A FRAMEWORK FOR IMPROVING THE PERFORMANCE OF APPLICATION SERVERS IN NEXT GENERATION NETWORKS

by

SUMANTRA RAJ KUNDU

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2007

ACKNOWLEDGEMENTS

During my four years at the University of Texas at Arlington, I had the opportunity to interact and work with many people. Foremost I would like to thank my Ph.D. mentors, Dr. Sajal Das and Mr. Kalyan Basu. The constant suggestions and feedback I received from Dr. Das during the research and writing of this thesis greatly influenced the quality of the work. Most of the experiments in networking and operating systems would not have been possible without his excellent support and encouragement. Throughout my Ph.D. studies it has been an intellectual challenge to work with Mr. Kalyan Basu. Much of my interest and problem abstraction skills in performance analysis and queuing theory concepts I owe to him. His incisive technical comments, encouragement, and unflagging enthusiasm have been a constant driving force in many of the projects we have worked during the course of the Ph.D.

I would also like to thank my thesis committee members, Dr. Hao Che, Dr. Mohan Kumar, and Dr. Yonghe Liu for taking the time to learn about my work. During the summer of 2006 and aftermath I had the chance to work with Mr. Bill Stouder-Studenmund and Mr. Thor Lancet Simon of NetBSD foundation. Their clear thinking and guidance about I/O schedulers and the virtual memory (VM) helped me define the stochastic algorithms we have implemented inside the NetBSD OS. It has always been a pleasure to work with them. I am grateful for the assistance from all the members of our research group at Center for Research in Wireless Mobility and Networking (CReWMaN) lab, especially Sourav Pal.

I would like to thank my fiancee, Indrani, for always being there with me. Her patience and bearing has been astounding. Finally, I would like to say a big 'thank-you' to my parents, sisters, and all my relatives for giving me the courage to follow my dreams.

This work has been supported by the National Science Foundation (NSF) under grant IIS-0326505 and from Google Summer of Code, 2006.

July 18, 2007

ABSTRACT

A FRAMEWORK FOR IMPROVING THE PERFORMANCE OF APPLICATION SERVERS IN NEXT GENERATION NETWORKS

Publication No. _____

SUMANTRA RAJ KUNDU, Ph.D.

The University of Texas at Arlington, 2007

Supervising Professors: Sajal K. Das and Kalyan Basu

Next generation networks (NGNs) such as IP Multimedia Subsystem (IMS) are completely built on the Internet Protocol (IP) suite. This has made IP the de facto standard for data networking, voice over IP (VoIP), and media rich applications such as streaming multimedia, ringtones, multi-player gaming, and high-definition video conferencing for remote interaction. A primary feature of such converged networks is that they use the same IP-based network for simultaneously delivering voice, video, and data. Such services are provided on application servers built using industry standard Advanced Telecom Computing Architecture (ATCA) based blade computing units with various flavors of commodity open source operating systems like Linux, xBSD, and OpenSolaris.

However, real-time and latency sensitive applications such as streaming multimedia require that the entire network path of packet delivery from the originating server to the end host be properly and appropriately configured so as to avoid unnecessary delay and jitter in the data transfer mechanisms. With the ease of deployment comes the challenge

of delivering such rich multimedia applications in NGNs since there exists no separate paths for voice and data as present in existing circuit-switched public switched telephone network (PSTN). Packet delivery in such converged architectures involves interaction between the storage disks, operating system (OS), network interface cards (NICs), and the various switches and routers - each of which is independently capable of introducting delay in the data transfer mechanism.

In this dissertation, we focus on understanding and improving the performance of application servers present in high traffic content delivery networks (CDNs) and hoisting latency sensitive applications with heavy I/O requirements. We start by identifying an architectural framework for traffic characterization that is expected to provide insights about the composition and dynamics (e.g., average packet size and data rate, protocol composition) of network traffic present in CDNs. Once the nature and type of network traffic arriving at the NICs have been identified, we attempt to identify packet processing bottlenecks due to the interaction between the NICs, OS, and the underlying hardware. We propose a closed form queuing model that aims to understand the packet processing capabilities of the NICs based on the available computing resources. We have shown that there exist limits beyond which a computing unit cannot process packets without overloading the CPU. Since the performance of latency sensitive processes can be negatively impacted by delays of the storage network and by the dynamics of the OS, we present solutions for prioritizing the reader processes and tweaking the pagedaemon in open source OS. Based on our implementation in the NetBSD kernel, we have observed an approximate 15%-20% improvement in the transactions per second (TPS) of latency sensitive applications. Finally, we believe that our framework and approach of identifying the basic components in network data transfer mechanisms are for most part generic and can be used for performance tuning and deploying application servers in NGNs with a variety of different services.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT v

LIST OF FIGURES xi

LIST OF TABLES xv

Chapter

1. INTRODUCTION 1

    1.1 Motivation 3

    1.2 The Problem 4

    1.3 Overview of Our Novel Approach 7

    1.4 Contributions of The Dissertation 9

    1.5 Outline of the Dissertation 12

2. AN ARCHITECTURAL FRAMEWORK FOR ACCURATE CHARACTERIZATION OF NETWORK TRAFFIC 14

    2.1 Components of Traffic Characterization 15

        2.1.1 Existing Approaches for Traffic Characterization 16

        2.1.2 Salient Features of Our Approach to Traffic Characterization . . . 17

    2.2 Definitions 18

        2.2.1 A Motivating Example 20

    2.3 Traffic Characterization: Problem Statement 21

    2.4 Measurement Framework: Architecture, Algorithms, and Modeling . . . 22

        2.4.1 Flow Collection Unit (FCU) 24

        2.4.2 How to Distinguish SLF FlowID from LLF FlowID? 25

        2.4.3 Storing and Accessing FlowIDs of SLFs 26

2.4.4 Determining the Parameters of Flow Collection Unit (FCU) . . . 28
2.4.5 Flow Management Unit (FMU) . . . 34
2.5 An Online Framework for Identifying LLFs . . . 36
2.6 Offline Estimation Using Kernel Density Estimator . . . 39
2.6.1 Estimating the PDF of Sampled Data . . . 39
2.7 Experimental Results . . . 41
2.7.1 Identifying the LLFs . . . 42
2.7.2 Entropy Distribution: LLFs and SLFs . . . 44
2.7.3 Estimating the Volume of Original Traffic . . . 45
2.7.4 Estimating the Density Function of Underlying Traffic . . . 46
2.8 Summary . . . 47
3. AN ANALYTICAL MODEL OF POLLING DEVICE DRIVERS . . . 49
3.1 Packet Processing in Commodity OS . . . 50
3.1.1 Packet Processing in Polling Device Drivers . . . 51
3.1.2 NIC Device Driver Configurable Parameters . . . 54
3.2 Related Work . . . 54
3.3 Polling Device Drivers: Analytical Model . . . 56
3.4 Moments of Overflow Traffic . . . 58
3.4.1 Queuing Model of the I/O Bus . . . 60
3.5 Estimating the Service Time of PCI Bus (μ_{pci}) . . . 61
3.6 Dynamics of the Polling Process . . . 62
3.6.1 Equilibrium Equations . . . 63
3.6.2 Bulk Size Distribution . . . 64
3.6.3 Constant Bulk Size . . . 65
3.6.4 Varying Bulk Size . . . 67
3.6.5 Average Packet Service Time (μ_e) . . . 68

3.7 Performance Evaluation 69
3.7.1 Experimental Platform 69
3.7.2 Testing Methodology 70
3.8 Experimental Results 72
3.8.1 Average CPU Utilization and Average Number of Interrupts . . . 74
3.8.2 Average Bulk Size Distribution 79
3.9 Summary 81
4. CONTROLLING WRITE CONGESTION FOR IMPROVING APPLICATION READ PERFORMANCE 82
4.1 Motivation and Background 83
4.2 Approaches that Impact the Performance of I/O Workloads 88
4.3 I/O Scheduling Algorithms 90
4.4 Dynamics of Page Flushing Process 92
4.5 Proposed WICA Algorithms 95
4.5.1 Deterministic WICA: Rate based Approach 96
4.5.2 Estimating R_i^d for w_i 97
4.5.3 Duration of Learning Period 98
4.5.4 Congestion Control Approach 101
4.5.5 Limitations of D-WICA 102
4.5.6 Probabilistic WICA (P-WICA): Effective Bandwidth(EB) based Approach 103
4.5.7 Effective Bandwidth (EB) and Smoothing 103
4.5.8 Congestion Control Approach 106
4.5.9 Which WICA Algorithm to Choose? 106
4.6 Performance Evaluation 107
4.6.1 Workloads using Modified Postmark Macrobenchmark 109

4.6.2 One Reader and One Writer 110
4.6.3 One Reader and Many Writers 113
4.6.4 Situation Outside the Scope of WICA 114
4.7 Summary 116
5. CONCLUSIONS AND FUTURE WORK 118
5.1 Directions for Future Work 119
5.1.1 Identifying Heavy-hitters in Network Traffic 119
5.1.2 Extensible Framework for NICs 119
5.1.3 Establishing QoS Inside the VM 120
REFERENCES 121
BIOGRAPHICAL STATEMENT 130

LIST OF FIGURES

Figure Page

1.1 Conceptual architecture of the Next Generation Networks (NGNs) 2

1.2 A single frame snapshot of streaming multimedia quality under perfect network conditions and negligible load on the streaming server. 4

1.3 Snapshot of the same frame of Figure 1.2 with 0.09% packet loss and with negligible load on the streaming server 5

1.4 Snapshot of the same frame of Figure 1.2 under perfect network conditions but with non-negligible load on the streaming server 5

1.5 Primary components that play pivotal role in the packet delivery mechanism of a typical application server. The arrows indicate direction of data traversal 8

1.6 Suggested steps for monitoring and tuning the performance of application servers in new generation converged networks such as IMSThe 'plus' indicates 'either-or-both' operations 12

2.1 Structure of Internet Protocol Version 4 (IPv4) packet header. The fields used for computing the five-tuple is shown in bold 19

2.2 Architecture of FastFlow. The two FCU units (Active/Standby) have been designed to specifically capture short lived flows 23

2.3 Illustration of continuous scan clock (CCLK) and sample scan clock (SCLK) used in our measurement architecture 24

2.4 Architecture of the Flow Collection Unit (FCU) used in our measurement study 26

2.5 Linear scale plot of the FlowID density function for traffic collected from our internal LAN. Observe the heavy-tailed nature of FlowIDs 30

2.6 Offline classification of traffic streams. The pdf of the LLFs in sampled traffic stream is estimated using the non-parametric Parzen window technique . . 40

2.7 Framework for capturing flow packets in the internal LAN in our CReWMaN lab. Port mirroring is used to send traffic to a dual port GigabitIntel server adapter from which the packets are capturedusing Ethereal in promiscuous mode 42

2.8 Time series of the occurrence of LLFs as estimated using our algorithm versus as predicted using existing approach 43

2.9 Histogram of LLFs and the length of typical sequences in traffic traces for traffic collected from our internal lab network 44

2.10 Ratio of entropy between Long Lived Flows (LLFs) and Short Lived Flows (SLFs) 45

2.11 Temporal distribution of Entropy of Short Lived Flows (SLFs) 46

2.12 Log-log scale plot of the FlowID density function defined using five-tuple in our experiment 47

3.1 Description of network packet processing in polling device drivers 52

3.2 Basic queuing model of packet processing polling device drivers 56

3.3 Processor sharing model involing the receiver and transmit side PCI/PCI-X bus 57

3.4 State Space of the PCI bus where n_1 and n_3 refer to the size of the buffers Q_1 and Q_3 respectively 60

3.5 Timeline highlighting bulk removal of packets by the CPU from the receive descriptor ring during each invocation of the polling function 62

3.6 State Transition Diagram of Bulk Arrival at the CPU work queue during the polling process 63

3.7 CDF of bulk size Distribution for packet size = 64bytes, rxFIFO = 32MB, rxDescriptors = 1024 65

3.8 CDF of bulk size distribution for packet size = 512bytes, rxFIFO = 32MB, rxDescriptors = 1024 65

3.9 CDF of bulk Size Distribution for packet size = 1400bytes, rxFIFO = 32MB, rxDescriptors = 1024 66

3.10 Motherboard architecture used in our experiments. SmartBits 6000C using dual Terametrics card (LAN3327A) inconjunction with SmartFlow was used to generate and analyze traffic from the system-under-test 69

3.11 Average CPU Utilization with rxFIFO = 32MB, rxDescriptors = 1024 . . 73

3.12 Average CPU Utilization with rxFIFO = 32MB, rxDescriptors = 512 . . 74

3.13 verage CPU Utilization with rxFIFO = 32MB, rxDescriptors = 128 . . . 74

3.14 Average CPU Utilization with rxFIFO = 48MB, rxDescriptors = 1024 . . 75

3.15 Average CPU Utilization with rxFIFO = 48MB, rxDescriptors = 512 . . 75

3.16 Average CPU Utilization with rxFIFO = 48MB, rxDescriptors = 128 . . 76

3.17 Average Packet Latency with rxFIFO = 48MB, rxDescriptors = 1024 . . 76

3.18 Average Packet Latency with rxFIFO = 48MB, rxDescriptors = 512 . . . 77

3.19 Average Packet Latency with rxFIFO = 48MB, rxDescriptors = 128 . . . 77

3.20 Average Packet Latency with rxFIFO = 32MB, rxDescriptors = 1024 . . 78

3.21 Average Packet Latency with rxFIFO = 32MB, rxDescriptors = 512 . . . 78

3.22 Average Packet Latency with rxFIFO = 32MB, rxDescriptors = 128 . . . 79

3.23 Average number of Interrupts generated with rxFIFO = 32MB, rxDescriptors = 128 80

4.1 Example of decreased Transactions per second (TPS), 45% in the worst case, of a single reader process in the presence of a sole background writer process with varying transactional load.The results were obtained using a modified version of Postmark on a NetBSD 3.1 systemwith 512MB of RAM, 4KB page size, and read priority (RPRIO) as the I/O scheduling policy . 85

4.2 Temporal variations of the page pool inside the Virtual Memory (VM) during a typical run of the experiment of Figure 4.1.Notice the availability of free pages inside the VM during the lifetime of the experiment 86

4.3 High Level diagram showing the location of the Write Congestion Indication Algorithm (WICA) inside the Virtual Memory (VM).Notice that WICA traps and monitors the page flushing of the writer processes only 87

4.4 Stochastic “Burstiness" in the rate of generation of dirty pages for sampling bin size of 1000ms 92

4.5 Rate of decay of Auto Coorelation Function (ACF) for sampling bin size of 1000ms. Notice the high value of ACF 93

4.6 Presence of “Burstiness” in the rate of generation of dirty pages at reduced sampling bin size of 100ms 94

4.7 Rate of decay of Auto Coorelation Function (ACF) for sampling bin size of 100ms 95

4.8 Disappearance of “Burstiness” in the rate of generation of dirty pages for sampling bin size of 15ms 96

4.9 Rate of decay of Auto Coorelation Function (ACF) for bin sampling bin size of 15ms 97

4.10 No "Burstiness" in the rate of generation of dirty pages for sampling bin size of 5millisecs 98

4.11 Rate of decay of Auto Coorelation Function (ACF) for bin sampling bin size of 5ms 99

4.12 In deterministic rate based modeling technique (D-WICA), the rate of growth of dirty pages is assumed proportional to the number of dirty pages generated 100

4.13 Systematic sampling, with random start, in order to estimate the rate of generation of dirty pages by the writer processes. Note that such a sampling strategy provides accurate estimation of the rate only when the ACF is close to zero 100

4.14 On-Off fluid flow model for determining the effective bandwidth of each of the writer process 103

4.15 Evaluation technique with modified version of Postmark for creating reader and writer processes. In each of the experiments, the readers and the writers were independently created at random instants of time and the effective transactions per second (TPS) of the reader processes were recorded . . . 108

4.16 Postmark results for the Reader process. Observe the increased benefit of P-WICA over D-WICA with increase in the transactional load 110

4.17 Transaction time for the Writer process. Observe the sharp increase in the case of D-WICA algorithm 111

4.18 Postmark results for the Reader process in the presence of three writers . 112

4.19 Postmark results for the Reader process in the presence of 20 writers. This identifies failure condition in our experimental settings for the WICA algorithm 114

4.20 Transaction time for the Writer process experiencing the highest transaction time among competing writers, in the scenarion where WICA fails 115

LIST OF TABLES

Table Page

2.1 Notations Used in FastFlow Architecture 22

2.2 LLFs actually present in our lab network traffic trace vs. those estimated by length of the typical sequence 44

2.3 Total number of packets actually present in the traffic trace versus those estimated (N^{est}_{LLF}) by FastFlow 46

3.1 Typical configurable parameters available for tuning in off-the-shelf NICs and commodity OS 55

3.2 CPU Utilization: rxFIFO = 32KB, rxDescriptors = 1024 70

3.3 CPU Utilization: rxFIFO = 32KB, rxDescriptors = 512 71

3.4 CPU Utilization: rxFIFO = 32KB, rxDescriptors = 128 71

3.5 CPU Utilization: rxFIFO = 48KB, rxDescriptors = 1024 72

3.6 CPU Utilization: rxFIFO = 48KB, rxDescriptors = 512 72

3.7 CPU Utilization: rxFIFO = 32KB, rxDescriptors = 128 73

4.1 Notations Used in WICA 96

4.2 Configurable parameters for the Postmark macro benchmark used in our experimental evaluation 107

4.3 Characteristics of the system used in our experiments. 108

CHAPTER 1

INTRODUCTION

The value of present-day Internet is driven by the wide range of application services such as video conferencing, movie-on-demand, IP telephony, online gaming, peer-to-peer communication and numerous other services it is able to provide. Today, the emerging IP Multimedia Subsystem (IMS) architecture [88] is realizing this vision by using a complex array of networks, service applications, and content servers in the infrastructure network. At one end, we have the application servers residing in the application services and in content delivery networks (CDNs) acting as the data sources while at the other end, we have different end points such as cellular, commercial, and residential networks consuming the data. The Internet acts as the intermediary data delivery mechanism for all them and is composed of core and edge routers switching packets along appropriate network links. Figure 1.1 provides a high level architecture of such a layout.

With the evolution and growth of the Internet, it is now possible to control bandwidth allocation, define various packet queuing policies, tune packet dropping thresholds in the core and edge routers for protecting and prioritizing traffic flows belonging to different class of service (CoS) (e.g., via diffserv). All these help to reduce and control end-to-end packet loss, delay, jitter and allow for fine grained traffic aggregation and control across multiple time scales. But before setting the router parameters and tuning application servers in CDNs for optimal performance, it is important to know the composition and characteristics of the network traffic. For example, processing units in application servers and routers exposed to OC-48/192 networking links and carrying large number of control packets with average size of 128 bytes, experience considerable load

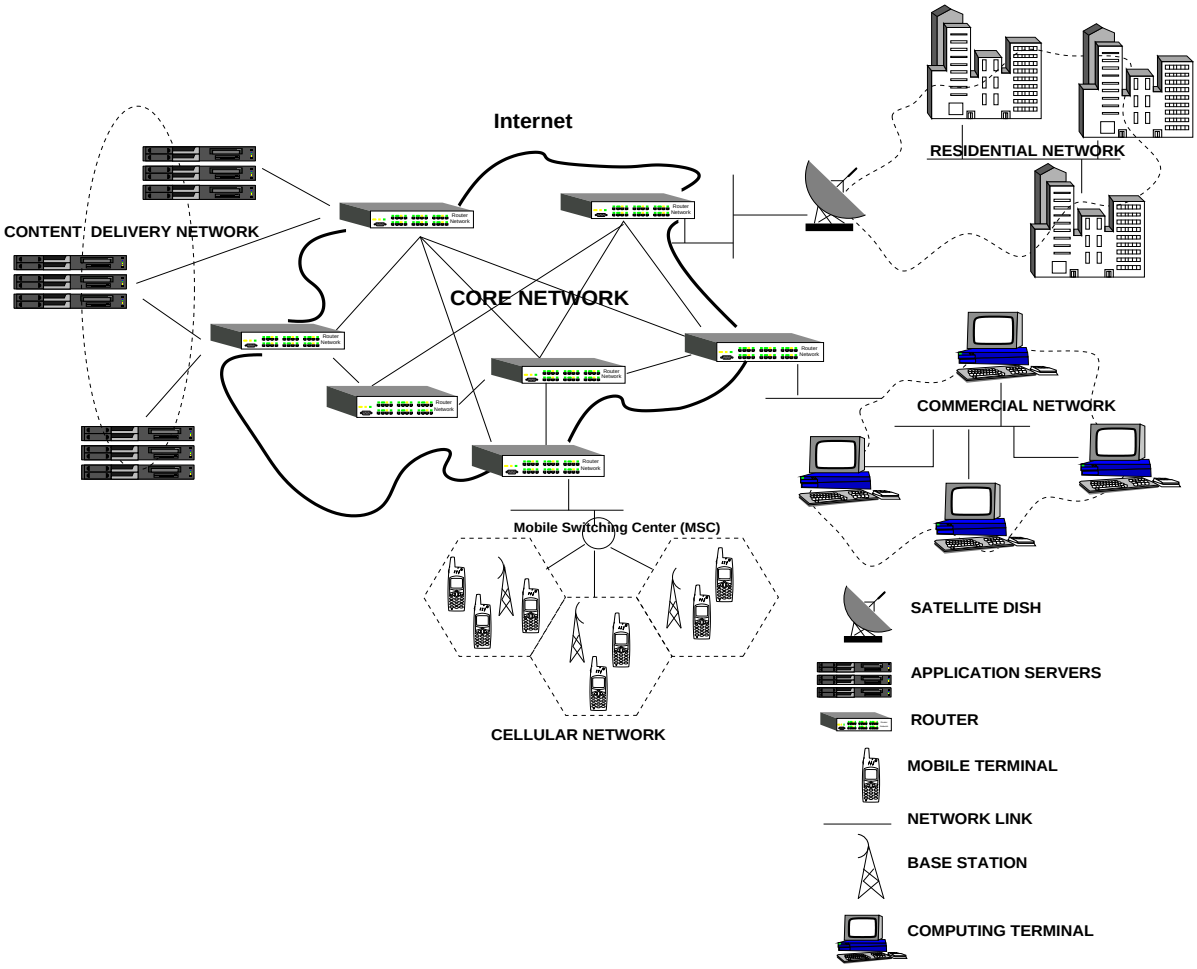

Figure 1.1. Conceptual architecture of the Next Generation Networks (NGNs).

on the CPU due to the microsecond granularity of the packet inter-arrival time. Thus, the packet inter-arrival time estimation is important in order to properly understand the allocation of computing resources. A barrage of such control packets has the potential to starve throughput of connections carrying application payload. Consequently, care must be taken to prevent such unwanted interaction of network packets.

At the same time, with the convergence of voice, video, and data packets in all IP-based networks, the volume of traffic is expected to grow considerably. The sheer volume of data that needs to be analyzed makes scalable traffic characterization a big challenge

even with the support from the most advanced hardware. Thus, this new converged architecture presents the research community with challenges along two major areas: (i) how to estimate the traffic characteristics of the network in OC-48/192 links and, (ii) how to guarantee the end-to-end latency for the delay sensitive multimedia streams.

1.1 Motivation

Real-time and latency sensitive applications such as streaming multimedia require that the *entire network path* of packet delivery from the originating server to the end host be properly and appropriately configured so as to avoid unnecessary delay and jitter in the data transfer mechanisms. For a pleasing end user experience, the ability to identify and control unwarranted latency in the data delivery mechanism is critical. For example, streaming multimedia require precise sequencing and synchronization of the video frames (I, B, P) for avoiding pixel loss in the video playout mechanism. Since it is well known that packet loss translates to media loss, reliable multimedia delivery dictates that the inter-arrival time gap between the network packets at the receiver network buffer should not be more than 150ms with less than 0.01% network packet loss.

Enforcing such strict end-to-end requirements over the Internet that is shared by different types of traffic is no easy task. And with the advent of new generation networks (NGNs), such as IMS, that are being increasingly built using the Internet Protocol (IP) suite, it is expected that the volume of traffic converging at the network elements (e.g., routers, switches) is going to grow further. Unlike existing specialized networks overlaid on a circuit-switched public switched telephone network (PSTN), such NGNs have no separate paths for voice and data services.

Thus, deploying and delivering rich multimedia applications in NGNs to the end hosts (wired and wireline) is going to be a challenge since packet delivery inherently involves interacting between the storage disks, operating system (OS), network interface

cards (NICs), and the various switches and routers; each of which is independently capable of introducting delay in the data transfer mechanism. To compound matters further, as multimedia transitions from standard to high-definition movie format, the demand for a higher data rate (greater than 2 Mbps) is expected to place more strict requirements on packet delivery and processing mechanisms. Consequently, tracing, identifying, and proposing a unified solution that aim to remove performance bottlenecks along the entire path of packet delivery is going to be a daunting if not an impossible task.

1.2 The Problem

Traditional design approaches have placed the burden of achieving the required end-to-end performance primarily on the network delivery mechanisms. Thus, a lot of effort has been directed in performance evaluation of routers [26], packet classification techniques [22], buffering mechanisms [1], network congestion detection and avoidance [29] schemes for controlling the uncertainties posed by the underlying network. The general expectation is that the application severs plugged on to the network is expected to meet the required performance figures.

Figure 1.2. A single frame snapshot of streaming multimedia quality under perfect network conditions and negligible load on the streaming server. Movie used from [96].

Figure 1.3. Snapshot of the same frame of Figure 1.2 with 0.09% packet loss and with negligible load on the streaming server.

Figure 1.4. Snapshot of the same frame of Figure 1.2 under perfect network conditions but with non-negligible load on the streaming server.

However, to drive a new era of system performance for applications operating in NGNs that demand strict end-to-end performance, it is not enough to consider the performance of the network alone in isolation. Instead the path of data delivery from the storage disks to the physical memory through the NIC cards become as important as the

network itself. To emphasize this point, in Figures 1.2, 1.3, 1.4, we provide snapshots of the same image captured from a video sequence encoded at 1 Mbps that is being streamed over a 1000 Mbps Gigabit Ethernet (GbE) network from a high performance streaming server. The streaming server has a database unit that stores the profile of all the registered users. Under immaculate conditions (no network impedance, no database updates), the overall delivered multimedia quality suffers no pixel loss and is considered to be perfect. This is shown in Figure 1.2. However, when the network[†] is experiencing congestion and randomly drops 0.09% of the multimedia packets, it affects the frame reconstruction capability of the media player. As a result, the remote viewer experiences unacceptable video quality. Similar degradation in the media quality is observed in Figure 1.4 when the streaming server experiences varying transactional I/O load from background database jobs due to multiple client access (update operations), thereby increasing the waiting time of the streaming application for the requested data block. Unfortunately, such variations in packet formation delay at the application level affects the audio-video synchronization of the remote media player. Consequently, there occurs pixel distortions and hence the viewing experience is far from perfect.

Although the above experiments were carried out in controlled laboratory settings that simulated the scenario of multiple client accesses, it proves the point that while the underlying network is an important transfer block in the path of packet delivery, it is equally important to consider the performance of the application server in the backdrop of the various OS jobs and the network in which it is expected to perform. All this means that the approach of designing and optimizing application servers that are sensitive to the data arrival process must also evolve to take into account both the network traffic profile along with the data retrieval mechanisms of the OS. At the same time, optimizing

[†]In our experiments we simulate varying network conditions using the click modular router [52].

the performance of both the server [†] and the network might not be feasible since most of the time the router related tuning parameters (queue type, bandwidth reservation, packet prioritization, etc.) are not available to the end hosts. The natural question that arises is: *How do we optimize the performance of latency sensitive application servers present in high traffic network domains with heavy I/O requirements?*

1.3 Overview of Our Novel Approach

In this dissertation we aim to answer the above question by looking into the interaction between the different components of the host OS and also trying to understand the profile of the underlying network traffic in which the latency sensitive application servers are expected to be placed. With the arrival of NGNs, these application servers are being increasingly built on open source Advanced Telecom Computing Architecture (ATCA) [89] based blade computing units with Linux or xBSD based OS. The infrastructure providers are using ATCA platform for their application server design such as IMS Call Session Control Function (CSCF) controllers and content providers are using it for building content servers in CDNs. Changes are also happening at the core network infrastructure where network planners are using designs based on network processing technology for high capacity core routers supporting OC-192 or OC-768 links. Apart from lowering the cost of ownership and deployment time period, such standardized architectures imply that the optimization and performance solution techniques proposed in this dissertation can be mostly be ported across different network and hardware profiles with little or no modifications. Thus, within the limits of practicability, our proposed technique can be considered to be generic.

To illustrate the concept, we consider the case of content delivery servers in our

[†]Note that our focus is exclusively on latency sensitive application servers plugged into high traffic domains as in the content delivery networks (CDNs). From now on, we use the term 'servers' to refer to such category of application servers.

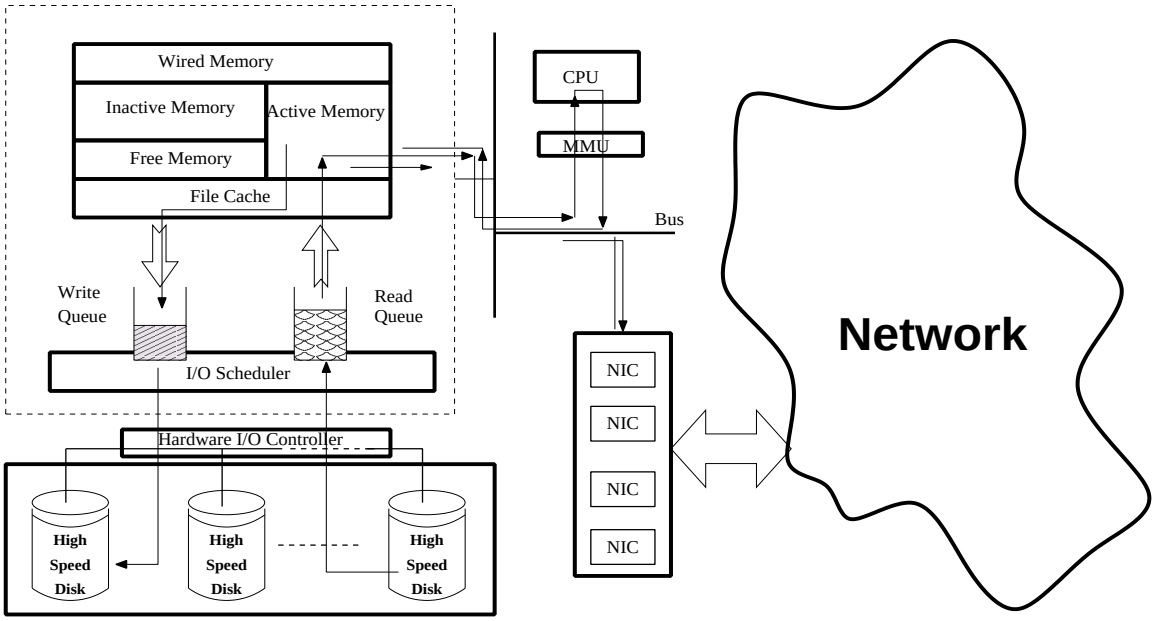

Figure 1.5. Primary components that play pivotal role in the packet delivery mechanism of a typical application server. The arrows indicate direction of data traversal.

work. The architecture and the design concept presented in our research is equally applicable to IMS CSCF servers or other application servers. Since complete path tracing and optimization is impossible, we separate the network delivery mechanism from the packet processing capability of content delivery servers. If we separate out the network delivery mechanism, the performance of such servers that send data over the network from storage disks is dependent on the following main components: (i) the performance of the I/O system that retrieves the data from the hard disks, (ii) virtual memory (VM) that allocates physical memory pages for the data to be mapped in, (iii) scheduling techniques of the CPU, and (iv) finally on the packet transmission mechanisms of the NIC from the OS buffers to the networking links. Such an architecture is shown in Figure 1.5 consisting of the network, the NIC cards, physical memory, and the I/O schedulers. In identifying such component parts, we have purposefully avoided going into the file system performance intricacies, CPU and I/O architectures, and various hardware and software

optimization techniques available in the literature. While such granular level optimization is expected to further improve the performance of the servers, it is outside the scope of this dissertation work. Instead, we aim to look at approaches on how most common OS kernels running over ATCA hardware with known network traffic profile can be designed to support latency sensitive applications such as streaming multimedia.

1.4 Contributions of The Dissertation

This dissertation is based on the assumption that application servers built using open source architectures such as ATCA and running commodity OS kernels like Linux, xBSD, or OpenSolaris can be performance tuned for serving latency sensitive applications by understanding the OS packet delivery mechanisms and the network environment in which they are expected to operate.

The major contributions of this dissertation are summarized as follows:

- **A New Framework for Network Traffic Characterization:** Understanding the nature and composition of network traffic interfacing with the servers is the first step towards planning and deploying large scale CDNs. An accurate diagnosis is achieved by carefully observing the interaction of traffic flows across various degrees of freedom like duration (time), size or protocol composition. There are two parts to the problem of traffic characterization: (i) *how* to measure the flows, and (ii) *infer* the overall traffic behavior and composition from the collected data. A lot of effort has been directed to the second problem of traffic inference and associated problems of flow dimensioning like short lived flows (SLFs) vs. long lived flows (LLFs), flow sampling (uniform, random, stratified), and on deriving statistical models (Pareto, Weibull, Poisson) that define the nature of traffic flows across the various degrees of freedom. Surprisingly though, the first problem of how to collect the flows has received limited attention.

In Chapter 2, we propose an alternate approach to traffic characterization by closely linking the flow measurement architecture with the estimation algorithm. Our measurement framework stores complete information related to SLFs while collecting partial information related to LLFs. For real-time separation of LLFs and SLFs, we propose a novel algorithm based on typical sequences from Information theory [13]. The probability density function (pdf) and the sample space of the underlying traffic is estimated using the non-parametric Parzen window technique [56] and likelihood function defined over the Coupon collector problem [21]. We validate the accuracy and performance of our estimation technique using traffic traces from the internal LAN in our laboratory and from the National Library for Applied Network Research (NLANR).

- **An Analytical Framework to Evaluate the Performance of Gigabit Network Interface Cards:** Application servers that are the focus of this dissertation, usually have multiple Gigabit Ethernet (GigE) NICs that involve considerable packet processing at high line speeds (greater than 300 Mbps average data rate). Since per packet interrupt processing is impractical at such high data rates, these NICs generally use device polling to transfer the packets to the OS. Network processing bottlenecks may arise due to complex interaction between the NICs, host OS, and the underlying hardware. These may be due to the inherent limitations present in the network adapter (e.g., limited on-chip buffer space), hardware architecture (e.g., I/O bus width), or simply due to improper allocation and configuration of the system resources. Considering the fact that an exhaustive experimentation involving all the parameters is not possible, in Chapter 3, we develop a closed form queuing model to investigate the impact of PCI/PCI-X I/O bus, on-chip packet buffers, receive and transmit descriptor rings of the OS, and processor service time on the dynamics of network packet processing. Our experiments indicate

that while device polling is an invaluable approach to prevent interrupt livelock at high line rates, it exhibits high average CPU utilization (greater than 90% for 128 bytes packet at 500Mbps) with increase in packet arrival rate. Furthermore, there occurs non-negligible costs in terms of interrupt generation when the system needlessly switches between polling and interrupt modes either due to: (i) overflow at the on-chip receive buffer, or (ii) due to non-availability of the PCI/PCI-X bus. We also show how results from our analytical framework can provide useful guidelines in evaluating the capability of such high performance networking systems.

- **Identifying and Controlling Write Congestion for Improving the Performance of Reader Processes:** *Write congestion* is a phenomenon when the effective transactions per second (TPS) of latency sensitive systems start decreasing in the presence of a large number of writer processes generating bursty workload patterns of disk access. We analyze the underlying behavior of such situations and propose two algorithms - deterministic (or rate based) and stochastic (or effective bandwidth (EB) based) - for improving the effective TPS of reader processes. For the rate based approach, we show by measurements how the presence of memory (Long-Range Dependence) in workload patterns can influence the decision of choosing the sampling type and the length of the sampling window. The stochastic approach is based on the established theory of EB in which we consider a set of fluid On-Off writers that independently generate dirty pages inside the virtual memory (VM). Each of the writers has an EB threshold that depends on the peak rate of generation of dirty pages and on the dynamics of an equivalent Poisson process. The rate based approach is suitable for lightly loaded systems in skewing the disk access towards the readers. However, in the presence of large variance in workload patterns, it unduly penalizes the writers. On the other hand, at the cost of increased complexity of implementation, the effective bandwidth based approach is suitable

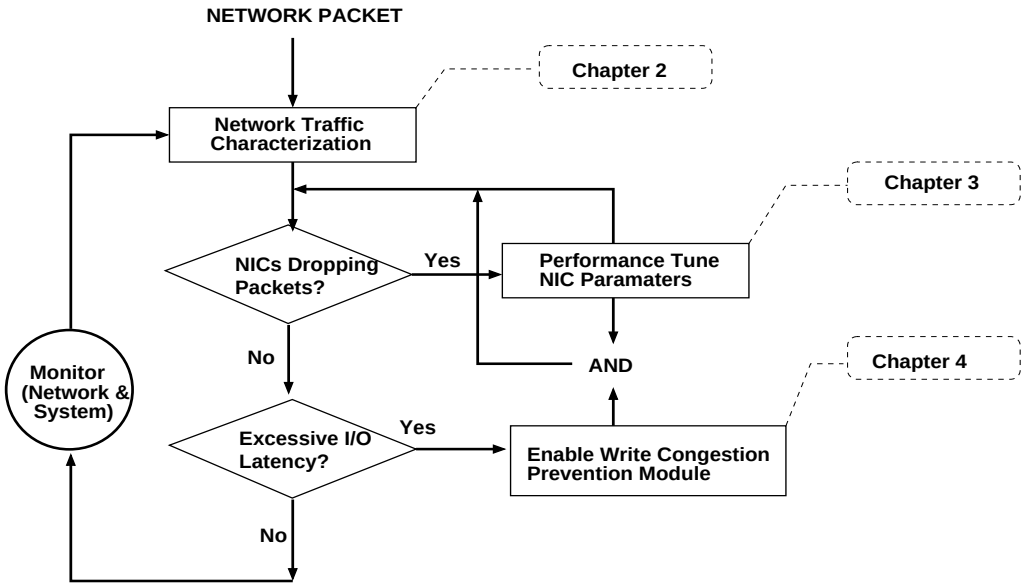

Figure 1.6. Suggested steps for monitoring and tuning the performance of application servers in new generation converged networks such as IMS.

for heavy duty servers. It is more resilient towards wide scale fluctuations of work load patterns and provides optimistic benefit in alleviating write congestion.

1.5 Outline of the Dissertation

Figure 1.6 describes a flow of actions that underline our approach for improving the performance of the application servers performance in the next generation converged networks such as CDNs. Before designing the CDNs, it is important to understand the dynamics of the traffic profile that the network is expected to handle. Keeping this in mind, Chapter 2 proposes an architecture for accurate characterization of network traffic. Next comes performance tuning of the NICs and OS parameters for preventing unwarranted packet loss along the receive path of data transmission. Considering the fact that an exhasutive experimentation involving all parameters is not possible, Chapter 3 proposes a queuing model and provides guidelines on how to set appropriate value of the parameters under various conditions. Next we focus our attention to reducing the

latency in the data retrieval mechanism inside the OS. In Chapters 4, we present two write congestion control algorithms (rate based and stochastic) and study their performances under various tansactional I/O loads. Finally, we conclude the dissertation in Chapter 5 with directions for future work.

CHAPTER 2

AN ARCHITECTURAL FRAMEWORK FOR ACCURATE CHARACTERIZATION OF NETWORK TRAFFIC

Understanding the dynamics of traffic crossing boundaries of network appliances (e.g., routers, switches, load balancers, firewalls, etc.) is an important step towards planning and deploying a large scale content delivery network (CDN). An accurate diagnosis is achieved by carefully observing the interaction of traffic flows (defined in Section 2.2) across various degrees of freedom like duration (time), size or protocol composition. The method of accurately estimating the nature of traffic flow along various degrees of freedom is referred to as the *traffic characterization*. Such characterization is essential for precise traffic engineering and is utilized for estimating network resource usage, bandwidth provisioning [18], differentiated services [8], traffic shaping [19]. For application servers inside the CDNs, such traffic characterization provides invaluable insights about the average packet size and data rate that the NICs and the server OS need to handle. For example, if the network is carrying lots of control packets with average packet size of 128 bytes or less, then it is advisable to map interrupt lines to specific processing units in multi-processing capable CPUs and also enable device polling with high bulk service rates. On the other hand, in CDNs with no such traffic profile enabling, such features will needlessly put load on the CPU. Thus, an accurate traffic characterization enables us to tune the performance of the resources of the server according to the needs and demands of the network traffic. The rest of the chapter is organized as follows. In Section 2.1 we define packet flows as they are related to network traffic. The challenges involved in characterizing network traffic at high line speeds (10Gbps or more) are exposed in Section 2.2. In Section2.4, we propose a flow measurement architecture that tackles the

problem of how to collect traffic flows at high line speeds. Data sampling and recovering of lost sample points for estimating the statistical distribution of the underlying traffic on data collected using our architectural approach is dealt with in Section 2.6. Validation of our approach is done in Section 2.7.

2.1 Components of Traffic Characterization

Traffic characterization involves two parts. The first part is paying attention to *how* to measure the flows and the second part is on *inferring* the overall traffic behavior and composition from the collected data. A lot of effort has been directed to the second problem of traffic inference and associated problems of flow dimensioning like short lived flows (SLFs) vs. long lived flows (LLFs), flow sampling (uniform, random, stratified), and on deriving statistical models (Pareto, Weibull, Poisson) that define the nature of traffic flows across the various degrees of freedom. Surprisingly though, the first problem of how to collect the flows has received limited attention. That being said, the recent work by Kumar et. al [43] [44] focused on using efficient data structures for flow characterization. Their work is a major step towards closely knitting the architecture and measurement technique together. However, the authors did not study the underlying architecture required for such a framework (except for identifying the need for multiple fast memory modules) and also did not take into account the idiosyncratic behaviour of heavy-tailed nature of the Internet flow. As we shall see, solutions to the two problems of collection and inference of network traffic are closely related; a measurement architecture specifically tuned for flow collection provides a strong basis for accurate traffic estimation.

If we consider the longevity of individual traffic flows, it is apparent that SLFs are the most difficult to capture since they die out fast (less than 1 second); but the individual size of such flows (i.e., the number of packets) is small (less than 100 packets). Also, they account for a substantial portion of the network traffic. In our experiments,

we have observed similar behaviour with the average lifetime of SLFs being less than a second (around 250 millisecs) with the combined traffic carried by them being around 40% of the entire traffic volume. On the other hand, LLFs exist in small numbers (around 3%), have longer lifetime (average duration of 30 minutes), and approximately account for 25% of the total traffic. From this observation, it is apparent that solutions that are unable to capture SLFs and take into account the ephemeral nature of such flows will have limited accuracy in estimating the composition of the underlying traffic. In sharp contrast, various statistical sampling techniques can be feasibly applied to LLFs since the traffic carried by them is substantial with each flow lasting for several minutes, hours, and even days.

Our proposed architectural framework for capturing and estimating the traffic distribution is based on the above observation. It is designed to *feasibly* store *complete* information related to SLFs with manageable complexity and hardware cost. At the same time, it uses systematic sampling to collect sufficiently large number of samples of LLFs; together which provides for accurate traffic flow characterization. In the next section, we briefly review existing approaches for traffic characterization.

2.1.1 Existing Approaches for Traffic Characterization

Precise knowledge and understanding of the properties and characteristics of the network traffic provide an important yardstick for accurate traffic engineering. An well-established mechanism for traffic characterization is to install a network tap at the point of measurement for a certain interval of time, collect packet samples, and infer flow characteristics from the collected data. Based on this philosophy but differing in the methods, two broadly different approaches currently exist for traffic flow characterization. They can be categorized as:

- *Traffic analysis based on different sampling methods:* Random sampling, simple random sampling, or stratified random sampling based traffic analysis has been studied in [10]. In [18], the authors studied the impact of sampling techniques on LLFs and suggested methods to infer properties of original traffic from sampled flow statistics. Correlated sampling strategy was proposed in [19] to account for the heavy-tailed distribution of flow lengths. Approaches that predict flow properties not available in sampled traffic volume have been studied in [20]. In [25], a theoretical study was undertaken that proved that it is possible to exactly infer the number of packets per flow which is not possible in traditional packet based sampling.
- *Characterization of traffic using a combined approach of efficient data structures and related statistical estimation techniques:* In [43] [44], the authors used a combination of bloom filters and maximum likelihood estimation techniques to predict the density function of the underlying traffic. Probabilistic data strutcures such as sketches have been used for identifying the heavy-hitters in large data streams [12]. Such approaches have been shown to yield better traffic characterization with definite bounds on estimation errors. Though novel, the approach of [43] and [44] suffer from architectural limitations related to the mismatch of speed between various memory hierarchies: Static Random Access Memory (SRAM), Dynamic Random Access Memory (DRAM), and mechanical disk storage unit.

2.1.2 Salient Features of Our Approach to Traffic Characterization

The primary limitations of the above approaches are the degree of accuracy of the results and scalability of the algorithms due to the large volume of datasets that need to be analyzed. It seems that a better solution could be achieved if it is first possible to identify and separate flows which carry most of the packets. An added benefit would be to

remove the storage limitations posed by the measurement architecture. Once such a goal is achieved, it would then be possible to store information for identifying scarcely occuring flows in the data stream with full accuracy while estimating the information for flows with a large presence. Our study is based on such a philosophy. It aims at linking the measurement architecture with the estimation algorithm such that information related to SLFs could be captured with complete accuracy while information related to LLFs can be predicted using non-parametric statistical techniques.

The salient features of our approach are summarized as follows:

- We propose an architecture framework called *FastFlow* that uses the principle of *typical sequences* [13] for separating the SLFs and LLFs. Experimental results indicate that typical sequences can identify LLFs with around 90% accuracy. Further, the low computational overhead of identifying typical sequences makes our framework suitable for real life implementation.
- We provide a fast SRAM based traffic flow update algorithm that avoids the complexity and overhead of traditional hash based solutions. It is based on binary content-addressable memory (BCAM) using addressing inversion for logic gating.
- We use non-parametric Parzen window technique [56] to estimate the the probability density function (pdf) of the underlying traffic.

2.2 Definitions

In this section, we define the terminology used in rest of the chapter. We define *flows* to refer to those packets with similar attributes. For example, a flow might consist of packets having identical values of five-tuple (source address, destination address, source port, destination port, protocol) as shown in Figure 2.1), or comprise of packets matching specific payload information (e.g., group of all TCP packets with payload containing the string "crewman"). Thus, flows can be characterized by packet headers,

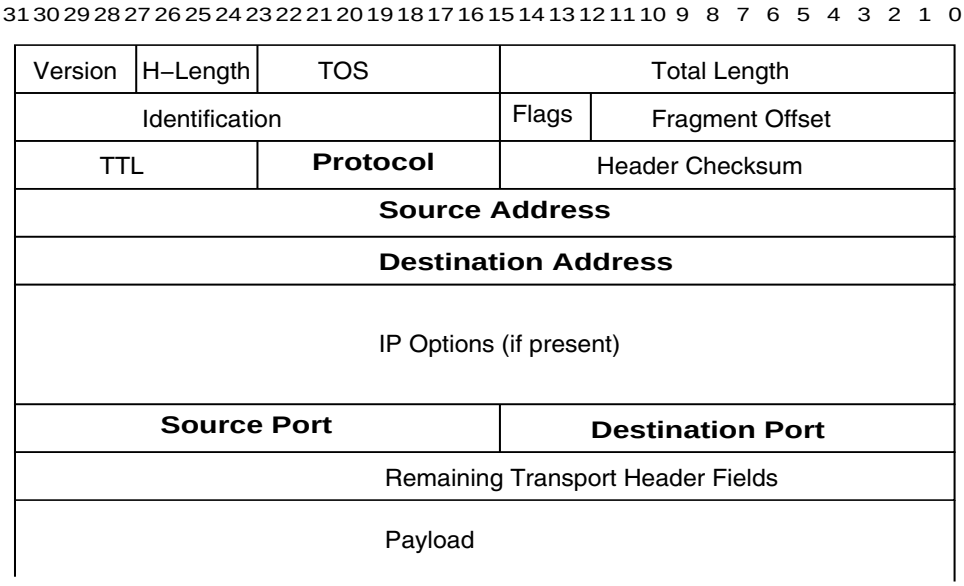

Figure 2.1. Structure of Internet Protocol Version 4 (IPv4) packet header. The fields used for computing the five-tuple is shown in bold.

payloads or a combination of both.

indent The *size* of a flow is the *number of packets* belonging to the flow, and the *duration* of a flow is its lifetime. For example, the size of a TCP flow is the number of packets exchanged till the last packet containing the FIN bit is sent (during normal termination) while the duration is the time interval between the first and last packets of the flow. In order to associate a packet with a flow, it is necessary to define a *Flow Identification Tag* (abbreviated as FlowID) using appropriate flow definition. For example, if flows are defined using the above five-tuple, then the FlowID is 104 bits long and can be used to separate packets belonging to different flows. Thus, the length of FlowID depends on how traffic flows have been defined. The next section provides a numerical case study that highlights the challenges related to the exhaustive flow collection. The performance figures quoted in the example are obtained by using the packet inter-arrival time of $\left\lceil \frac{R \times u}{P} \right\rceil$,

where R is the theoretical data rate, u is the link utilization factor, and P is the size of the packet in bytes.

2.2.1 A Motivating Example

To understand how traffic characterization is limited by resource constraints, consider the following example: On an OC-192 link (10 Gbps) with 80% link utilization and average packet size of 500 bytes, the average packet inter-arrival time is 500ns. Within the packet inter-arrival time, the hardware unit responsible for collecting flow statistics has to:

- Extract flow information by parsing fields of the packet.
- Compute the FlowID.
- Locate the FlowID in memory.
- Increment the counter corresponding to the FlowID.

In order to execute the above sequence of operations, the packet processing unit requires more than one memory access. Today's high capacity, high performance off-the-shelf static RAM (SRAM) has access time as low as 10ns with average size of 1-4 MB. Dynamic RAM (DRAM) have much higher densities (1 Gbits) but have equivalent access time of around 50ns or more [1]. If we use a hash table in SRAM, we need approximatly 160 bits [43] per hash table entry in order to store 32 bit wide counters that record the frequency of occurrence of flows in the underlying traffic. Thus, with 5MB SRAM, it is possible to store information corresponding to 0.25 million flows each 32 bits wide. Considering the fact that the number of flows in the Internet backbone links can reach 0.5 million [44] or more during the measurement interval, clearly a single SRAM module will not suffice. Also, in our illustrative example, we have assumed the ideal situation of one counter update per packet. In reality, the update overhead is considerably large since the useful clock cycles have to be expended in order to extract the flow information

from the packet, compute the FlowID and take into consideration the overhead of hash table update [44].

Thus, at 10 Gbps with 80% link utilization, the 5MB SRAM module will be filled up (assuming uniform hash table update) once every $5.3ms$; requiring that we move 0.25 million flows of 160 bits from SRAM to DRAM as fast as possible in order to prevent the SRAM overflow. At 50ns access time, the DRAM operation will take (assuming bus width of 64 bits) around $63ms$. Clearly, the low density, access speed mismatch, and high cost (4:1) [1] of SRAM compared to DRAM, make it practically infeasible for exhaustively capturing the information related to all the packets. Hence, a näive brute force approach of storing flow information is impractical (if not impossible) [20] as it would create memory hotspot problems, consume valuable processor cycles, and also might render the system unresponsive for prolonged periods of time. Added to these are the traditional issues associated with storing, mining, and analysis of large datasets.

2.3 Traffic Characterization: Problem Statement

Consider a definite flow measurement interval (T_1, T_2). It can be infinite if flow characterization is always on or can be a finite interval during which the algorithm is active. Let $[\mathcal{F}] = \{F_1, F_2, \ldots, F_i, \ldots, F_N\}$ be a sequence of N FlowIDs from $\{1, 2, \ldots i, \ldots N\}$, where each FlowID, F_i, is an index i.e., a number between 1 and N used to identify each flow in the underlying traffic. Denote $|F_i|$ to represent the number of packets belonging to the flow with FlowID F_i. It is important to note that the sequence $[\mathcal{F}]$ is sorted by increasing cardinality of the number of packets present in each FlowID. Our goal is to estimate N and the probability density function (pdf) of the network traffic from the available data sequence $[\mathcal{F}]$. As elucidated in Section 2.1, the accuracy of such a technique is primarily governed by the amount of information we are able to collect about the SLFs and LLFs. Under such circumstances, the flow characterization problem

Table 2.1. Notations Used in FastFlow Architecture

Notation	Meaning
N	Number of FlowIDs
$\mathcal{F}$	The sequence of FlowIDs
$\mathcal{F}^{'}$	The new sequence formed from $\mathcal{F}$ after clustering operation
(F_i)	Index used to identify a FlowID
K	size of the ingress buffer
A	Stochastic process denoting interarrival time of ingress buffer
B	Stochastic process denoting service time of ingress buffer
C_A	Coefficient of variation of interarrival time of ingress buffer
C_B	Coefficient of variation of service time of ingress buffer
ρ	Average system occupancy
W	Mean waiting time of FlowID in ingress buffer
γ, δ	Shape and Scale parameter of Pareto distribution
m	Width (in bits) of each FlowID
c	Number of bits associated with each FlowID counter
S	Size (in bits) of SRAM module
M	Total number of FlowIDs collected during a measurement interval
p	Probability of occurence of a LLF packet
CL	Confidence Limit required for FMU initialization
$H(X)$	Entropy of random variable X
$\psi(.)$	Gaussian kernel function
h	Width of the kernel density function

can be broken down into the following three sub-problems that serve as useful starting points for the overall solution: (i) architecture, algorithm, and performance modeling of the measurement architecture, (ii) algorithm for classifying LLFs and SLFs, and (iii) estimating techniques for N and the pdf of the network traffic from the available data set, $\mathcal{F}$. In the following sections, we provide solutions to each of these sub-problems. The notations used in our study are present in Table 2.1.

2.4 Measurement Framework: Architecture, Algorithms, and Modeling

We start with a high level description of the proposed architecture, called FastFlow, and analyze its component parts. This architecture shown in Figure 2.2 is envisioned to be a hardware module that plugs as a Smart Interface Card (SIC) into the network appliance. FastFlow is a self-contained hardware module with a Flow Identification Unit

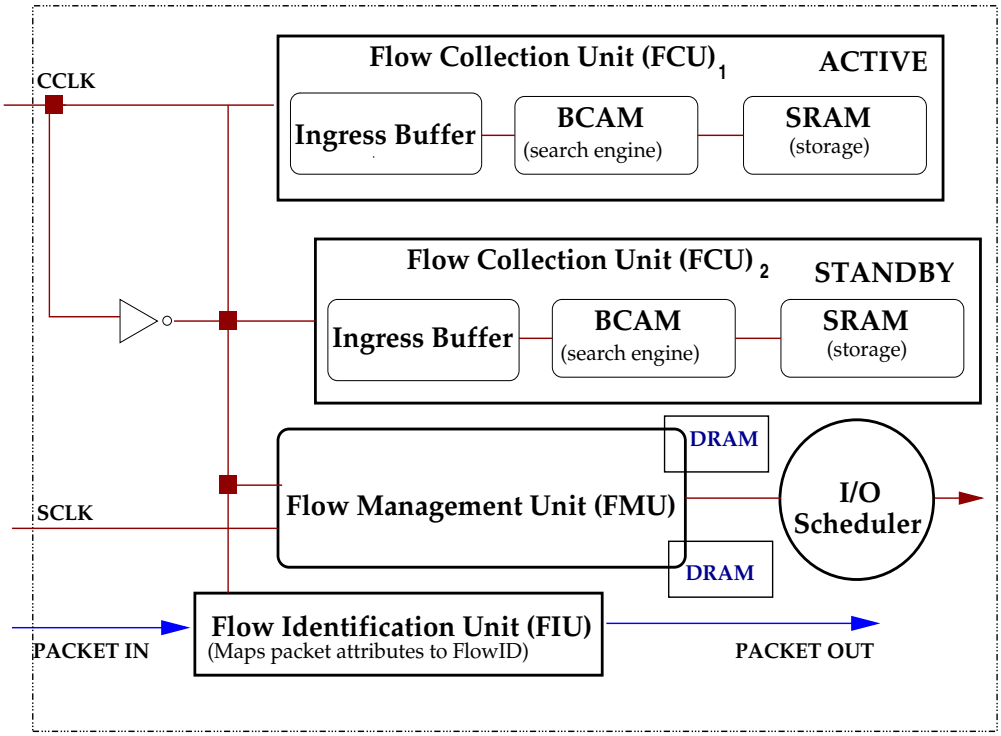

Figure 2.2. Architecture of FastFlow. The two FCU units (Active/Standby) have been designed to specifically capture short lived flows.

(FIU), Flow Management Unit (FMU), and two Flow Collection Units (FCUs). The FIU is responsible for extracting and mapping the attributes of a packet to the corresponding FlowID. When the SIC is active, packets flow through the FIU and are immediately placed in the original line of flow after the information relevant to the calculation of the FlowID has been extracted from the packet. This allows the measurement unit to work transparently from the rest of the appliance. Fundamental to the design is the assumption that a unified measurement approach for collecting information for both SLFs and LLFs is not technically very efficient (see Section 2.1).

Hence, instead of a single packet sampling approach where the sampling frequency is independent of the flow type, we use dual packet scanning clocks. The clocks are derived from the same reference clock and are assumed to be behave ideally (i.e., no

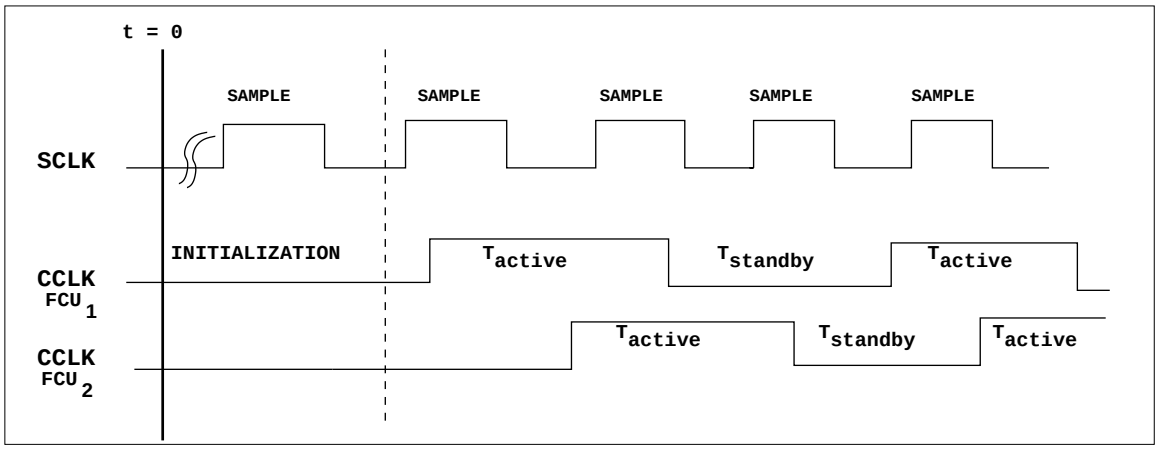

Figure 2.3. Illustration of continuous scan clock (CCLK) and sample scan clock (SCLK) used in our measurement architecture.

skew, delay, jitter). As shown in Figure 2.3, we refer to the clocks as continuous scan clock (CCLK) and sample scan clock (SCLK).

It is important to note that in Figure 2.3, the clock rates of CCLK and SCLK are set to different frequencies although they are derived from a common timebase. This is due to the nature of the data collection operation they are supposed to conduct. While the CCLK is aimed at capturing all the SLFs in Active/Standby mode, the SCLK collects samples of LLFs using variations of systematic sampling. The dual FCU makes it possible to capture information related to the SLFs with complete accuracy and within feasible system resource constraints. We now describe each of the individual units in more detail.

2.4.1 Flow Collection Unit (FCU)

Figure 2.4 provides the architectural description of the FCU responsible for collecting only the SLF FlowIDs from the tapped traffic. It consists of a no-bus-latency (NoBL) SRAM [81] memory module configured as ingress buffer, two BCAM based search and index engines, and a high speed SRAM storage unit. The FCU works in pairs and operates in the Active/Standby mode. Only one of the units is active at any point of time and the

operation of the units is synchronized by the CCLK signal (see Figure 2.3). When the active unit is collecting SLF FlowIDs, the standby unit is emptying its SRAM content to the FMU and viceversa. This alternating sequence of behavior ensures that no SLF FlowID is lost due to overflow of the SRAM storage unit.

During system initialization phase, the first FCU (denoted as FCU_1 in Figure 2.2) is Active by default and the second FCU (i.e., FCU_2 in Figure 2.2) is in Standby mode. The Active FCU works as follows: FlowIDs computed by the FIU is received by the FCU and are placed on the ingress FIFO queue. As the ingress queue is serviced and FlowIDs are emptied, we need to (i) identify the FlowID that belongs to an SLF since we store only SLFs in FCU; and (ii) decide on how to access the counter corresponding to the SLF FlowID in storage SRAM with minimal delay and processing overhead. These issues are addressed in the next section.

2.4.2 How to Distinguish SLF FlowID from LLF FlowID?

The decision as to which FlowID to store is done by configuring a BCAM as a search engine, referred to as the search BCAM (or SCAM), over a list of compiled LLF FlowIDs. This list is computed by FMU in real time and downloaded to the FCU when the unit is in Standby mode (how the LLF FlowID list is compiled online is explained in Section 2.4.5.2). The BCAM allows entries of binary 0 and 1, and content searches are of *fixed lengths* only. Furthermore, the size and width of the BCAMs are configurable [83]. All these make the BCAM attractive for our design since searches over FlowIDs are of fixed lengths.

The FlowID retreived from the ingress buffer is used as a key to the SCAM such that a classificiation decision can be made in a single table lookup. Each SCAM table entry contains the LLF FlowID, output address, and a valid bit (see Figure 2.4). The value of the valid bit is set to 0 when the corresponding table entry is empty. This helps

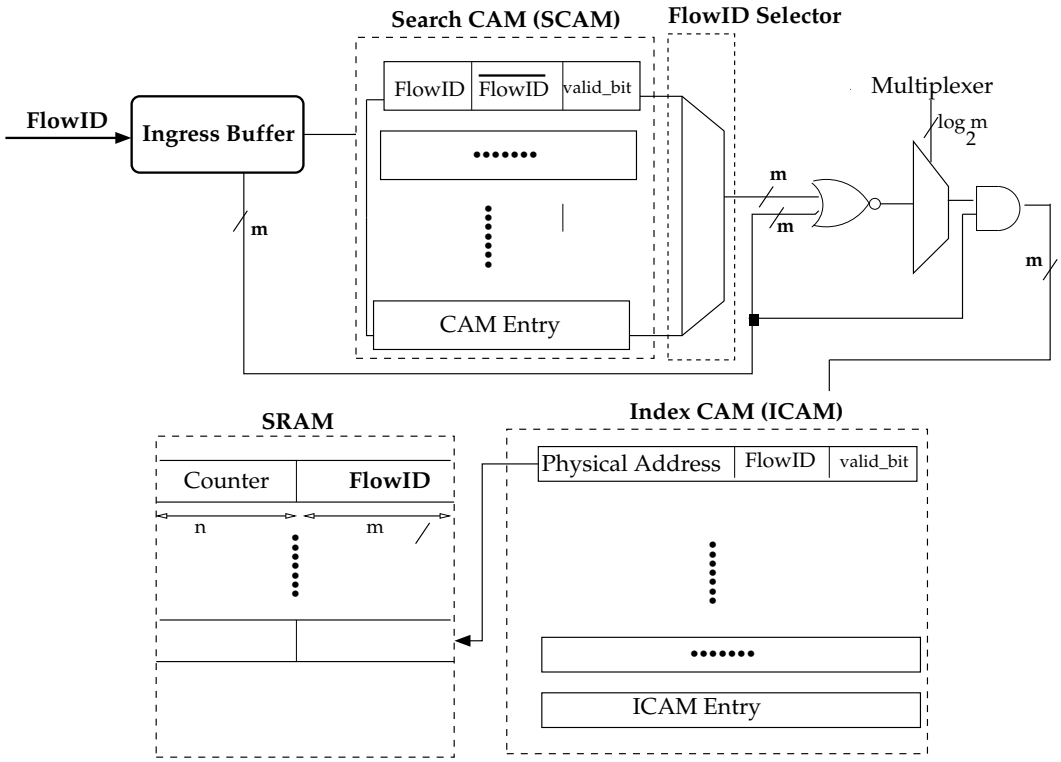

Figure 2.4. Architecture of the Flow Collection Unit (FCU) used in our measurement study.

us reduce power consumption by avoiding unnecessary searches over empty table lists. A search is successful if a SCAM entry matches the input FlowID. If a match occurs, then the current FlowID belongs to a LLF and is *discarded*. Otherwise, the FlowID is stored in SRAM and the counter corresponding to the FlowID is incremented by one. In the next section, we describe our algorithm for storing and updating the FlowID information inside the active FCU.

2.4.3 Storing and Accessing FlowIDs of SLFs

The SRAM storage unit records the frequency of occurence of each SLF FlowID when the FCU is operating in the Active mode. This is done by associating a counter

(of definite length) with each FlowID. The algorithmic approach to store FlowIDs is to pick a suitable hashing function that allows deterministic insertion and update operation of the SRAM. However, hash functions present additional computational and memory overhead, thus giving rise to collisions. An approach allowing hash collisions and later *estimating* the consequent counter loss has recently been proposed in [44]. Since we aim to measure the frequency of SLF FlowIDs with *complete accuracy*, such a scheme will not suit our purpose. Instead, we present an architectural framework that is practical, avoids hashing (and its associated complexity), and achieves *constant insertion time*.

Refering to Figure 2.4, each table entry of SCAM contains three fields: the LLF FlowID, the logical *inversion* of the LLF FlowID, and the valid bit. Let us denote these fields by `FlowID`, $\overline{\texttt{FlowID}}$ and `validbit`. Thus, each entry of the SCAM is a tuple of size three. The algorithm for updating and inserting new FlowIDs in SRAM works as follows. During the SCAM search operation, all the entries are compared in parallel with the target FlowID. If the FlowID matches a table entry, then the corresponding $\overline{\texttt{FlowID}}$ is returned. Otherwise, the *first* $\overline{\texttt{FlowID}}$ present in SCAM is returned. The $\overline{\texttt{FlowID}}$ obtained is then propagated to a NOR gate along with the original FlowID. If we receive an LLF FlowID which is present in SCAM, the storage circuitry connected to the output of the NOR gate is disabled and the FlowID discarded. Otherwise, the output of the NOR gate is connected to a multiplexer (see Figure 2.4) so as to enable the SRAM storage circuitry. The Index BCAM (or ICAM) is utilized for mapping the FlowID to the physical address of the SRAM. This is because the number of bits of the FlowID and the SRAM physical address could be different. Finally, the counter corresponding to the physical address is incremented by one.

It is important to note that the ICAM is accessed only when the matching fails (i.e., the FlowID does not belong to an LLF). If the FlowID is not present in the ICAM, then the ICAM is subsequenly updated (for example, see [76] for TCAM update solutions

Algorithm 2.1 Update: Algorithm for Inserting/Updating FlowID in SRAM

```
1: Retrieve FlowID from ingress buffer shown in Figure 2.4.
2: found == search_SCAM (FlowID)
3: if found === true then
4:    return
5: else
6:    memory_address == search_ICAM (FlowID)
7: end if
8: if memory_address ≠0 then
9:    *(memory_address)++
10: else
11:   update_ICAM (new_address, FlowID)
12:   *(new_address) = 1
13: end if
```

without complete locking) and the next available physical memory block is allocated † for this FlowID. Using our approach, we do not need to bother about apriori allocation of space in SRAM and hence are able to avoid the problem of skewed memory usage associated with a hash based storage solution. Also, the insertion and update operations can be done in constant time. These sequence of operations are denoted as `search_ICAM()` and `update_ICAM()` in Algorithm **Update** 2.1.

2.4.4 Determining the Parameters of Flow Collection Unit (FCU)

In this section we derive mathematical results for determing the size of the ingress buffer, the size of SRAM, and the frequency of CCLK.

2.4.4.1 Size of Ingress Buffer

We model the ingress buffer as a G/G/1 queue employing FIFO scheduling policy. Referring to Table 2.1, let $\{A_i : i \geq 1\}$ and $\{B_i : i \geq 0\}$ denote the inter-arrival intervals

†If memory is organized in contiguous blocks, then the next physical address is the current address + length of an SRAM entry, assuming proper memory alignment. The value of the current physical memory address is stored in a register (not shown in Figure 3).

and the service times of the G/G/1 queue, respectively. The queue is empty at the beginning of the Active mode. Let W denote the mean waiting time of a FlowID in the queue. Let $E(W \geq t)$ denote the average waiting time in the queue under steady state conditions. Then, using the approximation given in [66], we have:

$$E(W) \approx \frac{\rho}{1-\rho}.\frac{(1+C_B^2)((2-\rho)C_A^2+\rho C_B^2)}{2(2-\rho+\rho C_B^2)}.E(B) \tag{2.1}$$

where C_A and C_B respectively denote the coefficient of variation (CoV) of interarrival times and service times of the processes A and B, and ρ is the average occupancy of the system. In our case, the service time is constant since using SCAM and ICAM, the SRAM update time is $O(1)$. Thus, $C_B = 0$ and let $E(B) = K$ be a constant. Then, Equation (2.1) can be simplified as

$$E(W) \approx \frac{\rho}{1-\rho}.\frac{C_A^2}{2}.K \tag{2.2}$$

Now, the average system occupancy can be defined as:

$$\rho = E(B)/E(A) = K/E(A) \tag{2.3}$$

Using Equation (2.3), we can simplify Equation (2.4) as:

$$E(W) \approx \frac{\sigma_A^2}{2\left[E(A)\right]^2\left[E(A)-K\right]}.K^2 \tag{2.4}$$

where σ_A^2 is the variance of the interarrival process A.

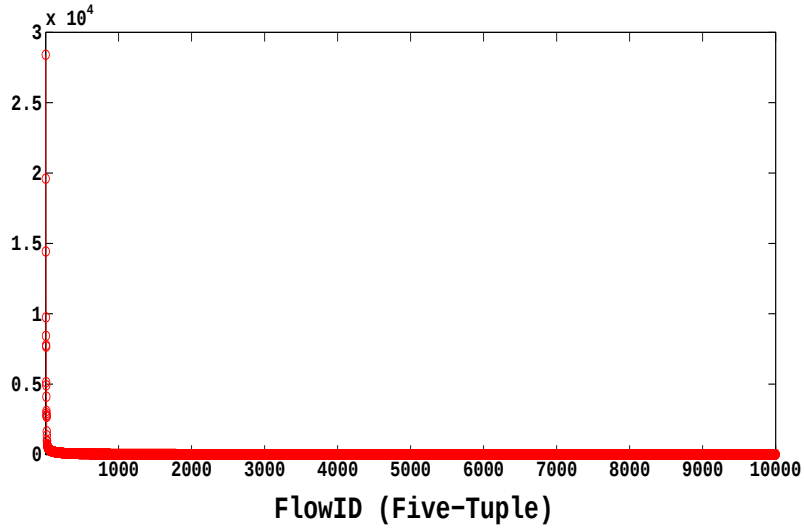

Figure 2.5. Linear scale plot of the FlowID density function for traffic collected from our internal LAN. Observe the heavy-tailed nature of FlowIDs.

2.4.4.2 Pareto FlowID Arrival

In Figure 2.5, we plot the distribution of FlowIDs captured from the internal LAN in our laboratory. Approximately 1 million packets were captured over a duration of two weeks. We observer that the distribution of FlowIDs clearly follows a heavy-tailed distribution. In other words, a few FlowIDs (around 1-2% in our experiment) carry around 0.85 million packets while a substantial portion (around 40%) of FlowIDs have only one packet. Such a distribution is extreme and on a linear scale the curve is almost L shaped. Assuming Pareto distribution, the cumulative distribution function (cdf) of FlowID is given by:

$$F(x) = 1 - \left(\frac{\delta}{x}\right)^{\gamma}, \; x \geq \gamma \tag{2.5}$$

In Equation (2.5), γ is the shape factor and δ is the scale parameter. The mean $E(\Omega)$, and variance σ_{Ω}^2, of the Pareto distribution are given by:

$$E(\Omega) = \left[\delta\gamma/(\gamma-1)\right], \text{ for } \gamma > 1 \tag{2.6}$$

$$\sigma_{\Omega}^2 = \left[\delta^2\gamma/\left[(\gamma-1)^2(\gamma-2)\right]\right] \ \ \gamma > 2 \tag{2.7}$$

Using Equations (2.4), (2.6), and (2.7), we can find a general expression for $E(W)$, the average waiting time of FlowIDs in the system. Let L denote the average number of FlowIDs in the FCU, and λ denote the FlowID arrival rate. Then, by Little's theorem:

$$L = \lambda E(W) \tag{2.8}$$

Let η be the maximum number of FlowIDs that can be accomodated in the FCU. Therefore,

$$L \leq \eta \tag{2.9}$$

$$\Rightarrow \frac{K^2(\gamma-1)\lambda}{2\gamma^2(\gamma-2)[\delta\gamma - K(\gamma-1)]} \leq \eta \tag{2.10}$$

Rearranging Equation (2.10), we obtain:

$$\frac{\lambda}{\eta} \leq \frac{2\gamma(\gamma-2))\left[\delta\gamma - K(\gamma-1)\right]}{K^2(\gamma-1)\lambda} \tag{2.11}$$

Equation (2.11) provides the relationship between the FlowID arrival rate (λ) and the maximum average number of packets (η) present in the FCU. Using Maximum Likelihood function for the heavy-tailed distribution under consideration, it is possible to estimate

the parameters γ and δ. For example, in Figure (2.5), they are estimated as $\gamma = 2.1$ and $\delta = 800$. On substituting these values in Equation (2.11), we obtain:

$$\frac{\lambda}{\eta} \leq \frac{1347 - 0.8822K}{K^2\lambda} \Rightarrow \eta \geq (7.42 \times 10^{-4})K^2\lambda^2 \tag{2.12}$$

Equation (2.12) estimates the size of the ingress buffer that needs to be provisioned for a definite line rate and average system service time. For example, if the line rate is 1 Gbps and the average service time is 1 ms, then the buffer should be able to store 100 FlowIDs on an average, in order to prevent overflow. Thus, considering different values of η and δ that depend on the traffic pattern, it is possible to estimate the size of the ingress buffer.

2.4.4.3 How Long is FCU in Active Mode?

Denote T_{active} as the time for which the FCU remains in the Active mode. Since SCAM and ICAM have constant access time, under stable conditions, T_{active} *will be dominated by the average time it takes to fill the SRAM, under worst case scenario.* Let the width of each FlowID be m bits and let c be the number of bits associated with each FlowID counter. Then, the width of each SRAM entry is $(m + c)$ bits. Let S (in bits) be the size of the SRAM module (see Figure 2.4). Then, the number of unique FlowID counters that could be stored is $[S/(m + c)]$ and the maximum value each counter can attain is 2^c.

Let $T_{overflow}$ be the minimum time required for the SRAM to overflow. Denote α to be the probability that during this time the total number unique of FlowIDs observed is less than the maximum number of SRAM entries and for any particular FlowID, the counter value is less than 2^c. Strictly speaking, though the arrival process of the FlowIDs in the FCU is heavy-tailed in nature, recent research [9] conducted over large volumes of the Internet traffic observed the Poisson nature of the packet inter-arrival time. Further-

more, convoluting a Pareto disribution is mathematically intractable. Consequently, we model the FlowID inter-arrival time by an exponential distribution in order to determine the value of T_{active}.

Let the inter-arrival time of FlowIDs be independent and follow exponential distribution. We assume the worst case scenario where all the FlowIDs received during T_{fcu} are unique. Let $Y_1(t), Y_2(t), \ldots, Y_k(t)$ be a sequence of independent, exponential random variables with parameter λ and probability density function:

$$y(t) = \lambda e^{-\lambda t},\ t \geq 0 \tag{2.13}$$

Here we have resonably assumed that the arrivals of FlowIDs is the same as the packet arrival rate, λ. The value of k is equal to $[S/(m+c)]$ which is the maximum number of SRAM entries. We are interested in the distribution of the sum of the random variables, $Y_i(t)$. Using the convolution theorem, we have:

$$\begin{aligned} P\left(\sum_{i=1}^{k} Y_i(t) < f\right) &\Rightarrow (\underbrace{y * y * y \ldots}_{k\ terms}) \\ \Rightarrow (y * y_{k-1}) f(t) &= \frac{\lambda^k t^{k-1}}{(k-1)!} e^{-\lambda t} \equiv y_k(t) \end{aligned} \tag{2.14}$$

which is the Erlang distribution with density function $y_k(t)$. Thus,

$$P\left(\sum_{i=1}^{k} Y_i(t) < f\right) = \frac{\lambda^k t^{k-1}}{(k-1)!} e^{-\lambda t} \tag{2.15}$$

Hence, the probability (α) that within T units of time, the SRAM will not overflow is given by:

$$\int_0^T y_k(t) \leq \alpha \tag{2.16}$$

Using Integration by parts, we obtain the following relationship:

$$\lambda^k e^{-\lambda T} \sum_{i=0}^{k-1} \frac{T^i}{k!\lambda^{k-1}} \geq (1-\alpha) \tag{2.17}$$

The value of T_{active} obtained from Equation (2.17) is a *lower bound* on the FCU switching time for a given confidence level, α. A Monte-Carlo approach can be used to arrive at an approximate solution of T_{active}. Thus,

$$T_{fcu} \approx T_{active}$$

based on the reasons identified at the beginning of this section.

2.4.5 Flow Management Unit (FMU)

The FMU uses external DRAM (see Figure 2.2) to store information associated with the individual FlowIDs so as to achieve massive storage capacity. In this section, we derive bounds on the FMU initialization time, describe the algorithm used to separate the SLF and LLF in real-time, and calculate the duration for which the FCU remains in Standby mode.

2.4.5.1 FMU Initialization Time from Statistical Confidence Limit

At the beginning of the measurement interval, the FMU must create a list of LLFs that will be used to intilize the ICAM of the FCU in Standby mode. This initial list is created using the concept of typical sequences in information theory and described in Section 2.4.5.2. However, the accuracy of the *initial* list of LLFs depends on the *duration* of the initialization phase. The longer the initialization phase, the better is the confidence limit (CL), but at the cost of longer run time. Hence, given a user specified CL, our

aim is to determine the number of samples that needs to be collected. Let M be the total number of FlowIDs collected during the initialization period. Assume the samples be independent and let τ denote the probability of occurence of an LLF packet. Then the probability of collecting m LLFs out of M FlowIDs can be modeled as a binomial distribution with $P_M(m)$ i.e

$$P_M(m) = \binom{M}{m} \tau^k (1-\tau)^{M-m} \tag{2.18}$$

Let ε be the number of LLFs FlowIDs captured after k samples. Then:

$$CL = Prob(\varepsilon > m) = 1 - \sum_{i=0}^{m} P_M(i) \approx \sum_{i=0}^{m} \frac{(M\tau)^i}{i!} e^{-M\tau} \tag{2.19}$$

We are interested in determining M based on the specified CL. Equation (2.19) can be rearranged to calculate the value of M that provides a lower bound on the number of FlowIDs that needs to be collected in order to achieve the desired CL.

$$\Rightarrow M = \left\lceil \frac{\ln\left(\sum_{i=0}^{m} \frac{(M\tau)^i}{i!}\right)}{\tau} \right\rceil + \left\lceil \frac{-\ln(1-CL)}{\tau} \right\rceil \tag{2.20}$$

For an approximate solution, suitable numerical methods can be used. Note that we need to compute Equation (2.19) only once, when the measurement architecture is being initilialized with an empty list of LLFs. Since the samples of LLF are collected from the underlying network traffic at intervals governed by the frequency of SCLK, the duration of initialization phase is simply $M/freq_{sclk}$. Here we make the tacit assumption that an LLF sample is collected during one pulse of SCLK.

2.4.5.2 Determining the List of LLFs at Run Time

The list of LLFs is used to populate the ICAM of the Standby FCU and compiled at run time. As explained in Section 2.2, the list gets updated whenever packets are sampled using the SCLK, or when the FCU transits from Active to Standby mode. The FMU maintains two tables: the array of counters of FlowID implemented according to [58], and the list of current LLFs.

In the next section, we address the following two questions: (i) how to identify LLF from all the FlowIDs, and (ii) how to update the list at run time?

2.5 An Online Framework for Identifying LLFs

Let us now consider an ergodic and discrete random process where each F_i is an independent variable drawn from the state space of $[\mathcal{F}]$ consisting of all possible FlowIDs. However, the random variables are not identically distributed. Denote $\{f_i\}$ to be the set of possible outcomes of F_i with $f \in [\mathcal{F}]$. Let us represent the probability mass function (pmf) of the sequence $\{F_i\}_{i=1}^{N}$ by: $\mathrm{P}\left(F_1 = f_1, \ldots, F_N = f_N\right) = p\left(f_1, \ldots, f_N\right)$. Let $H(\mathcal{F})$ $= \mathrm{H}(F_1, F_2, \ldots, F_N)$ denote the *joint entropy* of the sequence $\{F_i\}_{i=1}^{N}$ and let $\bar{H}_{\mathcal{F}}$ be the *entropy rate* of $\{F_i\}_{i=1}^{N}$. Then, $H(\mathcal{F})$ and $\bar{H}_{\mathcal{F}}$ can be defined as follows [13]:

$$\mathrm{H}(\mathcal{F}) = \mathrm{H}(F_1, F_2, \ldots, F_N) = \sum_{i=1}^{N} H(F_i|F_{i-1}, \ldots F_1) \tag{2.21}$$

$$\bar{H}_{\mathcal{F}} = \frac{1}{\mathrm{N}}\mathrm{H}(\mathcal{F}) \tag{2.22}$$

Since according to our assumption, the F_is are independent, Equation (2.21) reduces to: $\mathrm{H}(\mathcal{F}) = \sum_{i=1}^{N} H(F_i)$ which is the summation of the individual entropies of the flow. At this point, it is worth mentioning that it is possible to *estimate* $\mathrm{H}(\mathcal{F})$ without considering individual flow entropies [45]. However, this is not considered in this work.

Algorithm 2.2 Typical Sequence: Algorithm for calculating the list of LLFs

1: Intilialize list $L := \text{null}$
2: Collect FlowIDs $\{F_i\}$ in the initialization phase
3: $n :=$ number of FlowIDs in the current run of the experiment
4: Calculate the occurence probability $p\,(f_i)$ for each $\{F_i\}$
5: Calculate $H(F)$ and $\bar{H}_F$ using Equation (2.21)
6: **while** $i \Leftarrow N$ **do**
7: **if** p **then**
8: $(f_i) \leq 2^{-n\bar{H}_{\mathcal{F}}}$
9: add F_i to L
10: **end if**
11: **end while**
12: List L contains the set of LLFs
13: Download list to Standby SRAM
14: **Online Update** (see Algorithm 2.1)
15: Upload contents of SRAM to FMU when FCU is in Standby mode
16: Update corresponding FlowID counters
17: Update the list of LLFs
18: Go to Step 3

Definition 1. *The set of LLFs present in a sampled traffic is represented by the sequence,* $\{F_1F_2 \ldots F_{N'}\}$*, where* $N' \ll N$ *denotes the total number of LLFs.*

This definition provides us with the set of all packets which belong to the set of LLFs. Since our aim is to identify the sequence $\{F_1, F_2, \ldots, F_{N'}\}$, we need to isolate the sequence of FlowIDs that occur the highest number of times. If we visualize the sequence, $[\mathcal{F}]$, as an information source, then the existence of the above sequence of FlowIDs is governed by the probability of occurrence of a jointly typical sequence based on Asymptotic Equipartition Property (AEP) [13] in Information Theory. Note that the results based on AEP hold true only when the number of FlowIDs present in the sampled traffic volume is very large. Now considering the fact that there can be several sets of typical sequences, we have the following lemma for the set of LLFs:

Lemma 1. *For traffic volumes with large number of* FlowIDs *(i.e.,* $N \to \infty$*), the occurrence of the sequence* $\{F_1, F_2, \ldots, F_{N'}\}$*, where* $N' \ll N$*, is equiprobable and approx-*

imately equal to $2^{-N\bar{H}_{\mathcal{F}}}$.

Lemma 1 follows directly from the property of AEP. In view of the above, we can say that out of all the possible FlowIDs, that sequence which belongs to the typical set has the maximum concentration of probability. The sequences outside the typical set are *atypical* and their probability of occurrence is extremely low. As evident from the above lemma, a typical sequence implies that FlowIDs in the typical set are associated with a large number of packets. If we consider the distribution of FlowIDs in the Internet traffic, we can easily correlate this property with the Zipf distribution of Internet flows. Hence, it is not surprising that most of the LLFs belong to the typical set. However, what is the guarantee that such a sequence really exists?

Definition 2. *The joint entropy,* $H(\mathcal{F})$*, for a stationary stochastic process of* N *elements is a decreasing sequence in* N *and has a limit equal to its entropy rate.*

Definition 2 implies that the probability of correctly identifying LLFs *increases* with the corresponding increase in traffic volume. This observation is fundamental since it enables us to *scalably* create an approximate list of LLFs (i.e., a typical sequence), while avoiding unnecessary complex computations.

Based on the above discussion, Algorithm **Typical Sequence** enumerates our approach for identifying the LLFs in the underlying traffic. It also provides a high level working of the FastFlow architecture and is composed of two phases: an initilization phase and an online update phase. During the initialization phase, the initial list of LLFs is built in the FMU. This list is downloaded to the the Standby FCU. Note that based on the underlying traffic patterns, the list of LLFs is automatically identified by the FMU and communicated to the FCU.

2.6 Offline Estimation Using Kernel Density Estimator

In this section, we describe our estimation algorithm used for predicting the characteristics of the underlying traffic. We do not make any assumption on the distribution of traffic that we are trying to estimate and thus, resort to non-parametric techniques for estimating the density function. We have used the Gaussian kernel density estimator [56] for estimating the pdf of the given data set. Non-parametric estimators like histogram estimators can also be used for traffic characterization. However, histogram estimators are not smooth and depend on the bin properties (start and the end points of the bins). On the other hand, the kernel density estimators do not suffer form these limitations and provide an excellent tool for pdf estimation.

Figure 2.6 highlights the architecture for offline estimation. The contents of DRAM attached to FMU (see Figure 2.2) are paged to a high speed disk using suitable achedluing algorithms. In this study, we do not focus on any of the associated overheads of I/O scheduling. At the end of the measurement interval, all the FlowIDs present are grouped to form the sequence $\mathcal{F}^{'}$. We employ Parzen window technique on this sequence $\mathcal{F}^{'}$ while the likelihood estimator from Coupon Collector's problem [21] is employed on the set of all LLF FlowIDs. We elaborate on these concepts in the next section.

2.6.1 Estimating the PDF of Sampled Data

Let $\hat{f}_h(x)$ be the pdf of the random variable $\mathcal{X}$ we are trying to estimate for the sequence $\mathcal{F}^{'}$. Since we have considered a Gaussian kernel, from [56], we have:

$$\hat{f}_h(x) \approx \frac{1}{Nh} \sum_{i=1}^{N} \psi \left(\frac{x - x_i}{h} \right) \tag{2.23}$$

where $\{x_i\}_{i=1}^{N}$ are the data points of $\mathcal{X}$ and $\psi(\cdot)$ is a suitable kernel smoothing function of width h, also referred to as the bandwidth of $\psi(\cdot)$. In this approach, the estimated pdf

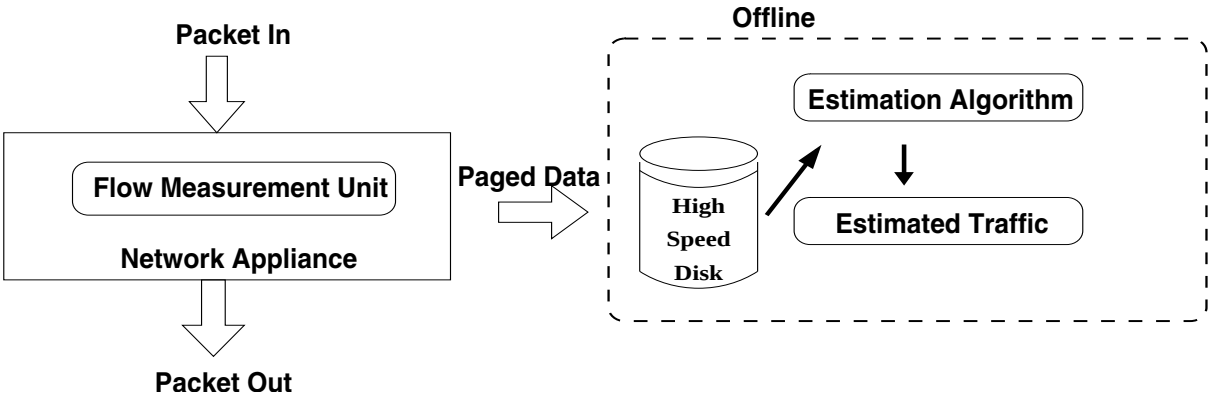

Figure 2.6. Offline classification of traffic streams. The pdf of the LLFs in sampled traffic stream is estimated using the non-parametric Parzen window technique.

is a linear combination of kernel functions centered on the individual x_i's. In Equation (2.23), the bandwidth factor h is the most important term in the estimation process [59]. The optimal value of the kernel window h can be calculated by minimizing the integrated mean square error (IMSE) between the actual pdf, $f(x)$ and the estimated pdf, $\hat{f}_h(x)$. That is,

$$\text{minimize} \left\{ \int \left\{ \hat{f}_h(x) - f(x) \right\}^2 dx \right\}.$$

In general, the process of finding the optimal window size is cumbersome as we do not know beforehand the nature of the density function that we are trying to estimate. Since the shape (degree of smoothness) of $\hat{f}_h(x)$ is closely related to the kernel function used, we use the Gaussian kernel function in our study in order to eliminate "noises" in the pdf estimation. Thus:

$$\psi(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right) \tag{2.24}$$

Corresponding to the Gaussian kernel, the bandwidth h can be approxmated using *Silverman's rule of thumb* [68] that satisfies the IMSE criteria. Consequently, h is defined as: $h = 1.06\,\widehat{\sigma}\,N^{-1/5}$ where $\widehat{\sigma} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N}}$ denotes the sample standard deviation.

It is important to observe that $\psi(\cdot)$ decreases with increase in distance from the origin; indicating that samples which are statistically far away from the cluster of density point can be eliminated without significantly impacting the nature of the estimated distribution. Such outliers can be eliminated without any loss in accuracy.

Estimating the Number of LLFs: In our measurement architecture, the information related to the LLF is incomplete and needs to be accurately estimated. Let N_{total} be the total number of FlowIDs collected and let N_{LLF} be the number of LLF FlowIDs observed during the measurement interval. From N_{total} and N_{LLF}, we have to estimate $\hat{N}_{LLF}$, the total number of FlowIDs present in the original stream. Assuming uniform FlowID arrival, $\hat{N}_{LLF}$ is estimated using the general solution of the Coupon Collector's problem. We use the Maximum Likelihood Estimator [21] in estimating $\hat{N}_{LLF}$. This is given by the smallest j which satisfies the inequality

$$\frac{j+1}{j+1-N_{LLF}}\left(\frac{j}{j+1}\right)^{N_{total}} < 1 \tag{2.25}$$

In our approach, all the SLFs are collected. Therefore, we have an exact value of N_{SLF}, the total number of SLFs present in the underlying traffic. Consequently, the total number of FlowIDs present is simply the summation of N_{SLF} and $\hat{N}_{LLF}$.

2.7 Experimental Results

In this section, we evaluate the performance of our algorithm using packet traces obtained from the National Library for Applied Network Research (NLANR) [85] and from our internal lab network (see Figure 2.7). For NLANR, we use three traces: (i) 20040130-133500-0.gz, (ii) 20040130-13400-0.gz, and (iii) 20040130-134500-0.gz. These are IP header traces captured by NLANR at the end of January 2004. The cumulative duration of the three files is 900 seconds containing 23.2 million packets. They subse-

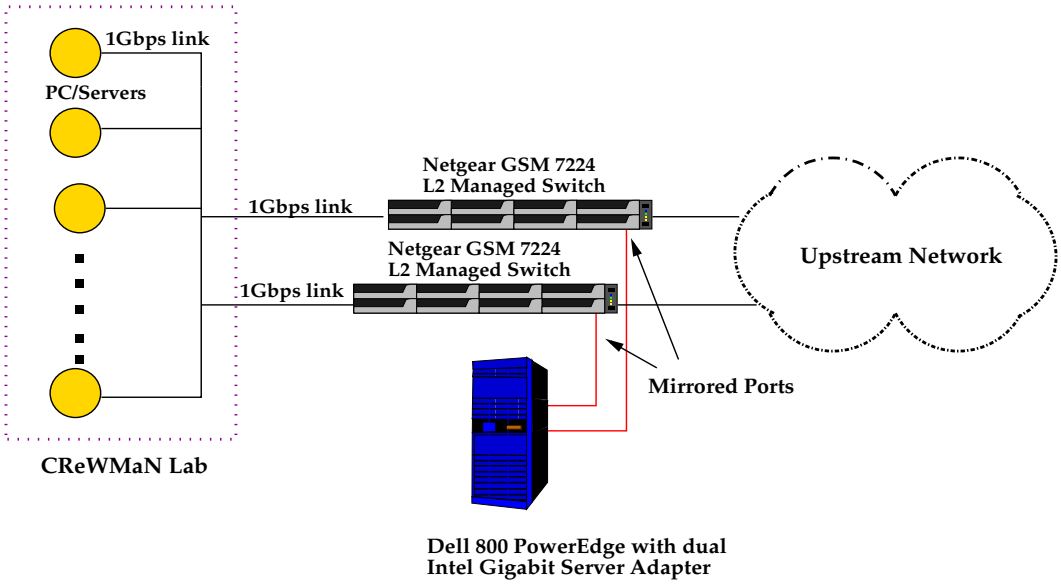

Figure 2.7. Framework for capturing flow packets in the internal LAN in our CReWMaN lab. Port mirroring is used to send traffic to a dual port Gigabit Intel server adapter from which the packets are captured using Ethereal in promiscuous mode.

quently map to 618, 225 FlowIDs, where each FlowID is defined using the aforementioned five-tuple. For our internal LAN, we use four traces collected during 2005 at CReWMaN Lab at UTA. They are labeled as (i) $Capture_1$, (ii) $Capture_2$, (iii) $Capture_3$, and (iv) $Capture_4$. In estimating the LLFs, we compare our approach with that of [51] where classification of packets as LLFs are based on the observation that the average duration of such flows typically exceed 15 mins.

2.7.1 Identifying the LLFs

In Figure 2.8, we plot the number of LLFs predicted using our classification algorithm and compare that with the approach in [51]. We have used the *frequency* of occurrence of the packets as the basis for calculating the flow probabilities. Apart from the already known fact that the proportion of LLFs is small in number (0.0035% in our case), two important conclusions can be drawn immediately:

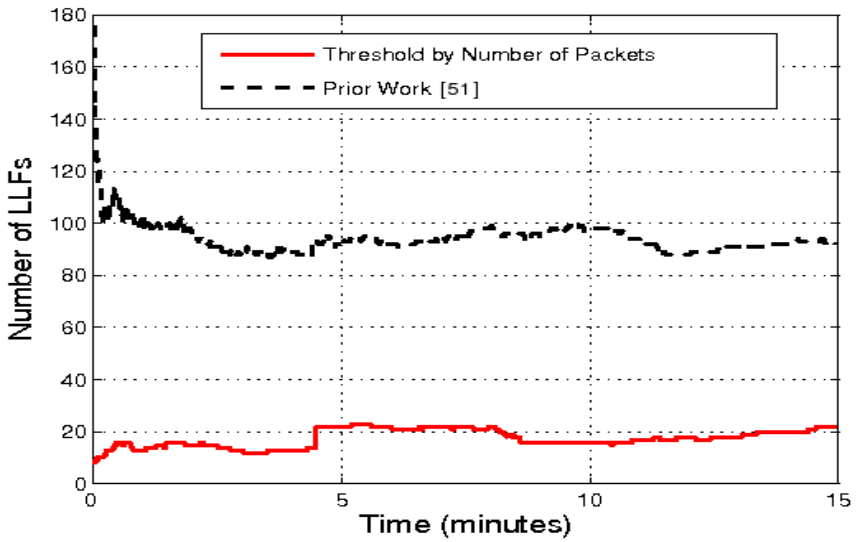

Figure 2.8. Time series of the occurrence of LLFs as estimated using our algorithm versus as predicted using the approach of [51].

- LLFs detected during the *initial phase* (first 5 mins for the traffic traces under consideration) using our classification algorithm identify FlowIDs that exhibit *bursty behavior*. A close analysis of the traffic traces reveals that this is indeed the case and is due to the fact that such FlowIDs cause immediate concentration of the probability mass function of the entire traffic sample.
- The *proportion of LLFs* classified using the frequency of occurrence of packets is almost equal in extent to those detected by considering the volume (bytes) of traffic. Notice that, using the approach of [51], the number of LLFs is estimated at around 85-90. This we believe is an overestimate as the approach of [51] exhibits a temporal *decreasing trend*.

In Figure 2.9, we plot the number of LLFs present in the original traffic stream and compare it with the length of the typical sequences estimated using our approach. In all the four cases, the accuracy of prediction is within 90% of the length predicted via a fixed cut-off approach of 15 minutes longevity.

Table 2.2. LLFs actually present in our lab network traffic trace vs. those estimated by length of the typical sequence

Trace	#Packets	#LLFs	#Typical Sequences
Capture_1	5101225	122430 (2.4%)	107126 (2.1%)
Capture_2	3684712	110542 (3%)	101330 (2.1%)
Capture_3	3675312	118713(3.23%)	113935 (3.1%)
Capture_4	2987316	112779 (3.77%)	107245 (3.5%)

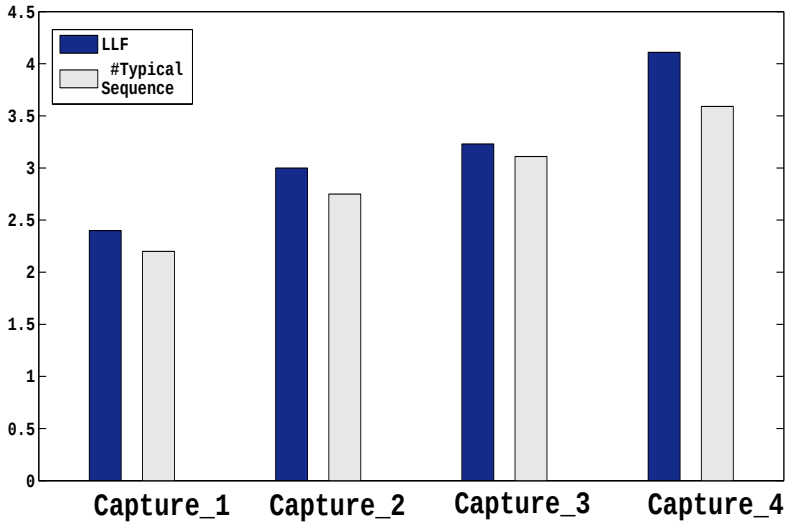

Figure 2.9. Histogram of LLFs and the length of typical sequences in traffic traces for traffic collected from our internal lab network.

2.7.2 Entropy Distribution: LLFs and SLFs

In Figures 2.10 and 2.11, we show the *temporal variation* of the ratio of entropy between the SLFs and LLFs. During the first 500ms of the input traffic, it is observed that there occurs a *dip* in the entropy of the LLFs. This is due to the presence of bursty LLFs which causes a temporary increase in the probability of LLFs. However, as the experiment continues, the entropy of the SLFs increases while the entropy of the LLF flows decreases. This trend indicates *decrease in randomness* of LLFs. Since the entropy of the typical set is a *decreasing sequence* (see Definition 3) with respect to the number

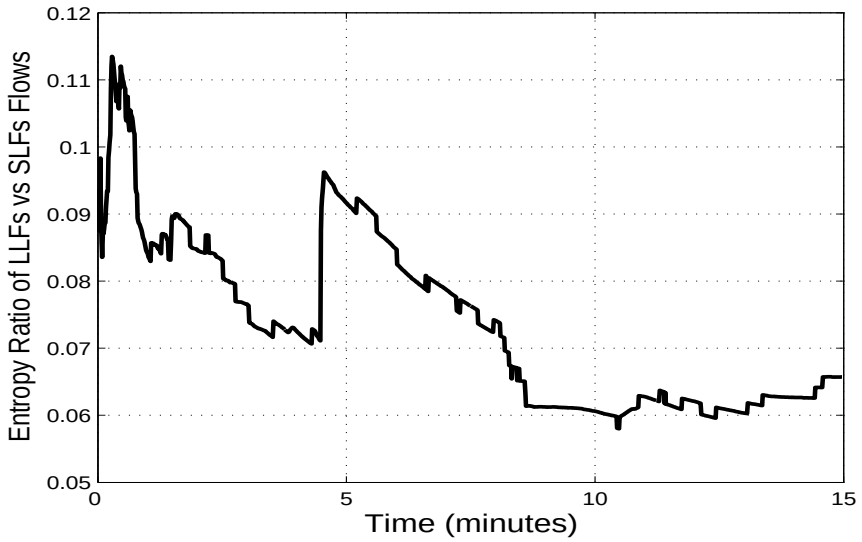

Figure 2.10. Ratio of entropy between Long Lived Flows (LLFs) and Short Lived Flows (SLFs).

of FlowIDs, the probability and proportion of FlowIDs classified as LLFs increase. This unique trend of entropy variation guarantees conservative, yet accurate flow classification of high traffic volumes.

2.7.3 Estimating the Volume of Original Traffic

In order to create the scenario of systematic sampling of LLFs, the list of LLFs was initially calculated by including all FlowIDs whose duration was more than the mean lifetime of the packet trace. This was set to N_{LLF}^{actual}. Then, the typical sequence was calculated over the whole trace. This was followed by sorting all the FlowIDs calculated from the typical sequence; after which we sampled packets at the sampling rate of 10%. The sampled data provided us with the value $N_{LLF}^{sampled}$. The Maximum Likelihood estimator was executed (with 95% confidence level) on the captured traces in order to obtain N_{LLF}^{est}. Table 2.3 provides statistics of the accuracy of the numer of packets estimated using the Coupon Collector algorithm [21].

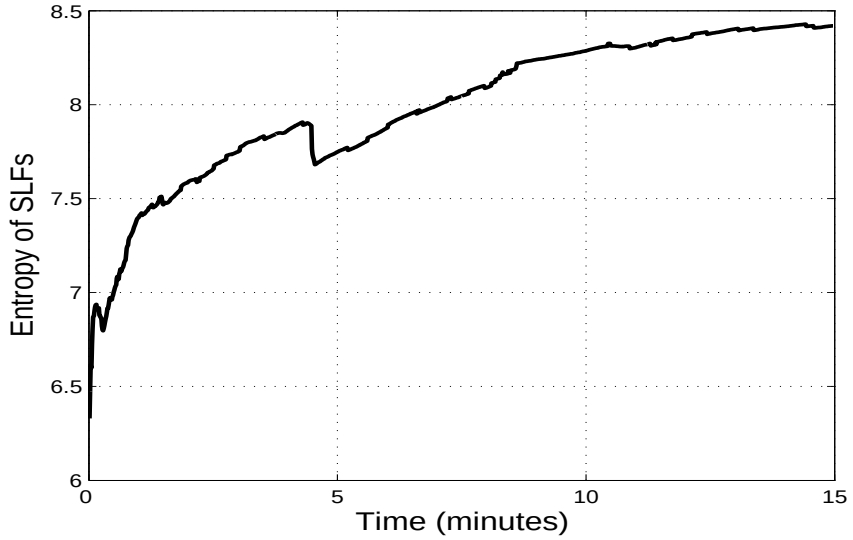

Figure 2.11. Temporal distribution of Entropy of Short Lived Flows (SLFs) .

Table 2.3. Total number of packets actually present in the traffic trace versus those estimated (N_{LLF}^{est}) by FastFlow

Trace	**#Packets**	**#Packets (estimated)**	**% Accuracy**
Capture_1	5101225	4680885	91.76%
Capture_2	3684712	3525901	95.69%
Capture_3	3675312	3418040	93%
Capture_4	2987316	2569092	86%

2.7.4 Estimating the Density Function of Underlying Traffic

In Figure 2.12, we plot the pdf of the flows present in the original traffic stream versus the pdf of flows estimated using the kernel density function. Observe that even with the exclusion of outliers, the estimated pdf match the real value distribution with high acccuracy. The data points used are from *Capture_2*.

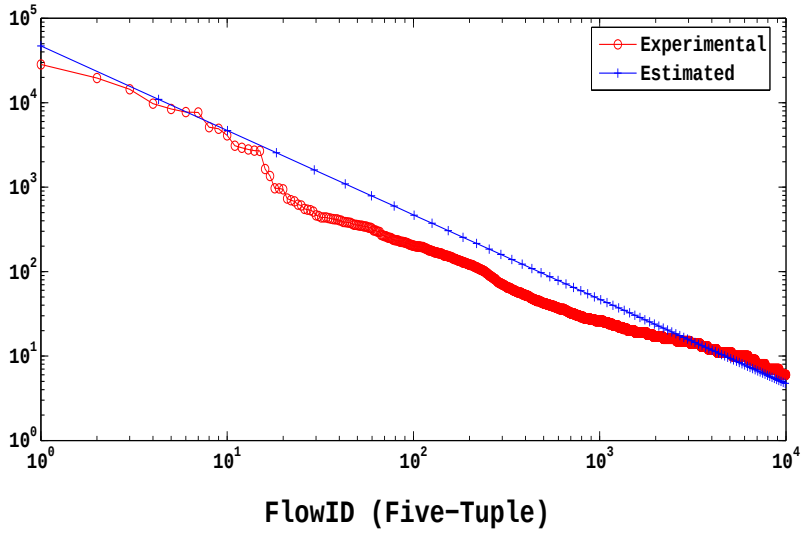

Figure 2.12. Log-log scale plot of the FlowID density function defined using five-tuple in our experiment.

2.8 Summary

Research in the field of traffic analysis has mainly focussed on improving the estimation techniques based on packet sampling which ignores the presence of SLFs. However, as highlighted in this chapter, SLFs comprise a major portion of the network traffic and hence, cannot be ignored. We have taken a non-traditional approach to flow measurement by closely integrating the measurement architecture with the statistical estimation technique. Considering the fact that the Internet traffic is heavy-tailed, we have proposed a novel flow measurement architecture using which all SLFs can be *feasibly* captured with *complete accuracy*. Data related to all non-SLFs is sampled at regular intervals for later analysis. Since the flow identification algorithm inside our architectuture needs to work in real time, we have proposed a flow classification algorithm based on the concept of *typical sequences*. Experimental results have validated our assumption that typical sequences can identify LLFs with very high accuracy. Lossy information obtained due to sampling of all LLFs is estimated using a likelihood function defined over the Coupon

Collector's problem. Finally, we have estimated the distribution (pdf) of the underlying traffic using a non-parametric Parzen window technique.

Armed with the capability to derive the characteristics of the network traffic, we are able to infer the average packet size, protocol composition, and average line rate of the networking link that will be feeding the NICs of the application servers. In the remaining chapters of this dissertation we will examine how to understand and handle the packet processing capabilities for minimizing the latency of the application servers.

CHAPTER 3

AN ANALYTICAL MODEL OF POLLING DEVICE DRIVERS

Once the characteristics of the network traffic in which the application servers in CDNs is expected to work is known, it is important to make sure that the NICs are working at their optimal capacity for preventing latency in the network data transfer mechanisms. In NGNs, such servers are based on Advanced Telecommunications Computing Architecture (ATCA) [89] based blade computing units with different flavors of open source operating system (OS) such as Linux, xBSD, or OpenSolaris. The multiple Gigabit Ethernet (GigE) network interface cards (NICs) present in them involve considerable packet processing at high line speeds. Network processing bottlenecks may arise due to complex interaction between the NICs, host OS, and the underlying hardware. These may be due to inherent limitations present in the network adapter (e.g., limited on-chip buffer space), hardware architecture (e.g., I/O bus width), or simply due to improper allocation and configuration of the system resources (e.g., limited packet receive buffers in the OS). Such bottlenecks often manifest themselves in the form of network packets being dropped with the system being operated at maximum possible processing capacity. Thus, the challenge is not only to operate the NICs at full link capacity at an acceptable value of the system load [15], but also to correctly identify, isolate and remove performance hotspots that may occur when definite resources in the critical path of packet processing gets consumed. On a side note, it should be noted that due to bursty nature of the arriving traffic pattern, performance tuning the receive side of the NIC is more important than the transmit side which is under the control of the OS.

In this chapter, we introduce a queuing model that can be measure the packet

processing capabilities in commodity OS. In Section 3.1, we provide a brief overview of packet processing in polling device drivers. Survey of existing work is provided in Section 3.2. In Section 3.3, we build our queuing framework for analyzing performance complexities. Experimental results using Spirent Communications *Smartbits* 6000C [79] hardware traffic generator in conjunction with *SmartFlow* [78] are present in Section 3.8

3.1 Packet Processing in Commodity OS

Packets arriving from the network are initially placed on the NIC buffer before they are handed over to the OS. There are primarily two different ways the packets can be transferred from the NIC buffer to the OS kernel: (i) *registered interrupts*, and (ii) *device polling*. In the interrupt driven scheme, the NIC generates an interrupt to notify the CPU that packets are available in its receive buffer. This triggers a series of actions that involve interrupting the CPU, executing the Interrupt Service Routine (ISR) of the device, and finally scheduling the packet for subsequent processing. Since raising an interrupt stops the target CPU from executing its current task, most NIC vendors implement various forms of *interrupt coalescing* [91] in which the interrupt generation rate is moderated depending on the packet arrival rate.

Both the interrupt and polling modes in modern NICs involve the Direct memory Access (DMA) chipset for removing the packet from the hardware buffer to the OS memory. Such devices usually act as their own bus master which allows them to access the RAM independently of the CPU and the data is transferred using multiple DMA channels across the I/O bus. Consequently, the transfer of a DMA packet actually consists of a number of physically discontiguous transfers and the average rate of the data transfer is around 1Gbps. However, since the PCI/PCI-X I/O bus (current ATCA standard) is shared by other peripheral devices, too many master devices sharing the same bus can cause the inter-access latency to be unusually high.

While an interrupt based approach improves the responsiveness of the system, it wastes useful CPU cycles at high interrupt loads especially at line rates above 200Mbps. In contrast, device polling shields the CPU from frequent interrupt processing and has been observed to be an efficient approach in packet processing at high line rates (greater than 200Mbps for a typical Linux kernel). For example, the new API (NAPI) [90], available in Linux kernels 2.4.20 and beyond, provides a well-defined interface for registering polling device drivers with the OS. It has been successfully proven to be an effective approach for suppressing interrupt storms at high link loads and simultaneously prevent *interrupt livelock* [50]; wherein the host CPU spends most of its time processing interrupts raised by NICs. Due to their widespread deployment and use in network servers at high line rates, the performance of such polling device drivers is the focus of this chapter. In the next section, we describe how packets are processed in most polling device drivers.

3.1.1 Packet Processing in Polling Device Drivers

At the *receiver side*, network packets arriving on the wire, are first placed on the receive FIFO (rxFIFO) buffer of the network adapter. Inside the OS kernel, receive and transmit packet buffers (collectively referred to as OS buffers) are described by *descriptors* arranged in a circular ring as shown in Figure 3.1. The descriptors are pointers which contain reference to the actual memory location in RAM where the packets are placed by the hardware. These memory locations, referred to as *packet buffers*, determine the maximum data that can be placed. Thus, it is not common for a packet to span multiple descriptors. The *OS descriptors* accessible by the NIC hardware are collectively referred as the *receive DMA ring*. As packets arrive from the network, they are retrieved from the rxFIFO and placed on the receive DMA ring using DMA.

In polling aware drivers, interrupts from the NIC are *disabled* after the arrival of the *first* network packet. This ensures that future interrupts from the device are masked

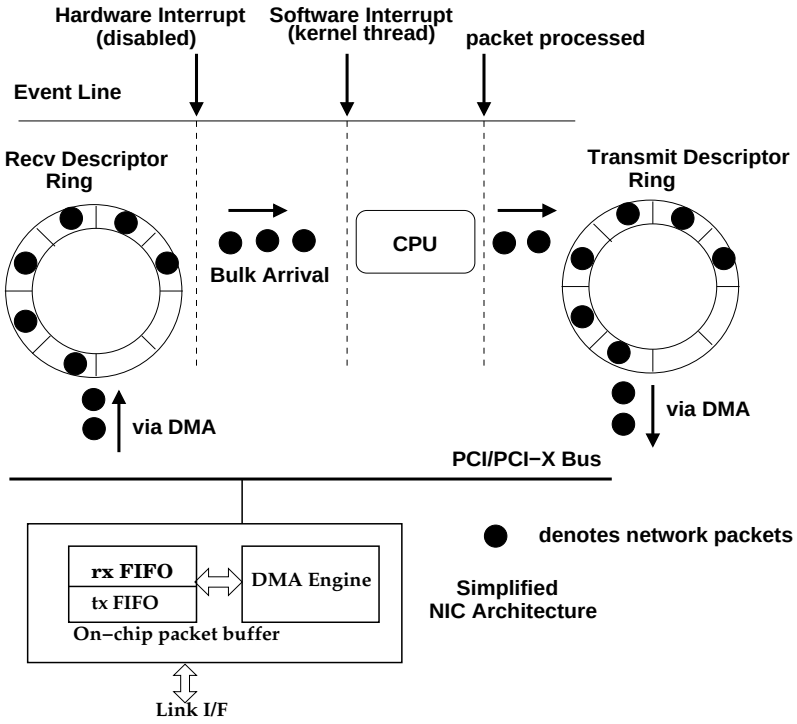

Figure 3.1. Description of network packet processing in polling device drivers.

and the CPU is prevented from the task of frequently servicing the interrupt handler (IH). At the same time, the IH places a pre-defined device polling function on the CPU poll list for clearing the receive DMA ring. However, if during any polling cycle the receive DMA ring is observed to be empty, interrupts from the NIC are re-enabled and the device removes its polling function from the CPU poll list. Such dynamic adaptation ensures that at light traffic load, useful CPU cycles are not wasted due to needless packet polling. However, as soon as packets start building in the receive DMA ring, interrupts from the NIC are again disabled and polling re-enabled. This process of transitioning between interrupt and polling phase continues silently in the background and is heavily dependent on the the *rate* the receive DMA ring is filled by the NIC hardware and serviced by the device polling function.

Packet polling, thus delegates the task of retrieving and processing network packets

to the OS. Since interrupt lines of the NIC are masked, it is the responsibility of the polling function to empty the receive DMA ring and initiate packet processing by calling appropriate CPU scheduling function. This is achieved by moving a certain *quota* of packets from the DMA ring to the CPU work queue. To the CPU, packets thus appear to arrive in bulks and are immediately marked for processing (see Figure 3.1).

Since at high line rates (usually greater than 200Mbps for 64 bytes packet), the packet processing function can completely monopolize the CPU, inside the Linux kernel, a special high priority OS thread (`ksoftirqd_CPUn`) is scheduled after a certain quanta of packets (`netdev_max_backlog`) have been processed during a single invocation of the packet processing handler function (`net_rx_action`). Thus, in the strictest sense of the term, packets *might* not be immediately placed for servicing if the CPU work queue has pending jobs from the system. Also, in addition to NAPI, Intel NICs (8256EB) used in our experiments, utilize their own timer based interrupt moderation and coalescing schemes [91] for interrupt suppresion along the receive path. Considering the fact that such manufacturer specific schemes might skew the results of our experiments, they were explicity disabled in all our measurements so as to isolate and evaluate the performance of polling device drivers in controlled environments. It should be noted that while NAPI works to lessen the host CPU load by focussing on the *receiver side* of network packet processing, frequent interrupt exchange between the NIC hardware and OS during packet *transmission* might also impact the performace of such polling schemes. However, at present, there exists no such NAPI counterpart for the transmit path (from the OS transmit buffer to txFIFO), although various interrupt balancing and moderation schemes are supported by the NIC hardware. Such approaches were also disabled to prevent them from influencing the results.

3.1.2 NIC Device Driver Configurable Parameters

Polling device drivers require careful tuning of several configurable parameters in order to identify the maximum packet processing capability of the system. Some of these parameters are: (i) the number of packets processed during a single invocation of the polling function, (ii) the size of the DMA rings (receive and transmit), (iii) the size of on-chip rxFIFO, and (iv) the number and size of packets transferred over the I/O bus from rxFIFO to the receive the DMA ring.

3.2 Related Work

There exist considerable research on measuring and quantifying delay statistics in the switching elements [26], improving the quality-of-service (QoS), throughput and latency of IP traffic in the Internet [7] [29]. However, limited work has been reported that explores the *dynamics* of packet processing when off-the-shelf NICs are interfed with links operating at GigE rates in commodity OS.

In [27], a measurement study of different 32 and 64 bit GigE NICs was conducted and the throughput observed with variations of packet size. In [32], the authors measured the performance of four server quality motherboards that support GigE NICs. They also reported the performance of PCI/PCI-X bus, variation of UDP throughput with packet size, and CPU utilization in great detail. In [54], the performance of GigE NICs was benchmarked across different manufacturers and subsequently the idea of estimating application level behaviors was proposed based on the results of microbenchmarks.

However, none of the existing works provides an analytical framework that explores the interaction between the NIC resources, I/O bus, OS buffers, and the CPU. The reported measurement results are mainly parametric in nature, and are not easily usable for applications with different arrival and service process dynamics. Consequently, in

the face of various tunable parameters (see Table 3.1) and their closed form interaction, performance tuning of such systems becomes difficult to reconcile.

In the light of the above discussion and considering the fact that an exhaustive experimental analysis involving all system parameters is not practically feasible, we try to answer the following questions in this chapter:

Table 3.1. Typical configurable parameters available for tuning in off-the-shelf NICs and commodity OS

Parameter	**Location**	**Explanation**
rxFIFO	on-chip	receive (rx) packet buffer size
txFIFO	on-chip	transmit (tx) packet buffer size
rxdesc	OS	#entries in the rx descriptor ring
txdesc	OS	#entries in the tx descriptor ring
rxdesc_buffer	OS	receive descriptor buffer size
txdesc_buffer	OS	transmit descriptor buffer size
pollpackets	OS	max. #pkts processed during polling
bulksize	OS	maximum bulk size

- How do we ascertain that a given network processing element with definite I/O bus width and frequency, OS buffers, CPU capability, and multiple NICs is capable of gracefully handling certain data rate worth of input traffic?
- What are the appropriate values of various tunable parameters of the system (network adapters and host OS) such that the maximum performance is achieved for a given configuration?

In this chapter, we first aim at addressing such performance tuning ambiguities by creating a simple queuing model that effectively captures the performance of network adapters from the *viewpoint of OS kernel*. The proposed model takes into account the closed form interaction between the NICs and OS buffers, I/O bus, and the host CPU. We highlight

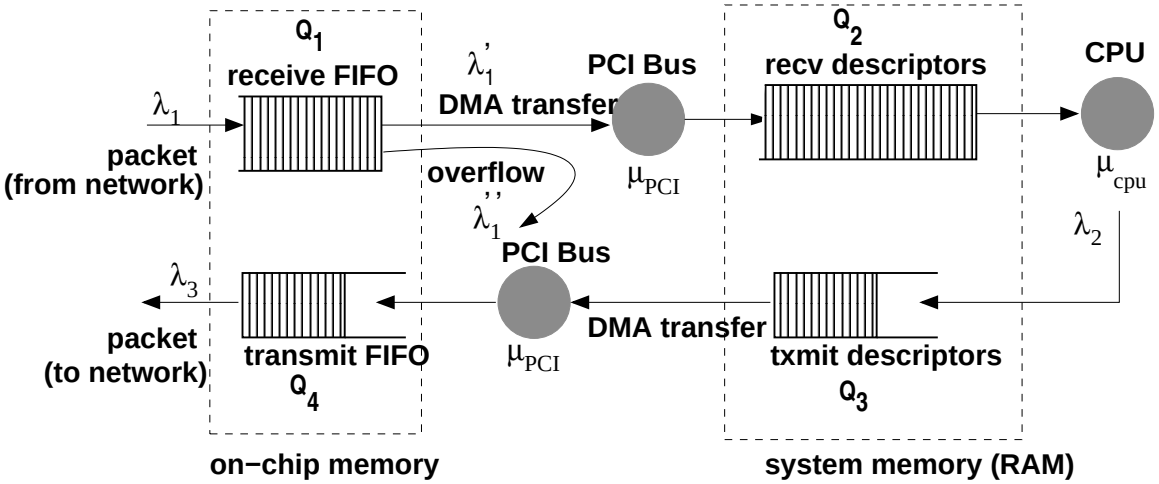

Figure 3.2. Basic queuing model of packet processing polling device drivers.

how our model can be used to derive standard network characteristics like variation of average latency, throughput, processor utilization with specific values of tunable parameters and variations of packet size and line rate in an operational system.

The second goal of the chapter is to understand the *dynamics* of network packet processing as packets are removed from the receive DMA ring by the CPU. Such information can provide invaluable information and insights about the dependency of network performance on the packet size, line rate, on-chip receive FIFO, size of receive DMA ring, and host CPU utilization. Although our approach is generic in nature and applies to polling device drivers (with appropriate kernel support), we specifically consider the Linux 2.6.11.6 kernel, PCI-X I/O interconnect bus, and Intel 82546EB GigE adapters as empirical case studies to validate our work.

3.3 Polling Device Drivers: Analytical Model

We consider the scenario where the system is configured to work as a router forwarding packets between different subnets. No processing (filtering, analysis, traffic shaping, etc.) is intended apart from standard routing functionalities. Consequently, under stable

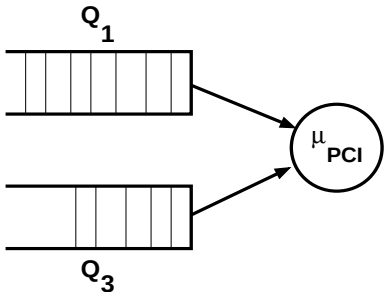

Figure 3.3. Processor sharing model involing the receiver and transmit side PCI/PCI-X bus.

operating conditions, we can safely assume that all packets from the receive DMA ring are successfully sent to the outgoing txFIFO after route processing. Such a process can be modeled by a network of four queues, namely Q_1, Q_2, Q_3, Q_4, as shown in Figure 3.2. Collectively they model the buffers present on the NIC and the host OS.

When a packet arrives on the wire, it is stored on the receive FIFO Q_1, and is subsequently transferred over the PCI bus to the receive DMA ring, Q_2. In polling mode, packets arrive *in bulk* at OS queue Q_2 and are processed by the the device polling routine of the CPU. Packet loss in Q_1 (refer to Figure 3.2) can happen when either queue when the service time of the PCI bus (μ_{pci}), or the CPU service time (μ_e) exceeds the packet arrival rate (λ_1) at Q_1. It should be noted that when the DMA ring (Q_2) gets filled up or the I/O bus is busy, it causes the NIC buffer (Q_1) to overflow since the hardware cannot fetch any more receive descriptors or bus master the PCI/PCI-X IO bus.

Hence, due to packet overflow, the traffic flowing out of Q_1 is *not Poisson* in nature. At the same time, assuming a Poisson arrival process at the PCI/PCI-X IO bus from the output of the buffers Q_1 and Q_3, makes the analysis of polling process tractable. Such an oddity is resolved by finding the moments (mean and variance) of overflow traffic from Q_1 using Equivalent Random Theory [11] and mapping it to an equivalent r stage Erlang process using the Method of Stages [41]. Such an approach is commonly used to analyze

queues where the service time is not exponential in nature. In the next few sections, we derive the following:

- Moments of overflow traffic from Q_1.
- Model of the PCI/PCI-X bus.
- Analysis of the polling process involving bulk service.

3.4 Moments of Overflow Traffic

Although it is well known that network traffic exhibits strong long-range dependency, it has recently being observed that multiplexing of traffic at high speed links often results in traffic assuming a Poisson distribution [9]. This has motivated us to assume that the traffic arriving at Q_1 to be a Poisson process. However, packets might overflow from rxFIFO, Q_1. Hence, the residual traffic flowing out of Q_1 has a certain coefficient of variation and consequently is not Poisson in nature. We model this residual traffic by mapping it to an equivalent $M/E_r/1$ process with the same mean and variance.

Let ρ_1 be the traffic intensity at Q_1. Then, by definition, $\rho_1 = (\lambda_1/\mu_{pci})$ where μ_{pci} is the average service time of the PCI/PCI-X I/O bus and is a dimensionless quantity. Let $\rho_1^{'}$ denote the residual traffic intensity after overflow. Then, $\rho_1^{'} = \rho_1(1 - b_1)$ where b_1 is the blocking probability of buffers Q_1 and Q_3 sharing the I/O bus. Denote α_m and β_v to be the mean and variance of overflow traffic from Q_1. Then α_m and β_v are given by [11]:

$$\alpha_m = \rho_1 b_1 \tag{3.1}$$

$$\beta_v = \alpha[1 - \alpha + \frac{\rho_1}{m + 1 + \alpha - \rho_1}] \tag{3.2}$$

The probability that there are n_1 packets present in Q_1 is given by:

$$p(n_1) = \frac{\rho_1^{n_1}/n_1!}{\sum_{j=0}^{n_1} \rho_1^j/j!} \tag{3.3}$$

Let $m_{residual}$ and $v_{residual}$ denote the mean and variance of the residual traffic from Q_1. Then:

$$\lambda_1^{'} = m_{residual} = (1 - \mathrm{b_1})\lambda_1 \tag{3.4}$$

$$\text{and } v_{residual} = (\lambda_1 - \beta) \tag{3.5}$$

$\mathbf{M/E_r/1}$ Queue: In reality, the distribution of the service time of the PCI bus is not exponential in nature. However, the PCI service time can be viewed as a process consisting of several serial stages. Thus, all packets leaving Q_1 are visualized to pass through r stages of service and the resulting traffic flowing out of Q_1 can be assumed to be smooth. The nature of such "smoothness" depends on the coefficient of variation, C_b, which for Erlang Method of Stages [41] is given by:

$$\mathrm{C}_b = 1/\sqrt{(r)} \tag{3.6}$$

The value of r can be obtained by equating the variance of the Erlangian distribution for r stages, $(1/(\mathrm{r}\mu_{pci}^2))$, with that of Equation (3.5). Thus:

$$r = \frac{1}{\mu_{pci}^2 v_{residual}} \tag{3.7}$$

The fallout of such an approach is that *the analytical model tends to be conservative if the number of Erlangian stages are small.* However, the error (degree of overestimate) reduces with the increase in the value of r. But it makes the analysis mathematically

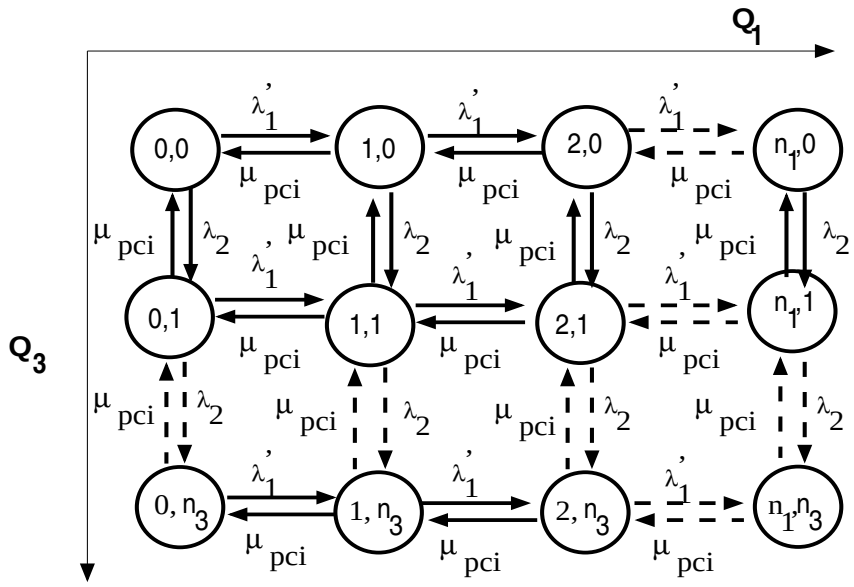

Figure 3.4. State Space of the PCI bus where n_1 and n_3 refer to the size of the buffers Q_1 and Q_3 respectively.

feasible and practical for real systems. Based on this discussion, we assume the traffic flowing out of Q_1 is Poisson in nature.

3.4.1 Queuing Model of the I/O Bus

The I/O bus receives two distinct types of network traffic (refer to Figure 3.2): (i) residual traffic (packets after overflow) from queue Q_1, and (ii) packets transmitted from queue Q_3. We assume the service time of the PCI bus to be negative exponential.

Under steady state conditions, the state space of the buffers can be described by a two-dimensional birth and death process with states (i,j), where i and j denote the number of packets present in Q_1 and Q_3 respectively. Let n_1 and n_3 denote the size of queues Q_1 and Q_3. Then, the state space is valid only when $(i+j) < n_1 + n_3]$.

Let the equilibrium probability be denoted by $P(i,j)$ where i and j denotes the number of packets in Q_1 and Q_3 respectively (i.e., the stationary probability). Assuming

that overflow traffic from Q_1 has no impact on Q_3 and there occurs no priority between packets from Q_1 and Q_3, $P(i,j)$ is given by:

$$P(i,j) = \frac{(\lambda_1^{'}/\mu_{pci})^i}{i!}\frac{(\lambda_2/\mu_{pci})^j}{j!}C;\ \text{for } 0 \leq i \leq n_1, 0 \leq j \leq n_3 \tag{3.8}$$

$$C = \left[\sum_{(i,j)\subseteq\{0,n_1,n_3\}} \frac{(\lambda_1^{'}/\mu_{pci})^i}{i!}\frac{(\lambda_2/\mu_{pci})^j}{j!}\right]^{-1}$$

In Equation 3.8, C is the normalization constant. We are interested in the blocking probability, b_1 of queue Q_1 i.e. the probability that the NIC card is denied access to the I/O bus. It is obtained from the marginal distribution of the joint probability distribution of Equation (3.8) and is defined as:

$$b_1 = \sum_{j=0}^{n_3} P((n_1+n_3)-j, j) \tag{3.9}$$

3.5 Estimating the Service Time of PCI Bus (μ_{pci})

Let p be the number of packets which are transferred during a single instant of DMA operation and let B be the size of each packet (in bytes). Then, $B*p$ bytes are transferred over the PCI bus before the NIC DMA controller relinquishes its hold[†]. Let the width of the PCI bus be w bits and the its clock frequency be f_{io} Hz. Then, the service time of the PCI bus, μ_{pci}, is given by:

$$(1/\mu_{pci}) = \frac{8*B*p}{w*f_{io}} seconds \tag{3.10}$$

We are now in a position to model the polling process of the device driver.

[†]Due to mismatch between the size of rxFIFO and size of a packet buffer in the DMA ring, a single packet is usually spread over physically discontinuous regions in main memory. This is commonly referred to as "scatter-write".

3.6 Dynamics of the Polling Process

Referring to Figure 3.2, packets arriving at Q_2 with Poisson intensity $\lambda_1^{'}$ are placed in the receive DMA ring buffer. When the polling function gets scheduled, certain number of packets are placed on the processor work queue on an FCFS service during the polling cycle. Packets arrive at the processor work queue in bulks and the bulk size is determined by the rate of arrival of the packets at the receive OS buffer (Q_2). Thus, the process can be modeled as a bulk $M/M/1$ queue and the bulk transition is shown in Figure 3.5.

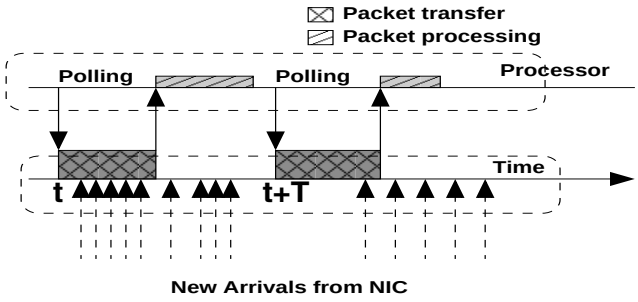

Figure 3.5. Timeline highlighting bulk removal of packets by the CPU from the receive descriptor ring during each invocation of the polling function.

Let $T_{\mathcal{P}}$ denote the inter-polling interval and $\mathcal{K}$ be the number of packets which are moved from Q_2 during each invocation of the polling process. It is evident that the bulk size $\mathcal{K}$ is *variable* and depends on: (i) the *residual number* of packets from the previous polling instant, (ii) and on the number of new arrivals during the inter-polling time instant $i * T$, where $i = [0, 1 \ldots \infty]$ is the i^{th} polling instance (refer to Figure 3.5).

Let the state variable be the total number of packets in the queue Q_2 *including* the ones being served by the processor. Denote π_n as the steady-state probability that there are n packets in the system (CPU work queue plus Q_2). Then, we have the bulk-arrival state transition process for the CPU where the polling instants define the bulk arrival

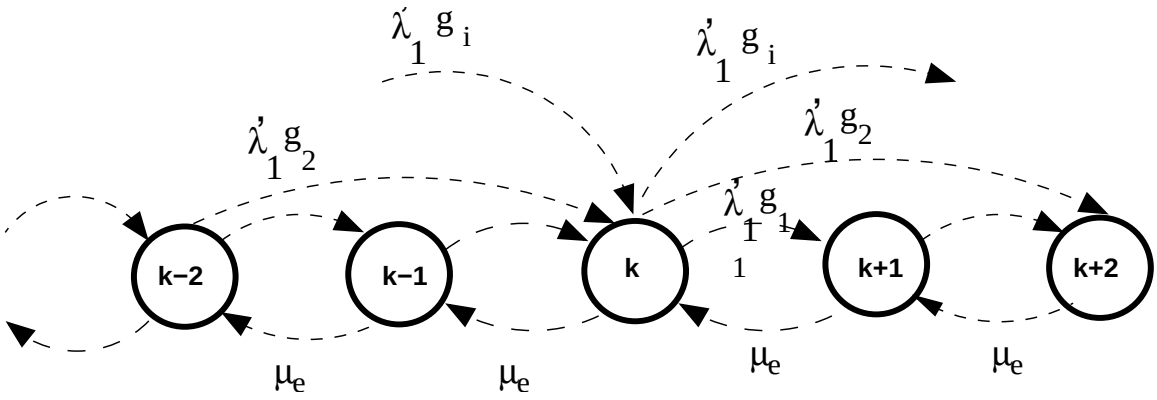

Figure 3.6. State Transition Diagram of Bulk Arrival at the CPU work queue during the polling process.

process. This is illustrated in Figure 3.6. State transition occurs at the time boundary when new packets arrive or when the processor has serviced an existing packet.

3.6.1 Equilibrium Equations

Let g_m denote the probability that the bulk size is m. Thus, $\sum_{m=1}^{\infty} g_m = 1$. Under steady state conditions for the bulk arrival system of Figure 3.6, we have from flow conservation:

$$\left(\lambda_1^{'} \sum_{m=1}^{\infty} g_m + \mu_e\right) \pi_{\mathcal{K}} = \left(\mu_e \pi_{\mathcal{K}+1} + \lambda_1^{'} \sum_{m=0}^{\mathcal{K}-1} \pi_m g_{\mathcal{K}-m}\right) \quad \mathcal{K} \geq 1 \tag{3.11}$$

$$\lambda_1^{'} \pi_0 = \mu_e \pi_1 \tag{3.12}$$

Notice that Equation (3.11) can be further reduced by observing $\sum_{m=1}^{\infty} g_m = 1$. We are interested in finding out the average queue size and the average service time of Q_2, which lies along the packet received path. Using z-transforms, the solution to Equations (3.11)(3.12) for bulk $M/M/1$ queue is given by [41]:

$$\Pi(z) = \frac{\mu_e(1-\rho_b)(1-z)}{\mu_e(1-z) - \lambda_1^{'} z[1-G(z)]} \tag{3.13}$$

In Equation (3.13), $\Pi(z)$ and $G(z)$ are the z-transforms of the pdf of the queue size and the bulk distribution, while ρ_b is the queue utilization factor. By definition, $G(z)$ is given by:

$$G(z) = \sum_{m=1}^{\infty} g_m z^m \tag{3.14}$$

Using Equation (3.14) is not straightforward since it requires us to express the z-transform of the bulk arrival into various powers of z. Instead, if we are able to find a distribution that best describes the bulk size distribution of Q_2, then we are done. Once the z-transform of the pdf of the bulk size is obtained, it is straightforward to derive the average buffer occupancy since according to [41]:

$$\rho_b = \frac{\lambda_1^{'} G^{'}(1)}{\mu_e} \tag{3.15}$$

$G^{'}(1)$ is the average bulk-size of the arrival process and is easily obtained from equation (3.14) by differentiating $G(z)$ w.r.t z and setting $z = 1$.

The average size of Q_2 can be obtained by differentiating Equation (3.13) w.r.t z and setting $z = 1$. Thus,

$$E\left[Q_2\right] = \frac{d\Pi(z)}{dz} | z = 1 \tag{3.16}$$

Applying L'Hospital's rule twice to Equation (3.16), we obtain:

$$E\left[Q_2\right] = \frac{2\lambda_1^{'} G^{'}(1) + \lambda_1^{'} G^{''}(1)}{\mu_e(1 - \rho_b)} \tag{3.17}$$

3.6.2 Bulk Size Distribution

Before mathematically formulating the bulk size distribution at queue Q_2, let us see the nature of the bulk size as obtained experimentally. It will provide us with a clue of the nature of the underlying process.

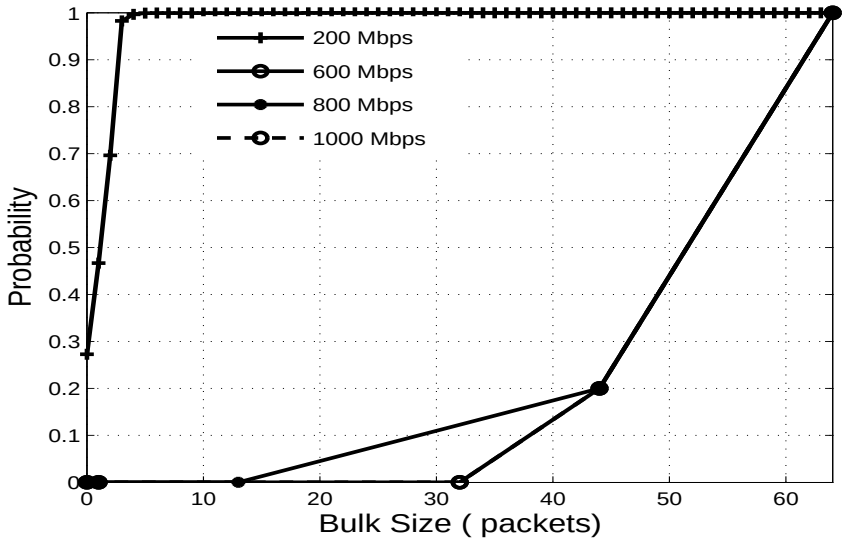

Figure 3.7. CDF of bulk size Distribution for packet size = 64bytes, rxFIFO = 32MB, rxDescriptors = 1024.

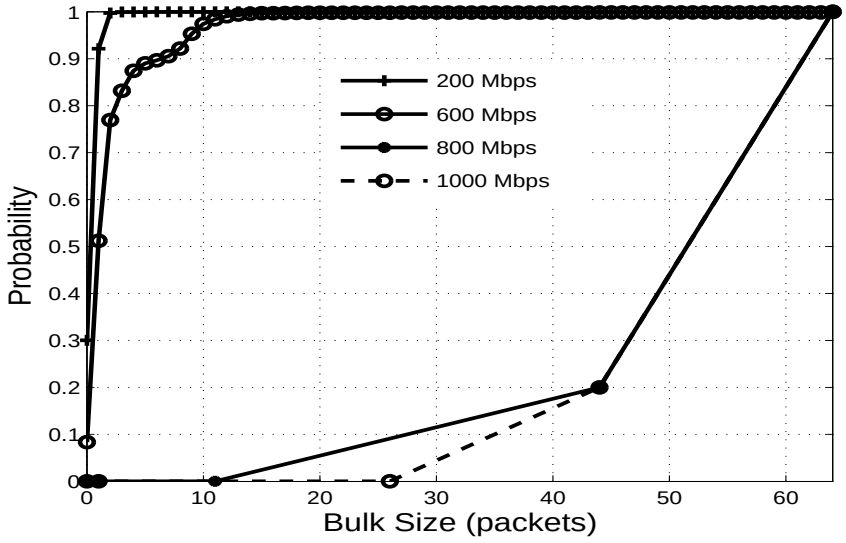

Figure 3.8. CDF of bulk size distribution for packet size = 512bytes, rxFIFO = 32MB, rxDescriptors = 1024.

3.6.3 Constant Bulk Size

At line rate of 200Mbps or less from Figures 3.7, 3.8, and 3.9, observe that the bulk size is almost constant during the entire run of the experiment. At the same time it is

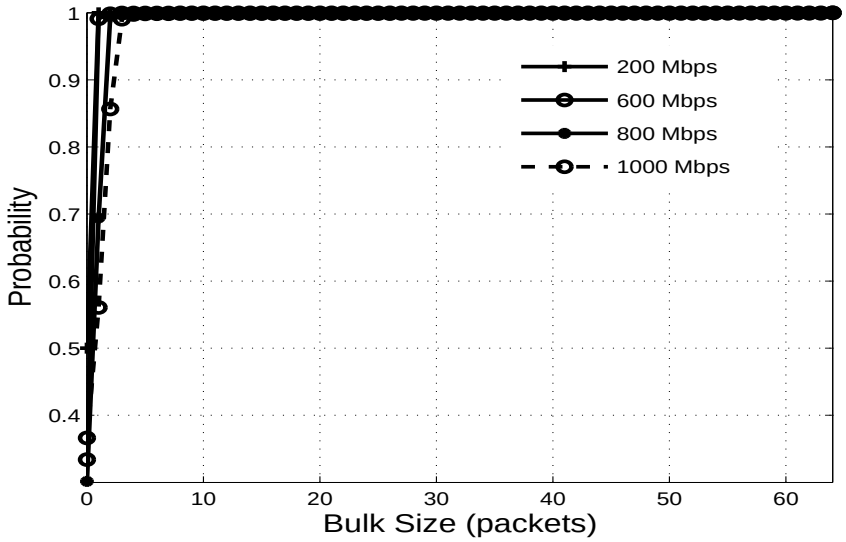

Figure 3.9. CDF of bulk Size Distribution for packet size = 1400bytes, rxFIFO = 32MB, rxDescriptors = 1024.

observed that no packets are dropped by the NIC. Lack of fluctuation or modulation of the bulk arrival indicates that the system is at stable steady state conditions. In order to derive the average queue size, let us assume that the bulk size has a constant value of $\mathcal{K}_b$. Thus, $G(z) = z^{\mathcal{K}_b}$. If we plug this into Equation (3.17), we have:

$$E[Q_2] = \frac{(\mathcal{K}_b + 1)}{2}\left(\frac{\rho_b}{1-\rho_b}\right) \tag{3.18}$$

where $\rho_b = (\lambda_1^{'}\mathcal{K}_b)/\mu_e$. Now the average time T_e a packet spends in the system, (queue Q_2 plus CPU), T_e, depends on the arrival rate $\lambda_1^{'}$. Applying Little's result, we have $E[Q_2] = \lambda_e T_e$. Consequently, T_e is given by:

$$T_e = \frac{(\mathcal{K}_b + 1)}{2\lambda_1^{'}}\frac{1}{(1-\rho_b)} \tag{3.19}$$

3.6.4 Varying Bulk Size

We have seen that the appearance of the constant bulk size indicates that the system is more or less operating with very little packets drop at the NICs. But, the CPU is kept busy all the time, te utilization of queue Q_2 is very high, and very few packets are dropped by the NICs. However, there occur scenarios primarily due to traffic bursts, CPU overload, or I/O bus contention when packets are dropped at the NICs (queue Q_1) as a result of which the bulk size distribution at Q_2 starts varying. From Figures 3.7, 3.8, and 3.9, we can see the variations of the cdf of the bulk size distributions at increased line rates. Notice how the peak of the curve has started shifting to the left. Based on the shape, we fit find the bulk size can be best described by a Rayleigh distribution. The pdf of the random variable $\mathcal{X}$, $g_{\mathcal{X}}(x)$, following a Rayleigh distribution with variance σ^2 is given by:

$$g_{\mathcal{X}}(x) = \frac{x}{\sigma^2} e^{-x^2/2\sigma^2} \tag{3.20}$$

$$\tag{3.21}$$

To find the z-transform we first find the Fourier transform of $f_X(x)$ and obtain the z-transform by recognizing that $z = e^{j\omega}$, where j is the complex coefficient in the Fourier transform. Thus,

$$G(\omega) = \int_{-\infty}^{\infty} \frac{x}{\sigma^2} e^{-x^2/2\sigma^2} e^{-j\omega x} dx \tag{3.22}$$

Using Integration by parts, we obtain:

$$G(\omega) = \sqrt{\frac{\pi}{2}}\ \sigma(j\omega)\ e^{-\frac{\omega^2\sigma^2}{2}} \tag{3.23}$$

Substituting $z = e^{j\omega}$ and recognizing that $j^2 = -1$, we have,

$$G(z) = \sqrt{\frac{\pi}{2}}\ \sigma \left[z\ z^{-\sigma^2/2} \right] \tag{3.24}$$

Equation 3.24 represents the bulk size distribution when the bulk size distribution is observed to modulate at Q_2. After obtaining $G^{'}(1)$ and $G^{''}(1)$, we can calculate the expected queue length of Q_2. Thus,

$$E\left[Q_2\right] = \frac{\sigma^2 - 1}{2} \left(\frac{\rho_b}{1 - \rho_b} \right) \tag{3.25}$$

Note that the expected queue length depends on the variance, σ^2, of the Rayleigh distribution.

3.6.5 Average Packet Service Time (μ_e)

As discussed earlier, in NAPI compatible device drivers, the system moves between the interrupt mode and the polling mode depending on the rate of traffic arrival and the polling process. Let p_{int} be the probability that the packets are pulled in the interrupt mode and p_{poll} be the corresponding polling probability. We assume that in the interrupt mode, interrupts are generated at the rate of one per packet. Denote L as the *average* bulk size in the polling mode. Then, the number of interrupts generated, N_{int}, can be roughly estimated as

$$\mathrm{N}_{int} = \left(\lambda_1^{'} * p_{int} + \frac{\lambda_1^{'}}{L} * p_{poll} \right) \tag{3.26}$$

Denote $\mathcal{T}_1$ to be the average interrupt holding time and $\mathcal{T}_2$ to be the average packet processing time (calculated over interrupt and polling mode). Then, the average CPU *occupancy* is given by:

$$\rho_{cpu} = N_{int} * \mathcal{T}_1 + \lambda_1^{'} * \mathcal{T}_2 \tag{3.27}$$

If the service time of the CPU is assumed to be negative exponential w.r.t to the occupancy, then the value of $1/\mu_e$ can be obtained for a specific value of ρ_{cpu} from Equation (3.27).

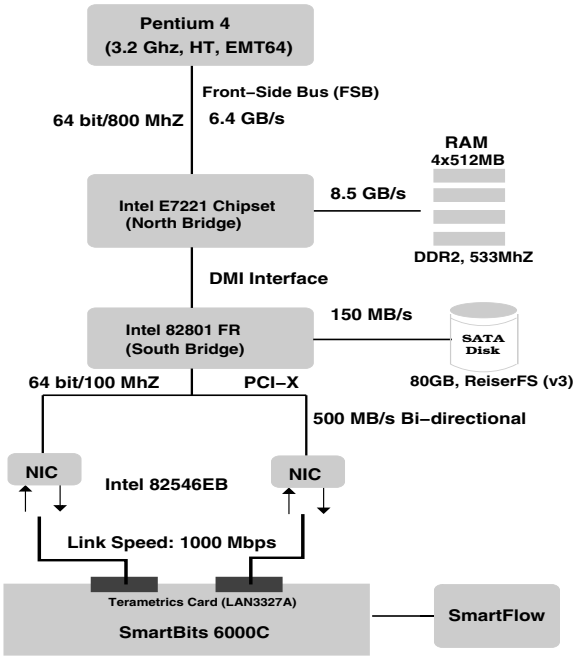

Figure 3.10. Motherboard architecture used in our experiments. SmartBits 6000C using dual Terametrics card (LAN3327A) in conjunction with SmartFlow was used to generate and analyze traffic from the system-under-test.

3.7 Performance Evaluation

In this section we validate the performance of our queuing model with results obtained from our experimental setup.

3.7.1 Experimental Platform

We evaluate the performance of our approach by conducting experiments on the motherboard shown in Figure 3.10. It has a core speed of 3.2 GHz with EMT64 (i.e., 64-bit extensions of x86 architecture) Pentium IV processor with 800MHz front-side bus

Table 3.2. CPU Utilization: rxFIFO = 32KB, rxDescriptors = 1024

Line Rate (Mbps)	Packet Size (bytes)	Experimental	Model
100	64	55%	60%
	512	19%	23%
	1400	8%	5%
200	64	80%	78%
	512	37%	43%
	1400	11%	7%
400	64	80%	78%
	512	55%	78%
	1400	28%	26%
800	64	98%	100%
	512	62%	59%
	1400	55%	58%

(FSB) and with 1MB L2 cache. Both Hyperthreading [6] and IRQ affinity [48] were disabled in all our experiments. The motherboard has 2GB of dual-channel, DDR2 RAM operating at 533MHz and configured as 4x512MB DIMMs. To prevent the impact of other processes competing for CPU resources, the machine was booted in the single user mode with no X-server and was also disconnected from the external network.

The network adapter used in our experiments is Intel 82546EB [80] with 64bit PCI-X/66MHz interface. It has a 64KB on-chip packet buffer shared between rxFIFO and txFIFO, supports advanced hardware interrupt moderation schemes and uses the Open Source Linux kernel driver, `e1000` [80] (version used is `6.2.15`). The OS used in the experiments is Linux with `2.6.11.6` kernel compiled with multiprocessor (SMP) support.

3.7.2 Testing Methodology

We study the accuracy of our queuing model and the performance of NAPI when Linux is configured to work as a router using `ip_forward`, `arp` and `route`. Such an approach has two distinct advantages: (i) it empowers us to control the path of packet

Table 3.3. CPU Utilization: rxFIFO = 32KB, rxDescriptors = 512

Line Rate (Mbps)	Packet Size (bytes)	Experimental	Model
100	64	58%	63%
	512	23%	25%
	1400	8%	12%
200	64	64%	66%
	512	37%	42%
	1400	17%	22%
400	64	98%	100%
	512	59%	55%
	1400	30%	27%
800	64	98%	100%
	512	59%	55%
	1400	58%	62%

Table 3.4. CPU Utilization: rxFIFO = 32KB, rxDescriptors = 128

Line Rate (Mbps)	Packet Size (bytes)	Experimental	Model
100	64	58%	55%
	512	19%	22%
	1400	8%	12%
200	64	58%	63%
	512	38%	43%
	1400	17%	22%
400	64	79%	85%
	512	57%	66%
	1400	28%	32%
800	64	81%	86%
	512	67%	73%
	1400	57%	66%

processing inside the router for detailed forensic analysis, and (ii) it allows us to explore the impact of traffic dynamics (traffic rate, packet size, burst size) on NAPI in a controlled environment.

Traffic Generation: We use *Smartbits* 6000C chassis [79] from Spirent Communications along with two Terametrics LAN-3327A modules in conjunction with *SmartFlow* application software [78] for generating traffic (see Figure 3.10). Each of the LAN-3327A modules is capable of generating pseudo-random traffic (with different load levels

Table 3.5. CPU Utilization: rxFIFO = 48KB, rxDescriptors = 1024

Line Rate (Mbps)	Packet Size (bytes)	Experimental	Model
100	64	67%	76%
	512	8%	12%
	1400	20%	22%
200	64	72%	75%
	512	19%	22%
	1400	38%	42%
400	64	81%	83%
	512	30%	33%
	1400	55%	57%
800	64	95%	99%
	512	55%	57%
	1400	73%	66%

Table 3.6. CPU Utilization: rxFIFO = 48KB, rxDescriptors = 512

Line Rate (Mbps)	Packet Size (bytes)	Experimental	Model
100	64	62%	67%
	512	20%	23%
	1400	9%	5%
200	64	62%	67%
	512	37%	44%
	1400	15%	24%
400	64	98%	100%
	512	55%	57%
	1400	32%	39%
800	64	88%	97%
	512	65%	67%
	1400	55%	65%

and packet size distribution) up to 1000Mbps. *Smartflow* uses the capability of Terametrics card to simultaneously generate and analyze various types of traffic and network parameters (latency, packet loss, throughput).

3.8 Experimental Results

All the results reported in this study are for line rates 100, 200, 400, and 800 Mbps and with packet sizes of 64, 512, and 1400 bytes. The maximum bulk size was set to

Table 3.7. CPU Utilization: rxFIFO = 32KB, rxDescriptors = 128

Line Rate (Mbps)	Packet Size (bytes)	Experimental	Model
100	64	60%	60%
	512	20%	22%
	1400	8%	12%
200	64	65%	72%
	512	37%	42%
	1400	15%	20%
400	64	74%	81%
	512	58%	62%
	1400	32%	35%
800	64	72%	75%
	512	63%	66%
	1400	52%	55%

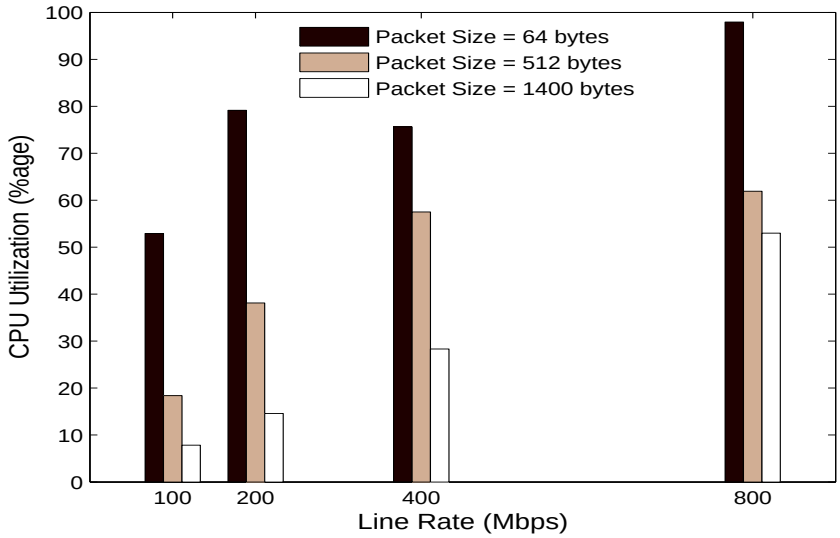

Figure 3.11. Average CPU Utilization with rxFIFO = 32MB, rxDescriptors = 1024.

64 packets and the polling function yielded the CPU every 300 network packets. In this dissertation, we report our observation of average CPU utilization, bulk distribution and the average latency with variations of line rates and packet size.

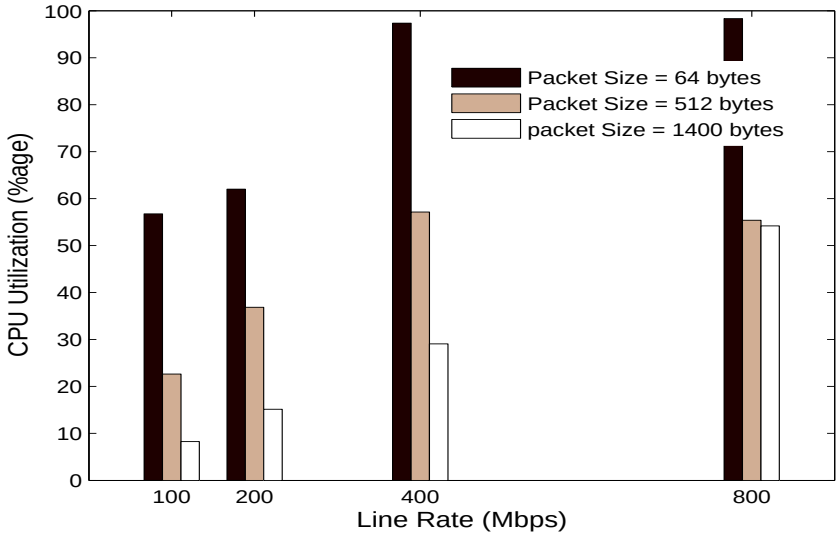

Figure 3.12. Average CPU Utilization with rxFIFO = 32MB, rxDescriptors = 512.

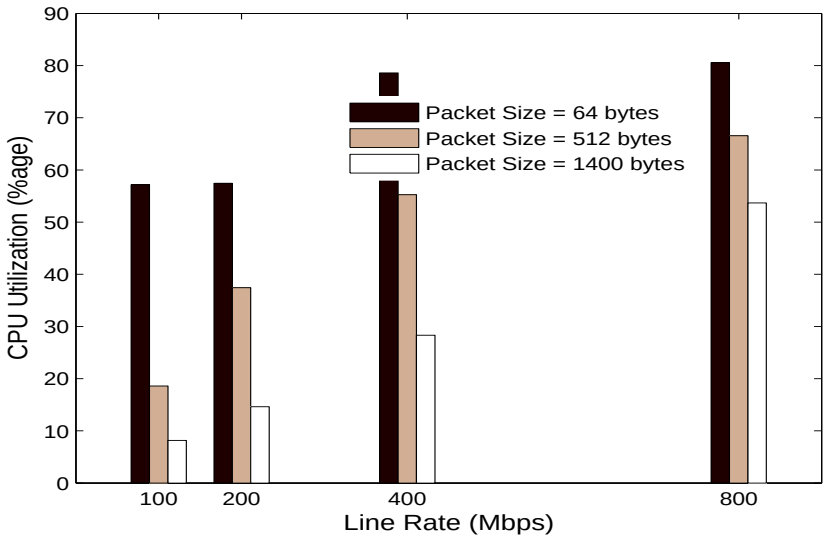

Figure 3.13. verage CPU Utilization with rxFIFO = 32MB, rxDescriptors = 128.

3.8.1 Average CPU Utilization and Average Number of Interrupts

In Figures 3.11, 3.12, 3.13, we plot the average CPU utilization with variations of packet sizes at different line rates for rxFIFO = 32MB. The size of the DMA ring (i.e., number of rxDescriptors) is varied from 128 to 1024. We observe that for 64 bytes

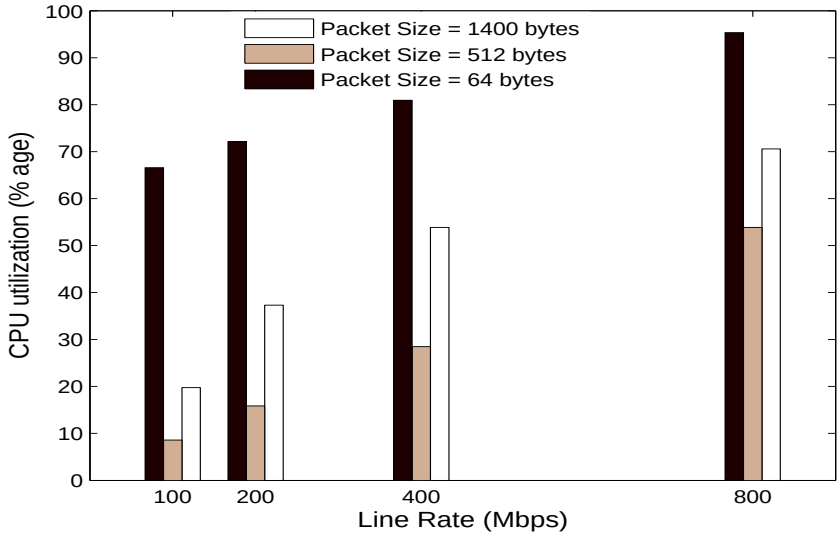

Figure 3.14. Average CPU Utilization with rxFIFO = 48MB, rxDescriptors = 1024.

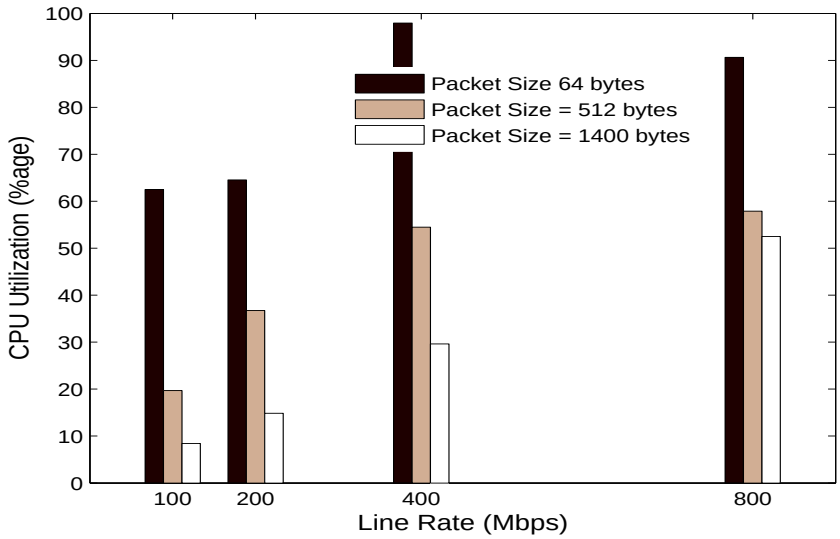

Figure 3.15. Average CPU Utilization with rxFIFO = 48MB, rxDescriptors = 512.

packet size at line rate of 800 Mbps, the average CPU utilization is almost 100%. This value is substantially lower at the line rate of 100Mbps (around 50%). However, as the size of the packet is increased from 64 bytes to 1400 bytes, the average CPU utilization decreases due to decrease in the traffic intensity at the DMA buffer. As the size of the

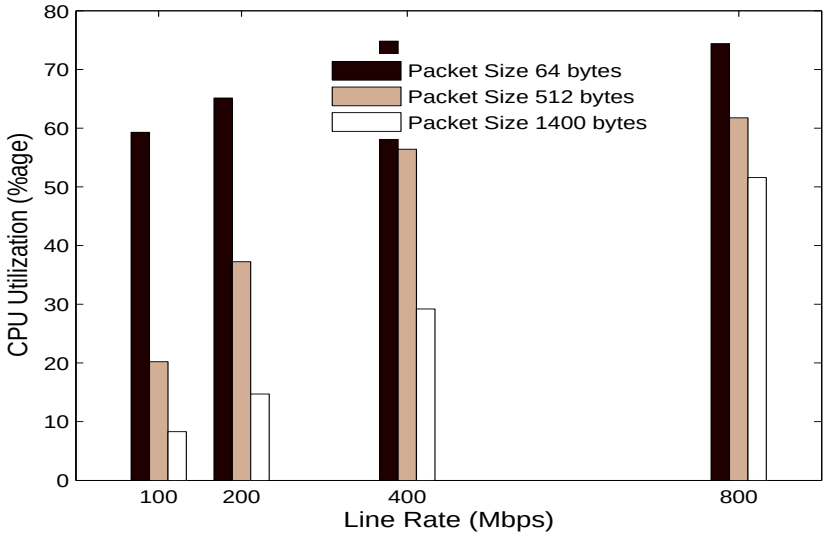

Figure 3.16. Average CPU Utilization with rxFIFO = 48MB, rxDescriptors = 128.

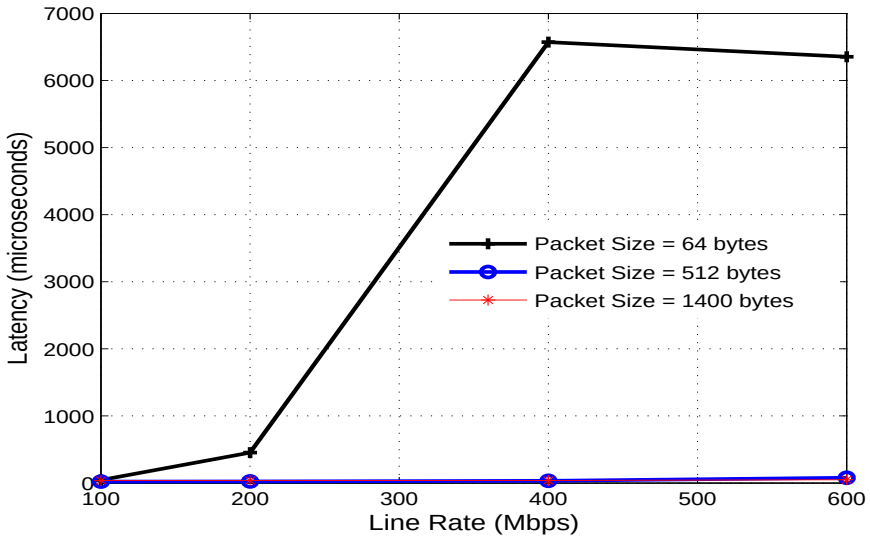

Figure 3.17. Average Packet Latency with rxFIFO = 48MB, rxDescriptors = 1024.

on-chip FIFO is increased to 48MB, the average CPU utilization decreases for packet size of 512 bytes. For 64 bytes packets at similar line rates, the corresponding figures are higher since there occurs no overflow at the rxFIFO and the DMA ring is always kept full. If we plug the values of theses parameters in the analytical model, for 64

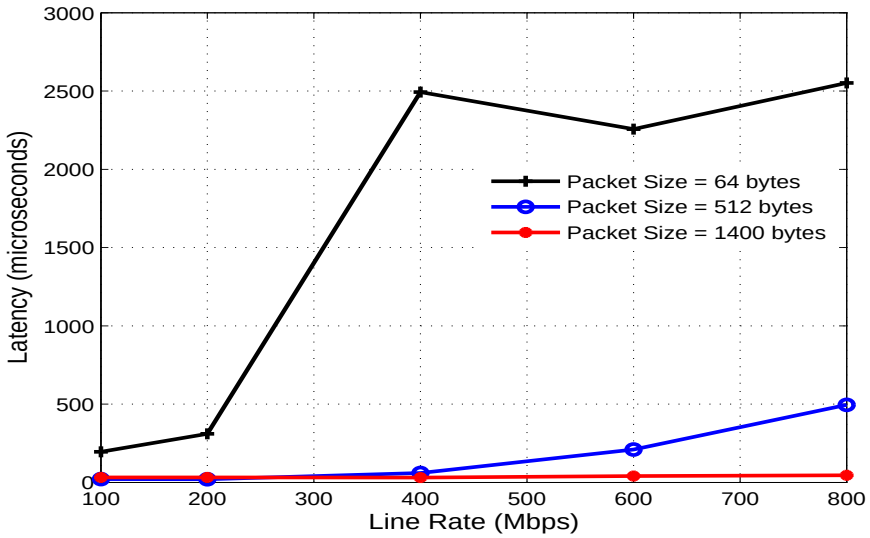

Figure 3.18. Average Packet Latency with rxFIFO = 48MB, rxDescriptors = 512.

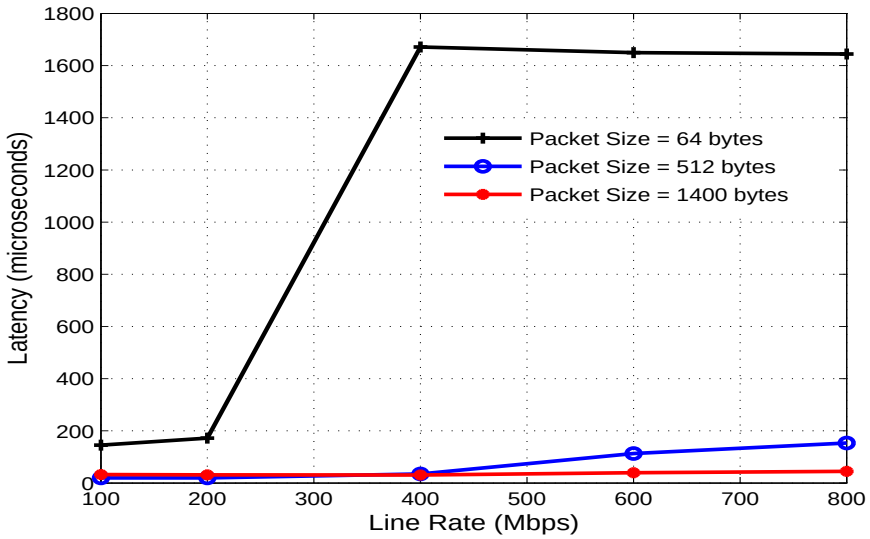

Figure 3.19. Average Packet Latency with rxFIFO = 48MB, rxDescriptors = 128.

bytes packet at we 800 Mbps, we obtain CPU utilization in the range of 90%-95%. The variation is due to the approximation in the degree of accuracy of the factorial values in Equation (3.8). Another aspect worth mentioning at this point is the average number of interrupts generated by the system and plotted in Figure 3.23. It shows the point of

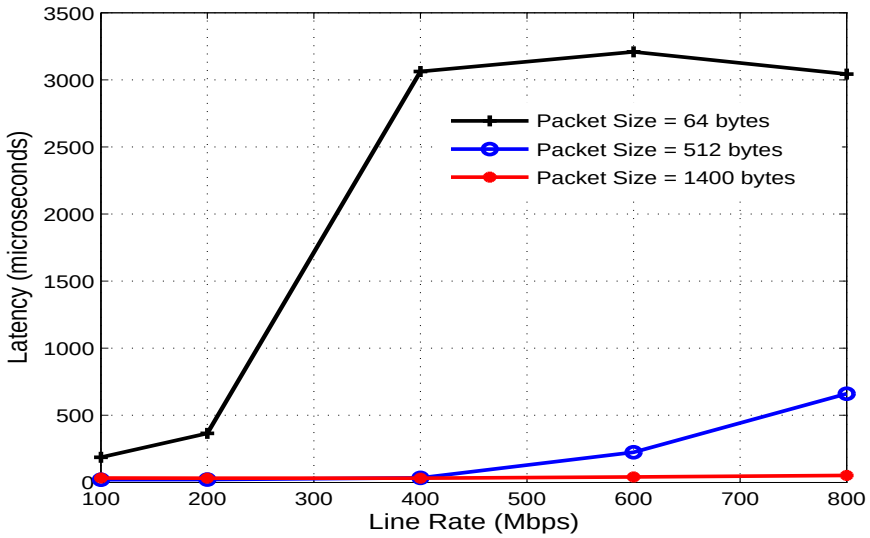

Figure 3.20. Average Packet Latency with rxFIFO = 32MB, rxDescriptors = 1024.

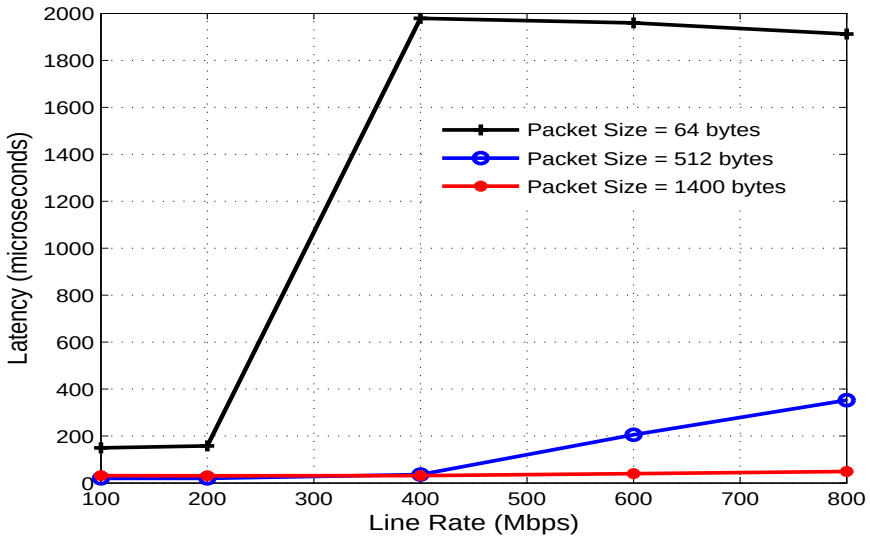

Figure 3.21. Average Packet Latency with rxFIFO = 32MB, rxDescriptors = 512.

activation of NAPI when the line is increased beyond 200 Mbps. The average number of interrupts is high for 64 bytes packet since the NAPI algorithm immediately switches to the interrupt mode whenever it finds the DMA ring empty. This can, however, be

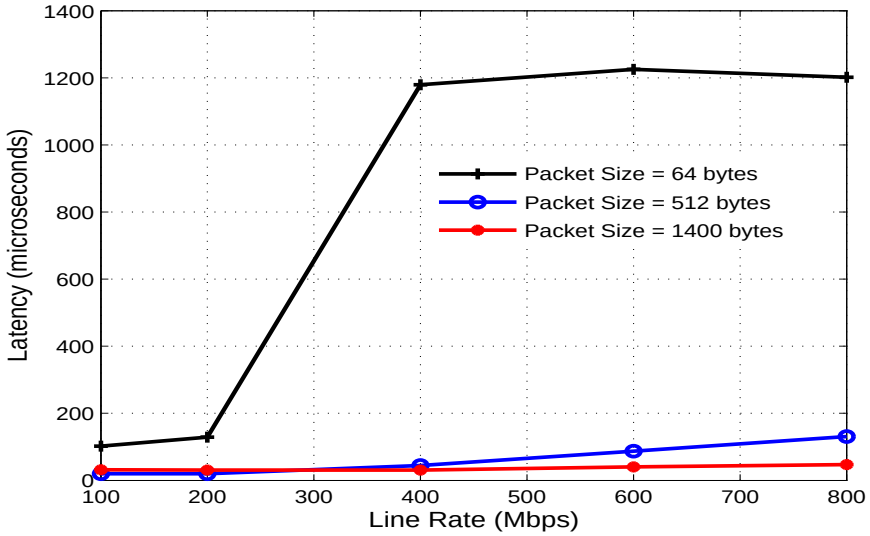

Figure 3.22. Average Packet Latency with rxFIFO = 32MB, rxDescriptors = 128.

improved by introducing lazy transitions wherein transitioning between the NAPI and interrupt mode occurs after a certain interval of time.

3.8.2 Average Bulk Size Distribution

In Figures 3.7, 3.8, 3.9, we plot the average bulk size distribution with variations of packet size at line rates varying between 200 Mbps to 1Gbps. Remember that the maximum bulk size that could be pulled during each invocation of the polling has been fixed at 64 in this study. We observe that with the increase in traffic intensity at the DMA ring, as evident in the case of 64 bytes packet, the proability that the polling function finds the DMA ring filled up increases. This is manifested in the figures by a corresponding increase in the probability of bulk size with increase in line rate, keeping the packet size constant. The value of the bulk size was obtained by modifying the e1000 driver to capture such statistics.

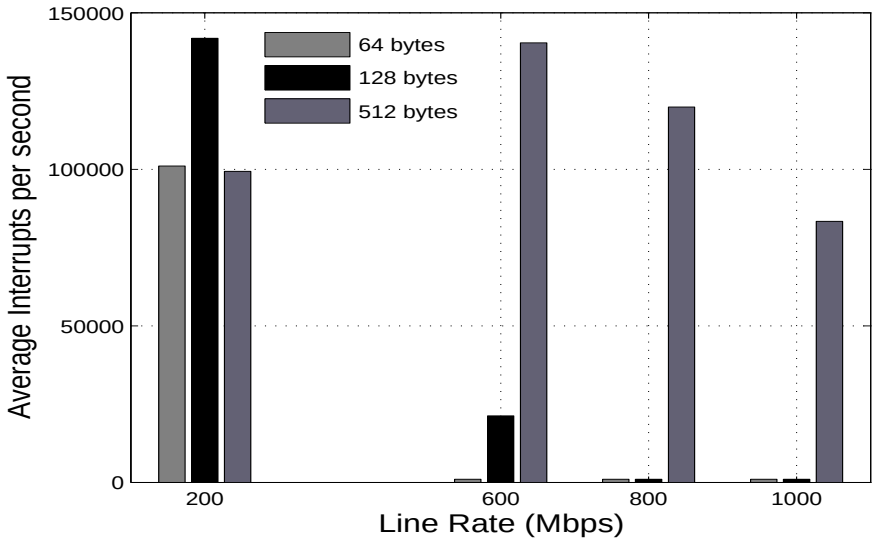

Figure 3.23. Average number of Interrupts generated with rxFIFO = 32MB, rxDescriptors = 128.

3.8.2.1 Average Packet Latency

In Figures 3.17, 3.18, 3.19, we plot the average latency as reported by SmartBits with rxFIFO = 48MB, rxDescriptors = 1024. Both the packet size and the line rates are varied. As expected, the average latency is highest in case of 64 bytes packet as compared to 512 bytes and 1400 bytes. This is due to the fact that 64 bytes provides the highest traffic intensity among the test cases and consequently suffer packet loss at the rxFIFO. This observation is supported by the Figures 3.20, 3.21, 3.22 where increasing the size of the rxFIFO decreases the average latency, if and only if there occurs no packet loss at that line rate. If we plug in the values of the parameters in our analytical model, we obtain: (i) the blocking probability at that specific line rate which can be used to infer the probability of packet loss, and (ii) the average delay of the system. In all the cases, the accuracy between the predicted and experimentally measured values was between 20% error, in the worst case scenario.

3.9 Summary

In this chapter we have presented a closed form queuing model for understanding the dynamics and mechanisms of device polling for packet capture in high speed networks. Since exhaustively evaluating the system capabilities is not feasible, such an analytical framework provides important indications of system bottlenecks and hotspots that might occur when off-the-shelf NICs are interfed with links operating at Gigabit line rates. As a case study, we have focused on the Linux kernel and Intel GigE NICs and CPU.

We have observed that while device polling is an invaluable approach for preventing interrupt livelock at high line rates, it exhibits high CPU usage with increase in packet arrival rate. Also, there occurs non-negligible costs in terms of interrupt generation when the system switches from polling mode to interrupt state due to either: (i) overflow at the on-chip rxFIFO, and (ii) blocking of the I/O bus. Such observations were not reported in earlier work. It indicates that the performance of polling device drivers can be improved by introducing “lazy transitions", wherein transitions between the interrupt and polling modes does not occur immediately but after a certain period of time.

Keeping architectural variations like I/O bus width, CPU frequency, system memory speed as constant parameters, we have also observed that the performance of polling device drivers is heavily dependent on the maximum bulk size that can be serviced by the host CPU.

CHAPTER 4

CONTROLLING WRITE CONGESTION FOR IMPROVING APPLICATION READ PERFORMANCE

Once the traffic profile of the network has been understood and the NIC cards properly tuned, it is important to monitor the average I/O latency experienced by the application services. Since we are dealing with primarily read oriented systems with background I/O jobs, it is important to monitor the average read transactions per second (TPS)[†] of such the systems. Performance tools such as `sar`, `iostat`, `vmstat` along with benchmarking tools `Postmark` [95], `Iozone` [93], `Bonnie` [94], can be used to investigate system disk I/O. These tools report the average I/O traffic of the system taking into consideration the file cache of the system. If it is found that excessive disk I/O is causing the read TPS to suffer, the system is most probably taking a performance hit due to dirty page flushing by the page daemon and excessive write system calls. We call this phenomenon as *write congestion* and is the focus of this chapter.

Write congestion is a phenomenon when the effective TPS of latency sensitive systems, such as streaming multimedia servers, start decreasing in the presence of a large number of writer processes generating bursty workload patterns of disk access. In this chapter, we evaluate the performance of the readers in the presence of the writers under varying transactional load. The negative impact of bursty writes on the performance of the reader processes is explored in the process. In order to control such write congestion, we introduce two approaches: (i) *deterministic*, in which the writers are penalized assuming exponential growth rate in the frequency of generation of dirty pages, and (ii) *stochastic*, where the value of effective bandwidth (EB) for each of the writers is used to

[†]We refer to the execution of a single `read()` or `write()` system call as a single transaction.

identify conditions that might lead to congestion and govern the loss probability of the per–process *virtual buffer* inside the VM.

If physical memory is not the source of bottleneck, the deterministic approach has negligible implementation overhead and is suitable for *lightly loaded servers* with *smooth* `write()` request arrival patterns. On the other hand, the EB based stochastic algorithm has complex decision phases but can handle *large fluctuations* in the arrival process of dirty pages. As such, it is suitable for *demanding servers* catering to a large pool of readers and writers.

As observed in our implementation with NetBSD 3.1, on an average the performance of the readers is observed to improve by 15%-20% as compared to the baseline case of no controlling of the write congestion phenomenon. Also, there exists *strong association* between the dynamics of the page daemon and the performance of the write congestion algorithm proposed in this chapter.

The rest of the chapter is organized as follows. In Section 4.2 we provide an outline of previous work. The correlation between the time scale of measurement and memory in workload data is studied in Section 4.4. We introduce two variants of our algorithm in Sections 4.5 and evaluate their performance in Section 4.6.

4.1 Motivation and Background

In order to function within the bounds of soft real-time constraints, content delivery and processing entities such as streaming multimedia servers and back-end database systems, are especially concerned with managing and improving the average I/O throughput rather than focussing on reducing the average I/O latency of hard disk access. This is in stark contrast to user-oriented systems where interactive response, as compared to I/O throughput, is generally considered to be the performance metric of choice for OS programmers. Consequently, for such read-oriented systems, writes to the disk have lower

priority than reads. This is because writes are usually *asynchronous* in nature and are flushed to the disk (by the kernel) in a lazy fashion; with the primary intension being to defray the total I/O access latency. On the other hand, read operations are inherently *synchronous* and applications are blocked from processing unless the requested data are made available by the kernel in a timely fashion.

In most open-source and commodity operating systems such as xBSD, Linux, Solaris, and Windows, when a process executing on the CPU initiates I/O operation that requires accessing the hard disk, it is removed from the processor work queue and put to sleep pending the outcome of the operation. Two commonly used system calls for accessing the storage medium, *assuming non-availability of data in their respective caches*, are the `read()` and `write()` system calls. However, in a typical running system, both the `read()` and `write()` system calls equally compete for free pages inside the virtual memory (VM) and stress the I/O scheduler and the disk subsystem in a nonuniform and nondeterministic fashion. For most cases, this leads to a phenomenon wherein a large flood of `write()`s[†] (write transactions) decreases the ability of the system to service the `read()`s (read transactions) in a *timely fashion*. We refer to this situation as *write congestion* since the effective throughput of the system is reduced in the face of a barrage of writes.

Such a situation involving write congestion has been recorded and observed in our experiments (see Section 4.6). We have observed that on an average there occurs a decrease in the effective TPS of the reader processes (henceforth referred to as *readers*) when there exists background writer processes (henceforth referred to as *writers*) initiating "large" disk transfers. An instance of such an effect is illustrated in Figure 4.1 where a single reader suffers as much as 45% decrease in effective TPS in the presence of a

[†]Note that deferring and controlling read/write transactions is different from filesystem read/write operations.

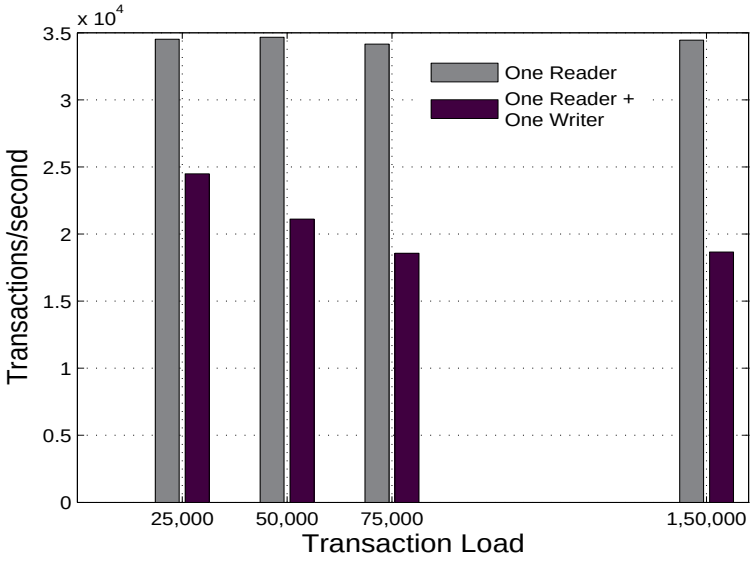

Figure 4.1. Example of decreased Transactions per second (TPS), 45% in the worst case, of a single reader process in the presence of a sole background writer process with varying transactional load. The results were obtained using a modified version of Postmark on a NetBSD 3.1 system with 512MB of RAM, 4KB page size, and read priority (RPRIO) as the I/O scheduling policy.

sole writer transferring a 300MB file to the disk. It should be noted that the decrease in TPS for the readers is not due to scarcity of free pages inside the VM since in all the experiments the system had a left over page pool that varied between 5%-20% (see Figure 4.2). Thus, in none of the instances of the experiment recorded in Figure 4.1, the page daemon got triggered and the `read()` system call penalized for non-availability of memory pages in the VM.

At an initial glance, it appears that the correct way to reduce such unwanted write congestion is to use some sort of read priortized I/O algorithm with aging of the write requests so as not to starve them. Unfortunately, changing the I/O algorithm alone will not suffice since the phenomenon of write congestion causes free memory pages to be rapidly consumed by the writer processes. Thus, even if the readers are scheduled at the I/O level with minimal delay, the effective `read()` service time still suffers since the VM

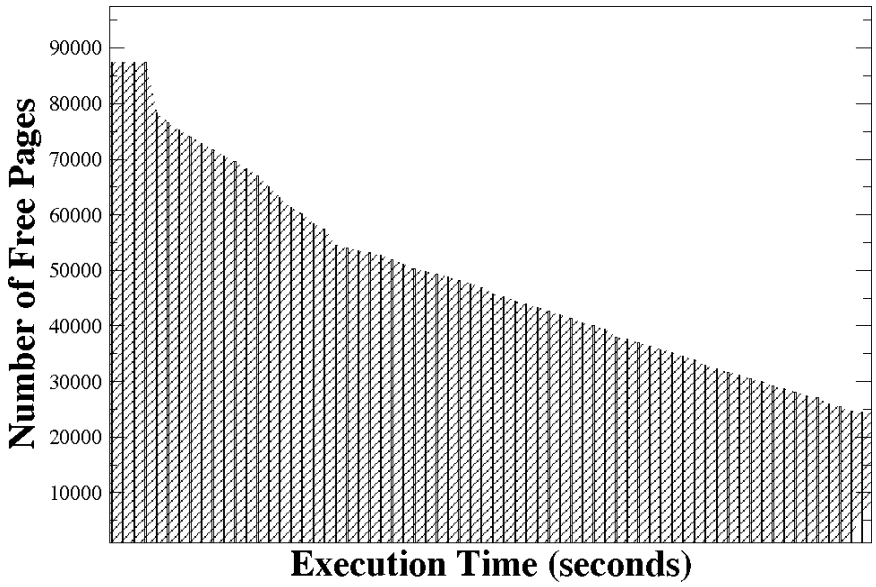

Figure 4.2. Temporal variations of the page pool inside the Virtual Memory (VM) during a typical run of the experiment of Figure 4.1. Notice the availability of free pages inside the VM during the lifetime of the experiment.

has to locate a free page into which the request can be read into. To make matters worse, if the number of free pages available in the system drops below a specific threshold, the VM schedules the page daemon to run with the highest exection priority. This, in turn, causes additional performance penalty since no reader/writer process can be serviced while the page flushing is in progress.

One of the possible ways to control such write floods is to enfore some kind of congestion control policy inside the VM so that majority of the free pages are not consumed by the writers. At the same time, the congestion control algorithm must be smart enough not to throttle reliability-based writes such as journaling logs and metadata updates related to the underlying file system. Furthermore, as write accesses require *exclusive locks* over reads which can use *share locks*, the congestion control algorithm must be careful not to overly prefer the readers in order to prevent write starvation.

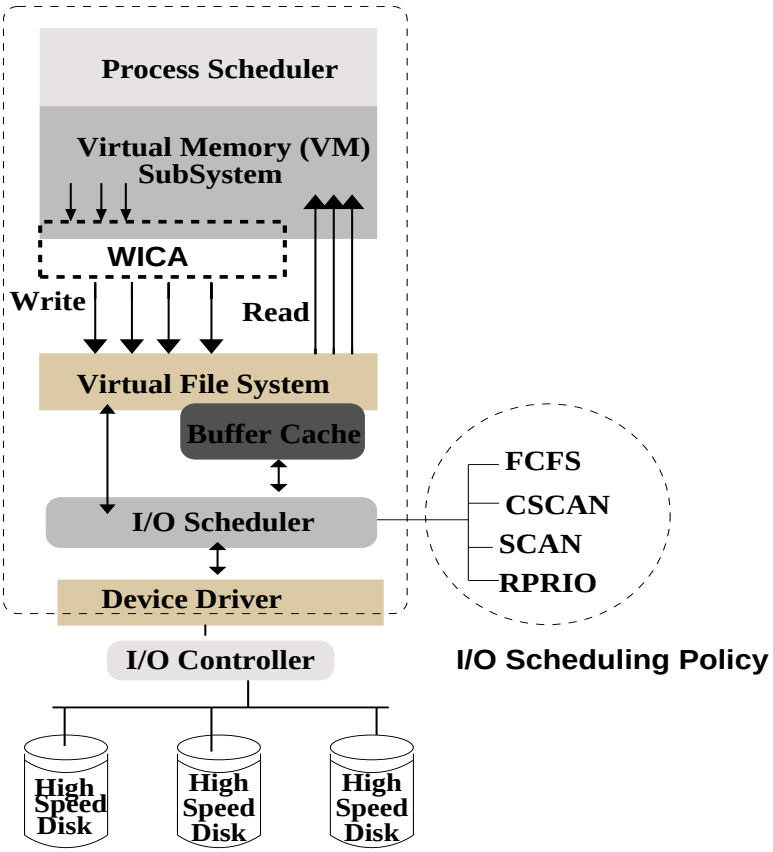

Figure 4.3. High Level diagram showing the location of the Write Congestion Indication Algorithm (WICA) inside the Virtual Memory (VM). Notice that WICA traps and monitors the page flushing of the writer processes only.

In this chapter, we present a ***WrIte Congestion indication Algorithm*** (WICA) that resides inside the VM and is capable of detecting and alleviating against sudden bursts of I/O write activity so as to balance and skew the disk access towards the readers. The location of WICA inside the VM is shown in Figure 4.3. Whenever a writer is observed to generate a burst of dirty pages that might give rise to write congestion, WICA kicks in and puts the process to sleep for a definite duration of time that depends on the dynamics of the activity of the writer process. Making the writers sleep for a *bounded period* of time is attractive since apart from slowing down the rate of generation of dirty pages and

filling the underlying I/O queue inside the I/O scheduler, it also indirectly eases pressure on the page allocation routine of the VM subsystem.

Thus, loosely speaking, WICA can be used to provide fairness among readers and writers competing for disk bandwidth. At this stage, it should be noted that WICA monitors the dynamics of dirty page flushing only for processes that engage the I/O subsystem in some form or fashion. However, in a running system, there also exists processes that are created in the system memory with no I/O activity whatsoever. Such pure memory based processes are not monitored by the current version of WICA. Consequently, there exists a certain possibility that WICA will not be able to favor the readers over the writers if most of the free pages are held by such memory based processes. However, such processes are ephimeral in nature and experimental evidence suggests that their role in impacting the performance of WICA is negligible.

4.2 Approaches that Impact the Performance of I/O Workloads

Approaches aimed at improving the performance of I/O operations for increasing the application throughput can divided into two categories:

- *Efficient buffer cache management schemes:* The primary aim of all buffer cache management schemes is to keep the pages belonging to the working set of an application in system memory. Some of the work incude page replacement strategies such as the well known Least recently Used (LRU) and its variations, Low Inter-reference Recency Set (LIRS) [30], Adaptive Replacement Cache (ARC) [49], Dual Locality (DULO) [31], and others. The LIRS algorithm improves upon the weak performance of LRU in scenarios of access patterns with poor locality by maintaining a record of the number of blocks accessed bewteen two consecutive accesses to the same block. This is referred to as the Inter-Reference Recency (IRR) of a block. Consequently, buffer eviction occurs on the basis of the block IRR value. The ARC

algorithm tries to track temporal locality by maintaining both the 'recency' and 'frequency' of cache access pattern. Since it is well known that random seek of the disk heads are constly maneuvers, the DULO algorithm tries to make the I/O requests more sequential in nature by considering both the temporal and spatial locality of page access patterns by the applications.

- *Intelligent data prefetching and caching schemes:* Heuristics that detect sequential block access patterns of an application and employ techniques that are able to prefetch the block before the request hits the buffer cache. Such approaches have been studied in [55][72]. Variants have been proposed in [57][42] that take into account the system wide file access history with probability of occurence of a specific pattern in the history. In[36], based on the LRU access pattern, blocks are fetched from the disk. In [28], the buffer cache is adaptively partitioned into random and sequential spaces based on the access patterns of the disk. The random space utilizes on-demand data acess while the blocks are prefetched in the sequential space.
- *Smart I/O request scheduling techniques:* In [69] is described an extensive survey of the Earliest Deadline First (EDF) scheduling algorithm commonly used in most real-time systems. The performance analysis of EDF presented in [34]. The SCAN [16] scheme mimics the elevator algorithm and aims to reduce the disk seek and rotation time. However, since SCAN is not sensitive to the timeline of the pending jobs, priority-based algorithms such as SCAN-EDF [60], SCAN-RT [35] and others have been proposed in the literature. All of them take into consideration the deadlines of the service tasks but suffer from extremely high disk-seek time. The SCAN-WRR [73] algorithm takes into account both the deadlines of real-time tasks while providing high disk throughput. A summary of various look ahead disk scheduling algorithms is presented in [71].

However, all the strategies identified above fail to work if the VM suffers from the phenomenon of write congestion identified in this study. To the best of our knowledge, no work has been reported in the literature that exposes the negative impact of write floods on the performance of the readers in a running system. We believe that our WICA approach serves to complement the existing work in I/O prefetching and caching by easing pressure on the VM system and the I/O scheduler. Since disk prefetching works on the surmise that logical file system blocks also occur in close physical proximity on the hard disk, there might be performance hit if such an assumption is invalidated. This can happen with the rising popularity of multi-disk environments employing volume managers. Situations might become worse if the rate of write requests or worse scatter write requests arriving at the I/O scheduler exceed the average I/O service time. Under such circumstances, WICA can help by slowing down the writers, within limits, so that the I/O scheduler does not become the source of bottleneck. It is worth repeating that WICA is not a generic approach suitable for all systems. Only high performance *read-oriented* systems such as streaming media servers, web servers, and other content delivery systems are expected to benefit from our approach.

4.3 I/O Scheduling Algorithms

The dynamics of reading and writing to the underlying block device is affected by nature and type of the I/O scheduling algorithm (also referred to as disk sorting algorithms). We start with a brief survey of the various disk sorting algorithms currently used in commodity OS such as NetBSD 3.1. Performance evaluation details are available in [65].

- **First Come, First Served (FCFS):** This is the simplest disk sorting algorithm where requests are served in the order of arrival at the I/O scheduler. Consequently, it is *independent* of the I/O queue size. If there occurs little spatial locality in

the requests, FCFS results in undue penalty in head movement of the disk and can result is very low disk utilization factor. However, the algorithm is fair to all requests and requires only one queue for maintenance.

- **SCAN:** The SCAN [16] or the "elevator" algorithm maintains two queues of arriving requests, sorted in ascending order of the block numbers. Requests positioned after the current block are placed on the first queue while those requests which came in after the head has passed their position, are placed on the second queue. Thus, at any instance of time the disk head services requests along one direction only. Hence on an average, requests in the middle of the disk experience better service time than those at the edge since the head passes over the center twice as often as compared to the edges. It has been observed that the average disk utilization *increases* with increase in the I/O queue length.
- **Cyclical SCAN (CSCAN):** Similar to SCAN, CSCAN [64] maintains two queues of arriving requests sorted in ascending order of the block number. However, when the head reaches the edge of the disk, instead of retracing its path, the head repositions itself to the first cylinder. Hence, there is no tendency of preferential servicing of requests which lie in the middle of the disk. For queues with large number of requests, the performance of SCAN and CSCAN are nearly similar [65].
- **Read Priority (RPRIO):** This algorithm also maintains two queues which are distinguished based on the *nature of the request*: read (read queue) or write (write queue). Requests in the read queue are serviced in the FCFS order, while those in the write queue are sorted in ascending order of block numbers. The algorithm alternates between the two queues and services them in bulk; the read batch size is larger than the corresponding write batch and is exposed as a tunable parameter.

Thus, the first step is to choose the most appropriate I/O scheduling algorithm based on the requirements. For our case, it is always the RPRIO algorithm. Next, we see the behavior of the underlying process that is generating the `write()` requests.

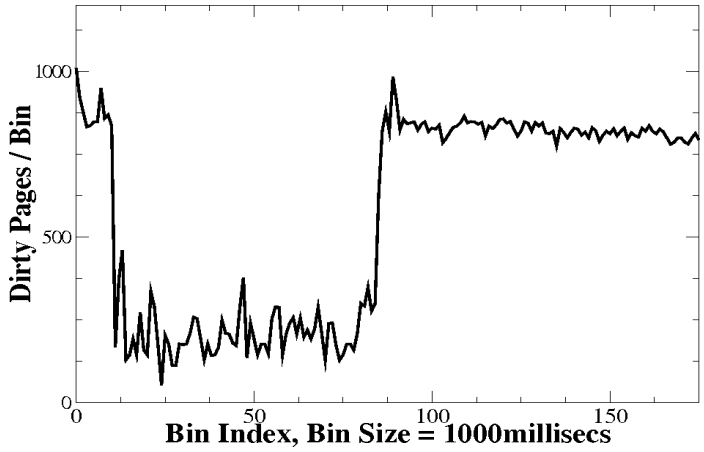

Figure 4.4. Stochastic "Burstiness" in the rate of generation of dirty pages for sampling bin size of 1000ms.

4.4 Dynamics of Page Flushing Process

The effectiveness of WICA is influenced by two main factors: (i) the nature of *burstiness* of individual process behavior, and (ii) the *statistical multiplexing* of the dirty pages being flushed from the VM. Since there exist strong correlations between the nature of fluctuations of the variance and the effectiveness of congestion control and queuing mechanisms [23], it is important to understand the variations of auto-correlation function (ACF) in the presence of multiple readers and writers. From Figures 4.4-4.11, it is apparent that there occurs sizeable "burstiness" in the manner pages are flushed from the VM. Furthermore, the degree of burstiness depends on the sampling interval. This motivates us to concentrate on examining the presence of long-range dependence (LRD) [46] among the datasets considering different sampling time window of observation.

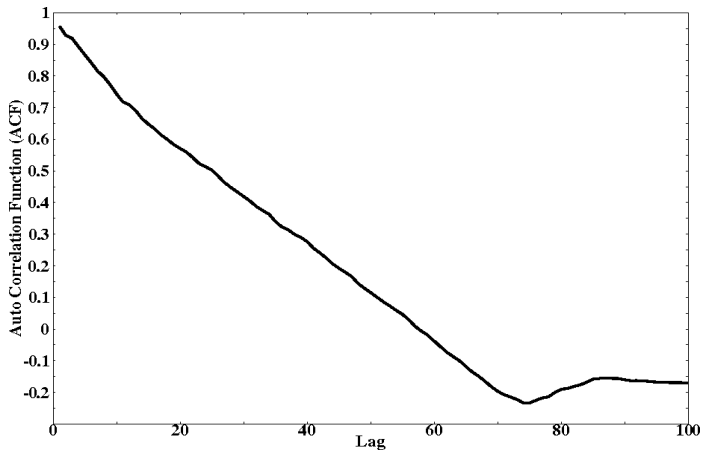

Figure 4.5. Rate of decay of Auto Coorelation Function (ACF) for sampling bin size of 1000ms. Notice the high value of ACF.

The autocorelation function (ACF), $\gamma(k)$, is a measure of similarity between the time sequence $X_{n(t)}$ and a time shifted version of itself at time lag k, denoted as $X_{n(t+k)}$. The ACF is defined as [53]:

$$\gamma(k) = \frac{E[(X_t - \mu)(X_{t+k} - \mu)]}{\sigma^2}$$

where μ and σ^2 are the mean and variance of the sequence $\{X_{n(t)}\}$. The value of $\gamma(k)$ varies between $+1$ to -1 and the value of 0 indicates that the series is independent at lag k. On the other hand, an absolute value of close to 1 indicates presence of "memory" in the time series.

Based on the above disscussion, from Figures 4.5 and 4.7, it is evident that the time sequence of Figures 4.4 and 4.6 with sampling interval of $1000ms$ and $100ms$ are *heavily correlated* (i.e., $\gamma(k)$ has a power law decay of the form $\gamma(k) \sim k^{2\mathcal{H}-2}$, for $0.5 < \mathcal{H} < 1$, H is the Hurst parameter) and has considerable "burstiness" or variability. Thus, even after multiplexing the dirty pages across different writers, it is not possible to smooth

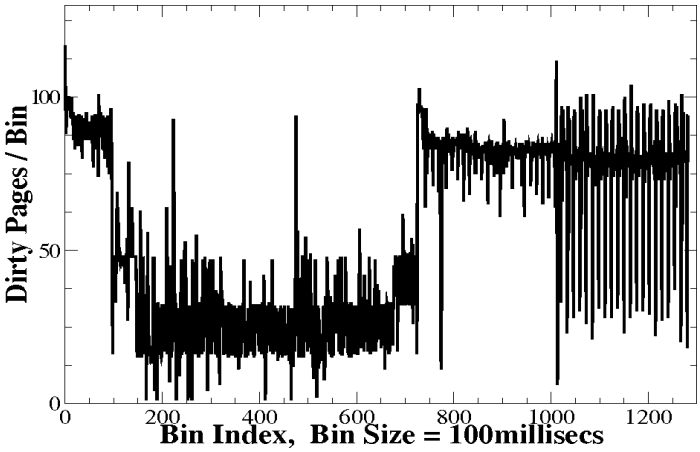

Figure 4.6. Presence of "Burstiness" in the rate of generation of dirty pages at reduced sampling bin size of 100ms.

out the peak behavior. This implies that at time scale resolution of $1000ms$ and $100ms$, WICA needs to account for the presence of long-range dependency (LRD) in order to correctly estimate congestion level. However, taking into account the LRD factor makes the analytical formulation intractable since determination of the Hurst parameter in $\gamma(k)$ variation is both complex and cumbersome.

On the other hand, as seen from Figures 4.9 and 4.11, there exists *limited correlation* across samples for the time series of Figures 4.8 and 4.10. This indicates that at *low time scale resolution* (less than $20ms$ in our experiments), the process becomes considerably smoother or exhibits short-range dependence (SRD) [46], with smaller variances and deviations. Thus, the ACF, $\gamma(k)$, *decays exponentially* instead of hyperbolically and the sampling instances can be considered to be independent.

Note that we have purposefully focused on the variation of $\gamma(k)$ for individual writers since greater variability in the multiplexed stream *does not* imply greater individual process variability. Consequently, the time scale of measurement plays an important role in understanding the underlying stochastic behavior, as also elaborated in [53].

From the above discussion, it is apparent that *although the page flushing process is*

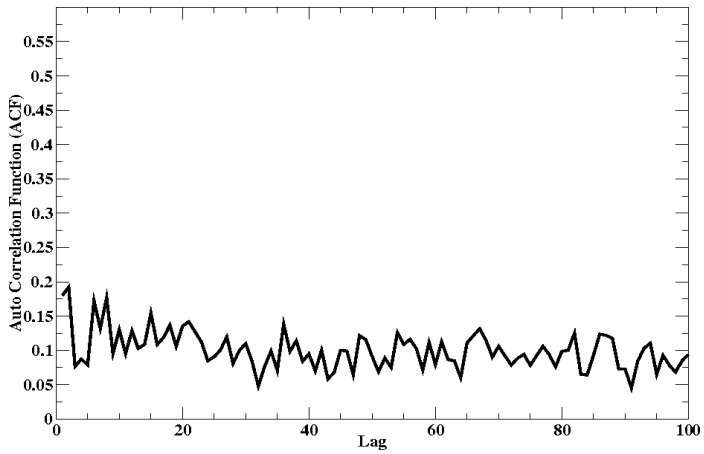

Figure 4.7. Rate of decay of Auto Coorelation Function (ACF) for sampling bin size of 100ms.

not Poisson, it also does not exhibit self-similar behavior. This implies that it is possible to accurately predict the average rate of dirty page generation if the sampling interval is made considerably "small". At such small time scales with low value of ACF, it is appropriate to use *systematic sampling with random start time* [47].

4.5 Proposed WICA Algorithms

We begin this section by looking at two variants of WICA algorithm: (i) deterministic rate based WICA (or D-WICA) where the rate of growth of dirty pages is proportional to the number of dirty pages generated, and (ii) probabilistic WICA (or P-WICA) which is based on the concept of effective bandwidth (EB) where each writer is visualized as a set of fluid On-Off writers. In the rest of the chapter, each individual writer process is designated as w_i. Table 2.1 summarizes the notations used in developing our WICA algorithms.

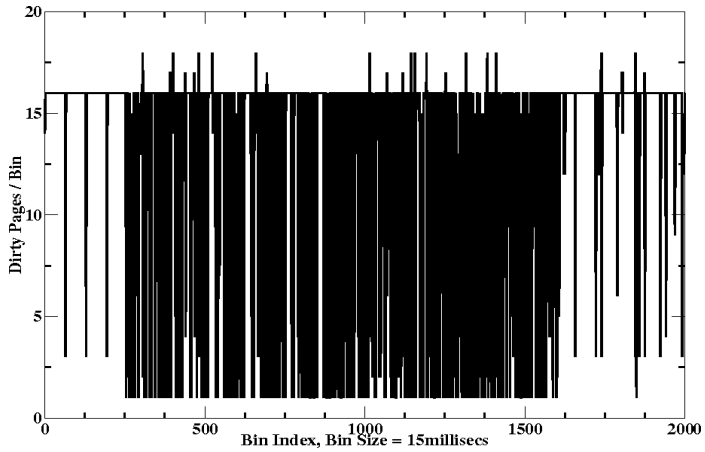

Figure 4.8. Disappearance of "Burstiness" in the rate of generation of dirty pages for sampling bin size of 15ms.

Table 4.1. Notations Used in WICA

Notation	Meaning
w_i	Writer process i
$\delta(T)$	Sampling time time window for D-WICA
$\gamma(k)$	Auto-correlation function (ACF) at time lag k
N_T^{dirty}	The number of dirty pages generated by a writer process
R_i^d	Rate of generation of dirty pages by writer process i
r_i^j	Instantaneous rate of generation of dirty page for process i and time instant j
β	EWMA smoothing constant
M_d	Total number of dirty pages collected during a measurement interval
p_d	Probability of generation of a dirty page
$P_{M_d}(k_d)$	Probability of collecting k dirty pages out M_d
S_i	Duration of sleep for writer process i
L_{sleep}^{max}	Max. sleep duration
σ_i	Penalty factor for writer process i
σ_i^{th}	Max. penalty or threshold
$A_i^d(t)$	Max. no. of dirty pages generated by w_i during time interval $[\tau,\tau+t]$
B	Fictitious buffer size for calculating effective bandwidth
$\alpha_i(s,t)$	Effective bandwidth for writer process i with space factor s and time factor t
ϵ	Probability of overflow of buffer B
EB_{th}	Maximum effective bandwidth
R_{peak}^d	Peak rate of dirty page generation of an equivalent Poisson process

4.5.1 Deterministic WICA: Rate based Approach

Consider a systematic sampling time window $\triangle T$ (see Figure 4.13) during which the birth rate, R_i^d, for writer w_i is assumed proportional to the number of dirty pages

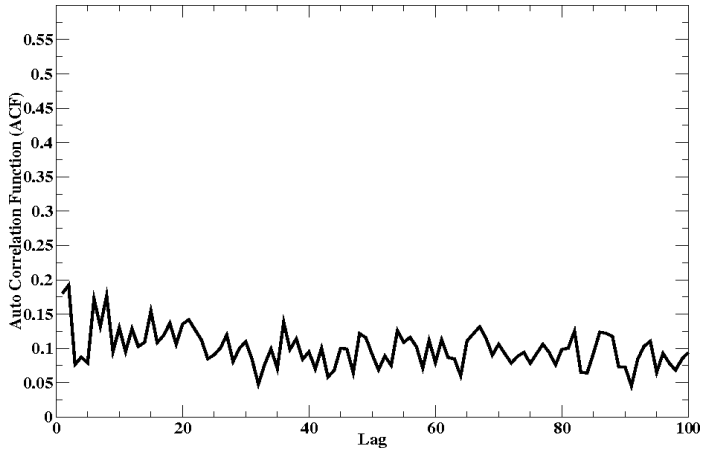

Figure 4.9. Rate of decay of Auto Coorelation Function (ACF) for bin sampling bin size of 15ms.

generated by it during the sampling interval. Assume the rate remain constant during the time interval $\triangle T$. Let $N_{(T)}^{dirty}$ and $N_{(T+\triangle T)}^{dirty}$ be the total number of dirty pages generated by writer w_i, at the end of time instants T and $(T + \triangle T)$. Then we have:

$$\frac{dN^{dirty}(t)}{dt} = R_i^d N(t), \text{ for } T \leq t \leq (T + \triangle T) \tag{4.1}$$

$$\Rightarrow N_{(T+\triangle T)}^{dirty} = N_{(T)}^{dirty} e^{R_i^d \triangle T} \tag{4.2}$$

In the above equation, the rate R_i^d varies for each of the writers and it is an unknown parameter. Hence, during initialization phase of the D-WICA, the rate is estimated using certain learning samples.

4.5.2 Estimating R_i^d for w_i

Consider a systematic sampling instance j. Denote R_i^j to be the smoothed rate of generation of dirty pages for writer w_i and r_i^j be the instantaneous value during the sam-

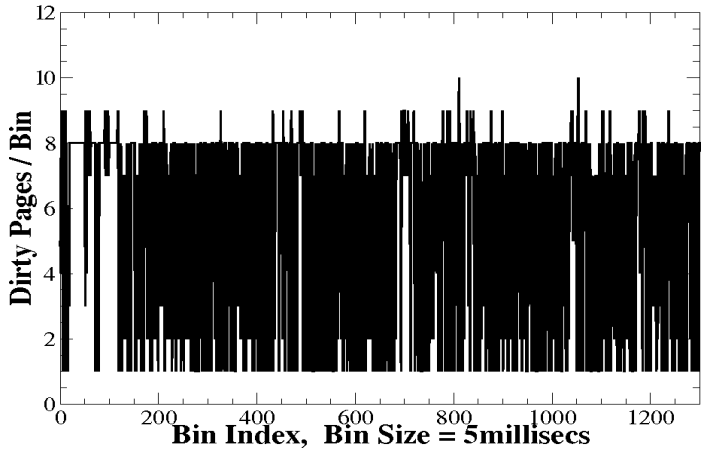

Figure 4.10. No "Burstiness" in the rate of generation of dirty pages for sampling bin size of 5millisecs.

pling instant j. Then, applying *first order exponential smoothing*, we have the following relationship:

$$R_i^j = \beta r_i^j + (1 - \beta) R_i^{j-1} \tag{4.3}$$

Equation (4.3) indicates that while the contribution of current sampling is governed by the parameter β, past history at sampling instant $(j - h)$ contributes $\beta(1 - \beta)^h$. As argued in the previous section, if we make the value of $\triangle T$ sufficiently small, we can considerably reduce the length of the history h. Consequently, the value of β can be made *large enough* (e.g., 0.7) in order to give more weight to the short term trend.

4.5.3 Duration of Learning Period

In order to apply Equations (4.1) and (4.2), it is important to gain an estimate of the initial value of the rate R_i^d, during which the S-WICA is not made active (learning phase of operation). The duration of time over which the initial value of R_i^d is estimated is referred to as the *learning period*.

Let M_d be the total number of dirty pages generated during a certain period of operation. Let us also assume the samples are independent due to SRD and let p_d

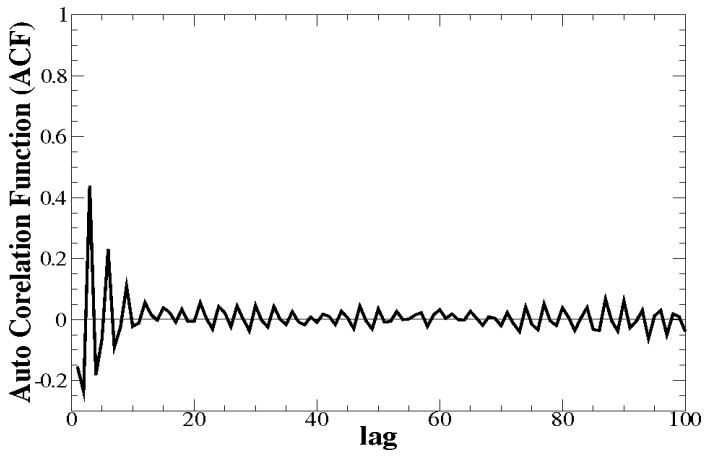

Figure 4.11. Rate of decay of Auto Coorelation Function (ACF) for bin sampling bin size of 5ms.

be the probability of occurrence of a dirty page. Then, the probability, $P_{M_d}(k_d)$, of collecting k_d out of M_d dirty pages due to systematic sampling, can be modeled as a binomial distribution. That is, $P_{M_d}(k_d) = \binom{M_d}{k_d} p_d^{k_d}(1-p_d)^{M_d-k_d}$. We are interested in collecting η_d pages (sample size) out of the total M_d (population size) dirty pages that have generated, assuming systematic sampling. The confidence limit (CL_d) for such an event is given by:

$$CL_d = P(\eta_d \geq k_d) = 1 - \sum_{i=0}^{k_d} P_{M_d}(i) \approx \sum_{i=0}^{k_d} \frac{(M_d p_d)^i}{i!} e^{-M_d p_d} \tag{4.4}$$

Note that the *accuracy* of Equation (4.4) depends on the *duration* of the learning period. The longer the learning phase, the better is the CL but at the cost of longer startup time. Hence, given a desired CL, our aim is to determine the *minimum number* of dirty pages to be collected before the WICA can be started with rate R_i^d.

Equation (4.4) can be rearranged to calculate the value of k_d and provides an

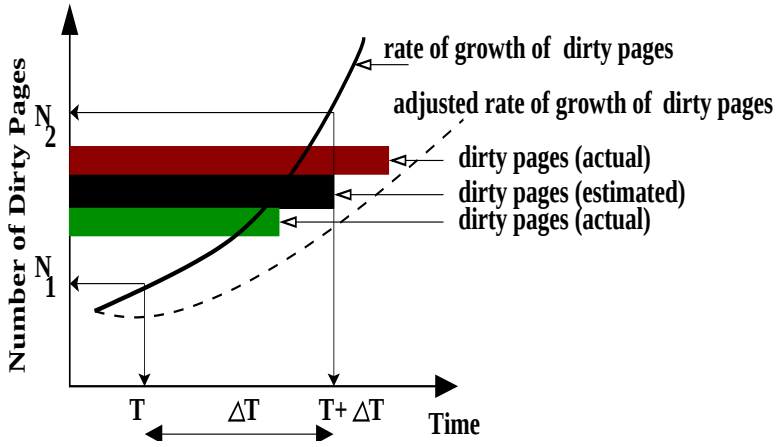

Figure 4.12. In deterministic rate based modeling technique (D-WICA), the rate of growth of dirty pages is assumed proportional to the number of dirty pages generated.

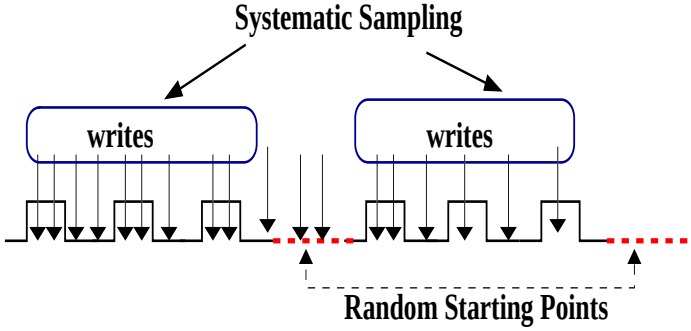

Figure 4.13. Systematic sampling, with random start, in order to estimate the rate of generation of dirty pages by the writer processes. Note that such a sampling strategy provides accurate estimation of the rate only when the ACF is close to zero.

approximate guideline for the number of dirty pages that needs to be collected in order to achieve the desired CL.

$$M = \left\lceil \frac{\ln\left(\sum_{i=0}^{k_d} \frac{(M_d p_d)^i}{i!}\right)}{p_d} \right\rceil + \left\lceil \frac{-\ln(1 - CL_d)}{p_d} \right\rceil \tag{4.5}$$

For an approximate solution of Equation (4.5), suitable numerical methods can be used.

4.5.4 Congestion Control Approach

Once the value of R_i^d has been estimated, it is possible to apply Equation (4.2) for understanding and controlling the degree of congestion being initiated by writer w_i. The basic approach is as follows: at the end of the sampling period $\triangle T$, the total number of dirty pages generated by w_i during the sampling interval $\triangle T$ is noted. Let this be $\mathcal{N}_{(T+\triangle T)}^{dirty}$. This is compared against the value obtained from Equation (4.2). If $\mathcal{N}_{(T+\triangle T)}^{dirty} < N_{(T+\triangle T)}^{dirty}$, w_i is assumed to be operating within limits. Otherwise, w_i is declared to create write congestion and is forced to sleep for a duration, $\mathcal{S}_i$, which is given by:

$$\mathcal{S}_i = \left[\frac{\mathcal{N}_{(T+\triangle T)}^{dirty} - N_{(T+\triangle T)}^{dirty}}{N_{(T+\triangle T)}^{dirty}} \right] * L_{sleep}^{max} \tag{4.6}$$

where L_{sleep}^{max} is the default sleep duration and a tunable parameter. In our experiments, it was set to $1500ms$. Although Equation (4.6) is able to detect write congestion for w_i, it is prone to system instability wherein a recently waken up writer generates bursty workload and is again put to sleep. Such open-loop feedback instability can be avoided by introducing a *penalty factor* (σ_i) in Equation (4.6), with σ_i^{th} being the maximum value or threshold. Thus, Equation (4.6) becomes:

$$\mathcal{S}_i = \left(\frac{\sigma_i}{\sigma_i^{th}} \right) \left[\frac{\mathcal{N}_{(T+\triangle T)}^{dirty} - N_{(T+\triangle T)}^{dirty}}{N_{(T+\triangle T)}^{dirty}} \right] * L_{sleep}^{max} \tag{4.7}$$

Our experimental observation indicates that setting $\sigma_i^{th} = 30$ is sufficient for most systems.

In the deterministic rate based congestion control approach, we considered the scenario where the estimated rate of growth of dirty pages for writer process i w_i was assumed to be exponential. The growth rate was measured over a certain history of sampling instances that is dependent on the time scale of measurement. At very "small"

time scales, the collected samples can be assumed to be independent (SRD). On the other hand, at "large" time scales there exists significant correlation between the collected samples (LRD). Once the writer process crosses the estimated number of dirty pages during the sampling time interval, it is penalized by forcibly putting it to sleep for a certain time duration. This sleep duration, in turn, is proportional to the difference between the estimated and observed number of dirty pages. Since the rate based approach is essentially an open-loop control system, a penalty factor was introduced for "dampening" the system.

4.5.5 Limitations of D-WICA

Unfortunately, such deterministic rate based approach suffers from the following shortcomings that might reduce its effectiveness in the presence of a large number of readers and writers processes.

- In general, it is *difficult to determine* the sampling duration that makes the workload short-range dependent (SRD). Thus, random sampling at arbitrarily chosen time scales, might inaccurately estimate the rate of the writer process.
- D-WICA is unable to *quickly detect and react* to temporary bursty phases of dirty page generation since: (i) the burst rate might be worse than exponential, and (ii) due to memory or long range dependence (LRD) and non-linear behavior in the workload generation process, there might be *hysteresis loops* that would increase the settling time of such a rate based algorithm.

Thus, instead of focussing on the average and peak rates of dirty page generation (i.e., the extremes), a different approach would be to estimate the resources that a single process is consuming in the face of bursty workloads. Such an approach is not only more accurate but also enables us to do away with the complexity of sampling bin width and hysteresis

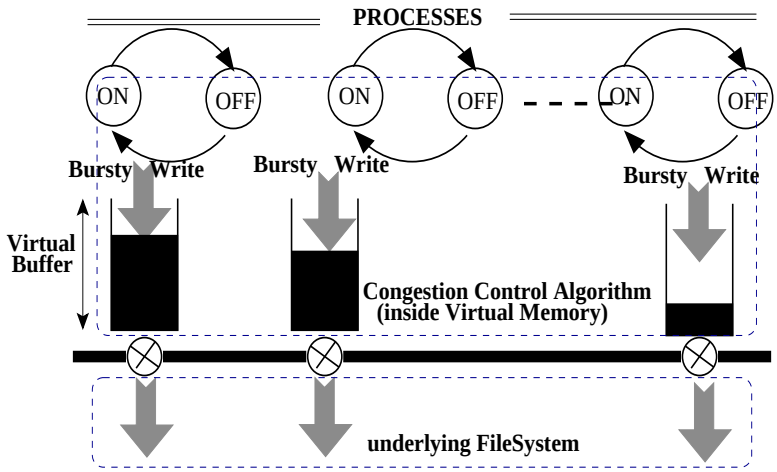

Figure 4.14. On-Off fluid flow model for determining the effective bandwidth of each of the writer process.

effect as explained above. Such resource utilization and corresponding service guarantee of this sort is explained in the theory of *effective bandwidth* (EB) [38].

4.5.6 Probabilistic WICA (P-WICA): Effective Bandwidth based Approach

In this stochastic approach, we use the theory of EB in order to find good approximations of the probability of overflow of the *virtual buffer* (see Figure 4.14) for each of the writers. We use the term "virtual buffer" since inside the VM the writers do not actually manage any real queue of dirty pages. Such fictitious FIFO queue, maintained by each of the writers, provides early warning about write congestion since the effective capacity of the virtual buffer is kept smaller than the capacity of the underlying I/O buffer insider the I/O scheduler (see Figure 4.3).

4.5.7 Effective Bandwidth (EB) and Smoothing

The theory of EB has been widely studied and used for providing statistical guarantee in variable bit-rate traffic (see [38] [70] [4]). Here we only provide the details relevant to our work.

Let $A_i^d(t)$ be the number of dirty pages generated by w_i during time interval $[\tau,\tau+t]$. Assuming the workload stream to be stationary and a large virtual buffer capacity, B, the effective bandwidth, $\alpha_i(s,t)$ for w_i is defined as [38]:

$$\alpha_i(s,t) = \lim_{t\to\infty} \frac{1}{st} \log\left[E(e^{sA_i^d[\tau,\tau+t]})\right] \tag{4.8}$$

In Equation (4.8), the parameter s (the space axis) and t (the time axis) define the operating point of the virtual buffer. For any writer process w_i, the value of $\alpha_i(s,t)$ fluctuates between the mean rate ($s = 0$) and the peak rate ($s \to \infty$) of generation of dirty pages. Thus, if the underlying stochastic process is smooth (low index of dispersion), $\alpha_i(s,t)$ would have values close to the mean rate; however, for bursty workloads (significantly high index of dispersion), it would experience significant deviations from the mean rate.

For a fixed value of t, the effective bandwidth is a function of s and characterizes the decay rate of the virtual queue length distribution in the buffer. Hence, for a virtual buffer with capacity B, the probability of overflow (asymptotic tail distribution), $P_{overflow}$, is given by:

$$P_{overflow} = P(\phi \geq B) \leq \epsilon \approx e^{-sB} \tag{4.9}$$

where ϕ is the number of dirty pages generated by writer w_i and ϵ is the probability of overflow of the virtual buffer and is a tunable parameter.

The value of $\alpha_i(s,t)$ is generally *estimated* using a suitable estimator for a given trace of a workload [4]. Since the VM operates within real-time constraints, it is of vital importance that the calculation of $\alpha_i(s,t)$ does not introduce additional delay and CPU overload in the system. In our study, we use the recursive on-line block algorithm

of [75] for estimating $\alpha_i(s,t)$ for any measurement instant (m) and observation interval (t). With $m = 1$ and $M_1 = 0$ as initial conditions, we have from [75]:

$$M_m = \left[(1 - \frac{1}{m-1})M_m + \frac{1}{m}exp^{sX[(m-1)t,mt]}\right] m = m + 1 \tag{4.10}$$

$$\alpha_m(s,t) = \frac{1}{st}[\ln(M_m]) \tag{4.11}$$

Equation (4.11) provides an estimate of the effective bandwidth at the end of any measurement instance m. Two things are worth mentioning here: (i) the recursive nature of Equation (4.10) makes it possible to calculate M_m over the lifetime of writer process w_i with negligible overhead, and (ii) considering the complexity of saving the floating point registers in the kernel, we consider the logarithms to be of base 2 in our implementations.

There are two performance criteria of the virtual buffer that can be used to observe and ease write congestion inside the VM: (i) the *delay parameter*, which places an upper bound on the average queuing delay, and (ii) buffer *loss constraint* governed by Equation (4.9). In this study, we use the loss constraint, ϵ, for a fixed buffer size B. It should be noted that with FIFO queuing discipline and infinite-buffer approximation, the delay performance criterion translates into an equivalent loss probability requirement [4]. Under such conditions, the operating point s (the space parameter) can be determined as:

$$P_{overflow} \leq \epsilon; \Rightarrow e^{-sB} \leq \epsilon \tag{4.12}$$

$$\Rightarrow s \geq \left(\frac{-\ln \epsilon}{B}\right) \tag{4.13}$$

Consequently, using Equations (4.11) and (4.13), the effective bandwidth is estimated as:

$$\alpha_m(s,t) = \underbrace{B(\ln \epsilon)}_{constant} * \left[\frac{\ln(M_m)}{t}\right] \tag{4.14}$$

Since the average queuing delay is $1/(s\alpha_m(s,t))$ with average queue size $1/s$, it is approximately equal to $t/(\ln M_m)$ at observation instant m.

Since it is difficult to determine *apriori* the effective bandwidth of any workload stream without assuming any statistical distribution, we employ *smoothing techniques* such that the resultant effective bandwidth is kept within limits of the peak rate ($\mathcal{R}^d_{peak}$) of an *equivalent Poisson process* [39]. We refer to this value as the threshold effective bandwidth limit, EB_{th}. Thus, at the operating point s, we have the following relations:

$$EB_{th} = \mathcal{R}^d_{peak}\left(\frac{e^s - 1}{e^s}\right); \mathcal{S}_i = \left[\frac{\alpha_i(s,t) - EB_{th}}{EB_{th}}\right] * L^{max}_{sleep} \tag{4.15}$$

4.5.8 Congestion Control Approach

Based on the above discussions, we can summarize the probabilistic WICA as follows: Each writer w_i maintains a virtual buffer with a fixed capacity, B, which is less than the capacity of the underlying I/O bufer, $\mathcal{B}_{io}$ with certain probability of overflow, ϵ. At certain time intervals, the traffic (as measured in terms of the number of dirty pages) is measured and the EB calculated. If the calculated EB is observed to exceed EB_{th}, then w_i is put to sleep for a duration that is dependent on the difference of the two effective bandwidths. Simultaneously, the virtual queue length for w_i is also set to zero. Thus, instead of assuming exponential growth rate, the effective bandwidth uses *the queue length based approximation.* At the cost of slightly more complexity of implementation, we shall see that the EB approach is more accurate and less prone to false negatives of write congestion.

4.5.9 Which WICA Algorithm to Choose?

Having seen two different approaches for throttling the writers, the question now is which algorithm is appropriate for effectively throttling the writers. The biggest attrac-

tion for the rate based approach is its inherent simplicity. As a matter of fact, unless the system starts working under a series of bursty load, we have observed the deterministic rate based approach to able to control write congestion in most of the cases. Hence, for systems that are lightly loaded, in terms of average CPU occupancy, number of reader and writers and disk utilization, deterministic WICA provides performance which is close to the stochastic approach. However, stochastic WICA is recommended whenever the system is deployed in work load intense environments and where there occurs considerable variability in the workload (i.e., generation of dirty pages) pattern.

Table 4.2. Configurable parameters for the Postmark macro benchmark used in our experimental evaluation

Parameter	**Explanation**
`nReader`	Number of Reader Processes
`nWriter`	Number of Writer Processes
`nTransactions`	Transactional Load
`rSize`	`read()` Block Size
`wSize`	`write() Block Size`
`nFiles`	Number of files created for reading/writing
`fLow`	Min. size of file (in bytes) created for reading/writing
`fHigh`	Max. size of file (in bytes) created for reading/writing

4.6 Performance Evaluation

In this section, we evaluate the performance of D-WICA and P-WICA with the parameters listed in Table 4.2. Both the algorithms are compared with the native NetBSD 3.1 system without any form of WICA. All the measurements were taken on a machine running NetBSD 3.1 with 800MHz Pentium III processor, 512MB of main memory (8 page colors and 4KB page size). The file system type was NetBSD fast file system (ffs)

Table 4.3. Characteristics of the system used in our experiments

CPU Parameters	Disk Parameters	FileSystem Parameters
CPU: Intel Pentium III (32bit)	Disk Type: SATA	Type: FFS
FSB: 533 MhZ	Capacity: 120GB	Softupdates: Off
Clock: 800 MhZ	Rotation Speed: 7200 rpm	
I/O Connect: PCI	#Heads: 16	
L1 Cache:	#Cyclinders: 232581	
L2 Cache:	#Sectors: 234441648	
	#Sectors/Track: 63	
	#Bytes/Sector: 512	
	Average Seek Time: 7.4msecs	
	Average Latency: 4.17msecs	

with soft dependency (soft-dep) turned off. We created a separate test partition, `/test`, on the hard disk that was initially empty when all the experiments were conducted. Thus, all the results are for an empty partition with no consideration for file system aging. In Table 4.3, we provide the characteristic of the disk used in our experiments.

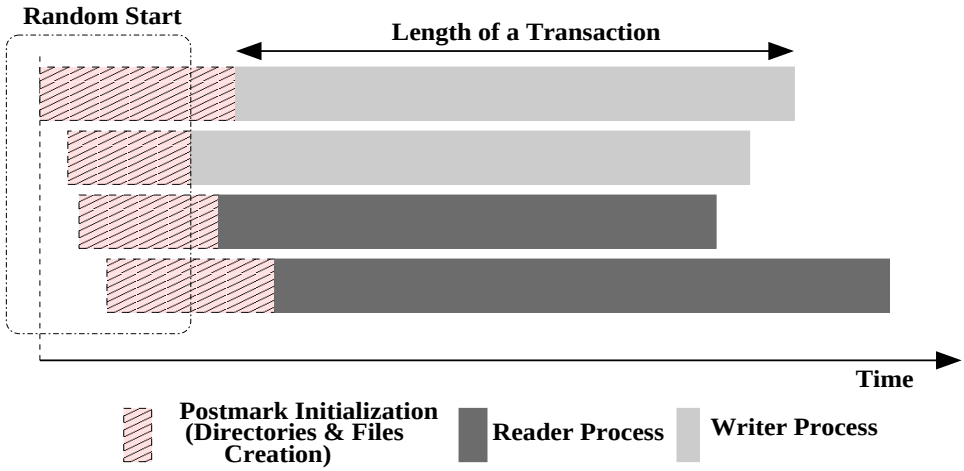

Figure 4.15. Evaluation technique with modified version of Postmark for creating reader and writer processes. In each of the experiments, the readers and the writers were independently created at random instants of time and the effective transactions per second (TPS) of the reader processes were recorded.

4.6.1 Workloads using Modified Postmark Macrobenchmark

Postmark [37] is a popular I/O intensive file system benchmark for evaluating the transaction rates of a dynamically changing small file environment as present in most web-based commerce transaction, e-mail, and netnews. Its effectiveness in stressing the underlying filesystem has been recorded in [95].

We modified Postmark (version 1.5) to create independent reader and writer processes hitherto referred as `pmark_r` and `pmark_w`, respectively. This empowered us to control the number of readers and writers actually executing inside the system. For both the readers and writers, the parameters shown in Table 4.2 were varied. Care was taken to ensure that for both the reader and writer processes, the working set was greater than the storage capacity of `ramfs` and that the reads and writes were never serviced from the file system buffer cache. As shown in Figure 4.15, `pmark_r` and `pmark_w` were created at random instants of time and the effective TPS of the reader processes were recorded.

Postmark operation: Details on the inner workings of the macrobenchmark are presented in Ref. [37]. Both the `pmark_r` and `pmark_w` processes start by creating *separate pools* of randomly selected text files with sizes uniformly distributed between the ranges, $fLow$ and $fHigh$ (see Table 4.2). This time is marked as *postmark initialization time* ($postmark_{int}$) in Figure 4.15 and it is in general a random variable. During the lifetime of the experiment, no further file deletion or creation operations are permitted by the readers or the writers.

After the initilization phase, the readers and writers move into the transaction phase which consists of *randomly* reading and writing to the file pool created during the period, $postmark_{int}$. The total time taken to completely read or write to the file size pool divided by the total number of transactions initiated provides a measure of the throughput of the operation (read/write) under test. It is subsequently reported as the effective

TPS.[†] In this study, we report the effective TPS of Postmark measurements for the `/test` directory. We choose RPRIO as the I/O sorting algorithm in all our experiments. Also, the sampling bin for D-WICA and the value of time t in the case of P-WICA was set to $20ms$ in for the results reported in this chapter.

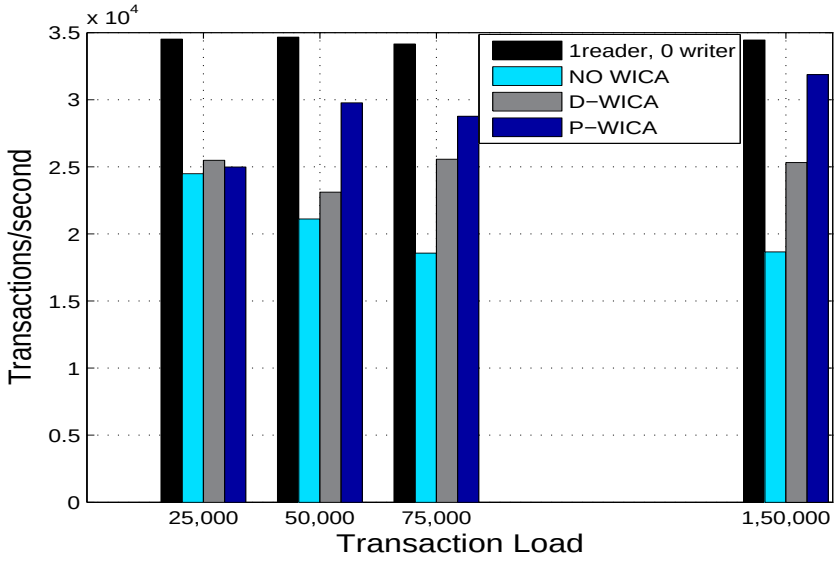

Figure 4.16. Postmark results for the Reader process. Observe the increased benefit of P-WICA over D-WICA with increase in the transactional load.

4.6.2 One Reader and One Writer

The initial test consists of only one writer and reader process in the system. The effective TPS for the native NetBSD system is shown in Figure 4.1 for a range of transactional load. The parameters used in the experiments are as follows: `rSize` = 512 bytes, `wSize` = 512 bytes, `nFiles` = 15000, `fLow` = 5000, and `fHigh` = 10000.

In Figures 4.16 and 4.17, we provide the effective TPS and the execution time of

[†]In our experiments, transaction time is defined as the wall clock time required to complete a `read()` and `write()` system call where the individual system calls each constitute a transaction.

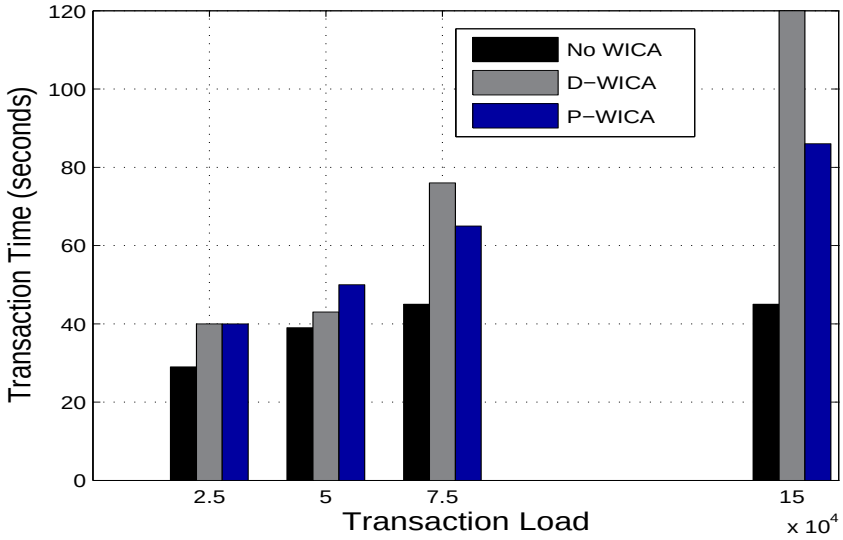

Figure 4.17. Transaction time for the Writer process. Observe the sharp increase in the case of D-WICA algorithm.

the experiment for four different scenarios: (i) the baseline case where there is only one reader and no writers in the system, (ii) a single reader in the presence of a background writer in the absence of WICA, (iii) with D-WICA, and (iv) with P-WICA. It is important to note here that there were sufficient pages in the VM to prevent triggering the page daemon. The figures reported represent the average values of five independent runs of the experiment.

Although one reader and writer is the simplest case in our experiment, it provides directions for important conclusions. Following are important conclusions that can be extended to a large system.

- At low transactional load, say 25,000 transactions, there is no significant difference in the performance of the D-WICA and P-WICA algorithm as can be seen Figure 4.16. The difference becomes apparent and significant as the system load is increased beyond 75,000 transactions. As a matter of fact, although not shown

in this study, we have observed that for lightly loaded systems, there occurs no significant difference in performace of the two algorithms.

- From Figure 4.17, it can be seen that D-WICA overestimates the degree of write congestion and puts the writer process into sleep more frequently and for a longer duration of time. This difference between the two algorithms becomes more apparent as the transaction load is increased. One possible reason is that the D-WICA is unable to quickly adjust to bursty loads and hence the estimated rate R_i suffers significant deviation from the true value. On the other hand, P-WICA is more optimistic in its approach and consequently the writer is penalized for lower duration of time.

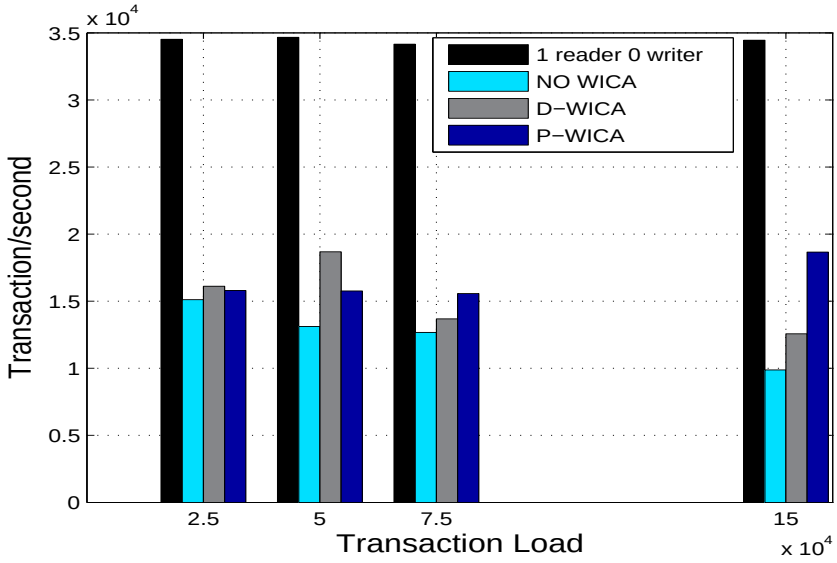

Figure 4.18. Postmark results for the Reader process in the presence of three writers.

4.6.3 One Reader and Many Writers

Since our aim is to study how WICA performs in the presence of write congestion, we purposely increase the number of writers (set to three in our case) in the system while keeping the number of reader process as one. In such situations, WICA should return the maximum benefit since the whole idea is to slow down the writers in favor of the readers. In Figure (4.18) we highlight such a case where the number of writers are increased to three while keeping only a single reader process in the system. Again, care was taken to ensure that the VM had enough free pages to meet the demand of the readers and writers. All other parameters of the experiment was the same as the previous case of single reader and writer.

Important conclusions that are extensible to a large system can be similarly drawn:

- Although not exposed in Figure 4.18, it was seen that the overall TPS is decreased (around 10%) if the I/O scheduling discipline is changed from RPRIO to FCFS. This implies that the underlying I/O scheduling algorithm does indeed affect the performance of reader and writer processes.
- Similar to the previous case, we observe that while both variations of WICA are effective in controlling write congestion, P-WICA triumphs with increase in the transactional load. However, the performance of D-WICA is better at transaction load of $50,000$. Initially this seems contradictory if we compare the result with Figure 4.16. However, the results proves the fact that in the case of *smooth load variations*, it is possible to accurately estimate the rate, R_i. Under such circumstances, D-WICA results in better identification of the congestion level.

Next, we identify a situation where the write congestion actually *fails*.

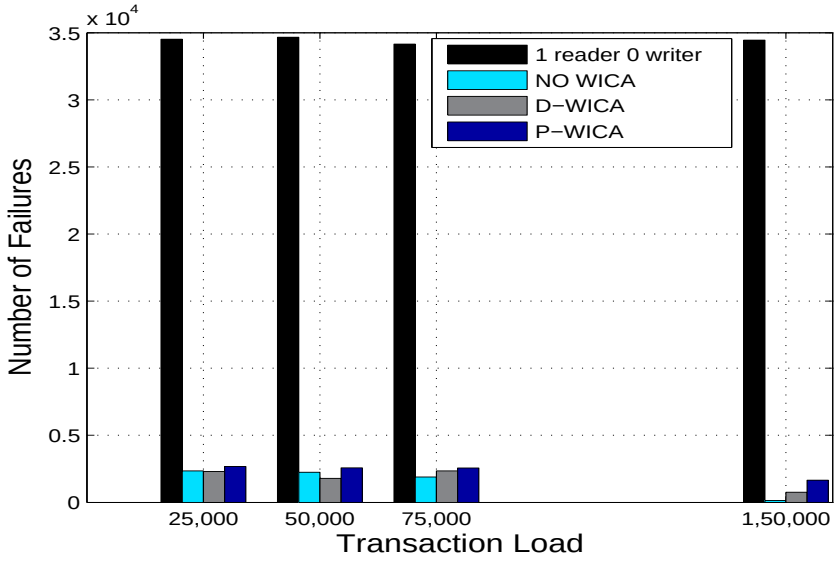

Figure 4.19. Postmark results for the Reader process in the presence of 20 writers. This identifies failure condition in ourexperimental settings for the WICA algorithm.

4.6.4 Situation Outside the Scope of WICA

In this section, we focus on identifying the condition where both the variants of WICA fail to provide any noticeable benefit in improving the TPS of the reader process. To identify such a situation, we increased the number of writer processes to 20 while keeping the reader process to one. The choice of 20 was made after several experimental settings that yielded the desired failure condition.

The ineffectiveness of the algorithms is immediately apparent from Figure 4.19. Also observe that extremely low TPS is experienced by the writer process. This situation happens when the free page pool inside the VM goes beyond the threshold limit that causes the page daemon to wake up. Thus, although the WICA tries to ease the VM system by penalizing the writers, the demand for free pages far outstrips the supply; so much so that the reader has no free page to map its data in the physical memory. Such a situation is beyond the scope of any congestion control algorithm that does not implement process admission control so that no new processes are admitted inside the

VM when the system is already struggling to identify free memory pages. The situation becomes worse if the number of writers in the system forces the VM to evict the current working set of pages to swap space.

When WICA the approach fails, it is observed that there occurs corresponding increase in the average transaction time of the writer processes. This is also due to the absence of free pages inside the VM. Since WICA is not an admission control algorithm, controlling such starvation is outside the scope of our approach. However, it is worth pointing out that, in general, P-WICA is a more optimistic in its approach and penalizes the writers cautiously. This is due to the ability of the P-WICA to better identify write congestion in the face of varying and bursty load patterns.

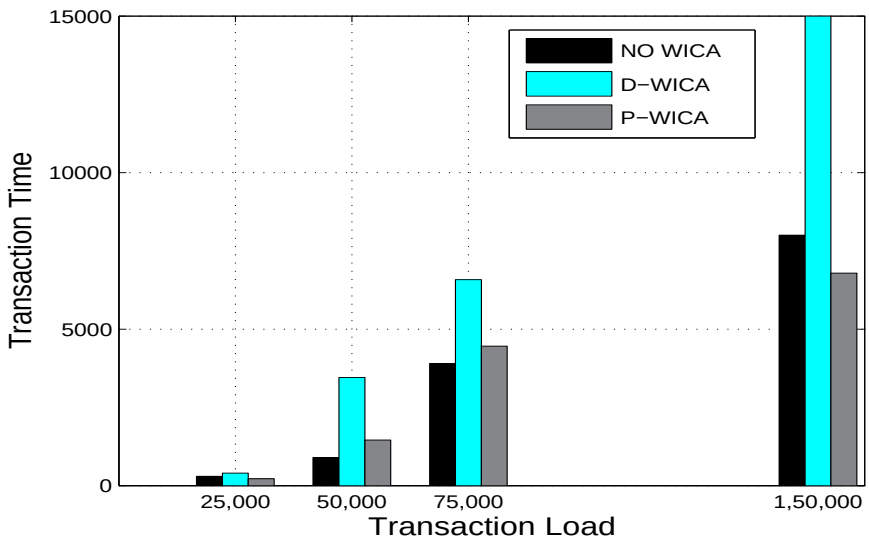

Figure 4.20. Transaction time for the Writer process experiencing the highest transaction time among competing writers, in the scenarion where WICA fails.

4.7 Summary

In this chapter, we have identified a scenario where the performance of the reader processes inside commodity operating systems is negatively impacted by the barrage of bursty writes initiated by the writers. We refer to this situtation as *write congestion*. For many real-time systems such as streaming multimedia servers and database systems where the average I/O throughput rather than reducing the I/O latency is of primary concern, write congestion can seriously degrade the overall system response time. To accommodate such write congestion, we have proposed two different algorithms.

The first one is deterministic, rate based approach where the birth rate of workload generation is assumed to be exponential. The second algorithm is stochastic in nature and based on the theory of *effective bandwidth*. On an average, it was observed that there occurs 15%-20% improvement in the effective TPS of the reader processes when WICA is used. However, the application space of the algorithms are different.

For lightly loaded systems, there is no significant variations in the patterns of workload generations. Hence, the deterministic approach is well suited in detecting write congestion. For systems where the workload is more bursty in nature, such as high end servers, the stochastic approach takes a more optimistic view in tackling and identifying congestion in the individual writer processes. We have implemented both these algorithms inside the NetBSD 3.1 kernel and reported Postmark performance figures with FCFS I/O scheduling discipline in this study.

An important observation was made regarding the nature of dirty page generation inside the VM. Using measurement data and time-variance analysis we have shown, that at higher order time scale (greater than $20ms$ in our experiments), the stochastic nature of dirty page generation is non-stationary and exhibits long-range dependence (LRD). However, there exists certain time scales (less than $20ms$ in our experiments), when stationary and Poisson models can be accurately used. The choice of suitable time scale

plays an important role in deciding the nature and type of the sampling process.

We have also identified a situation when the WICA fails to perform as expected. This happens when the free page pool inside the VM goes below a certain threshold that triggers the page daemon. In the absence of any free pages, the reader processes are unable to proceed and gets stalled for an inordinate amount of time.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have outlined guidelines that can be used to build next generation networks (NGNs) and improve the performance of read-oriented, latency sensitive application such as streaming multimedia services in such converged networks. In Chapter 2, we have proposed an architecture that closely integrates the traffic measurement architecture with the traffic inference algorithms. We have shown how it is possible to separate traffic belonging to short ived flows SLFs from long lived flows (LLFs) by applying the concept of typical sequence from Information theory. The simplicity of our algorithm allows us to feasibly capture the entire population of SLFs while inferring the LLFs using non-parameteric Gaussian kernel density functions.

In Chapter 3, we have provided a closed form queuing model that can be used to estimate the performance of NICs in commodity OS such as Linux, XBSD, and OpenSolaris. Considering the trend by network vendors to build application servers using Advanced Telecommunications Computing Architecture (ATCA) blade computing units that are expected to work in gigabit Ethernet networks, such a framework provides important pointers about the occurrence of performance hotspots due to incorrect setting of the NIC parameters, latency of the PCI/PCI-X I/O bus, and CPU overload.

However, the performance of latency sensitive applications is dependent not only on the network and the NICs but also on the OS. For read oriented applications, the performance is greatly affected by the activity of the writer processes and dirty page flushing by the OS. Thus, application servers in content delivert networks (CDNs) servicing latency and delay sensitive applications need to careful when initiating write I/O activity

that touch the disk in a non-uniform manner. In Chapter 4, we have identified a phenomenon called write congestion where the performance (as measured in TPS) of reader processes is negatively impacted by the flood of dirty pages being flushed by the VM. Since the underlying process exhibits LRD with no self-similarity, we have identified two algorithms, deterministic and effective bandwidth based, based on such dynamics that can be used to slow down the writers while skewing the disk access towards the reader processes.

Unfortunately, if the virtual memory (VM) system has insufficient free pages, then the solutions of Chapter 4 cannot be effectively used. Consequently, the performance of latency sensitive application can take a hit due to non-availability of data within a definite period of time.

5.1 Directions for Future Work

Several extensions are possible for the work presented in this dissertation. We briefly describe them below.

5.1.1 Identifying Heavy-hitters in Network Traffic

We have shown in this dissertation how flows can be separated based on their density of occurrence in the collected traffic samples. It would be beneficial to explore how effective our approach would be in identifying heavy-hitters in massive data sets as in the Internet backbone traffic. Another interesting result would be to bound the worst-case accuracy guarantee in traffic flow identification.

5.1.2 Extensible Framework for NICs

The queuing model presented in Chapter 3 was formulated considering a single CPU. A natural continuation is extension of the model to handle multiple CPUs especially

with the rapid advent of multi-core processing units. Also, we have assumed that the arrival process at the CPU work queue to be a $M/M/1$ bulk process. The effectiveness of such an assumption in scenarios where the CPU work queue is servicing jobs not only from the NICs but also from various processes of the system, would be an interesting finding. Another important direction would be to make the process of device polling dynamic. Current generation of polling device drivers require intimate knowledge of the system for extracting maximum performance benefits. Having such detailed knowledge of the working system might not be always feasible. It would be nice to have a framework that is able to derive and provide inputs to the polling system with the most appropriate values depending on traffic profile, CPU load, and I/O bus latency.

5.1.3 Establishing QoS Inside the VM

In current generation of operating systems available today, all processes have almost equal access to the free page list inside the VM. However, recognizing the type of request (read or write) from the processes would help the VM allocate the page pool more towards a specific type of process. Also, providing information about the performance of the read/write queues at the I/O level would help the VM understand the average service time of the storage system w.r.t the page arrival rate. Such mechanisms, though complex, might prove invaluable in future systems with concurrent existence of a large pool of readers and writer processes.

REFERENCES

[1] G. Appenzeller, "Sizing Router Buffers", *PhD Thesis*, Stanford University, Department of Electrical Engineering, Mar. 2005.

[2] M. Andrews and L. Zhang, "Minimizing end-to-end Delay in High-Speed Networks with a Simple Coordinated Schedule", *Journal of Algorithms*, vol. 52, issue 1, pp. 57-81, 2004.

[3] M. J. Bach, *"The Design of the Unix Operating System"*, Prentice-Hall, Inc., 1986.

[4] A. W. Berger and W. Whitt, "Extending the Effective Bandwidth Concept to Networks with Priority Classes", *IEEE Communications Magazine*, vol. 36, issue 8, pp 78-83, 1998.

[5] A. Barczyk, A. Carbone, J.P. Dufey, D. Galli, B. Jost, U. Marconi, Niko Neufeld , G. Peco, V.M. Vagnoni, "Reliability of Datagram Transmission on Gigabit Ethernet at Full Link Load",*Technical Document No. LPHE 2005-006, Laboratory for High Energy Physics, Swiss Federal Institute of Technology*, Apr. 2004.

[6] P. Benmowski, "Hyper-threading Linux", *Linux World*, Aug., 2003.

[7] T. Bonald and A. Prouti, "On Performance Bounds For the Integration of Elastic and Adaptive Streaming Flows", *ACM SIGMETRICS*, pp. 235-245, 2004.

[8] N. Brownlee, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", *IEEE Communications Magazine*, vol. 40, pp. 110-117, Oct. 2002.

[9] J. Cao, W. S. Cleveland, D. Lin, and D. X. Sun, "Internet Traffic Tends Towards Poisson and Independent as Load Increases", *Nonlinear Estimation and Classification*, pp. 83-109, Springer-Verlag, 2002.

[10] K. C. Clay, G. C. Polyzos, and H. W. Braun., "Application of Sampling Methodologies to Network Trace Characterization", *ACM SIGCOMM*, pp. 13-17, 1993.

[11] R.B. Cooper, "Introduction to Queueing Theory, 2nd ed.", London, Arnold, 1981.

[12] G. Cormode and S. Muthukrishnan, "What's Hot and What's Not: Tracking Most Frequent Items Dynamically", *ACM PODC*, pp. 296-306, Jul. 2003.

[13] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley, 1991.

[14] D. C. Cranor, "The Design and Implementation of the UVM Virtual Memory System", *Ph.D. Thesis*, Sever Institute of Technology, Department of Computer Science, Washington University, St. Louis, MO, USA, Aug. 1998.

[15] A. Currid, "TCP Offload to the Rescue", *ACM Queue*, vol. 2, issue 3, pp. 58-65, May 2004.

[16] P. J. Denning, "Effects of Scheduling on File Memory Operations", *AFIPS Spring Joint Computer Conference*, pp. 9-21, Apr. 1967.

[17] B. Doytchinov, J. Lehoczky, and S. Shreve, "Real-Time Queues in Heavy Traffic with Earliest-Deadline-First Queue Discipline", *Annals of Appl. Probability*, pp. 332-379, 2001.

[18] N. Duffield, C. Lund, and M. Thorup, "Properties and Prediction of Flow Statistics from Sampled Packet Streams" *ACM SIGCOMM Internet Measurement Workshop (IMW)*, pp. 159-171, Nov. 2002.

[19] N. Duffield, C. Lund, and M. Thorup, "Flow Sampling Under Hard Resource Constraints", *ACM SIGMETRICS*, pp. 85-96, Jun. 2004.

[20] N. Duffield, C. Lund, and M. Thorup, "Estimating Flow Distributions from Sampled Flow Statistics", *ACM SIGMETRICS*, pp. 325-336, Aug. 2003,.

[21] M. Finkelstein, H. G. Tucker and J. A. Veeh, "Confidence Intervals for the Number of Unseen Types", *Statistics and Probability Letters*, vol. 37, no. 4, pp. 423-430, Mar. 1998.

[22] P. Gupta and N. McKeown, "Algorithms for Packet Classification", *IEEE Network*, vol. 15, no. 2, pp. 24-32, Mar. 2001.

[23] S. D. Gribble, G. Singh Manku, E. A. Brewer, T. J. Gibson, and E. L. Miller, "Self-Similarity in File-System Traffic", *ACM SIGMETRICS*, pp. 141-150, Jun. 1998.

[24] M. Handley, O. Hodson, and E. Kohler, "XORP: An Open Platform for Network Research", *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 33, pp. 53-57, Jan. 2003.

[25] N. Hohn and D. Veitch, "Inverting Sampled Traffic", *ACM SIGCOMM Internet Measurement Workshop (IMW)*, pp. 222-233, Oct. 2003.

[26] N. Hohn, D. Veitch, K. Papagiannaki, and C. Diot, "Bridging Router Performance and Queuing Theory", *ACM SIGMETRICS*, pp. 355-366, vol. 32, Jun., 2004.

[27] P. Gray and A. Benz,"Performance Evaluation of Copper-based Gigabit Ethernet Interfaces", *IEEE Conference on Local Computer Networks (LCN)*, pp. 679-690, 2002.

[28] B. Gill and D. S. Modha, "SARC: Sequential Prefetching in Adaptive Replacement Cache", *USENIX ATC*, pp. 293-308, 2005.

[29] V. Jacobson. "Congestion Avoidance and Control", *ACM SIGCOMM*, vol. 18, pp. 314-320, Aug. 1988.

[30] S. Jiang and X. Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance", *ACM SIGMETRICS*, pp. 31-42, Jun. 2002.

[31] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities", *USENIX FAST*, Dec. 2005.

[32] R. H. Jones, S. Dallison, and G. Fairey, "Performance Measurements on Gigabit NICs and Server Quality Motherboards", *First International Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2003.

[33] N. Joukov, T. Wong, and E. Zadok, "Accurate and Efficient Replaying of File System Traces", *USENIX FAST*, pp. 337-350, Dec. 2005.

[34] M. Kargahi and A. Movaghar, "A Method for Performance Analysis of Earliest-Deadline-First Scheduling Policy", *Journal of Supercomputing*, vol. 37, no. 2, pp. 197-222, 2006.

[35] I. Kamel and Y. Ito, "Disk Bandwidth Study for Video Servers", *Technical Report, Matsushita Information Technology Laboratory*, Apr. 1996.

[36] S. F. Kaplan, L. A. McGeoch and M. F. Cole, "Adaptive Caching for Demand Prepaging", *ACM ISMM*, pp. 114-126, 2002.

[37] J. Katcher, "PostMark: A New File System Benchmark", *Technical Report TR3022*, Network Applicance Inc., Oct. 1997.

[38] F.P. Kelly, "Notes on effective bandwidths", *Stochastic Networks: Theory and Applications*, pp. 141-168, Oxford Univ. Press, Oxford, 1996.

[39] G. Kesidis, J. Walrand, and C. S. Chang, "Effective Bandwidths for Multiclass Markov Fluids and other ATM Sources", *IEEE Transactions on Networking*, pp. 424âĂŞ428, 1993.

[40] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A Low-Overhead, High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References", *ACM/USENIX OSDI*, pp. 119-134, Oct. 2000.

[41] L. Kleinrock, "Queuing Systems: Vol 1: Theory", John Wiley and Sons, Inc., 1975.

[42] T. M. Kroeger and D. D. E. Long, "Design and Implementation of a Predictive File Prefetching Algorithm", *USENIX Annual Technical Conference*, pp. 105-118, Jan. 2001.

[43] A. Kumar, J. Xu, O. Spatschek, and L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement", *IEEE INFOCOM*, Aug. 2003.

[44] A. Kumar, M. Sung, J. Xu, and L. Wang, "Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution", *ACM SIGMETRICS*, pp. 177-188, Aug. 2003.

[45] A. Lall, V. Sekhar, M. Ogihara, J. Xu and H. Zhang, "Data Streaming Algorithms for Estimating Entropy of Network Traffic", *ACM SIGMETRICS*, pp. 145-156, Jun. 2006.

[46] W.E. Leland, M.S. Taqqu, W.Willinger, and D.V.Wilson, "On the Self-Similar nature of Ethernet traffic (extended version)", *IEEE/ACM Transactions on Networking*, vol. 2, pp. 1âĂŞ15, 1994.

[47] T. Lindh, "Systematic Sampling and Cluster Sampling of Packet Delays", *PAM 2006*, Apr. 2006.

[48] R. Love, "CPU Affinity", *Linux Journal*, issue 111, Jul. 2003.

[49] N. Megiddo, D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache", *USENIX FAST*, pp. 115-130, 2003.

[50] J. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel", *ACM Transactions on Computer Systems (TOCS)*, vol. 15, issue 3, pp. 217-252, Aug. 1997.

[51] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets", *ACM SIGCOMM Workshop on Internet Measurment Workshop (IMW)*, pp. 115-120, 2004.

[52] R. Morris, E. Kohler, J. Jannotti, and M. Frans Kaashoek, "The Click modular router", *ACM Transactions on Computer Systems (TOCS)*, vol. 18, pp. 263-297, Aug. 2000.

[53] A. L. Neidhardt and J. L. Wang, "The Concept of Relevant Time Scales and Its Application to Queuing Analysis of Self-Similar Traffic", *ACM SIGMETRICS*, pp. 222-232, 1998.

[54] V. S. Pai, S. Rixner, and H. Kim, "Isolating the Performance Impacts of Network Interface Cards through Microbenchmarks", *ACM SIGMETRICS*, pp. 430-431, June 2004.

[55] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement through Readahead Optimization", *Linux Symposium*, Jul. 2004.

[56] E. Parzen, "On Estimation of a Probability Density Function and Mode", *The Annals of Mathematical Statistics*, vol. 33, pp. 1065-1076, 1962.

[57] J. Paris, A. Amer, and D. D. E. Long, "A Stochastic Approach to File Access Prediction", *ACM International Workshop on Storage Network Architecture and Parallel I/Os*, pp. 36-40, 2003.

[58] S. Ramabhadran and G. Varghese, "Efficient Implementation of a Statistics Counter Architecture", *ACM SIGMETRICS*, pp. 261-271, 2003.

[59] S. Raudys, "On the Effectiveness of Parzen Window Classifier", *Informatics*, vol. 2, no. 3, pp 435-454, 1991.

[60] A. L. N. Reddy and J. Wyllie, "I/O issues in a Multimedia System", *IEEE Computers*, vol. 27, pp. 69-74, Mar. 1994.

[61] J. A. Redstone, S. J. Eggers and H. M. Levy, "An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture", *In Proceedings of ACM ASPLOS*, pp. 245-256, Jun. 2000.

[62] J. Regehr and U. Duongsaa, "Preventing Interrupt Overload", *ACM Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 50-58, 2005.

[63] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet", *Usenix Technical Conference*, Nov. 2001.

[64] P. H. Seaman, R. A. Lind, and T. L. Wilson, "On Teleprocessing System Design, Part IV: An Analysis of Auxiliary-storage Activity", *IBM System Journal*, vol. 5, no. 3, pp. 158-170, 1966.

[65] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited", *Winter 1990 USENIX Conference*, pp. 313-324, Jan. 1990.

[66] J. G. Shantikumar, J. Buzacott, "On the Approximations to the Single-server Queue", *Internat. J. Prod. Res.*, pp. 761-773, 1980.

[67] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown, "Maintaining Statistics Counters in Router Line Cards", *IEEE Micro*, vol. 22, no. 1, pp. 76-81, Jan. 2002.

[68] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, 1986.

[69] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, "Deadline Scheduling for Real-time Systems: EDF and related algorithms", *Kluwer Academic Publishers*, 1998.

[70] S. Tartarelli, M. Falkner, M. Devetsikiotis, I. Lambadaris, and S. Giordano, "Empirical Effective Band-widths", *Global telecommunications Conference, IEEE GLOBECOM*, vol. 1, pp. 672-678, 2000.

[71] A. Thomasian and C. Liu, "Disk Scheduling Policies with Lookahead", *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 2, pp. 31-40, Sep. 2002

[72] A. Tomkins, R. H. Patterson, and G. Gibson, "Informed Multi-Process Prefetching and Caching", *ACM SIGMETRICS*, pp. 100-114, 1997.

[73] C. Tsai, E. Chu, and T. Huang, "WRR-SCAN: A Rate-based Real-time Disk-scheduling Algorithm", *ACM EMSOFT*, pp. 86-94, 2004.

[74] B. L. Worthington, G.R. Ganger, and Y. Patt, "Scheduling Algorithms for Modern Disk Drivers", *ACM SIGMETRICS*, pp. 241-251, 1994.

[75] J. Yang and M. Devetsikiotis, "On-line Estimation, Network Design and Performance Analysis with Effective Bandwidths", *ITC-17*, pp. 347-358, 2001.

[76] K. Zheng, H. Che, Z. Wang, and B. Liu, "TCAM-based Distributed Parallel Packet Classification Algorithm with Range-Matching Solution", *IEEE INFOCOM*, pp. 293-303, Mar. 2005.

[77] Cisco NetFlow, `http://www.cisco.com`.

[78] Spirent Smart Flow, `http://www.spirentcom.com/documents/94.pdf`.

[79] Spirent SmartBits 6000C, `http://www.spirentcom.com/documents/1050.pdf`.

[80] Intel(R) PRO/10/100/1000/10GbE Drivers, `http://sourceforge.net/projects/e1000`

[81] NoBL SRAMs and Bus Contention, `www.cypress.com`

[82] Netgear GSM 7224, `http://www.netgear.com/products/details/GSM7224.php`

[83] Network Search Engine (NSE) Family, `www.netlogicmicro.com/products/nse.html`

[84] Intel Pro/1000 MT Dual Port Server Adapter, `www.intel.com/network/`

[85] NLANR AMP Website, `http://pma.nlanr.net/Special/`

[86] tcpdump, `www.tcpdump.org`

[87] tcpdstat, `www.freshports.org/net/tcpdstat/`

[88] Third Generation Partnership Project, Technical Specification Group Services and System Aspects, TS 23.228 IP Multimedia Subsystem (IMS), Stage 2/3GPP2 X$\dot{S}$0013 − 002 − 0 v1.0, `www.3gpp.org`.

[89] Advanced TCA (ATCA), `www.picmg.org/newinitiative.stm`.

[90] NAPI, Linux Documentation available in Linux kernel source distribution under `/usr/src/Documentation/`.

[91] Interrupt Moderation using Intel Gigabit Ethernet Controllers, `www.intel.com`.

[92] `www.comlab.uni-rostock.de/research/tools.html`

[93] IOzone Filesystem Benchmark, `www.iozone.org`

[94] Bonnie++: A benchmark suite of hard drive and file system performance, www.coker.com.au/bonnie+

[95] Veritas File Server Edition Performance Brief: A PostMark 1.11 Benchmark Comparison, *Technical report*, Veritas Software Corporation, available online at: http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf, Jun. 1999.

[96] http://mama.agr.okayama-u.ac.jp/english/movie.html

BIOGRAPHICAL STATEMENT

Sumantra R. Kundu received his B.Tech. degree from Indian Institute of Technology Kharagpur, India, in 1999, his M.S. degree from The University of Iowa, Iowa City, in 2001; both in Electrical Engineering. He obtained his Ph.D. degree in Computer Science and Engineering from The University of Texas at Arlington, in 2007. His current research interest are in performance modeling and analysis of large scale systems.