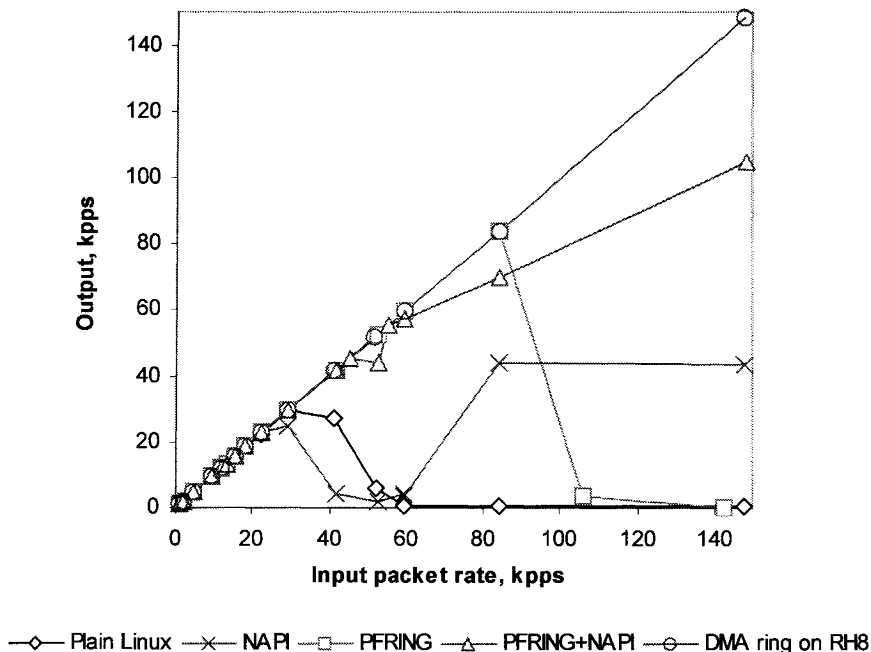
The architectures are ranked in decreasing order of performance, rank 1 indicates the best one. PFRING with NAPI gave a very small loss of 0.02 % at 29 kpps, which does not appear well in the Fig. 7.1.

NAPI actually performs slightly worse than Linux, NAPI starts loosing packets after 22 kpps, whereas Linux loose packets after 29 kpps. When no loss capacity is considered, PFRING performs better without NAPI. No loss capacity of PFRING is 84 kpps, whereas for PFRING with NAPI it is 22 kpps. However PFRING with NAPI gives lower average loss at higher packet rates (above 84 kpps) compared to NAPI, when NAPI polling is sustained. DMA ring is found to be superior compared to all these existing solutions in respect to no packet loss capacity. DMA ring does not loose any packets within this operating range ($\leq$ 148 kpps). Even with larger packets of 1024 bytes, DMA ring architecture does not suffer any packet loss at packet rate of 12.15 kpps (= 99.54 Mbps) which is highest possible packet rate possible for 100Mbps Ethernet. This data is not shown in Fig 7.1.

Interrupt based architectures, Linux and PFRING makes a rapid transition from no loss zone to the failure zone, when the packet loss rates shoots up towards 100 %. On the other hand, NAPI based architectures - NAPI and PFRING with NAPI makes a comeback once the NAPI polling is sustained, the packet loss rates for these two architectures starts dropping down after 30 kpps. In fact, beyond 84 kpps, PFRING with NAPI makes a better recovery compared to only NAPI. PFRING with NAPI suffer lower packet losses at higher packet rates. This is because PFRING with NAPI avoids unnecessary protocol processing in the kernel network stack. The PFRING architecture was proposed with NAPI based on similar observations at high packet rates [39]. However sustained NAPI polling starts too late, only after 60 kpps and it does not aid in avoiding packet loss at the lower packet rate range between 22 and 60 kpps.

The system throughput, derived from the loss behavior is presented in Fig. 7.2.

Fig. 7.2: System throughput performance profile
(64 Byte UDP packets)



All the loci lying below y = x diagonal line with 45° slope, indicates lower than ideal throughput and some packet losses. The point from which a line breaks off from the diagonal line indicates the no loss capacity for that architecture. Only DMA ring shows ideal throughput, its locus aligns with the y = x line. PFRING with NAPI is the next best one which follows the y = x line more closely than others. PFRING breaks off at a much higher packet rate, after 84 kpps and falls down towards "x" axis, which indicates a higher no loss capacity but a lower throughput at packets rates higher than its no loss capacity. When the average throughout of the entire range is considered (regardless of the packet loss at any specific packet rate), then the architectures can be ranked in the following order (best to worst) - DMA ring (perfect), PFRING with NAPI, PFRING, NAPI, Linux.

In addition to Fig. 7.2, the packet loss percentages for these architectures at three packet rates - 42 kpps, 59 kpps and 148 kpps are tabulated below (Table 7.3) for easy comparison. These three packet rates are above the no loss capacity of NAPI, PFRING, PFRING with NAPI. This table gives similar information as Fig. 7.1 and 7.2. The architectures are ranked in decreasing order of performance.

Table 7.3: Packet loss comparison

Rank	Architecture	Packet loss percentages at high packet rate		
		42 kpps	59 kpps	148 kpps
1	DMA ring on Redhat 8	0 %	0 %	0 %
2	PFRING + NAPI	0.02 %	16.5 %	29.39 %
3	PFRING	0 %	0 %	99.5 %
4	NAPI	89.9 %	93 %	70.7 %
5	Linux	34 %	99 %	99.7 %

If some packet losses are tolerated at high packet rates, only then, PFRING with NAPI gives an acceptable performance.

7.2.2 CPU utilization profile

Fig. 7.3 and 7.4 presents CPU utilization for Linux, NAPI, PFRING, PFRING with NAPI and DMA ring for 64 Byte UDP packets for different packet rates. Fig. 7.4 magnifies the plot for the packet rate range below 35 kpps. Smaller 64 Byte packets allow highest possible packet rates, which causes greater CPU utilization due to high per packet costs. Hence results with smaller packets show worst case behavior.

Fig. 7.3: CPU utilization profile
(64 Byte UDP packets)

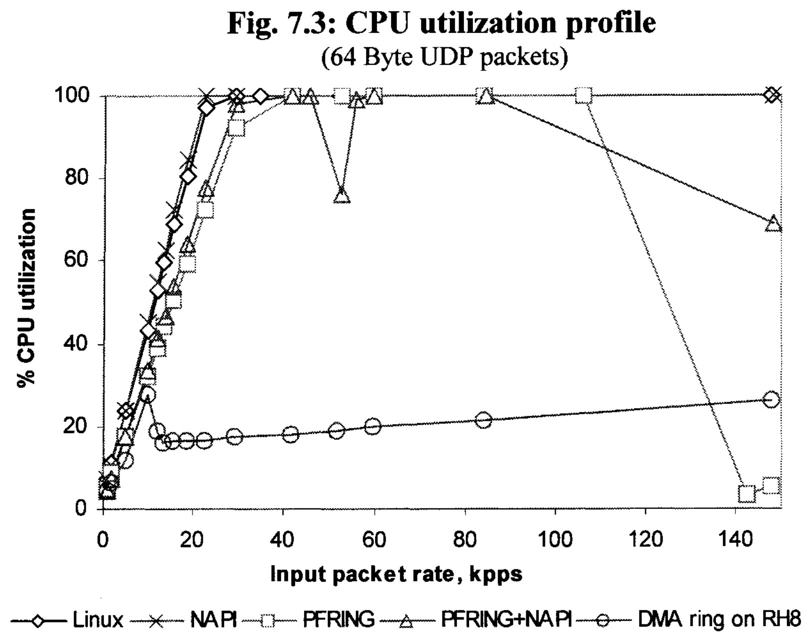
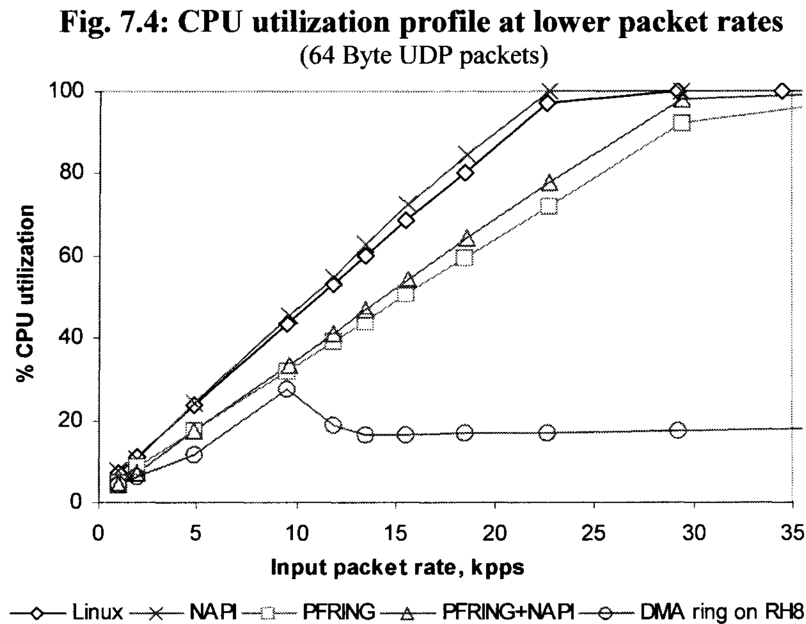


Fig. 7.4: CPU utilization profile at lower packet rates
(64 Byte UDP packets)



At all packet rates DMA ring consumes less than 28% CPU time whereas all other architectures hit 100% beyond 40 kpps and still cannot capture all the packets (Fig. 7.1 and 7.3). For all packet rate, DMA ring consumes the least CPU resources among all these architectures. NAPI polling cycle overheads are higher compared to Linux's interrupt servicing overheads due to additional operations (interrupt disable-enable, polling invocation) per cycle. This shows up as higher CPU utilization and higher packet losses in case of NAPI compared to those for Linux (Fig. 7.1, 7.3). The two PFRING architectures, PFRING and PFRING with NAPI have lower CPU utilization and packet losses compared to Linux and NAPI (Fig. 7.1, 7.2, 7.4), due to two reasons. Firstly, PFRING architectures implement a minimal integrated protocol processing compared to the kernel's inefficient layered protocol stack used in Linux and NAPI. Secondly, they do not employ the kernel thread and suffer from the associated context switching inefficiency. PFRING with NAPI consumes more CPU resources than PFRING due to NAPI's inefficiencies. CPU utilization of DMA ring remains same even for bigger packets (1024 bytes) because it implements DMA buffer sharing instead of traditional zero copy. This aspect is demonstrated later in Fig. 7.19, section 7.3.3. At very high packet rates, beyond 140 kpps, PFRING architecture seems to have collapsed (Fig. 7.3). The reason for this is not analyzed in the present study, but apparently it has something to do with high interrupt servicing overhead and high interrupt latency.

DMA ring is found to be superior in both respects -lower CPU utilization and no packet loss. This demonstrates the advantage of using - low fixed polling rate paced by hardware timer, single user space task to avoid context switching, minimal protocol processing and employing a single staging area (shared DMA buffer without explicit zero copy transfer and memory allocation). The individual contributions of each of these performance enhancing strategies are presented later in section 7.3.

7.2.3 Packet delivery latency profile

Table 7.4 compares the estimated average and 98th percentile statistics (10^6 samples) for packet delivery latencies of various architectures at different packet rates.

Table 7.4: Packet delivery latency profile

(for 64 Byte packets, figures in μsec)

Rank	Architecture	Statistics	2kpps	5kpps	18.5kpps	59kpps	84kpps	148kpps
1	DMA ring on Redhat 8	Avg.	14	14	61	61	61	61
		98th	25	21	131	136	136	136
2	PFRING	Avg.	18	18	18	155730	240608	
		98th	38	38	38	170658	266081	
3	PFRING + NAPI	Avg.	17	35	608	2718378	4655819	
		98th	38	18	18	12474085	12722675	
4	NAPI	Avg.	37	58	189	5269274	6728832	
		98th	58	58	78	12711075	12788045	
5	Linux	Avg.	37	37	38			
		98th	58	58	78			

The DMA ring operates in interrupt mode below 8.333 kpps packet rate and above it, DMA ring operates in polling mode. So for packet rates below 8.333 kpps, the packet delivery latency statistics are estimated by using Eqn. 6.1 and 6.2 and employing the associated instrumentation, as depicted in Fig. 6.3, in section 6.2.3. For packets rate above 8.333 kpps, Eqn 6.3 and 6.4 were used and corresponding instrumentation as in Fig. 6.5 was employed. The DMA ring polling statistics collected under worst case loading, i.e. at 148 kpps was utilized to conservatively estimate the figures in the range 59 to 148 kpps. The statistics of the other two hybrid interrupt-polling architecture, NAPI and PFRING with NAPI were similarly estimated utilizing Eqn. 6.1, 6.2, 6.5 and 6.6 and employing both the instrumentation architectures as in Fig. 6.3 and 6.5. Statistics for Linux and PFRING are estimated by the method for interrupt operation (Eqn 6.1 and 6.2, Fig. 6.3). Linux is considered to successfully operate below 22 kpps, and NAPI and

PFRING architectures are considered to successfully operate below 148 kpps, hence some slots in the Table 7.4, corresponding to high packet rates, are empty.

At low packet rates (below 8 kpps) DMA ring has the lowest packet delivery latency. For this range the architectures can be ranked in decreasing order of packet delivery latency performances as - DMA ring, PFRING, PFRING with NAPI, Linux and NAPI. NAPI suffer from small additional latencies due to interrupt disable-enable and polling cycle invocation and shutdown overheads.

In the packet range between 8 and 22 kpps, PFRING appears to be the best, Linux comes second, DMA ring is the third, NAPI is fourth and PFRING with NAPI is the last. NAPI actually increases the delivery latency jitter and PFRING with NAPI also suffer from it. Between 8 and 22 kpps, for both NAPI architectures in this packet rate range, the average expected latency value is higher than the 98th percentile. This indicates a very high worst case jitter. At high packet rates, above 22 kpps, the jitter in NAPI, PFRING and PFRING with NAPI explodes, but jitter in DMA ring are contained as the polling is paced by hardware timer, rather than a jittery software kernel timer.

Considering the entire range of packet rates, DMA ring appears to be the best, except for a narrow band between 8 kpps and 22 kpps, where it scores third position. At packets rates above 5 kpps, NAPI introduces severe jitter in the latency, whereas DMA ring shows stable and better behavior all throughout. NAPI actually deteriorates PFRING performance at lower packet rates, so PFRING should not be used with NAPI when mission critical packet capturing is involved.

An actual feel of the packet delivery latency statistics can be obtained by observing the frequency distribution histogram envelopes for interrupt latency and aggregate ISR, kernel task response and kernel to user space context switch times for these architectures. Only the frequency distribution envelope which is obtained by connecting the height of

the frequency bars at the time value mid points, is presented. Henceforth the terms "frequency distribution" and "distribution" are used to mean the frequency distribution envelope. All sample sizes were 1 million. Frequency is presented in log scale (0 to 6).

For Linux

Fig. 7.5 presents the frequency distribution of the interrupt latency measured at 18.5 kpps on Redhat 8 on the receiver hardware (PII 333Mhz, Dell desktop). This distribution is applicable even for NAPI and DMA ring in Redhat 8 on the same hardware. The distribution has a very sharp profile with the 99th percentile at 11.2 microseconds. The worst case interrupt latency observed is 364.8 microsecond (maximum "x" axis value).

Fig. 7.5: Frequency distribution of interrupt latency for Linux
(Redhat 8 on PII 333 Mhz)

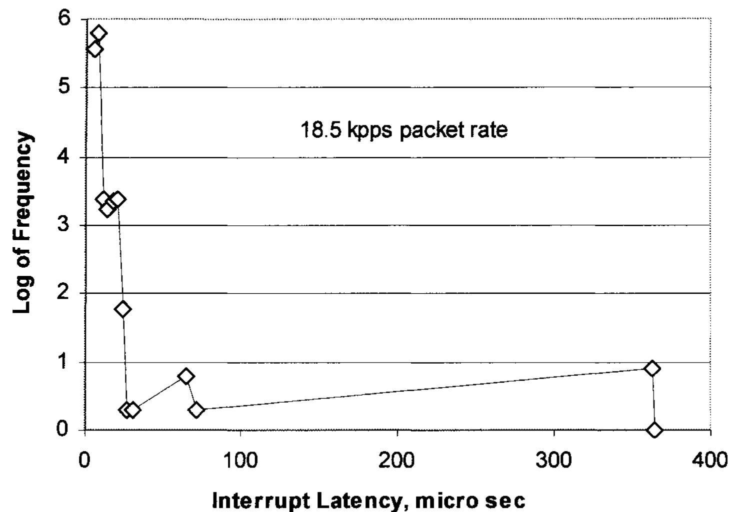
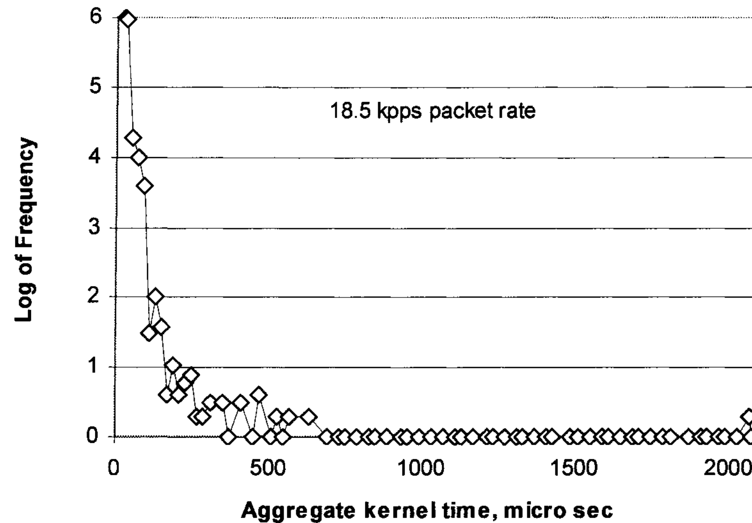


Fig. 7.6 presents the frequency distribution for the aggregate kernel time (ISR, kernel response plus kernel to user space context switch time) measured at 18.5 kpps on Redhat8. This distribution has a very sharp profile with the 99th percentile at 70 microseconds. The worst case aggregate kernel time is 2080 microsecond at 18.5 kpps.

However if power saving feature is enabled in the hardware, the worst case latency could be over 8000 microsecond.

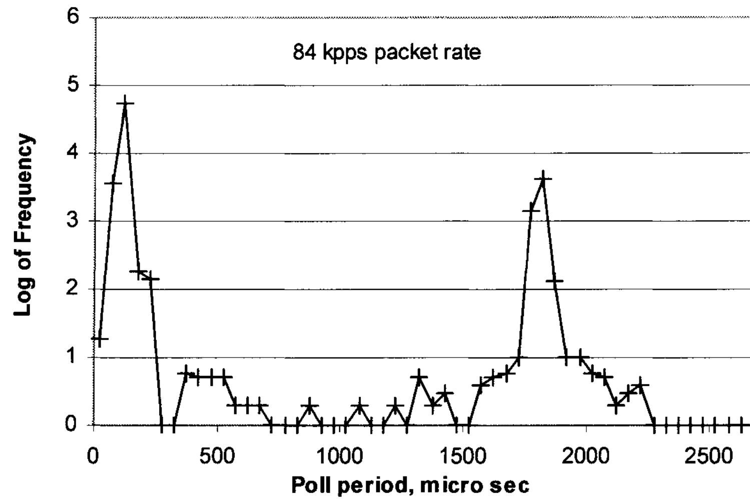
Fig. 7.6: Aggregate kernel time frequency distribution for Linux
(Redhat 8 on PII 333 Mhz)



For NAPI

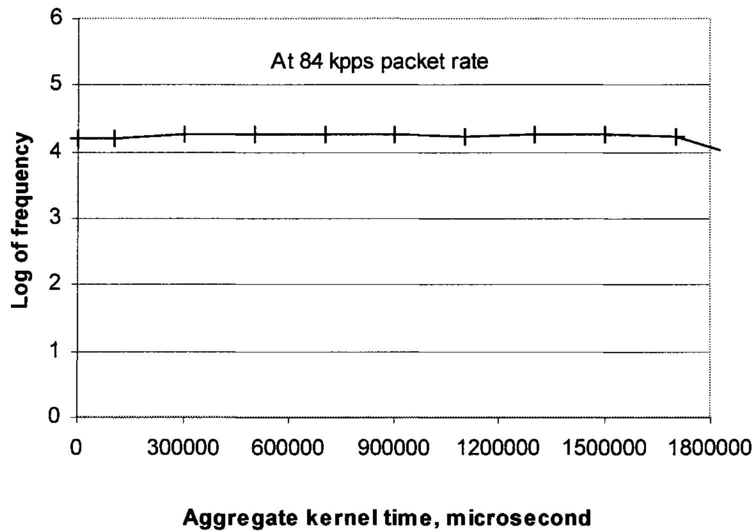
At packet rate below 40 kpps, when NAPI polling is not sustained, the interrupt latency and aggregate kernel time distributions will be similar to Fig. 7.5 and 7.6. Only the average aggregate kernel time will be few microseconds larger due to additional interrupt disable-enable and polling cycle overheads. For higher packet rates when the NAPI polling is sustained the packet delivery latency distribution is governed by the polling period distribution instead of interrupt latency distribution. The polling period distribution for Redhat 8 at 84 kpps is presented in Fig. 7.7 (next page). The distribution has two sharp modes at 125 and 1825 microseconds, and 99th percentile at 1840 microseconds. The primary mode is at least 10 times higher than the secondary mode. The worst case polling period jitter is 2620 microsecond.

Fig. 7.7: Frequency distribution of poll period for NAPI
(Redhat 8 on PII 333Mhz)



A part of the aggregate kernel time distribution is presented in Fig. 7.8 for the same scenario.

Fig. 7.8: Frequency distribution of the aggregate kernel time for NAPI
(Redhat 8 on PII 333 Mhz)



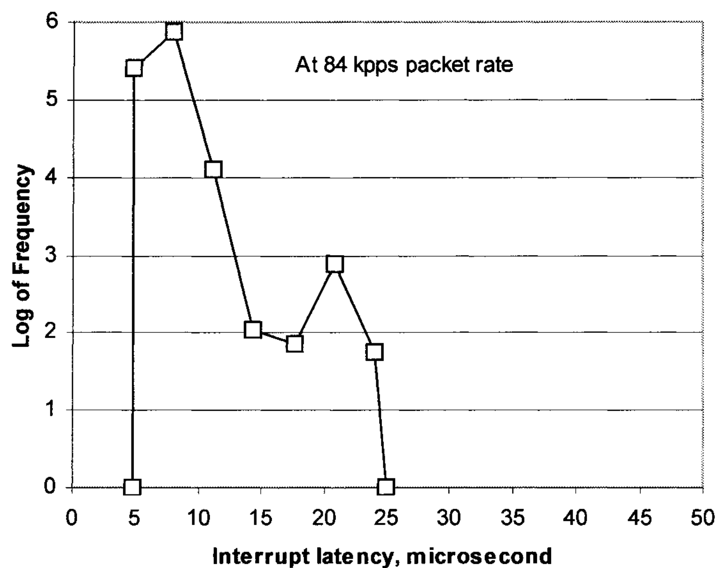
The distribution continues well beyond 1,800,000 microsecond. The distribution has a flat profile. The worst case aggregate kernel time is 6,728,566 microsecond at 84 kpps. This high value of worst case aggregate kernel time indicates user space task starvation

even though the low priority "softirqd" thread for NAPI polling is supposed to avoid that. As the aggregate kernel time latency jitters are several order larger than the polling period jitters, so the aggregate kernel time distribution determines the shape of the packet delivery latency distribution. It is evident that the packet delivery latency distribution will have a flatter / uniform profile due to flat profile of aggregate kernel time.

For PFRING

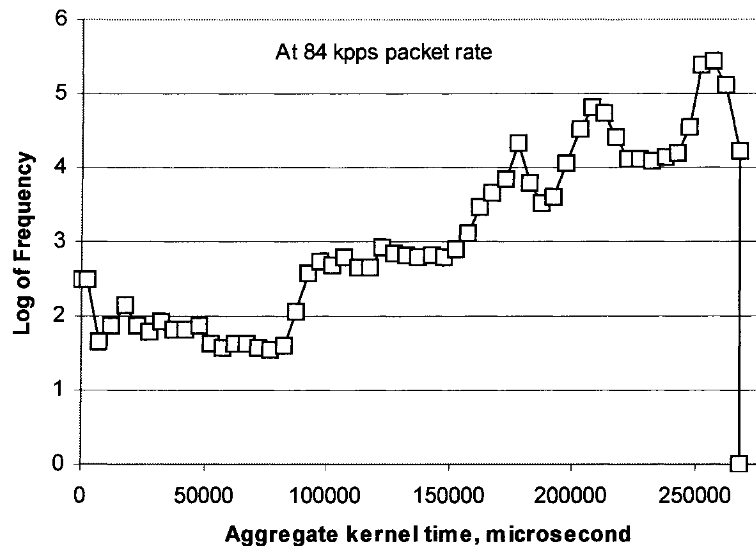
Fig. 7.9 presents the frequency distribution of the interrupt latency measured for PFRING at 84 kpps. This distribution is also applicable even for PFRING with NAPI. The distribution has a very sharp profile with the 99th percentile at 12.8 microseconds. The worst case interrupt latency observed is 25.6 microsecond.

Fig. 7.9: Interrupt latency frequency distribution
(Linux 2.4.23 with PFRING and RTirq patch on PII 333 Mhz)



A part of the aggregate kernel time distribution for PFRING at 84 kpps is presented in Fig. 7.10 (next page).

Fig. 7.10: Frequency distribution of the aggregate kernel time for PFRING
(Linux 2.4.23 with PFRING and RTirq patch on PII 333 Mhz)



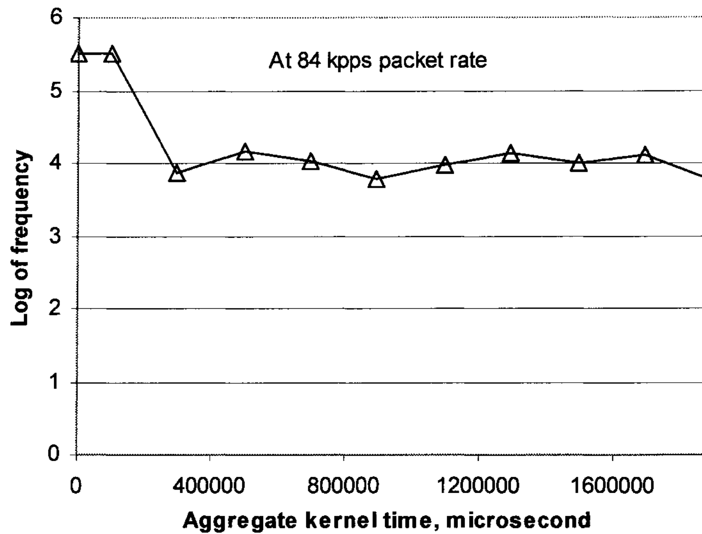
It is evident from this figure that PFRING have high aggregate kernel time jitter. The distribution is skewed towards higher latency values, there is moderately sharp mode around 2,57,500 microsecond. The worst case aggregate kernel time is 2,67,960 microsecond at 84 kpps. As the aggregate kernel time latency jitters are several order larger than the interrupt latency jitters, so the aggregate kernel time distribution determines the shape of the packet delivery latency distribution. Therefore the packet delivery latency distribution will also be skewed towards the higher end due to the skewed aggregate kernel time distribution. The mode of aggregate kernel time defines the mode of the packet delivery latency.

For PFRING with NAPI

At packet rate below 40 kpps when NAPI polling is not sustained, the interrupt latency and aggregate kernel time distributions are similar to Fig. 7.9 and 7.10 for PFRING with NAPI. For higher packet rates when the NAPI polling is sustained the packet delivery latency distribution is governed by the polling period distribution instead of interrupt

latency distribution. The polling period distribution at packets rates at 84 kpps and beyond, is similar to as in Fig. 7.7. The aggregate kernel time distribution for PFRING with NAPI at 84 kpps and beyond, is presented in Fig. 7.11.

Fig. 7.11: Frequency distribution of the aggregate kernel time for PFRING with NAPI
(Linux 2.4.23 with PFRING and RTirq patch, NAPI NIC driver on PII 333 Mhz)



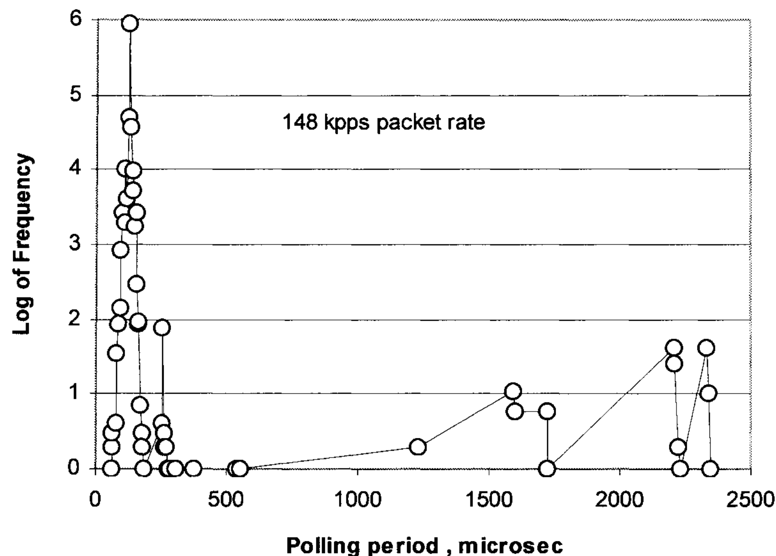
This distribution is skewed toward lower latency values, it has a moderately sharp mode below 100,000 microsecond, but rest of the portion has a flat profile with worst case latency at 11,970,280 microsecond. NAPI introduces additional aggregate kernel jitter when NAPI polling operates with PFRING, due to CPU starvation in user space tasks. The worst case jitter for PFRING with NAPI is 11,970,280 micro second, whereas for NAPI alone is 6,728,566, and for PFRING alone is 267,960. As the aggregate kernel time latency jitters are several order larger than the NAPI polling period jitters, so the aggregate kernel time distribution determines the shape of the packet delivery latency. Therefore the packet delivery latency is skewed towards latency values lower than 100,000 microseconds.

Even though the worst case packet delivery latency is higher in PFRING with NAPI compared to PFRING alone, but the packet loss is less in PFRING with NAPI. This two phenomena are not incongruent because the central tendency (mode) of the latency distribution determines the average packet loss not the worst case latency. Though the worst case latency in case of PFRING with NAPI might be higher, but its mode is lower compared to PFRING. For PFRING, the mode is skewed towards the higher latency value, at around 2,57,500 microsecond (Fig 7.10). Whereas, in case of NAPI with PFRING, the mode of packet delivery latency distribution is skewed towards lower latency values, below 100,000 microsecond (Fig. 7.11).

<u>For DMA ring on Redhat 8</u>

At packet rate below 8 kpps when DMA ring operates in interrupt mode, the interrupt latency and aggregate kernel time distributions similar to Fig. 7.5 and 7.6. For higher packet rates when the DMA ring operates in polling mode, then the packet delivery latency is governed by the polling period distribution. The polling period distribution for DMA ring on Redhat 8 at 148 kpps packet rate (worst load scenario) is presented in Fig. 7.12.

Fig. 7.12: Frequency distribution of poll period for DMA ring
(Redhat 8 on PII 333Mhz)



The profile is very sharp with a single mode at 125 microsecond and the worst case latency at 2345 microsecond. If video power saving feature is enabled, then the observed worst case poll period is 8195 microsecond.

7.2.4 Memory requirement profile

On the basis of memory utilization criteria, the architectures are ranked as in Table 7.5, in decreasing order of performance i.e. on increasing order of memory resource utilization. The memory accounting policy as depicted in section 6.2.4 was adopted. Linux and NAPI use less memory than DMA ring and PFRING architectures. DMA ring requires nearly half the memory than that required by PFRING architectures.

Table 7.5: Memory requirement profile

Rank	Architecture	DMA buffer size (Bytes)	Kernel packet queue and socket buffer size (Bytes)	Total memory requirement (Bytes)
1	NAPI	128*1536	65536	262144
2	Linux	32*1536	300*1536 + 65536	575488
3	DMA ring on Redhat 8	2048*1536		3145728
4	PFRING	32*1536	4096*1536	6340608
5	PFRING + NAPI	128*1536	4096*1536	6488064

7.2.5 Robustness of DMA ring architecture on Redhat 8

Only robustness of DMA ring architecture on Redhat 8 were profiled, other architectures were not tested for robustness as they did not pass the basic no loss capacity requirement within the observation range (<148 kpps). Table 7.6 presents the maximum tolerable data processing load (conservative estimate) for two DMA ring sizes, 1024 and 2048. The

robustness aspect of DMA ring with these two ring sizes is presented in Table 7.7 for loaded and unloaded condition.

Table 7.6: Maximum usable CPU resources for DMA ring on Redhat 8

Architecture	**Maximum tolerable event data processing load at 148 kpps** (as the percentage of additional CPU utilization)
DMA ring size 1024	25 %
DMA ring size 2048	70 %

At 148 kpps packet rate, the DMA ring architecture alone consumed 26% CPU for both ring sizes. With additional data processing load, the CPU utilization goes up above 26%. A load which cause total 51 % CPU utilization, is tolerable when ring size is 1024. This (51-26 =) 25% additional CPU load is caused by the data processing load over and above 26 % CPU consumed by the packet receiver. A ring size of 2048 is more robust, as it allows a higher data processing load (corresponding to total 96% CPU utilization). A DMA ring size of 2048 allows additional 70% CPU utilization, which is a significant portion of the remaining free CPU. Once the maximum tolerable load for these two architectures were ascertained, these architectures were tested for robustness (Table 7.7).

Table 7.7: Robustness of DMA ring on Redhat 8

Test	**DMA ring on Redhat 8**			
	DMA ring size = 1024		**DMA ring size = 2048**	
	No load	**51 % Loaded**	**No load**	**96 % Loaded**
GUI activity - rapid mouse clicks, mouse movement, switching consoles.	Pass	Pass	Pass	Pass
Hard disk activity - Launching programs, browsing /dev and other directories.	Pass	Pass	Pass	Pass
VGA monitor shutdown due to power saving (APM) feature.	*Fails*	*Fails*	Pass	Pass
Stress testing at 148 kpps, for 1 billion packets.	Pass	Pass	Pass	Pass

Although power saving feature is supposed to be disabled in a real-time packet receiver, but this feature was used to simulate transient worst case jitters. DMA architecture with ring size of 1024 failed in this test. This receiver lost packet whenever the system reactivated the AGP system. This happened probably because the host I/O bridge was locked for more than usual time during display reactivation (Fig. 3.6). Ring size of 2048 tolerated all these transient loads and it even withstood an overnight (> 8 hours) continuous operation with full 148 kpps load. So a ring size of 2048 is recommended whenever Redhat 8 is used.

7.3 Contribution of the design and implementation strategies

A combination of design and implementation strategies were behind superior performance of DMA ring architecture. The following sections present how these factors contributed to performance improvements and how much they contributed.

7.3.1 Benefit of polling

Polling alone can lower CPU utilization. But in addition to this fundamental advantage, a couple of better design and implementation choices were behind the superior performance of DMA ring. The transition from interrupt to polling could be sharply defined and controlled in DMA ring, therefore the polling operation could be further optimized by choosing an appropriate transition point. In DMA ring, polling was performed at a fixed low rate to amortize the polling overhead over many packets instead of an adaptive polling strategy as used or suggested in other implementations [39,57]. DMA ring implemented polling in user space which also helped to avoid high kernel to user space context switching time jitters which NAPI suffers from.

Fig. 7.13 and 7.14 illustrates how polling in DMA ring lowers CPU utilization. By setting the interrupt-polling threshold to zero the DMA ring architecture was forced to work only

in interrupt mode, which is termed as "DMA ring (intr)". The CPU utilization for normal polling "DMA ring" is compared to that of "DMA ring (intr)" in Fig. 7.13. So this study compares the same DMA ring with and without polling. From Fig. 7.13 it is evident that the CPU utilization for DMA ring with polling is lower for packet rates higher than 13 kpps. Polling reduce the CPU utilization once it is sustained. At packet rates below 5 kpps, both architectures work without polling hence have similar CPU utilization. The point on the packet rate axis, when polling is sustained is evident from Fig. 7.14.

Fig. 7.13: Benefit of polling on CPU utilization
(DMA ring, on Redhat 8 on PII 33Mhz)

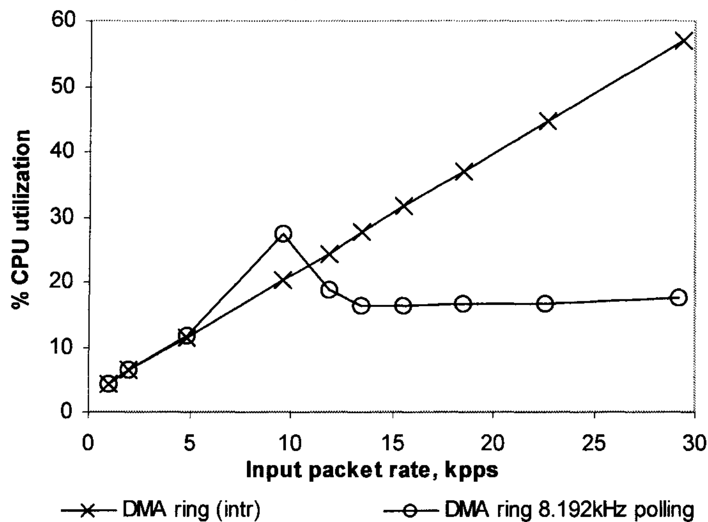


Fig. 7.14: Sharp mode transition in DMA ring: Interrupt to packet ratio
(DMA ring, Redhat 8, PII 333Mhz,)

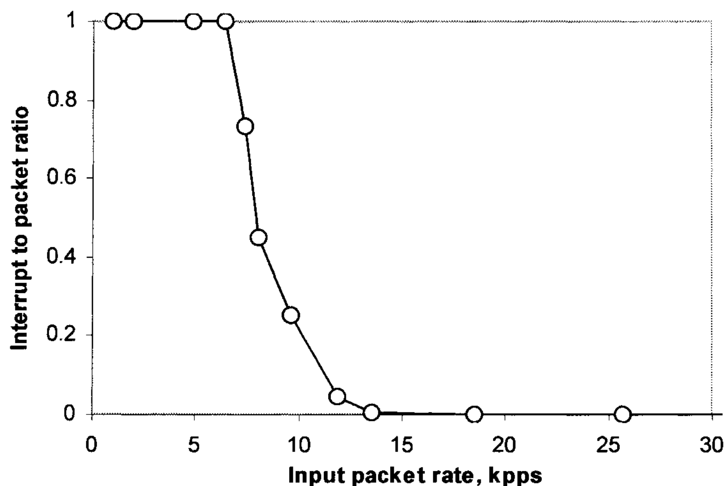
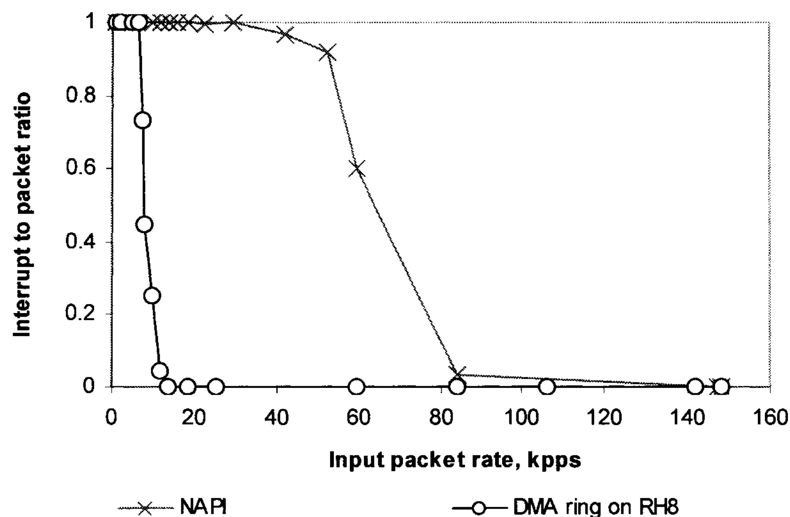


Fig. 7.14 presents the interrupt to packet ratio for DMA ring to illustrate the mode transition. The interrupt to packet ratio drops sharply at higher packet rates beyond 7 kpps, due to the interrupt mitigation effect of the hybrid polling mechanism. In a small band, between 7 to 10 kpps, where the hybrid architecture makes a transition from interrupt to polling mode the CPU utilization rises due to additional overheads incurred in switching the polling timer on and off repeatedly. Once polling is sustained beyond 13 kpps the timer remains switched on, and the architecture starts consuming lower CPU resources (Fig. 7.13). Due to sharp transition, this high CPU utilization region is very limited in case of DMA ring. Once polling sustains beyond 10 kpps, the CPU utilization nearly flat lines to a much lower value, less than 27%, whereas for interrupt operation it rises steeply above 50% in direct proportion to incoming packet rate (Fig . 7.13).

Fig. 7.15 compares the interrupt mitigation effect and mode transition of DMA ring with that of NAPI. This figure presents the interrupt to packet ratio for these architectures for the entire packet rate range (< 148 kpps).

Fig. 7.15 : Mode transition in DMA ring and NAPI: A comparison

(Redhat 8, PII 333Mhz)



In DMA ring, the transition between operation modes are more sharply defined, and can be controlled externally. DMA ring was able to make the transition to polling mode within a narrow band of 7 to 10 kpps, where as for NAPI this happens between 53 to 85 kpps. The transition threshold is set to a low value around 8.3 kpps for DMA ring to optimize the performance, therefore the sustained polling could begin at lower packet rates. DMA ring started polling when its CPU utilization was still low at 28 % (Fig. 7.3), whereas NAPI sustained its polling only above 85 kpps. Long before that, at 29 kpps NAPI was already overload (100% CPU utilization, Fig. 7.3) and was dropping packets (Fig. 7.1).

Packet delivery latency jitter and in-kernel packet processing jitter are synonymous in all architectures. It is evident from Fig. 7.7 and 7.12 that the polling period jitter for DMA ring and NAPI are in same order. For NAPI, the packet delivery latency jitter is the total of NAPI polling period jitter and the aggregate kernel time jitter (Eqn. 6.5, 6.6). Aggregate kernel time jitter is very high in case of NAPI (Fig. 7.8). On the other hand for DMA ring the packet delivery latency/ processing jitter is solely governed by the polling period jitter because the polling is executed in the user space and the packets are available directly in the mapped DMA buffer in user space. Due to user space polling, DMA ring has much lower packet processing jitter compared to NAPI. Due to lower packet processing jitter, buffer overflow and packet losses are avoided in DMA ring. A lower processing jitter also means a lower packet delivery jitter in case of DMA ring.

Due to lower poll frequency in DMA ring, more packets are processed per polling cycle, therefore high polling overheads are amortized over multiple packets. Table 7.8 presents the ratio of packets to polling cycle for DMA ring on Redhat 8 at high packet rates.

Table 7.8: Packets processed per polling cycle

Packet rate kpps	Number of packets received	Number of polling cycle invoked	Number of packets processed per polling cycle
59	1000,000	136831	7.31
84	1000,000	96946	10.32
106	1000,000	76898	13.0
142	1000,000	57299	17.45
148	1000,000	55105	18.15

It is evident from Table 7.8 that multiple packets are processed in each polling cycle at higher packet rates. Amortization of the cost of polling over multiple packets is demonstrated in the next paragraph.

Average polling cycle invocation time and packet processing time can be estimated from the dependency of CPU utilization on polling rate and packet rate (Fig. 7.16, 7.17). Fig. 7.16 gives the relationship between CPU utilization and polling rate at 15.6 kpps packet rate. Fig. 7.17 gives the relationship between CPU utilization and packet rate for 8.192 Khz polling rate.

Fig. 7.16: CPU utilization vs. polling rate
(15.6 kpps, DMA ring on Redhat 8, PII 333Mhz)

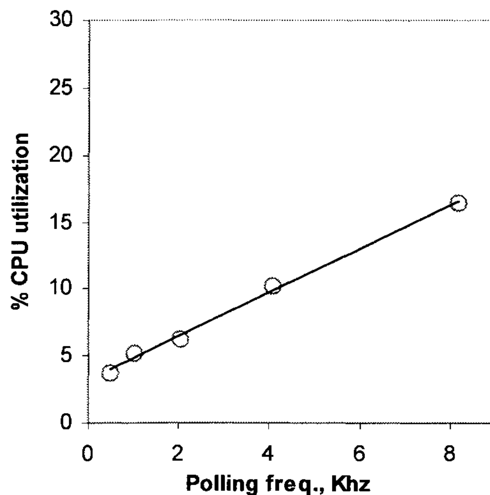
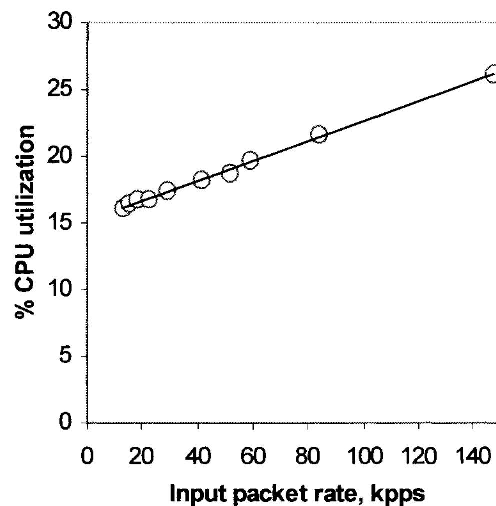


Fig. 7.17: CPU utilization vs. packet rate
(8.192kHz polling, DMA ring on Redhat 8, PII 333Mhz)



By regression the dependency of CPU utilization on polling and packet rates is obtained, which is -

$$\eta = 16.323 * 10^{-6} * f_{Poll} + 0.746 * 10^{-6} * f_{PR} + 0.03158 \text{ ....Eqn. 7.1}$$

where - η is the CPU utilization, presented as a fraction
f_{Poll} is the polling frequency in Hz.
f_{PR} is the incoming packet rate packets per second.

Comparing this relationship, Eqn 7.1 with Eqn. 5.4 of section 5.7, one gets the average polling overhead as 16.323 micro second, and the average packet processing time as 0.746 microsecond per packet and CPU utilization due to background tasks as 3.158 %. Just for information, the time required to process a packet with benefit of cache hits is 0.295 microsecond. During normal processing operation there is no benefit of cache hits as the cache locations for a packet are invalidated after each DMA transfer.

At 148 kpps packet rate, this high amount of polling overhead (16.323 micro second) is amortized over 18 packets (Table 7.8) to yield a lower per packet average of 16.323/18 = 0.91 micro second. Thus at 148 kpps, the per packet average response time is 0.91 + 0.746 = 1.656 microsecond. These overheads corresponds to a maximum packet processing capacity potential of 1119 kpps as given by Eqn. 5.5. Actual system capacity will be far less due to task response jitters. Instead of using fixed low rate polling, if the architecture is run in interrupt mode, then this high polling overhead of 16.323 microseconds would have got included to each packet processing response time. Instead of 1.656 microsecond the response would have been 16.323 + 0.746 = 17.069 microsecond, which corresponds to a maximum packet processing potential of 58 kpps. This would have deteriorated the CPU utilization and packet loss performance substantially.

7.3.2 Benefit of lower context switching frequency, integrated protocol processing, efficient border crossing, and no memory allocation

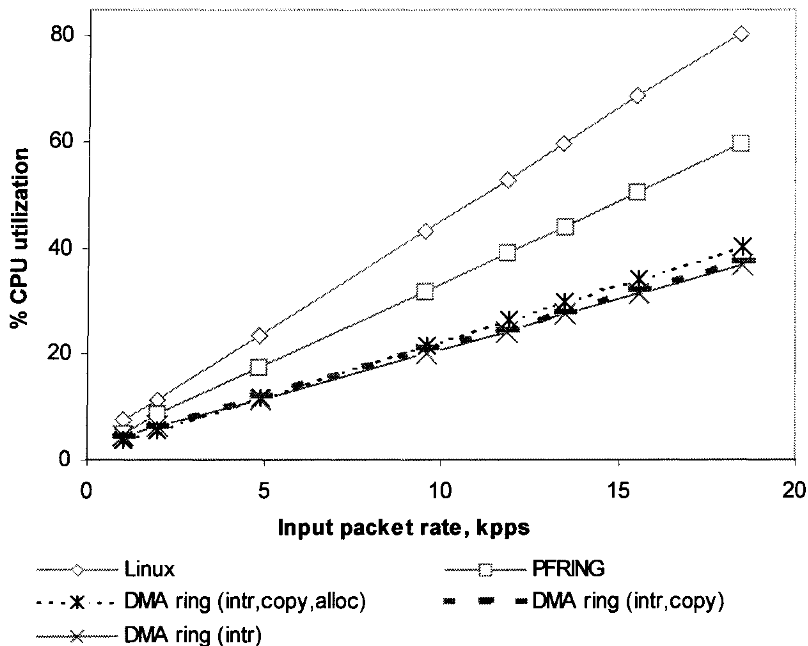
To analyze the contributions of these design and implementation strategies, a set of special test setups were employed. In these receiver setups the DMA ring is forced to operate in three modes -

- with interrupts -"DMA ring(intr)"
- with interrupts and packet copy -"DMA ring(intr, copy)
- with interrupts, packet copy and real-time memory allocation - "DMA ring(intr, copy, alloc)"

By setting the interrupt-polling threshold to a very high rate, the DMA ring architecture was forced to work only in interrupt mode, which is termed as "DMA ring (intr)" operation. In "DMA ring (intr, copy)" mode, a packet was copied to a separate staging packet buffer to complete the protocol processing. This added an additional copy operation for each packets received. The amount of data copied depended on packet length, so for 64 byte packets the performance loss would be less than that for bigger packets. The staging packet buffer on which a new packet is copied is reused for different packets. This common staging packet buffer is allocated during setup time. This "DMA ring (intr, copy)" mode is also forced to operate with interrupts. In "DMA ring (intr, copy, alloc)" mode, in addition to the interrupt and copy operation, a new packet buffer is allocated and de-allocated in the kernel along with DMA mapping and un-mapping. This simulates similar run time behavior of memory allocation in NIC driver layer and de-allocation in the socket layer. The protocol processing is carried out in the same manner in all these three modes as in normal DMA ring.

Fig. 7.18 compares the CPU utilization of these three modes against Linux and PFRING, to illustrate the contribution of each factor behind higher CPU utilization for 64 Byte packets.

Fig. 7.18: CPU utilization profile for different operation modes



Unlike Linux, PFRING does not employ softirq task for protocol processing. PFRING performs the minimal integrated protocol processing in the user space. Therefore PFRING has the advantage of avoiding redundant protocol processing and an extra context switching and over Linux. Linux suffer two context switches - one from ISR to softirq, and the second from softirq to user space task. Whereas PFRING suffers only one - ISR to user space. This is illustrated by a lower CPU utilization for PFRING compared to Linux (Fig. 7.18).

PFRING employed poll() function call to block itself and wait for the packet. Whereas DMA ring uses ioctl() function call to enter the kernel followed by a wait_event_interruptible() kernel API call to block itself. Lower CPU utilization of

"DMA ring(intr, copy, alloc)" compared to PFRING (Fig. 7.18) demonstrates the advantage of an efficient border crossing interface, i.e. ioctl() along wait_event_interruptible() over poll() function call, as used in PFRING [39].

Lower CPU utilization for "DMA ring(intr, copy)" compared to "DMA ring(intr, copy, alloc)" shows the performance gain achieved (Fig. 7.18) by avoiding run time memory allocation as used in the existing NIC drivers for Linux, NAPI, PFRING and PFRING with NAPI. The cost of run time memory allocation-deallocation and DMA mapping-unmapping is moderate, hence the additional CPU utilization due to these are not large.

With 64 Byte packets, the effect of copy is small, hence there is little difference in CPU utilization between "DMA ring (intr, copy)" and "DMA ring (intr)" (Fig. 7.18). All these three modes, Linux and PFRING have similar disadvantages when interrupt overhead is concerned as all of them operates with interrupts.

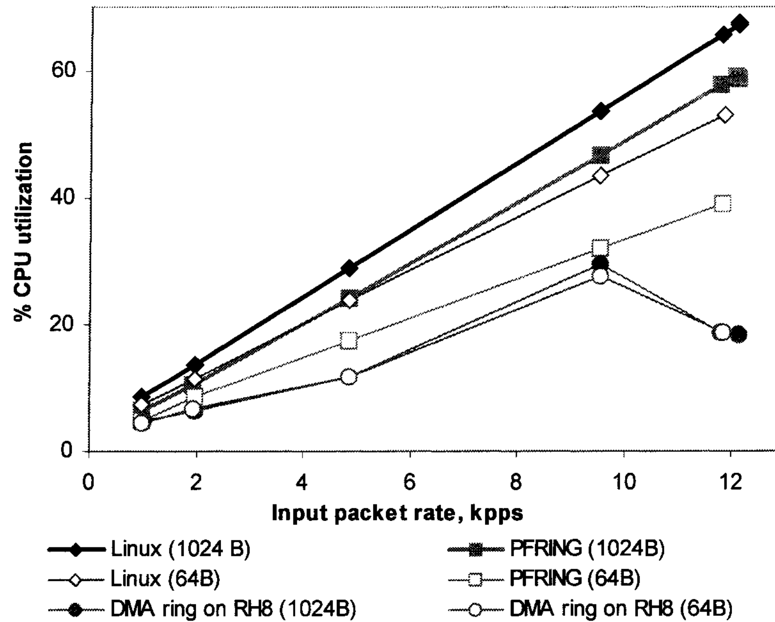
For a given packet rate, from the difference in CPU utilization of these cases, the positive effect of less context switching, integrated protocol processing, efficient border crossing, and run time memory allocation avoidance can be quantified.

7.3.3 Benefit of shared staging area

Fig. 7.19, next page, presents benefit of shared staging area (DMA buffer) by comparing the CPU utilization performance of Linux, PFRING and DMA ring for 64 Byte and 1024 Byte packets. Only a smaller packet rate range is covered (≤12 kpps) which corresponds to 100Mbps line speed for 1024 Byte packets. CPU utilization for PFRING and Linux is higher for 1024 Byte packets due to per byte processing costs. Whereas, the performance of DMA ring is same for both 64 and 1024 Bytes at all packet rates because DMA ring does not have any per byte cost component associated with copy operations. Sharing of staging area (DMA buffer) in DMA ring also does not penalize the performance at

smaller packet sizes unlike traditional zero copy implementations [42,43,44], therefore sharing of staging area has an advantage.

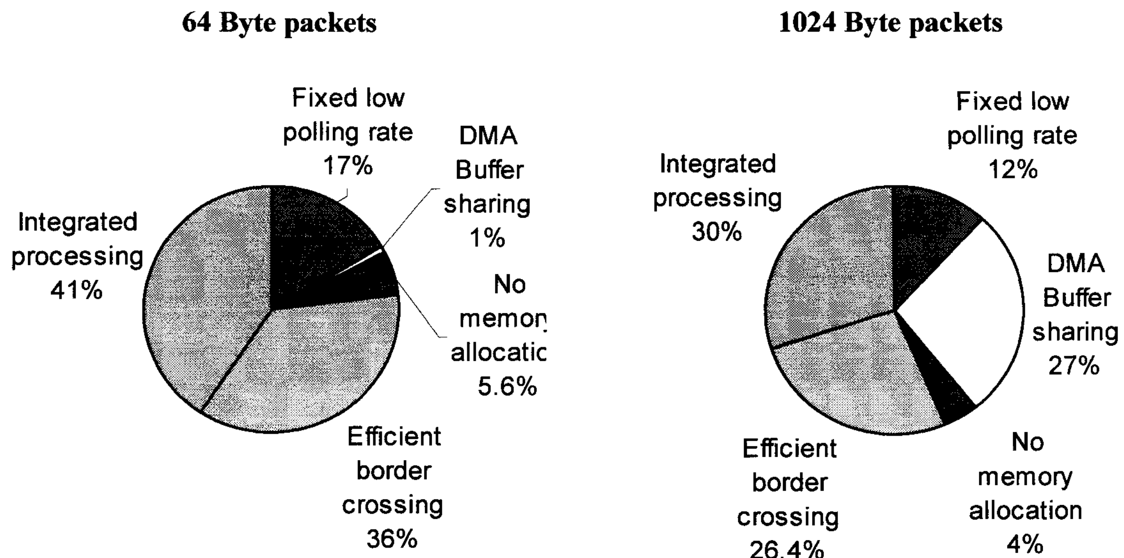
Fig. 7.19: Effect of packet size on CPU utilization for Linux, PFRING and DMA ring: Benefit of shared staging area



7.3.4 Summary

The relative contributions of various performance enhancing design and implementation strategies are summarized in Fig. 7.20. The contributions are quantified from Fig. 7.18 and Fig. 7.19. These factors are independent of each other hence their cumulative effect can be considered to be the aggregate of the individual effects. Fig. 7.20 presents the scenarios for both 64 and 1024 Byte packets at 12 kpps. The impact of DMA buffer sharing is higher in case of the larger packets as more time is saved by not having to copy or transfer large data buffers. For both packet sizes, the impact of minimal integrated protocol processing, context switch avoidance (indicated together as "integrated processing" in Fig. 7.20), efficient border crossing and fixed low rate polling is significant, these reclaims significant CPU time at all packet sizes. DMA buffer sharing gives additional significant advantage only if the packet sizes are large.

Fig. 7.20: Relative contribution of various performance improvement strategies to reduce CPU utilization (at 12 kpps)



7.4 Limitations of DMA ring on Redhat 8

Though the DMA ring architecture on Redhat 8 shows promising CPU utilization, packet loss and packet delivery latency performance, but it falls short in terms of memory utilization compared to Linux and NAPI. It requires more than ten times additional memory compared to Linux and NAPI (Table 7.5). Memory requirement can be reduced by choosing a smaller ring size but this also reduces robustness. Though the packet delivery latency is far better than the other architectures, but still the worst case packet delivery latency is large enough (2345 micro second) to make it unsuitable for many hard real-time applications. These two aspects - high memory requirement and worst case packet delivery latency are interrelated. Worst case task response jitter defines the buffer requirement and the memory utilization. These limitations are not inherent to DMA ring. In the next section the performance of DMA ring on a real-time platform is presented to demonstrate that this limitation can be managed by adopting a real-time platform.

7.5 Performance of DMA ring on real-time platform

RTAI version 3.1 with LXRT on Linux 2.4.24 is chosen as the real-time platform. This platform guarantees bounded task response jitter, therefore a small DMA buffer size of 64 was sufficient to tackle highest packet rate and yet manifest the best robustness. Both, "DMA ring on LXRT with PIT" and "DMA ring on LXRT with RTC" employs a DMA buffer size of 64 bytes. These two architecture manifest all the superior performances figures as "DMA ring on Redhat 8", like no packet loss, low CPU utilization and in addition it yields very low packet delivery latency jitter. The next sub-sections also demonstrate that reduction of memory utilization and packet delivery latency jitter is achieved without any extra cost. As there is no packet loss so that profile is skipped.

7.5.1 CPU utilization profile

Fig. 7.21 and 7.22 presents CPU utilization for "DMA ring on Redhat 8", "DMA ring on LXRT with PIT (8254) timer" and "DMA ring on LXRT with RTC timer". Fig. 7.22 presents the same plot for the packet rate range below 35 kpps. Smaller 64 byte packets allow highest possible packet rates, hence expose worst case behaviors.

Fig. 7.21: CPU utilization profile of DMA ring architectures
(64 Byte UDP packets)

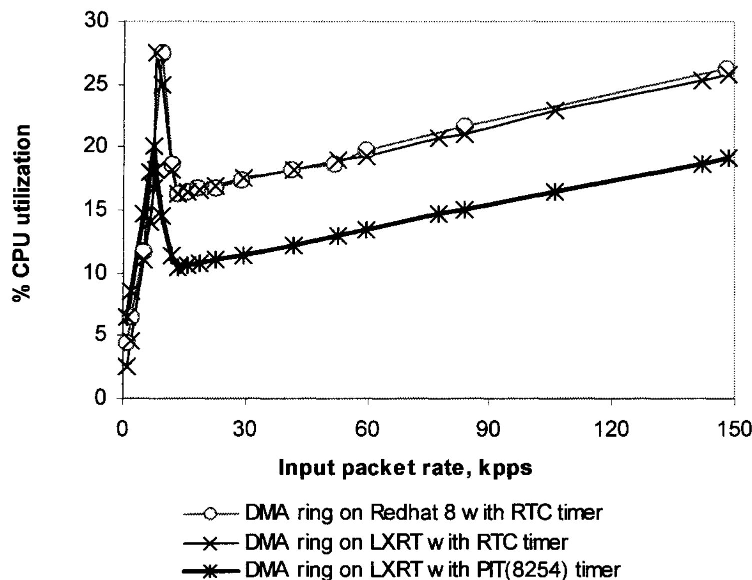
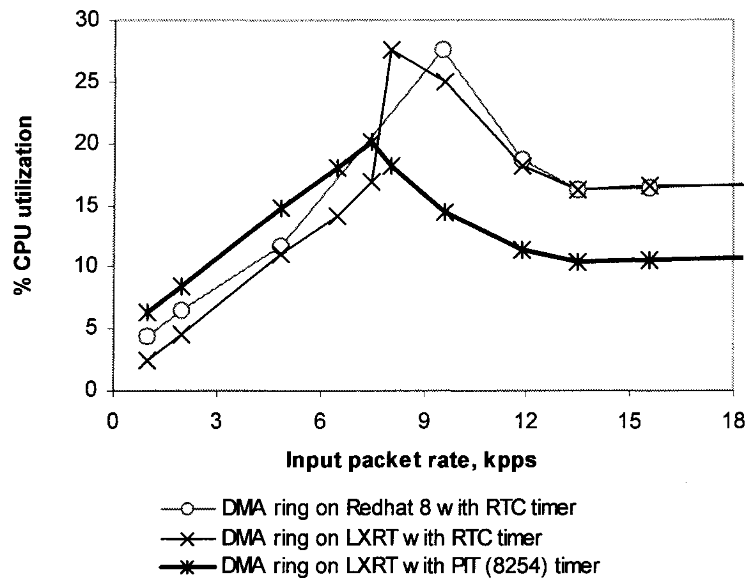


Fig. 7.22: CPU utilization profile of DMA ring architectures at lower packet rates
(64 Byte UDP packets)



The shapes of the CPU utilization profiles of DMA ring on LXRT are similar to that on Redhat 8. At lower packet rates when the DMA ring operates in interrupt mode, the CPU utilization of "DMA ring on LXRT with PIT (8254) timer" is slightly higher than other two architectures. This is because the PIT timer is operational as soon as the system starts, even when the system is not operating in polling mode. Some CPU resource is consumed by the RTAI co-kernel to service these PIT timer interrupts. Interrupt rate of PIT timer is set at 8.192 kHz which is sufficiently high to cause additional 3 to 4 % CPU utilization. In other two architectures, "DMA ring on LXRT with RTC timer" and "DMA ring on Redhat 8", the RTC timer, do not start till the packet rate cross the 8.333 kpps threshold. These architectures do not implement the PIT timer running at 122 microsecond periodicity, hence they do not have additional CPU utilization due to high frequency PIT timer interrupts. That explains 3 to 4 % higher CPU consumption in case of "DMA ring on LXRT with PIT (8254) timer" compared to other two architectures at range below 8 kpps.

On the other hand RTC timer in case of "DMA ring on Redhat 8" and "DMA ring on LXRT with RTC timer" starts beyond 8.333 kpps packet rate and cause additional 5 to 6% CPU consumption due to RTC interrupt servicing. From these figures it is evident that "DMA ring on LXRT with PIT (8254) timer" is the most efficient one when the entire operation range is considered. This establish that RTAI kernel timer is a better implementation choice compared to RTC timer. At all packet rates the "DMA ring on LXRT with PIT (8254) timer" consumes less than 17% of the CPU resources.

"DMA ring on Redhat 8" and "DMA ring on LXRT with RTC timer" are very similar, both uses RTC timer interrupts to pace the polling. Comparing the CPU utilization of these two, it can be concluded that CPU utilization does not deteriorate on using RTAI-LXRT over Linux.

7.5.2 Packet delivery latency profile

Table 7.9 compares the estimated average, 98th percentile and worst case statistics for packet delivery latencies of various DMA ring architectures at different packet rates (10^6 samples).

Table 7.9: Packet delivery latency profile for DMA ring architecture
(for 64 Byte packets, figures in μsec)

Rank	Architecture	Statistics	2kpps	5kpps	18.5kpps	59kpps	148kpps
1.	DMA ring on LXRT with RTC timer	Avg.	14.4	14.4	61.4	61.4	61.4
		98th	19.2	22.4	136.1	136.1	136.1
		Worst case	44.8	44.8	172.5	172.5	172.5
2.	DMA ring on LXRT with PIT(8254) timer	Avg.	14.4	14.4	61.4	61.4	61.4
		98th	19.2	22.4	141.1	141.1	141.1
		Worst case	41.6	44.8	182.5	182.5	182.5
3.	DMA ring on Redhat 8	Avg.	14	14	61	61	61
		98th	25	21	131	136	136
		Worst case	50	123	1935	2345	2345

For LXRT architectures the packet delivery latencies during interrupt operations could be directly measured by reading the NIC counter in user space (section 6.2.3), as these latencies were bounded within 183 microsecond. The poll period were measured by the same method as in "DMA ring on Redhat 8".

LXRT improves the worst case behavior by bounding the jitter, therefore for both LXRT architectures the jitter is low. But nothing can be concluded with sufficient confidence, about which LXRT architecture has lower packet delivery latency.

An actual feel of the packet delivery latency statistics can be obtained by observing the frequency distribution histograms for packet delivery latency and polling period jitter for these architectures.

Fig. 7.23 presents the frequency distribution of the packet delivery latency measured at 5 kpps packet rate on LXRT on the same receiver hardware (PII 333Mhz, Dell desktop). This same distribution is applicable for both LXRT architectures. The worst case interrupt latency observed is 44.8 microsecond.

Fig. 7.23: Frequency distribution of packet delivery latency in LXRT
(122 μ sec timer period, for RTAI 3.1 with Linux 2.4.24 on PII 333 Mhz)

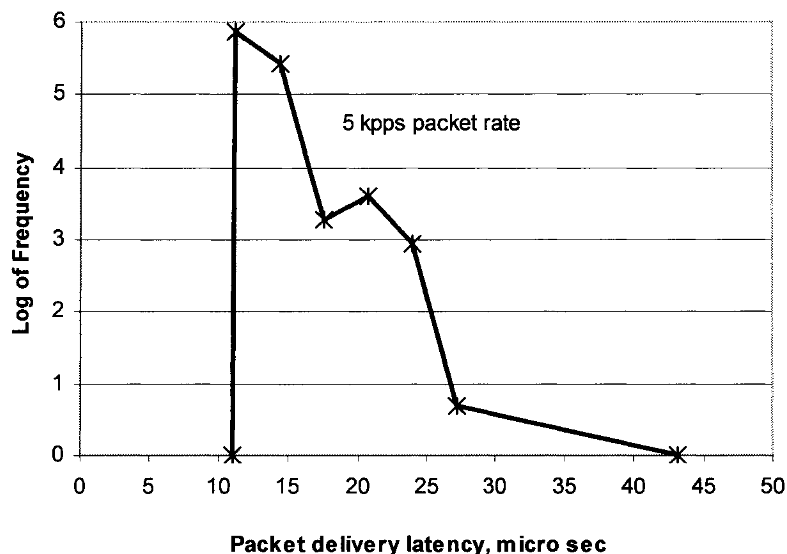


Fig. 7.24 presents the poll period of "DMA ring on LXRT with PIT(8254) timer" relative to that of "DMA ring on Redhat 8". The poll period for LXRT with PIT timer is bounded between 60 and 185 microseconds whereas for Redhat 8, it is distributed between 59 and 2345 microseconds.

Fig. 7.24: Frequency distribution of DMA ring poll period in Redhat 8 and LXRT

(for RTAI 3.1 with Linux 2.4.24 on PII 333 Mhz, Dell desktop)

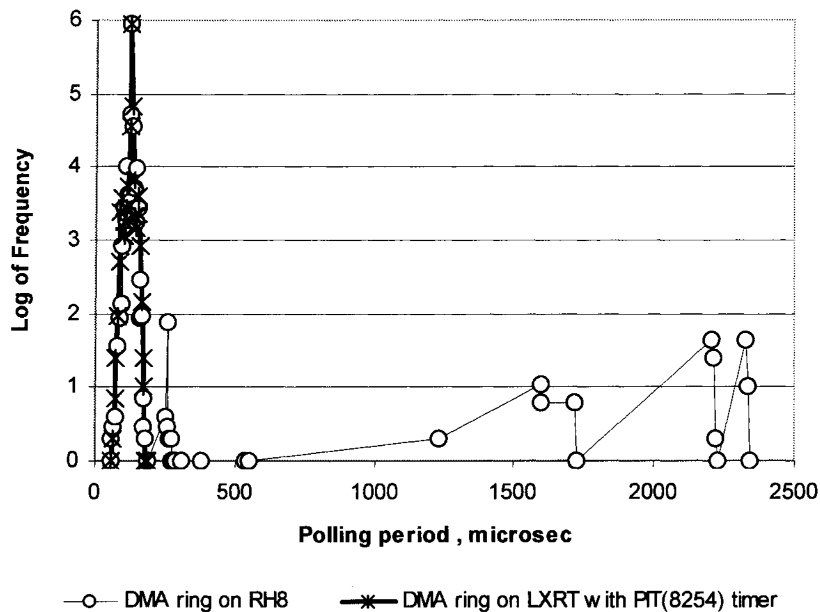
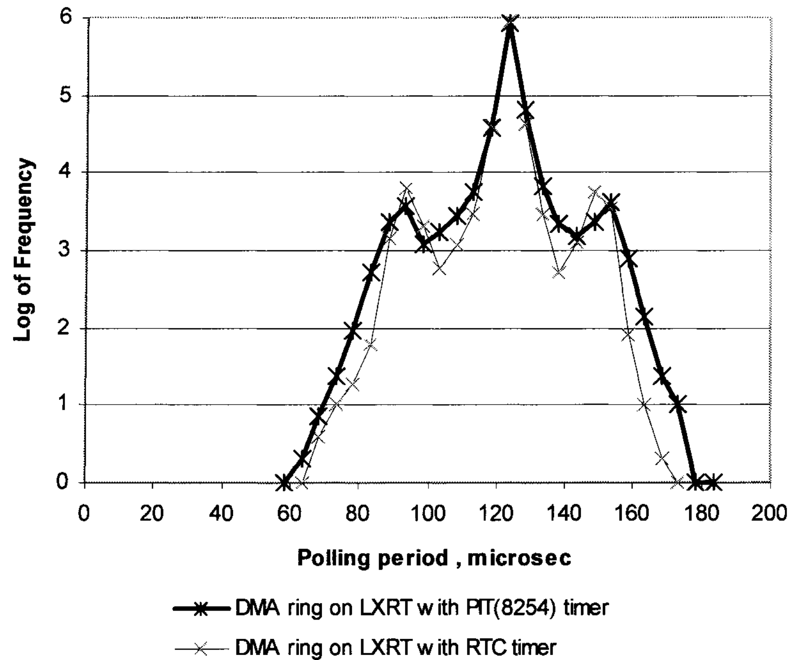


Fig. 7.25 compares the polling period distribution for "DMA ring on LXRT with PIT" with that of "DMA ring on LXRT with RTC" at 148 kpps (worst load scenario). Distribution for RTAI kernel timer based on PIT and RTC timer are very similar with worst case jitter of around 185 microseconds for both. The profile is distributed uniformly on both sides of the timer period at 122 microsecond. This comparison shows that in RTAI co-kernel for uni-processors, both schemes: polling paced by native RTAI timer (i.e. PIT in this case) and polling by an external hardware timer (i.e. the RTC timer in this case), yields similar advantage when jitter is concerned. This may not be the case

in RTAI co-kernels for SMP architectures, where native RTAI timers are differently implemented and may yield lower jitters than PIT and RTC.

Fig. 7.25: Frequency distribution of polling period paced by PIT and RTC timer in LXRT
(for RTAI 3.1 with Linux 2.4.24 on PII 333 Mhz, Dell desktop)



7.5.3 Memory requirement profile

On basis of memory utilization criteria, the architectures can be ranked as in Table 7.10 in decreasing order of performance i.e. on increasing order of memory resource utilization. Each packet buffer on the DMA ring in all the architectures were provisioned with 1536 Bytes (section 6.2.3). Both the LXRT architectures, i.e. "DMA ring on LXRT with PIT" and "DMA ring on LXRT with RTC" have same memory requirements, so both of them are represented by a single entry, "DMA ring on LXRT", in Table 7.10. DMA ring on LXRT does not require any socket buffer, hence with only 64 entries in the DMA buffer, it requires the least memory resource compared with Linux and NAPI.

Table 7.10: Memory utilization of DMA ring on LXRT and Redhat 8 compared with other low memory architectures

Rank	Architecture	DMA buffer size (Bytes)	Kernel packet queue and socket buffer size (Bytes)	Total memory requirement (Bytes)
1	DMA ring on LXRT	64*1536	-	98304
2	NAPI	128*1536	65536	262144
3	Linux	32*1536	300*1536 + 65536	575488
4	DMA ring on Redhat 8	2048*1536	-	3145728

7.5.4 Robustness

"DMA ring on LXRT with PIT timer" with DMA ring size of 64 can tolerate an additional load of more than 70%. With this load it pass all the robustness tests which "DMA ring on Redhat 8" passed, as depicted in section 7.2.5.

By using LXRT the DMA buffer size requirement is reduced, without sacrificing the robustness and other performance aspects. So it can be concluded that the apparent weakness of "DMA ring on Redhat 8" has been effectively addressed by LXRT.

7.6 Comparison of overheads in LXRT and Redhat 8

Average polling cycle invocation overhead for "DMA ring on LXRT with PIT" can be estimated from the dependency of CPU utilization on polling rate and packet rate, as done at the end of section 7.3.1. By regression, the dependency of CPU utilization on polling and packet rates is obtained as -

$$\eta = 8.128 * 10^{-6} * f_{Poll} + 0.746 * 10^{-6} * f_{PR} + 0.01 \quad \text{..........Eqn. 7.2}$$

where - η is the CPU utilization, presented as a fraction
f_{Poll} is the polling frequency in Hz.
f_{PR} is the incoming packet rate packets per second.

Comparing this relationship with Eqn. 5.4 of section 5.7, the average polling overhead is estimated as 8.128 microsecond. The average packet processing time is still the same 0.746 microsecond per packet as in Eqn. 7.1. The CPU utilization due to background tasks is only 1.0 % in LXRT whereas the same for Redhat 8 was 3.2 %. The polling overhead (8.128 microsecond) in LXRT is half of that in case of the Redhat 8 implementation (16.323 microsecond). This shows that RTAI's interrupt servicing and the rt_sleep() function call used to block and wait for the next polling period is more efficient than Linux's interrupt servicing, ioctl() and wait_event_interruptible() mechanisms. The CPU utilization due to back ground kernel tasks in LXRT is one third of that in case of Redhat 8. This may indicate that the RTAI scheduler is more efficient that the Redhat 8 Linux scheduler. So RTAI on Linux 2.4.24 may be a better choice in all respects compared to Redhat 8.

7.7 Summary

The key findings from the performance studies can be summarized as following -

- Packet loss in receiver is primarily due to buffer overflow.
- Low fixed rate polling yields better performance even though polling overhead may be quite high. This strategy amortizes polling overhead over many packets.
- Threshold based mode switching in a hybrid-interrupt-polling scheme allows optimization of the polling operation. Performance loss due to frequent mode switching is avoided.
- Reduced context switching, integrated minimal protocol processing, efficient border crossing, using a common staging area to avoid run time memory allocation and copy operation improves performance of a packet receiving architecture.

- The proposed "DMA ring" architecture has superior performance compared to existing solutions - NAPI, PFRING and Linux networking stack in terms of significantly lower CPU utilization, lower packet delivery latency and robustness (Table 7.11).

Table 7.11: Superior performance of DMA ring: Comparison with best of class solutions.

Performance Measure	DMA ring on LXRT (PIT)	DMA ring on Redhat 8 (RTC)	Contemporary best of class
No loss capacity	**> 148 kpps**	> 148 kpps	PFRING 84 kpps
Loss at 148 kpps	**0%**	0%	PFRING + NAPI 29.40%
CPU utilization	**< 17 %**	< 28 %	All 100%
CPU available for event Processing tasks	**> 70 %**	70%	All 0%
Packet delivery latency (micro sec) at 84 kpps	**Avg = 61 98th = 136 Worst case = 172**	Avg = 61 98th = 136 Worst case = 2345	PFRING Avg =240608 98th = 266081 Worst case = 267985
Memory requirement (Bytes)	**98304**	3145728	NAPI 262144
Robustness: can handle continuous load over 1 billion packets	**Pass not a single loss**	Pass not a single loss	No competing solution is available

- Minimizing protocol processing, efficient border crossing and polling can yield significant benefits at all packet rates. Shared DMA buffer render additional benefit only if the packet size is quite large.
- There is some cost associated with memory allocation, but it is not too high.
- If DMA ring is used on a GPOS like Redhat 8, then it requires more memory, however if it is implemented on a real-time platform, this weakness is addressed.

DMA ring on a real-time platform use the least amount of memory even less than half of what Linux networking stack requires.

- DMA ring on a real-time platform also bounds the worst case packet delivery latency to a small value.
- It is possible to optimize performance of DMA ring on a real-time platform by utilizing the native kernel timer to pace the polling instead of using external hardware timer interrupts. Choosing suspend-resume for task blocking and unblocking yields better DMA ring performance.
- Using a real-time platform with an additional kernel itself does not deteriorate the CPU performance of the system. There is no cost of implementing such real-time platform in the present problem context.
- In fact, the RTAI scheduler may be more efficient than Redhat 8 Linux scheduler.
- NAPI has the worst no loss packet capacity, it also deteriorates this performance aspect of "PFRING with NAPI" architecture.
- NAPI introduces high packet delivery jitters at high packet rates, it depreciates this performance aspect of the "PFRING with NAPI" architecture.
- NAPI suffer from livelock phenomena even though it was designed as a solution against it. This was because the NAPI polling is sustained only at very high packet rates, and because NAPI wastes much CPU resources due to frequent context switching, high polling overheads and kernel protocol stack inefficiencies.
- Even though NAPI was designed to employ low priority softirqd tasks to reduce CPU starvation of user space tasks, but user space tasks still starved. NAPI fails in both the design goals - livelock and user space starvation.

- In spite of these limitations of NAPI, it reduces packet losses in case of PFRING. PFRING with NAPI has far better throughput and lower packet loss compared to PFRING alone.
- PFRING collapses at very high packet rate.
- A user space mechanism can be better or as good as a kernel mechanism for network I/O processing if certain design pre-cautions are taken.

Chapter 8: Related Work and Contributions

The artifacts, as discussed in Chapter 4, could be readily deployed with off the shelf components to rig up a packet capturing solution. Other than those solutions, many other related research works are also available. These related works provide interesting insights and concepts. Some of these ideas are embodied as custom hardware or software components, therefore they cannot be readily deployed with off the shelf components in their present form. Some propositions either do not provide features to implement a required packet capturing architecture or they do not provide any special advantage in the present problem context. Hence these works were not classified as solutions under chapter 4. One interesting real-time networking stack, RTnet is described, even though it cannot be employed to solve the packet capturing problem in a general context. Nevertheless an analysis of RTnet is presented to demonstrate that its performance is expected to be worse than that of DMA ring.

However, the ideas behind these contributions are worth ässimilating. They aid in perceiving the position of the proposed DAM ring architecture in the concept space. These related works are briefly described in the following section. With this given background, the contributions of this thesis are presented. The differentiating features of the DMA ring are discussed and compared against these related works.

8.1 Related works covering aspects of network receiving performance

The existing literature may be classified under the following five categories:

8.1.1 Approaches to tackle interrupt servicing overheads

To accept interrupts at a high rate, operating system's role to deliver an event may be reduced [63]. The network interface can be brought closer to the CPU by reorganizing the

hardware. Instead of the usual scheme of raising interrupts request, dispatching them through OS and invoking the ISR, the network interface can directly deliver the network event notification to the user process without involving the OS. The network interface will manipulate the process state bits in the memory to force a running thread to preempt the current process. The network interface will also modify the state of a blocked thread, which was waiting for a network event, to make it runnable. This architecture can be implemented on a simultaneous multi threading (SMT) processors to hide the latencies and context switching involved in interrupt servicing. No performance results or implementation were presented.

"Bursty scheduling" of interrupt loads has been proposed to manage high interrupt rates in case of embedded systems [64]. The logic behind "bursty scheduling" is quite similar to DMA ring polling, interrupt is disabled if the interrupt rate is higher than a threshold and events are serviced in a group to amortize the event servicing overhead over many task cycles. This work presented very good performance results for a implementation based on a embedded system OS running on an extremely low power embedded CPU (4Mhz) with 10kHz interrupt rates. This work was implemented on a custom embedded system hardware, it is not applicable for Intel x86 PC architecture in its present form.

"Clocked interrupt" scheme to handle network events avoids the high interrupt overheads [65]. This scheme is actually similar to DMA ring polling, only the difference being, "clocked interrupt" is paced by software kernel timer. Therefore it is expected to suffer from high jitters due to OS limitations. This scheme was implemented on a modified AIX kernel on HP9000/700 hardware with "Afterburner" ATM/SONET network adapter. The AIX OS kernel was modified to increase the system clock rate from the standard 100 HZ value. The researchers opted for data copy instead of zero copy I/O. The researchers maintained that cost of zero copy packet buffer transfers are larger than cost of data copy. They studied ATM/SONET NICs with large message and packet sizes. They observed

that packet buffer transfer between network layers require stripping the DMA mapping from the individual packet buffers which are transferred out of the DMA buffer. When this packet buffer is replaced by a new one then DMA mapping of this fresh packet buffer have to be made. They observed that DMA mapping and tear down is costly compared to data copy mechanism, hence they opted for data copy.

The reality in Linux, on Intel x86 architectures with PCI NICs is different. First of all, individual packet buffers allocation and de-allocation is not required if a common staging area is employed for DMA transfer and protocol processing (section 5.1). Secondly, for PCI devices in Linux, the DMA mapping and stripping is straightforward and not very costly (Fig. 7.18, 7.20). DMA mapping simply means flushing the cache and computing the true physical address of the first location of the contiguous memory segment. These can be achieved by a few processor instructions for all packet sizes. Hence these arguments does not hold true for Linux on Inter x86 hardware with PCI NICs.

8.1.2 Optimized network processing

A work [37] on Myrinet NIC (not Ethernet) addressed the task balancing and buffer requirement issue in an interrupt driven multi layered network processing architecture. This work further optimized an available user level direct access network API, the Fast Message (FM) library for Myrinet. By forcing interleaved scheduling of task threads in different layers, the requirement of any intermediate buffer was entirely avoided. To avoid message losses it implemented flow control with the FM protocol primitives to match the sending rate with receiver capacity. A common staging area was used for all processing layers. To improve performance with large message sizes, it pipelined processing of incomplete messages in the receiver, i.e. the receiver starts processing of the received message fragments even though all the message fragments have not yet

arrived. This work did not address the kernel to user space border crossing costs. Performance improvement with these strategies were established with experimental data.

RTnet, a real-time network stack also implemented many other optimizations, this is separately covered later under section 8.1.5.

8.1.3 User space network access

Some work has been done to support direct access of network devices from user space without involving the OS. At least two complete user space driver development kits are available for Linux PCI NICs which claims to facilitate complete user space driver development.

Gelato project offers a library to implement an interrupt handler as real-time thread in the user space along with other PCI and DMA resource management APIs [66]. Another elaborate library, known as "User-Space Driver Development Kit" (USDDK) is available for similar purpose [67]. Both these libraries use mmap feature of Linux to avoid data copy during kernel to user space border crossing. These works were primarily proposed to avoid the high development costs associated with kernel module development. No published work was found that compared performance of user space drivers developed with these libraries, against the performance of kernel drivers.

USDDK library did not allow enough flexibility to easily implement the DMA ring architecture, rather it was worthwhile to implement DMA ring using native Linux kernel API. Gelato project library was not explored as Linux kernel API was found suitable enough for the job. Not much driver development work was involved which could benefit from these libraries. Most of the work involved modifying and porting the existing NIC kernel driver and developing the user space polling logic. These libraries have no especial proposition to make in these two areas.

uDAPL (user direct access programming library) is a Linux library developed for InfiBand Architecture adapters for cluster systems [68]. Fast access to shared remote distributed memory and messaging between cluster nodes can be carried out from user space using these API. The event dispatch to user space is either carried out by polling or by interrupts. This work also compared the performance of a user space driver developed utilizing this library with that of a kernel driver. The user space driver performs worse than the kernel driver with interrupt based operation. This work is not for IP network. This library cannot be used in UDP/IP context.

A publication on Linux user space hard disk driver is available [69]. The performance of this user space driver was poorer than the kernel driver, but they were comparable, the throughput was within 75% and the CPU utilization percentage was only additional 2 to 8 % for different data rates. This work argued the benefit of a small driver code footprint to result higher cache hits. This work did not address the problem how an user space driver resource can be effectively shared by multiple users and what would be its performance in such multi user environment. Multi user environment will mean multiple processes and threads and higher inter process interactions. All these are expected to deteriorate the performance significantly.

All these works on user space drivers give an idea that if development support or libraries are available then low cost user space drivers can be developed. The performance comparisons indicate that additional design strategies are needed to improve the performance of these user space drivers and add a multi-user interface to it.

8.1.4 Enhanced network interface cards

Network interface cards can be enhanced to provide interfaces for user space access by multiple users. A work proposed and implemented a NIC interface for user level drivers/applications for cluster networks (CLAN) [70]. The interface is suitable for

multiple user applications accessing the same NIC hardware to perform remote memory operations on other cluster nodes across the CLAN. This work involved developing/modifying the NIC firmware. The modified NIC firmware carried out many interface management tasks, the host CPU need not undertake them and this saved host CPU cycles. This work claimed to reduce the number of real-time system calls. The proposed NIC interface demonstrated better performance compared to another gigabit NIC interface, VIA on Giganet cLAN, in terms of higher throughput and lower CPU utilization.

"Lazy receiver processing" approach optimizes the network stack operation by offloading the packet de-multiplexing task to the NIC [71]. This work was implemented on 4.4 BSD Unix on SPARC with ATM adapter. Separate DMA buffers are maintained for each socket opened. The NIC examines the packet's destination port and places the packet in the appropriate DMA buffer. If a socket buffer is full then the NIC drops the packet. This saves the host CPU from unnecessary protocol processing in the kernel in case there is no socket to receive the packet. In case of normal BSD Unix (or Linux) kernels, packets are dropped in the socket layer after much wasteful processing in the lower stack layers. In addition to demux task offloading to NIC, the kernel stack is modified so that protocol processing is carried out on demand by the user task thread when it enters the kernel. Both these strategies improved the throughput capacity. This approach required OS kernel and NIC firmware modification.

In addition to executing interface management tasks on the NIC, some protocol processing tasks can be offloaded on the NIC hardware, which can save additional host CPU cycles [72]. The source code of the Aleton Gigabit Ethernet NIC firmware is openly shared by its manufacturer. So the researchers took this opportunity to develop a firmware for this NIC to implement an interface between the NIC and user space driver which by passes the OS kernel completely. The DMA to and from the user space were

carried out without involving the OS kernel. The higher throughput, lower packet delivery latency and CPU utilization of this scheme was very promising. However it is not clear how this scheme can work with multiple users.

Another work [73] implemented an adaptive interrupt mitigation in Myrinet NIC. This is also a kind of hybrid interrupt-polling system. The NIC implements a monoshot timer, and the host implements an interrupt enable-disable flag and a packet counter in its DMA memory. When a packet arrives, the NIC makes DMA transfer, raises an interrupt and starts the monoshot timer. If another packet arrives within the timer period then the NIC makes DMA transfer but does not raise an interrupt. For very high packet arrival rate the interrupt rate is limited to a mitigated interrupt rate given by the reciprocal of the timer period. Interrupts are generated only periodically even though packets may arrive very fast. This scheme adapts itself to the host's processing capacity and may raise the mitigated rate if the NIC observes that the host is able to quickly process the pending packets. This is implemented by a packet counter. The host CPU increments the counter for every packet processed. NIC polls the counter frequently to monitor the progress of packet processing on the host. This polling rate is governed by the same monoshot timer. If the NIC finds that host has already processed all the pending packets then the NIC cancels the timer, thereby allowing an interrupt to be raised earlier when the next packet arrives. This allows more frequent interrupts if the host can process them faster. To completely disable interrupts the host turns off the interrupt enable-disable flag. The NIC polls this flag along with the packet counter, and if the flag is off, the NIC does not raise any interrupt till this flag is again turned on by the host. The NIC periodically downloads the values of this flag and the packet counter from host memory by DMA transfer to examine them.

A very comprehensive treatment of multi user, user level NIC interfacing has been carried out in a D.Sc. thesis [74]. All aspects of hardware and software architectures in

the host and NIC side has been addressed. This work involved NIC chip design, fabrication, PCI NIC hardware, driver development and testing. This work proposed a NIC interface suitable for direct user space access in a multi user environment. All implementations involving enhanced NIC need custom NIC hardware.

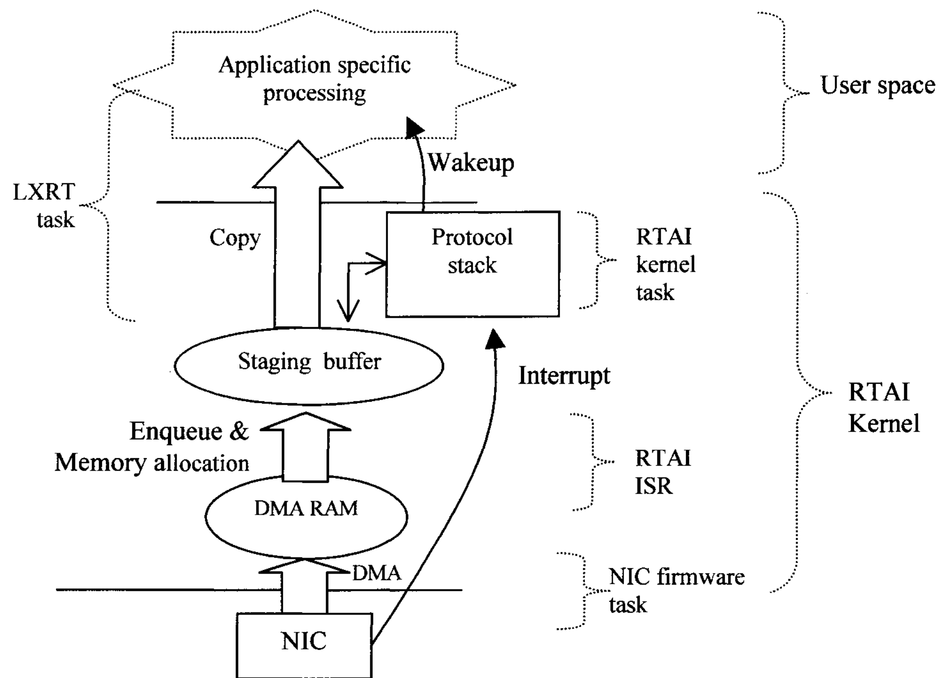
8.1.5 Real-time networking stack

A real-time networking stack is available for RTAI on Linux. The protocol stack is optimized to minimize the packet delivery latency in addition to a guaranteed packet transit delay [47,75,76,77]. It is the optimized stack that is interesting in the given problem context. RTnet implements UDP, ICMP and ARP protocols which is just sufficient for a distributed real-time application to operate over an IP network. This scheme requires a dedicated LAN segment where all the real-time nodes are tightly bound with each other [75,76]. All the participating real-time nodes have to implement the RTnet stack on RTAI. Network and system parameters about all the participating real-time nodes are collected at a central place during setup time. Every node is allocated a specific media access time slot when it sends packets. Real-time media access control and central archiving of node parameters are executed by two additional real-time networking protocol running over the usual UDP/IP, Ethernet and MAC protocol layers.

The RTnet task model and architecture layering (Fig. 8.1, next page) [47] is some what similar to Linux. However there are differences. The ISR does not belong to Linux domain, it is a RTAI ISR. The softirq task thread as in Linux is replaced by a high priority RTAI kernel task thread which takes care of the protocol processing. The user space task thread is a high priority LXRT task. There is no packet queue buffer or socket buffer however the NIC DMA buffer still exists. Instead of both packet queue and socket buffer there is one single staging buffer for each network socket opened. This staging buffer serves as a common staging area for protocol processing and socket buffering.

This staging buffer is created when a socket is opened from user space. The socket operating semantics is similar to BSD socket operation [78]. RTnet allows multiple users. The RTnet stack sits over RTAI kernel and can co-exists with Linux and its native network stack.

Fig. 8.1: RTnet network stack
(3 tasks, 1 copy, 1 memory allocation)



When a LXRT task tries to read a empty socket it blocks on a socket semaphore. This socket semaphore (a RTAI kernel semaphore) is not available if there are no packets in the staging buffer. The RTAI kernel task blocks on an interrupt semaphore (RTAI kernel semaphore) when there are no pending packets to process in the staging buffer. When a packet is received the RTAI ISR is invoked, the ISR transfers the packet from the DMA buffer to the staging buffer and releases/signals the interrupt semaphore on exit so that the kernel task is waked up to processes the packet. On completion of the protocol processing, the RTAI kernel task releases/signals the socket semaphore to wakeup the blocked user space LXRT task to carry out the event data processing.

Like Linux, the packet transfer from DMA buffer to staging buffer is carried by packet buffer transfer instead of copying for large packet sizes. The packet data is copied from staging buffer to user space memory on the return path of the socket read call. But this stack does not allocate memory to replenish the DMA buffer, because when the ISR hands over the packet to the staging buffer, it receives an empty packet buffer in return. This empty packet buffer is the one whose data has been already read by the user space task earlier. Sufficient numbers of empty buffers are pooled at setup time, so that the system do not starve out of empty buffers at run time. Only a small number of empty buffers and small DMA and staging buffer sizes are required as all three tasks can be balanced due to their true real-time priorities and manageable response jitters. So this architecture do not suffer from memory allocation inefficiencies but only from data copy, unnecessary protocol processing, context switching and DMA mapping-unmapping inefficiencies. Task synchronization with semaphore is a costly method and may not yield better performance than suspend-resume method as used in DMA ring on LXRT. Context switching, unnecessary protocol processing and data copy avoidance are three most significant advantages that yields higher performance (Fig. 7.20). Unlike DMA ring, RTnet does not exploit these key improvements. So it is most likely that DMA ring will have better receiving performance than RTnet. RTnet implementation also requires modification in the NIC driver, similar to what is required for DMA ring implementation.

RTnet cannot be used in a system where the packet sending nodes are numerous, where they do not implement a real-time OS or RTnet protocol and they do not belong to the same dedicated LAN segment. A NMS or NIDS application is one such application where RTnet cannot be deployed, therefore it was not deemed as a deployable solution.

8.2 Differences and advantages of proposed DMA ring architecture

DMA ring architecture reduces both per packet and per byte packet processing cost component, thus it yields superior performance at small and very large packet sizes. In most situations small sized packets below 600 bytes are most likely [36,37]. For network management applications the alarm message size is also below 500 bytes. In worst case scenario, smaller packet size implies higher packet rates. So managing per packet costs in a high packet rate scenario is more important than eliminating only data copy to tackle large packets.

DMA ring is a scalable software solution that does not require any modifications in the network adapter, neither in the host hardware, nor in the OS kernel. The single thread single buffer architecture has simple system dynamics, whose performance scales linearly with CPU speed. DMA ring works with off the shelf, readily available commodity components. It does not depend on any low availability, difficult to administer, custom patch. DMA ring can work on either Redhat 8 or any RTAI vanilla Linux combination. 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 CPU clock cycle counter (TSC) is the only CPU dependent function that needs to be modified.

DMA ring can work with any network protocol. DMA ring may even support ATM/SONET NICs. Memory requirement for a DMA ring size of 2048 for individual 64KByte packet buffers for Gigabit jumbo frames, is on the higher side and might pose practical problems, but in that case DMA ring for Gigabit NICs can be implemented on LXRT with smaller (64) ring sizes. It can work with any PCI NIC with commodity features. The modifications required in the NIC driver 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. The user space component is fairly generic and can work on any platform that supports Linux.

DMA ring is a non-adaptive hybrid interrupt-polling mechanism implemented partially in the user space. 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, minimal protocol processing, 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). The current DMA ring architecture only serves a single packet capturing process, which is sufficient for NMS and NIDS applications.

8.3 Future research directions and insights for further improvements

The DMA ring was tested only on uniprocessor hardware where multi threading has no advantage. The event data processing tasks may require more CPU resources, in such case, multiple concurrent threads on multiple processors can be employed to carry out event data processing. It might be possible to run concurrent user space threads for packet receiving and event data processing tasks without explicit task synchronization and balancing. An enhanced version of the DMA ring buffer can still act as the common staging area for all threads. This DMA ring memory will be shared across all Linux threads or processes. The DMA ring LKM is already thread safe, so most of the design enhancements for multi threaded operation can be easily implemented in user space. Another area worth studying is the performance of the suggested algorithm that decides

the mode switching. Under various packet arrival patterns its performance can be ascertained.

8.4 Conclusion and discussions

This thesis investigated the packet capturing problem in network management and network intrusion detection applications. The receiver capacity limitations largely determined the packet loss characteristics in such systems. The issues that limit the packet capturing performance in a receiver host were identified. In this regards, several possible architectural forms, the relationship between performance of these architectures and their structural parameters were discussed.

Linux OS based system was chosen for detailed analysis and the factors behind poor network receiving performance of Linux 2.4, 2.6 kernels and existing Linux based solutions like NAPI and PFRING were studied, ascertained and discussed. Several other possible relevant approaches and solutions were also considered and deliberated.

Based on the findings, a set of design and implementation strategies were proposed for a high performance packet capturing architecture. Utilizing these strategies, a high performance packet capturing architecture was designed and implemented for Redhat 8 and for RTAI 3.1 on Linux 2.4.24 with high availability, off the shelf components. The development work included modifying and porting the available Linux NIC driver and developing a user space driver component. This architecture avoided most of the identified problems that limited the performance of Linux, NAPI and PFRING. This architecture could receive packets at high arrival rates without any packet loss while consuming less CPU resources. The generic modifications to be carried on NIC drivers were delineated.

Performances of the proposed architecture, Linux, NAPI and PFRING were measured in experimental setups. Performance of the proposed architecture was compared with that of Linux, NAPI and PFRING against seven performance criteria. The test results demonstrated that the proposed architecture is superior in terms of lower CPU usage, no packet loss (higher throughput), lower packet delivery latency jitter and greater robustness. This study also indicated that a user space network processing mechanism can yield better performance under heavy network loads compared to existing kernel mechanisms, provided additional design efforts are undertaken.

The experimental work also profiled performances of NAPI and PFRING more exhaustively and elaborately than any other published literatures that could be found. Limitations of these two solutions were exposed. Many related works and solutions available to solve problems associated with packet capturing were surveyed and discussed. Some key implementation aspects with LXRT-RTAI were discussed which could ease development of other hard real-time data acquisition applications. Performance evaluation and measurement of network stack is a common practical problem. The evaluation and instrumentation methodology that was documented in this work could be used to measure performance of similar network stack.

References

All URLs are as of 1st June 2005.

[1] Davison, S., Hamilton, J., Intelligent alarm handling and analysis, in Stemming the Alarm Flood (Digest No: 1997/136), IEE Colloquium.17 Jun 1997, pp. 9/1-9/4.

[2] Brown, D.C., O'Donnell, M., Too much of a good thing?-alarm management experience in BP Oil. 1. Generic problems with DCS alarm systems., in Stemming the Alarm Flood (Digest No: 1997/136), IEE Colloquium., 17 Jun 1997, pp. 5/1-5/6.

[3] Jun, L., Khiang, W. L., Weng, K. H., Kay, C. T., Srinivasan, R., Tay, A., The intelligent alarm management system, in Software, IEEE, Mar/Apr 2003, pp. 66- 71, Vol. 20, Issue: 2.

[4] Hayes, P., Marshall, A., Real-time modeling of alarm generation and propagation in an SDH network, in Singapore ICCS '94. Conference Proceedings., 14-18 Nov 1994, pp. 182-186, vol.1.

[5] Dufey, J. P., Jost, B., Neufeld, N., Zuin, M., Event Building in an Intelligent Network Interface Card for the LHCb DAQ System, Nuclear Science, IEEE Transactions, Aug 2001 Volume: 48, Issue: 4, pp: 1323-1328.
http://lhcb-comp.web.cern.ch/lhcb-comp/daq/postscript/tns-00193-2000-final.pdf.

[6] R. Spencer, R. H. J., Mathews, A., O'Toole, S., Packet Loss in High Data Rate Internet Data Transfer for EVLBI. Proceedings of the 7th European VLBI Network Symposium, October 12th-15th 2004, Toledo, Spain.
www.hep.man.ac.uk/u/rich/VLBI/Spencertoledo.pdf

[7] Stallings, W., SNMP, SNMPv2, SNMPv3, and RMON 1 and 2, 3rd Ed., Addison-Wesley, 1999.

[8] Schaelicke, L., Slabach, T., Moore, B., Freeland, C., Characterizing the performance of network intrusion detection sensors. Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003).
http://www.cse.nd.edu/~spanids/papers/nids_perf_raid03.pdf

[9] Yoohwan, K., Wing C. L., Mooi C. C., Chao, J. H., PacketScore: Statistical-based Overload Control against Distributed Denial-of-Service Attacks.
http://www.princeton.edu/~rblee/ELE572Papers/Fall04Readings/packet_score.pdf.

[10] Bos, H., Bruijn, W., Cristea, M., Nguyen, T., Portokalidis, G., FFPF: Fairly Fast Packet Filters.
www.ist-scampi.org/publications/papers/bos-osdi2004.pdf .

[11] _____, TCP vs UDP.
http://syn.cs.pdx.edu/~jsnow/wireless_performance/tcp_udp.html

[12] Park, S., Ahn, S., Wooyong, S. H., Na, H. J., The reliable distributed network management platform, Distributed Computing Systems, 1995., Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of, 28-30 Aug 1995, pp. 439-445.

[13] Sterritt, R., Shapcott, C.M., Adamson, K. and Curran, E.P., High Speed Network First-Stage Alarm Correlator, International Conference Intelligent Systems and Control 2000, Hawaii, USA, pp391-397, 2000.

[14] Weiss, G.M, Predicting telecommunication equipment failures from sequences of network alarms, Handbook of Knowledge discovery and data mining, Oxford University Press. http://storm.cis.fordham.edu/~gweiss/papers/dmhandbook.pdf.

[15] Steinder, M., Sethi, A. S., The Present and Future of Event Correlation: A Need for End-to-end Service Fault Localization, Proc. SCI-2001, 5th World Multiconference on Systemics, Cybernetics, and Informatics, Orlando, FL (July 2001), pp. 124-129. http://www.cis.udel.edu/~sethi/papers/01/sci01.pdf

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

[17] Hasan, M., Sugla, B., Viswanathan, R., A conceptual framework for network management event correlation and filtering systems.
http://www.cs.toronto.edu/~zmhasan/ec-concep.pdf.

[18] Xue, F., Markovski, V., Trajkovic, l., Packet Loss in Video Transfers over IP Networks. www.ensc.sfu.ca/~ljilja/papers/iscas2001.ps

[19] Cui, W., Machiraju, S., Katz, R. H., Stoica, I., SCONE: A Tool to Estimate Shared Congestion Among Internet Paths.
http://sahara.cs.berkeley.edu/papers/CMKS04.pdf.

[20] ______, Local TCP / UDP Tests with the Loop-Back at SARA.
http://staff.science.uva.nl/~jblom/lambda/tcp_udp_tests/upgrade/mem_512mb/linux_2_4/sara_lb/

[21] Piotr, G., Linux Network Performance Studies for ATLAS HLT System, ATLAS T/DAQ Week, April 23, 2002.
http://piters.home.cern.ch/piters/TCP/seminar/TDAQ-April02/presentation.pdf

[22] Canalys, Linux small but growing fast in mobile device market, Aug. 3, 2004.
http://www.linuxdevices.com/articles/AT8941987592.html

[23] Singh, I., Embedded Linux-A Perspective.
http://www.techonline.com/community/ed_resource/feature_article/5440

[24] Galli, D., Loss- Free Datagram Transmission at Gigabit Rate.
http://www.bo.infn.it/lhcb/calcolo/galli31LHCbWeek.pdf.

[25] Bong, B., A Guide To Graphics Chipsets.
http://web.singnet.com.sg/~duane/b0000004.htm

[26] Adler, C., From: Clemens Adler <cadler@bnl.gov>
www.conservativecomputer.com/ myrinet/64bit/ds10.466.64bit.txt

[27] Lindahl, G., Myrinet Experiences Exchange Site: PCI Performance.
www. conservativecomputer.com

[28] rush@infimed.com, PCI DMA reads are too slow.
http://www.osronline.com/lists_archive/ntdev/thread1958.html

[29] Clark, D., Re: query about TCP header on tcp-ip. http://www.cs.clemson.edu/~westall/881/notes/vanj.93sep07.txt (now only cached copy available in Google).

[30] Hughes-Jones, R., PCI-X Activity and UDP measurements using the Intel 10 Gigabit Ethernet NIC. Dept. of Physics and Astronomy, University of Manchester, Oxford Rd., Manchester. http://dsd.lbl.gov/DIDC/PFLDnet2004/papers/Hughes-Jones.pdf

[31] Niessen, A. J., Linux in consumer products. http://www.bits-chips.nl/download/BC2002%20-%20Arnold%20Niessen.ppt.

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

[33] Brosky, S., Shielded CPUs: Real-Time Performance in Standard Linux, 2004. http://www.linuxjournal.com/node/6900/print,

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

[35] Stohr, J., Bulow, A. V., Farber, G., Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real-Time Software, 4th Intl Workshop On Worst-Case Execution Time (WCET) Analysis, June 2004. www.irisa.fr/manifestations/ 2004/wcet2004/Papers/Stohr.pdf

[36] Kay, J., Pasquale, J., The importance of non -data touching overheads in TCP/IP. In Proceedings of the ACM SIGCOMM Conference, pages 259-269, September 1993.

[37] Lauria, M., Pakin, S., Chien, A. A., Efficient Layering for High Speed Communication: Fast Message 2.x (1998). http://citeseer.ist.psu.edu/rd/92410033%2C518275%2C1%2C0.25%2CDownload/http%3AqSqqSqwww-csag.ucsd.eduqSqpapersqSqhpdc7-lauria.ps

[38] Wang, P., Liu, Z., Operating system support for high performance networking, a survey, Computer and Information Science Department, Indiana University, Purdue University. http://www.cs.iupui.edu/~zliu/doc/os_survey.pdf

[39] Deri, L., Improving passive packet capture: beyond device polling. http://luca.ntop.org/Ring.pdf.

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

[41] Kay, J., Pasquale, J., "Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000". Proceedings of the Winter 1993 USENIX Conference, pp. 249-258, January 1993.

[42] Skevik, K.A., Plageman, T., Goebel, V., Evaluation of a zero-copy protocol implementation. Department of Informatics, University of Oslo.

http://www.ifi.uio.no/english/research/groups/dmms/papers/33.pdf

[43] Thadani, M. N., Khalidi, Y.A., An efficient zero-copy I/O framework for Linux, Sun Microsystems Laboratories, Inc. http://www.smli.com/techrep/1995/abstract-39.html

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

[45] Chu, H. K. J., Zero-Copy TCP in Solaris (1996). http://citeseer.ist.psu.edu/cache/papers/cs/2392/http:zSzzSzwww.usenix.orgzSzpublicationszSzlibraryzSzproceedingszSzsd96zSzfull_paperszSzchu.pdf/chu96zerocopy.pdf

[46] Chase, J.S., Gallatin, A.J., Yocum, K.G., End-System Optimizations for High-Speed TCP. http://www.cs.duke.edu/ari/publications/end-system.pdf

[47] ______, RTnet Cross Reference, RTnet version 0.8.2. http://www.rts.uni-hannover.de/rtnet/lxr/source/Documentation/README.pools

[48] Chanteperdrix, G., Berlemont, A., Ragot, D., Kajfasz, P., Integration of real- time services in user-space Linux, 6th RTL Workshop. http://www.linuxdevices.com/files/rtlws-2004/DominiqueRagot-RTUser.pdf

[49] Cooperstein, J., Linux Kernel 2.6: New Features - III : Networking. http://www.axian.com/docs/linuxkerneltalk3.pdf

[50] Kuhn, B., The Linux "Real- time interrupt patch". January , 2004, http://linuxdevices.com/articles/AT6105045931.html.

[51] Kuhn, B., Interrupt priorisation in the Linux kernel , (AKA "Linux real time interrupts"). http://home.t-online.de/home/Bernhard_Kuhn/rtirq/20040304/rtirq.html.

[52] Yaghmour, K., Adaptive Domain Environment for Operating Systems, Technical report, OperSys.com, 2001. http://opersys.com/ftp/pub/Adeos/adeos.pdf

[53] Sarolahti, P., Real-Time Application Interface, Research Seminar on Real-Time Linux and Java, University of Helenski, Department of Computer Science, 2001. http://www.cs.helsinki.fi/u/sarolaht/papers/rtai.pdf

[54] Mantegazza, P., DIAPM RTAI Programming Guide 1.0. www.aero.polimi.it/~rtai/documentation/ reference/rtai_prog_guide.pdf

[55] Locke, C., D., Four Ways to Arm Linux for Real-Time Duty. http://www.rtcmagazine.com/home/article.php?id=100087

[56] Yodaiken, V., Barabanov, M., FSMLabs RTLinux PSDD: Hard Real- time with Memory FSMLabs RTLinux PSDD: Hard Real- time with Memory Protection. www.dedicated-systems.com/magazine/03q1/2003q1_2.pdf

[57] Vanderpool, B., Smith, Z., A Linux implementation of HIP, Project Report. http://web.demigod.org/~zak/documents/college/ece750-report/pdf.

[58] Abbott, M. B., Peterson, L. L, Increasing network throughput by integrating protocol layers, IEEE/ACM Transactions on Networking (TON) , Volume 1 , Issue 5 , pp. 600 - 610, October 1993.

[59] Salim, J. H. and Olsson, R., Beyond softnet, 5th Annual Linux Showcase & Conference, November, 2001. http://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf

[60] Sarma, D., McKenney, P. E., Issues with Selected Scalability Features of the 2.6 kernel, Proceedings of the Linux Symposium, July 21-24, 2004. www.linuxsymposium.org/proceedings/ reprints/Reprint-Sarma-OLS2004.pdf

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

[62] Intel Corporation, Small Packet Traffic Performance Optimization for 8255x and 8254x Ethernet Controllers Application Note (AP-453). ftp://download.intel.com/design/network/applnots/ap453.pdf

[63] Parker, M., Davis, A., Hsieh, W., Message-Passing for the 21st Century: Integrating User-Level Networks with SMT. School of Computing, University of Utah. http://www.cs.utah.edu/~ald/pubs/MTEAC.pdf

[64] Regehr, J., Duongsaa, U., Preventing Interrupt Overload, 2004. http://www.cs.utah.edu/~regehr/papers/lctes05/regehr-lctes05.pdf

[65] Chung, J. D., Brendan, C., Traw S., Smith, J. M., Event-signaling on the afterburner ATM Link Adapter. http://citeseer.ist.psu.edu/cache/papers/cs/17425/ftp:zSzzSzftp.cis.upenn.eduzSzpubzSztrpzSzhpcs95.pdf/event-signaling-on-the.pdf

[66] UserLevelDrivers - Gelato@UNSW WiKi. http://www.gelato.unsw.edu.au/IA64wiki/UserLevelDrivers

[67] Schleef, D., User-Space Driver Development KitReference Manual. www.schleef.org/usddk/usddk-docs.pdf

[68] Lentini, J., Pham, V., Sears, S., Smith, R., Implementation and Analysis of the User Direct Access Programming Library, 2nd Workshop on Novel Uses of System. Area Networks, SAN-2, February 2003.

[69] Leslie B., Heiser, G., Towards untrusted device drivers, Technical Report, UNSW-CSE-TR-0303, School of Computer Science and Engineering, March, 2003. ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0303.pdf

[70] Riddoch, D., Pope, S., A Low Overhead Application/Device-driver Interface for User-level Networking (2001). http://www-lce.eng.cam.ac.uk/lce-pub/public/djr23/pdpta2001_riddoch.pdf

[71] Druschel, P., Banga, G., Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. http://bbcr.uwaterloo.ca/~brecht/courses/856/readings-new/lrp-1996.pdf

[72] Shivam, P., Wychoff, P., Panda, D., EMP: Zero-copy: OS-bypass NIC-driven Gigabit Ethernet Message Passing. In Proc. 2001 International Conference on Supercomputing. www.osc.edu/~pw/emp/shivam-emp-sc01.pdf

[73] Bhoedjang, R., Verstoep, K., Bal, H., Rühl, T., Reducing data and control transfer overhead through network-interface support. In Proceedings of the 1st Myrinet User Group Conference (Lyon, France). 2000. http://www.cs.cornell.edu/raoul/papers/mug2000.pdf

[74] Dittia, Z., Integrated Hardware/Software Design of a High-Performance Network Interface, D.Sc. Thesis, Washington University Server Institute of Technology. http://www.arl.wustl.edu/Publications/2000-04/zDittia-2001.pdf

[75] RTnet Development Team, RTnet Documentation.
www.rts.uni-hannover.de/rtnet/doc.html

[76] Kiszka, J., Schwebel, R., The Alternative: RTnet.
http://www.rts.uni-hannover.de/rtnet/ad104705-en.html

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

[78] Kiszka, J., RTnet Application Programming Interface.
www.rts.uni-hannover.de/ rtnet/download/Dev04_API-Tutorial.pdf

[79] Rubini A, Corbet, J., Linux Device Drivers, 2nd Edition, O'reilly Publication.
Online edition : http://www.oreilly.com/catalog/linuxdrive2/

[80] Bovet, D. P., Cesati, M., Understanding the Linux kernel, 2nd Edition, 2003, O'reilly Publication.

[81] Shanley, T., Pentium Pro Processor System Architecture, 1997, Addison-Wesley Developers Press.