University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Masters Theses 1911 - February 2014

2011

Hardware Implementation of Queue Length Based Pacing on NetFPGA

Abhishek Dwaraki
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/theses

 Part of the Digital Communications and Networking Commons

Dwaraki, Abhishek, "Hardware Implementation of Queue Length Based Pacing on NetFPGA" (2011). *Masters Theses 1911 - February 2014*. 604.
Retrieved from https://scholarworks.umass.edu/theses/604

This thesis is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Masters Theses 1911 - February 2014 by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

HARDWARE IMPLEMENTATION OF QUEUE LENGTH BASED PACING FOR SMALL BUFFER NETWORKS

A Thesis Presented

by

ABHISHEK DWARAKI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

May 2011

Department of Electrical and Computer Engineering

© Copyright by Abhishek Dwaraki 2011

All Rights Reserved

HARDWARE IMPLEMENTATION OF QUEUE LENGTH BASED PACING FOR SMALL BUFFER NETWORKS

A Thesis Presented

by

ABHISHEK DWARAKI

Approved as to style and content by:

Tilman Wolf, Chair

Ramgopal Mettu, Member

Weibo Gong, Member

C.V.Hollot, Department Chair
Department of Electrical and Computer Engineering

To my parents.

ACKNOWLEDGMENTS

I would like to thank my advisor Prof.Tilman Wolf who has been a constant source of support and guidance through the duration of my Master's. I thank him for all the help and more importantly the opportunity to work on an interesting research topic.

At the same time, there have been many people involved directly or indirectly who would warrant acknowledgement here. Ramakrishna, Vikram, Vishwas, Sudheendra, Pavan and Priyamvada have been involved in this some time or the other and their contributions have been invaluable. Working on this thesis without their help would have been that much harder. I thank each one of them for being around whenever I needed them.

Sinan, Shashank and Sriram have always been there to help me with any issues I had and it is my privilege to have had them as my lab mates. I thank them for all the help.

And last, but not the least, I would like to thank my support system at UMASS, Amherst - Priyanka, Shruti, Saket, Shailesh, Santosh, Kartik, Rashmi, Supratim, Ajanta, Anindya, Lokesh and Akshaya for standing by me during this time.

ABSTRACT

HARDWARE IMPLEMENTATION OF QUEUE LENGTH BASED PACING FOR SMALL BUFFER NETWORKS

MAY 2011

ABHISHEK DWARAKI

B.E, VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELGAUM,

KARNATAKA, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Prof. Tilman Wolf

Optical packet switching networks are the foundation for next generation high speed internet and are fast becoming the norm rather than an option. When such high speed optical networks are taken into account, one of the key considerations is packet buffering. The importance of packet buffering plays an even bigger role in optical networks because of the physical and technological constraints on the buffer sizes that can be implemented. Existing protocols, in many real world scenarios do not perform well in such networks. To eliminate such scenarios where there is a high possibility of packet loss, we use the pacing algorithm proposed in [3]. The proposed pacing scheme aims to reduce or eliminate packet losses arising from packet bursts in small-buffer networks. This thesis deals with a proposed hardware design and implementation of the packet pacing system on a NetFPGA. Our results show that the packet pacer can be implemented with a low overhead on hardware resources.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION AND MOTIVATION

1.1 Introduction

Moving into the twenty-first century, it might not just be enough to say that information technology and services have made themselves an indispensable part of our lives, ranging from societal to commercial aspects. And at the forefront of this unstoppable advancement of information technology, we have one of the many areas that has become so ubiquitous - Computer Networks. High data rates, higher bandwidth availability and a deregulated telecommunication environment has resulted in cut-throat competition among service providers and has fueled the race to develop higher capacity links to service more users. This paradigm shift from co-axial cables is happening slowly, but steadily towards an all-optical core network. As we delve deeper into the intricacies of how data and information is transmitted and decoded over a network, what limitations we have to work with, how to makes things better etc, we come across scenarios that may not seem to be problematic at first, but they end up being the most complicated ones later.

One of these problems is related to the optical core. With optical equipment still in its infancy, having large optical buffers ends up being immensely costly and impractical. Having small buffers would mean having suitable traffic statistics to minimize packet losses and control congestion. If you look at things pragmatically, there are always two sides to the same coin. The pros of moving to the opto-electronic circuits and then slowly to all optical networks is the promise of having unmatched transmission rates over the network. But at the same time, it is very evident that

everything that goes with the network needs to keep up with this. Since a lot of networks connect to the optical core and pour their data in, transmission of data could become a tricky business if the link is not ready sometimes. In this eventuality, buffering the data becomes an even bigger issue, firstly, because the physical design itself creates constraints and secondly, high transmission rates compound the problem. The physical design of optical networks does not allow buffering in the legacy method of using FIFOs, but instead uses fiber delay lines to buffer the data. FDLs have a lot of downsides. They provide a fixed delay and are also cumbersome. More is discussed about this in Chapter 2. Continuing with the discussion, keeping the data buffered in the data plane at that high rate is a tough job and is complicated by the reason stated above. TCP, as we all know is inherently bursty. With thousands of TCP flows existing over the Internet, many of them are short flows [8] being dominated by user and session parameters and more predominantly by TCPs own congestion control scheme. As a result, performance degrades with normal TCP sessions causing more frequent packet drops [3]. Consequently, there is a high probability that the buffer on the ingress of a bottleneck link in the core will fill up fast resulting in buffer overflows. TCP itself is designed in this way with its congestion control mechanism. And considering the move to have all-optical cores, which use small buffers, there should be some other way of minimizing packet losses due to queue drops.

A previous thesis titled Analysis and Study of Queue Length Based Pacing in Small Buffer Networks had already dealt with the analysis and study of a pacing scheme called Queue Length Based Pacing (QLBP) [7]. It focused on how QLBP could help ensure better statistical properties by smoothing out bursty traffic flows in the Internet to improve the performance of small buffer networks. This thesis focuses on implementing a low-overhead pacer in hardware and applying it at the edge of the core network to regulate or engineer traffic in the core network and hence aid in

minimizing the packet loss rate. In the following chapters, we will see in detail how and why buffer sizing and pacing are important to us.

1.2 Motivation

The Internet has become ubiquitous in our lives today: all pervading, ever present, and ever required. As the demand continues to grow, the range and variety over which the Internet is used broadens to include newer horizons. Consequently, the demand for high bandwidth networks is constantly on the rise.

One of the ways to cater to this high bandwidth requirement is to implement an all-optical core network, like explained above. Why this is costly is dealt with in the next paragraph. Traditionally, the rule of thumb for buffer size estimation of a core router has always been $C * RTT$, where C is the link capacity and RTT is the effective round trip time of the packets going through the router. As it is very evident, for gigabit capacity links and RTTs in the order of milliseconds, the buffer sizes would grow in factors greater than 10^4 at least. With all optical routers using fiber delay lines for optical buffering, it becomes impossible even with the latest technology to build such large optical buffers. Recent studies have also shown that smaller buffers could be used if the traffic is properly spaced out when it arrives so as to minimize losses [2] [5]. At the same time, there is another school of thought which points to the fact that small buffers in case of congested links and large TCP flows could lead to high losses [11]. From these instances, it is easy to see why small buffers and packet pacing over routers using small buffers is of interest to us.

1.3 Contributions

Our main contributions are as below:

1. To implement the pacing algorithm as a prototype on the NetFPGA hardware board.

2. Prove that the traffic egressing the prototype is paced to an extent.

3. Analyze the performance using metrics like throughput obtained with pacing enabled, packet drop rate etc.

1.4 Thesis Organization

The rest of the thesis is organized as follows; Chapter 2 presents some background on Optical Networks, how buffering is accomplished and why it is important. Chapter 3 provides a discussion on pacing, how we consider it to be a possible solution to our buffering issue and how it slots into our idea of optical networks. Queue Length Based Pacing, the algorithm and its design are discussed in Chapter 4. The design and architecture of the hardware implementation is discussed in chapter 5, along with a brief introduction to the NetFPGA platform being used. The results of the implementation are presented in Chapter 6. Chapter 7 summarizes and concludes this thesis document by listing future enhancements that can be made.

CHAPTER 2

OPTICAL NETWORKS AND BUFFERING: THE HOW, WHY AND WHY NOT

With the problem and the motivation stated, in this chapter, we are going to look a bit into the past about all the recent developments in increasing bandwidth and how the Internet is progressing towards an all-optical core network.

2.1 Introduction

Traditionally, Time Division Multiplexing (TDM) was originally used for carrier access. Explicitly stated, during a particular time quantum; one sender had exclusive access to the channel. This solved the problem of collisions, but resulted in under-utilization in many instances. To address this issue, another concept, namely, Wavelength Division Multiplexing (WDM) came into existence. With WDM, the transmission spectrum of the physical medium was split into multiple wavelengths, each sender using a separate wavelength for its transmission. This worked just fine, since each sender had its own wavelength which was sent through the fiber, and was de-multiplexed at the other end. As with TDM, the signal at the other end was the aggregate of all the input signals, but each signal was transmitted independently of the others, with its own dedicated resources on the carrier signal. As WDM grew in popularity, it started having problems handling the increasing number of multiplexed senders.Subsequently, Dense WDM came into existence.

Dense WDM spaces wavelengths closer than WDM and hence accommodates more transmitters. It can also carry varying signals in the sense that all the signals need

not be of the same transmission rate and protocol type. This was a breakaway from the traditional method since it allowed for a variety of signals to pass through at the same time without interfering with each other.

With the increase in DWDMs bandwidth and data transfer rates, if we go by the traditional thumb rule of calculating the buffer sizes to be used, then, for such minimal delays over such high bandwidths, buffer sizes end up being millions of packets, which, let alone in optical engineering, would be well nigh impossible in terms of electronic FIFOs too. The case for optical networks becomes especially severe since buffer sizes are pretty small and the implementation is not as an optical buffer, but as a Fiber Delay Line (FDL).

2.2 Fiber Delay Lines as Optical Buffers

As instead stated above, optical networks provide high bandwidth and faster data transmission rates as compared to electronic transmission links. Analogous to what we have in normal, electronic packet switched networks (EPS), an optical core results in an optical packet switched network (OPS). And again, we end up at the same point in our discussion so far, buffering.

Buffering in OPSes is accomplished using FDLs. The motivation for using buffers is simple: to store packets if the link is busy transmitting some other data. But with optical lines, it becomes difficult mainly because of the high data rates and bandwidth. EPSes normally use SRAM based memories to implement FIFO buffers, which is in accordance with the operational capability of an SRAM to match that of the EPS. This is not the case with OPSes. Firstly, to have large buffers in OPSes is infeasible, both from an engineering standpoint as well as a cost effective standpoint. The other downside is that since OPSes work at such high speeds, it is practically impossible to use SRAM based electronic devices to do the buffering. There exists an inherent operational speed mismatch between a CMOS gate-based device and an

optical transmission medium. This is what makes the use of SRAM based FIFO buffers impossible to use with optical routers and hence the use of FDLs.

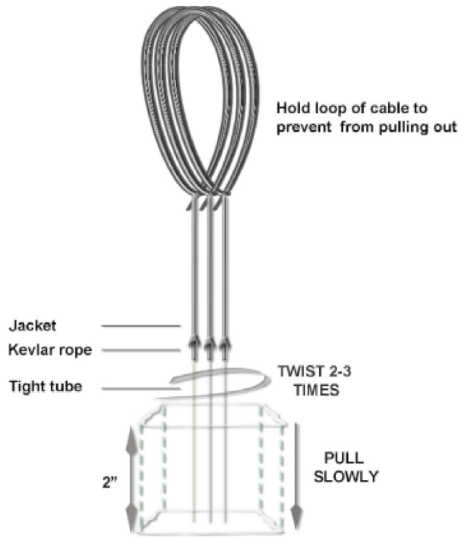


Figure 2.1. Fiber Delay Line

FDLs also have some typical characteristics that make it hard to implement small buffers using them:

1. FDLs are actual optical wires, which means that they take up space, limiting the number of FDLs that can be provided per router.

2. FDLs, once implemented are fixed delay devices.

3. Lastly, but one of the more important factors, FDLs are long wires, and due to this, encounter signal attenuation inside the routers [7].

From the above discussion, it becomes amply clear that buffer sizing in optical networks is of paramount importance and the answer does not lie entirely in increasing or decreasing the buffer size. We do need to work or look into areas other than buffer sizing. But how does this issue of buffer sizing affect our implementation in other areas?

2.3 Buffer Sizing - Is it the only solution?

Since large buffers are out of the questions for OPSes, would reducing the buffer size work?

Even if buffering works to an extent [4], there still are some other considerations for reduction in buffer size. Like in most other engineering problems, this is more often a question of trade-off with some other characteristic behavior. For example, since FDLs are fixed delay entities, you cannot make them too long, which would induce a very high delay that is not desirable. On the other hand, you cannot make the buffers too small either, since we cannot reduce the buffer size without the traffic being statistically paced. Also, when it comes to FDLs, the longer they are, the more cumbersome they become since they are physical entities that consume space and are also infeasible to implement in practical applications.

Applying this to TCP transmissions, their inherent bursty nature causes packet losses in small buffer networks without any pacing being applied when there are multiple TCP flows. Small number of flows can be handled by small buffer sizes. Some potential solutions that have been explored have been those of limiting the flow rate per flow, showing that O(log W) buffers suffice, where W is the congestion window size [4]. But this holds good only when pacing has been applied to all the flows and the link is under-utilized [7]. Further analyzing this scenario, it is evident that a potential flaw might be the presence of small number of flows in the network. And as a result of this, there is a high level of under-utilization since all the flows are limited in addition to only few flows being present.

Figure 2.2 shows how users are aggregated to various DSLAMs. These DSLAMs are then multiplexed onto different aggregation switches/routers before being routed onto the IP core network. Most ISPs connect to the IP core, which is a common cloud. Since access links are much slower than their core counterparts, it is a possibility that the traffic is already paced before it reaches the core. This would be true and might

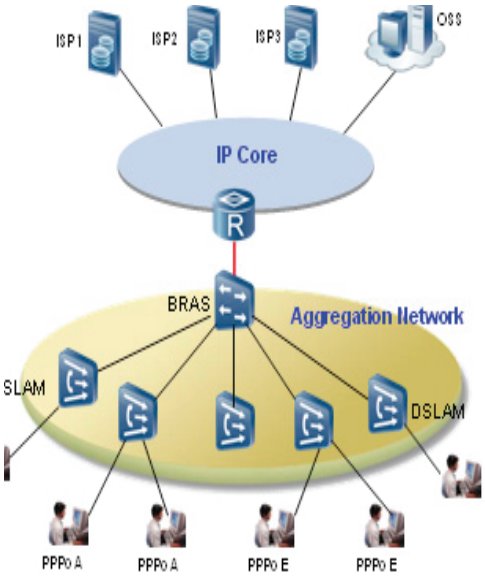


Figure 2.2. Differentiation between Access and Core Networks

not matter much when normal computers are used. But if high speed machines are communicating, then the end to end delay would become really evident and of course, undesirable.

There are a number of solutions that come to mind when it comes to pacing. We would naturally think that it would be a good idea to pace the TCP agent at the sender itself. This idea, coupled with TCPs own congestion control initially seems utopic, but is not the case. It means changing and re-deploying all the TCP protocol stacks which is definitely infeasible. Hypothetically, even if we did pursue this method, there is no guarantee that all the flows aggregated at the access edge of the network will not be synchronized and hence result in packet drops.

The next and most convenient method would be to pace the traffic at the edge of the network and prepare it before it even reaches the core. This would facilitate the core to focus on bandwidth maximization without factoring in small buffer scenarios and packet drops. Noted below is some work till date regarding pacing at the access nodes of OPSes:

1. It is possible to achieve a low packet drop with small buffers by applying pacing [11] . But this applies a fixed delay to packets using a leaky bucket algorithm and does not detail how to calculate the optimum delay.

2. [12] proposes a delay based algorithm that dynamically adjusts the pacing times so that the end-to-end delay is bounded.

2.4 TCP Buffer Sizing

In this section, let us consider TCP as a case study to see how it relates to buffer sizing and why we consider this issue to be of prime importance. We saw a lot of scenarios in the last section whether buffer sizing was not the only option. We have already established pacing as an alternative to buffer sizing. But the impending question still remains: Why is buffer sizing so very important?

With todays market trend for memories, it might be a simple solution at first sight to put in a huge memory. But what are its implications? When you look at the router, you need to consider the hardware perspective. Huge memory makes the router denser in terms of an electronic footprint and increases the power consumption. Also, more and more applications on the Internet (which is supposed to be the benchmark for networks in terms of traffic composition and statistics) are delay and jitter bound applications [10]. These applications cannot afford to have their packets buffered infinitely.

One of the reasons that we are considering TCP as a case study is because of its design. It is designed in a way that no matter how large the buffer is, TCP will ramp up its sending rate to such an extent that packet loss will occur and the sending rate will drop to half its value. Then it slowly ramps up again until another packet loss occurs. The concept behind this saw-tooth congestion control is to make full use of the bandwidth available without letting under utilization occur.

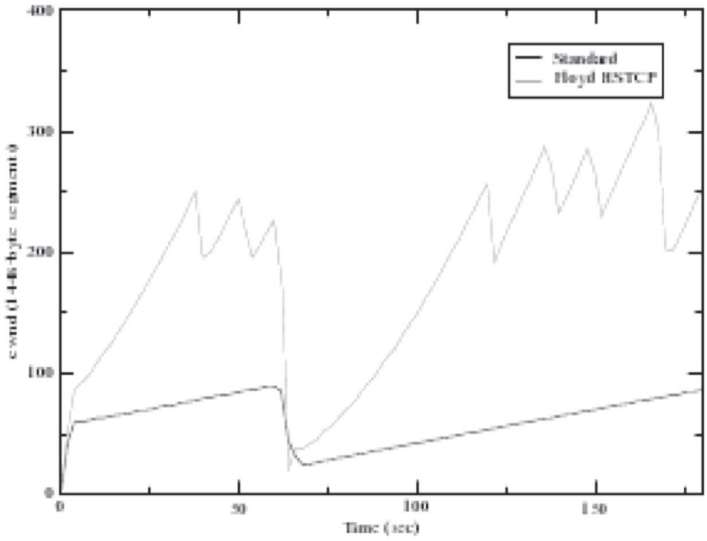


Figure 2.3. TCP Sawtooth Congestion Control

Figure 2.3 is the TCP saw-tooth graph with two plots, one for standard congestion avoidance and the other for high-speed congestion avoidance [6]. Also, it is a well-documented fact that TCP needs the full value of $C*RTT$, where C is the link capacity and RTT is the round trip time, at every bottleneck link to deliver full capacity transmission [14]. We can observe the Additive Increase Multiplicative Decrease (AIMD) nature of congestion avoidance from this.

Consider a sender-receiver pair with a router in between them. The sender-router link is higher in capacity than the router-receiver link and hence acts as the bottleneck. The packets end up being buffered at the router. When an acknowledgement is missed, the congestion window at the sender is halved to prevent any further contribution to congestion at the router. This means that the sender already has many packets still to be acknowledged in the network. The queues/buffers at the router are so designed as to prevent packet loss as far as possible. But sometime, the TCP transmission rate will hit a point where a packet is lost and is not acknowledged. This is when the rate drops and can be visualized as a momentary pause in transmission. During this time,

the buffer at the router should not be so small as to drain and let the bottleneck link go idle. This would lead to under-utilization of the link.

It is intuitive from the TCP graph that the buffer size is the difference in bytes between the crest and the trough of the saw-tooth graph. Also, it is equally intuitive that for different scenarios, rates of transmission and congestion levels, the saw-tooth is going to vary, resulting in varying buffer requirements. A simple example of this would be the sawtooth graph that is shown above. It has different slopes of the saw-tooth for different speeds.

There are numerous ways in which we could extrapolate this, for example, long and short flows and they will also be different in nature. The Internet has evolved at such a rapid pace that anything related is no longer allowed to remain constant for a long time. With the advent of the Internet some decades ago, there were not many concurrent flows and hence a $C * RTT$ capacity buffer would suffice to keep the bandwidth fully utilized. Also, long haul links were used then and they were expensive and hence systems were designed in a manner as to utilize these efficiently [7]. Nowadays, with bandwidth available in excess with the advent of optical networks, a minor drop in bandwidth does not affect the network that much and the focus has shifted from maintaining maximum permissible bandwidth to minimizing packet loss and providing quality of service. With this paradigm shift in focus, we need to take a look at buffer sizing all the time to suit the needs of the ever-changing network requirements.

CHAPTER 3

PACKET PACING IN NETWORKS

3.1 Packet Pacing: Why it is essential

Till now, we have dwelt on the burstiness of traffic in the Internet and the changing buffer size requirements. Let us now take a step back and see why packet pacing would make a difference.

1. Like discussed above, the move towards an all-optical core network dictates the usage of small buffers with high-speed optical links. The inability to use large buffers in this case to hold packets makes it all the more essential to control the pacing of packets.

2. The Internet operates with TCP as the predominant transport layer protocol, apart from the odd use of UDP when the requirement mandates it. The inherent bursty nature of TCP traffic itself, coupled with the existence of multiple short TCP flows and small buffers makes data loss a real probability and packet pacing a necessity.

3. This approach is in accordance with the move towards next generation Internet architectures and is interesting in the sense of its applicability. Since majority of the traffic over the Internet is TCP, and most of it happens to be from short TCP flows, it becomes all the more imperative to consider this as a part of the next generation Internet architecture.

3.2 Small Buffers and TCP Packet Packing

As dealt with in Chapter 2, optical networks are becoming the order of the day. In order to fabricate a true optical router, a couple of problems have to be solved first. The first problem is buffering packets until the link becomes available for transmission. This problem involves holding the packet in the data plane of the optical domain, which is not an easy thing to do. If you consider the routers that exist today, the buffering is done by electronic FIFOs.

In the move towards optical networks, ongoing research has showed that integrated photonic circuits can buffer a few packets on-chip before switching them onto the optical delay line. Larger optical buffers are out of the question since they are modeled on the basis of FDLs. Larger buffers would mean larger coils of FDLs which is not feasible. Also FDLs model delay lines and not true FCFS buffers. TCP pacing performance studies by Aggarwal et al. [1] has shown that pacing can improve the throughput, implying reduced delays in some cases but at the same time can also impact performance. If we take a look at the link utilizations in the core of the Internet, we can see that they are operated at extremely low link utilizations. There are many causes to this, including non-optimized configurations as well as staggered traffic aggregation dependent on varying geographical utilization. The core network ends up having 20-30% utilization. If this is the case, losing a bit of performance due to small buffers in fully optical core networks that operate at high network rates is not going to result in a performance degradation. It is well known that TCPs sliding window mechanism and its built in congestion control tend to generate heavy bursts of traffic. This behavior is exaggerated when high bandwidth lines are multiplexed or aggregated together before they enter the core network. The bursts produced by each TCP flow ends up loading the ingress buffers and hence produce long wait times and eventually packet drops [15]. As discussed above, recent studies have shown that

when TCP traffic is statistically paced and prepared before it arrives at the core, small buffers are sufficient to service the requirements of non-bursty traffic [1] [4] [8].

We could argue a case like stated in the previous chapter saying that access networks are magnitudes of speed slower than their core counterparts. This would hold good with restrictions. But something that has not been accounted for in this analysis is that when it comes to the core network, there are thousands of smaller networks pouring in their traffic. If each of these smaller networks generated bursty traffic that eventually ended up inside the core, then the assumption that the slow link speed of the access network would make up for pacing would not be valid anymore. There are some real scenarios where packet pacing could prove to be beneficial. For applications that are delay and jitter bound, like video streams, bursty traffic would mean loss of packets sometimes. And for video streams, loss of packets means loss of data continuity in the frame. Consider for the same time sensitive application if the packets were paced, they would egress at a much smoother pace and hence would maintain they delay and jitter bounds. There has been some research in this area [9] [10].

Moving the discussion again to the TCP segment, there are some instances where a certain area of research has given us insights into how things actually work, but at the same time have missed out on accounting for some important points. Villamizar and Song, in their paper High Performance in TCP ANSNET have experimented with the performance and behavior of TCP with different buffer sizes. Their analysis led to the conclusion that the famous $C * RTT$ thumb rule for buffer sizing holds good in their WAN testbed, where they defined delay or RTT as the amount of time taken for a round trip by a packet from a single TCP flow trying to saturate the network. Mathis et al also state this on a more general scale [6]. What Villamizar and Song did not measure or take into account was the existence of multiple flows having different delays [14]. This would definitely have a different bearing on the performance since all the TCP flows are individually trying to further their gains and

are contending for the link. In this case, $C * RTT$ will not be enough. For TCP to perform at full bandwidth, a buffer size of $CXRTT$ would have to be provided at every single bottleneck link [6]. Morris studied the effects of multiple contending TCP flows and analyzed the data and performance statistics. He proposed a method in which the buffer size is dynamically proportioned on the basis of the number of contending flows [8].

3.3 Packet pacing in the network

Having discussed the effects of buffer sizing and the need for packet pacing, we now proceed to see how pacing can be deployed in the network.

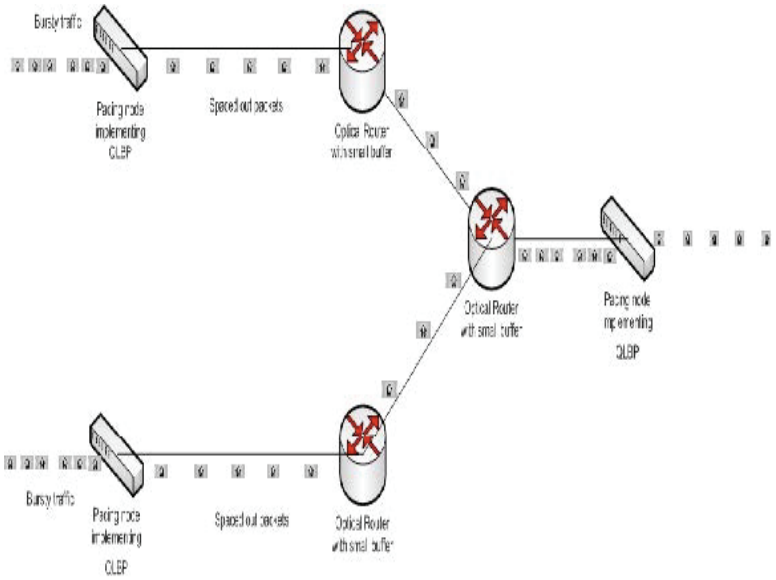


Figure 3.1. Network Pacing serving ingress and egress criteria

Figure 3.1 shows nodes implementing pacing. Two criteria are met and both are shown in this figure for comparison. We have been talking about pacing all this while. We will not look at some criteria as to where pacing can be applied. Pacing is applied at the 3 edge routers/switches on the access network (considering that the 3 routers in between are part of the core). With all this talk about pacing packets,

we need to remember that we need to pace traffic before it reaches the core ONLY if it is bursty. There is no point in pacing traffic that is not bursty. That would only consume resources onto no useful end.

The first criterion or role that pacing could play here would be to act on the ingress router at the edge of the core and enforce pacing rules to incoming bursty traffic. If the traffic is not bursty by not conforming to a certain characteristic, for example, if the inter-arrival times are not comparable to a pacing parameter that defines bursts, then there would be no need to pace this traffic. Traffic that is bursty can be held in the buffers to space the packets out accordingly and then sent out to the core.

The second role would be as an egress router which checks if the traffic delivered to it by the core could again cause bursty behavior in the access network. There is not much of difference in the way these two criteria behave in the data plane. Data is just forwarded in the data plane. The difference exists in the control plane architecture for both. The ingress router criterion checks to see if a statistical characteristic is present or not and then paces the traffic on the basis of that. The egress router criterion checks to see if the pacing characteristic has been achieved and if the core network contributed any bursty behavior. If it did, then the egress router paces the traffic again before delivering it to the access network.

CHAPTER 4

QUEUE LENGTH BASED PACING

In this chapter, we will take a look at the concept of Queue Length Based Pacing and how it actually tries to regulate packets in the network. With all the discussions about pacing in the previous chapter, the actual implementation of a pacing node becomes interesting since there are a lot of scenarios to consider. In this chapter, we look at one of the proposed solutions, that being Queue Length Based Pacing, henceforth QLBP. The principle behind QLBP is the dependence of the algorithm on the instantaneous queue length in comparison to its maximum queue length. The amount of delay a packet undergoes is dependent on the state of the delay queue when it enters it. In brief, if the packet enters a loaded queue, it experiences a low amount of delay than it would normally have it if entered a queue that was empty. A packet that enters an empty queue is the one that is going to experience the maximum delay.

4.1 QLBP Design

This scenario will be better explained with the time-line in figure 4.1. We first show the burst arriving and then how QLBP spaces packets out.

What the time-line in figure 4.1 shows is that packets arrive in a burst and are held in the queue. The pacing controller determines when the packet at the head of the queue is going to go out of the queue. Packets arrive at times T1 to T6 closely spaced together, which means the inter arrival time of the packets is not too long. When they depart, they are held in the delay queue for different amounts of time by the pacing controller.

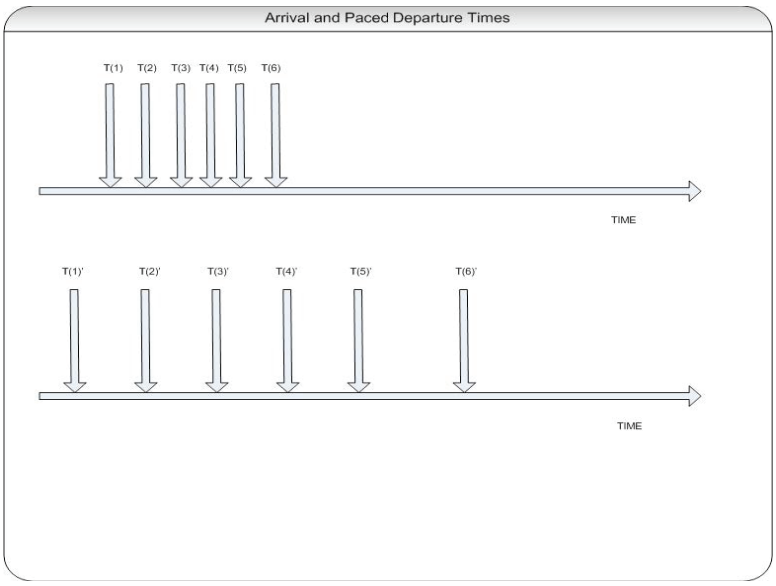


Figure 4.1. Pacing showing arrival and departure time of packets

The design of the algorithm is such that it takes into account the instantaneous queue length with respect to the maximum length of the queue and calculates the delay based on the present load. If the queue happens to be lightly loaded, the packet is held for a longer duration of time. If a packet is enqueued into an almost full FIFO, it will be sent out almost instantaneously because we do not want to cause any packet drops due to the design.

Going into the details of how this happens, the pacing controller calculates the delay for the packet at the head of the queue and then blocks the queue. It then starts a timer and waits for it to expire. In the eventuality of another packet ingressing without this timer expiring, the next transmission time is again recalculated and the timer re-initialized to the newly calculated value. If no packet ingresses and the timer expires, the packet at the head of the queue is transmitted and the transmission time is recalculated for the next packet in the queue. This behavior should produce the sort of spacing shown on the time-line previously.

It would be apt to familiarize ourselves with the QLBP notation and system from wherein we can move on to the actual algorithm in the next section. The notation is as follows:

- $\lambda(t)$: This is the arrival rate of input traffic at any given time t.
- $\mu(t)$: This is the pacing rate that the controller applies to a packet at any given time t.
- $d(t)$: This is the amount of delay time that the pacing controller will hold the packet for.
- $q(t)$: This is the length of the delay queue at any given instant t.
- D_max : This is the maximum amount of delay that the pacing controller can apply to a packet. This helps in keeping the delay bound and not letting it grow indefinitely.
- Q_max : This is the maximum queue length of the delay queue.
- S_p : This is the size of the packet being paced, rather the size of the packet at the head of the queue.
- S_max : This is the maximum packet size that can be transmitted over the Internet.
- μ_{min} : This is the minimum threshold of input traffic that the traffic controller takes into account. Any rate of traffic greater than this will be paced. Conversely, any rate of traffic lower than this will not be paced.
- μ_{max} : This is the maximum pacing rate that the pacing controller can possibly produce.

The algorithm as explained in the QLBP paper [3], the transmission rate $\mu(t)$ at time t is determined by queue length $q(t)$ as follows:

$$\mu(t) = \begin{cases} \frac{\mu_{max}-\mu_{min}}{Q_{max}} q(t) + \mu_{min}, & q(t) < Q_{max} \\ \mu_{max}, & q(t) \geq Q_{max} \end{cases} \tag{4.1}$$

The meaning of this pacing scheme is interpreted as follows: If the instantaneous queue length is lesser than the maximum queue length, then the delay is dependent on the instantaneous length of the delay queue and the pacing parameters. For any queue lengths that are greater than or equal to the maximum permissible length, there is no delay and the pacing rate is the same as the rate of the incoming traffic.

The parameter Q_{max} is the maximum queue length at which the delay becomes negligible because of the queue being full. At this queue size, the delay is equal to the transmission rate of the packet.

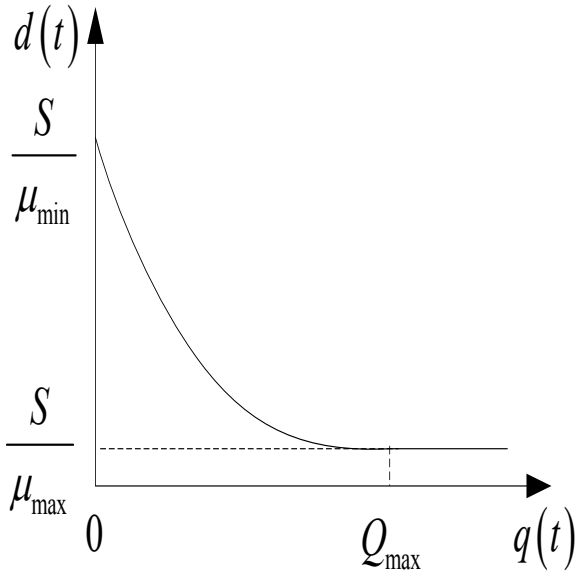


Figure 4.2. Pacing Delay of QLBP (from [3])

4.2 The QLBP Algorithm

The packet pacing scheme approximates constant bit rate traffic from bursty traffic by delaying the transmission of packets in the burst. Since the arrival times of packets are not known, the pacer uses a dynamically adaptive scheme to adjust the delay of packets so that the traffic rate mimics CBR.

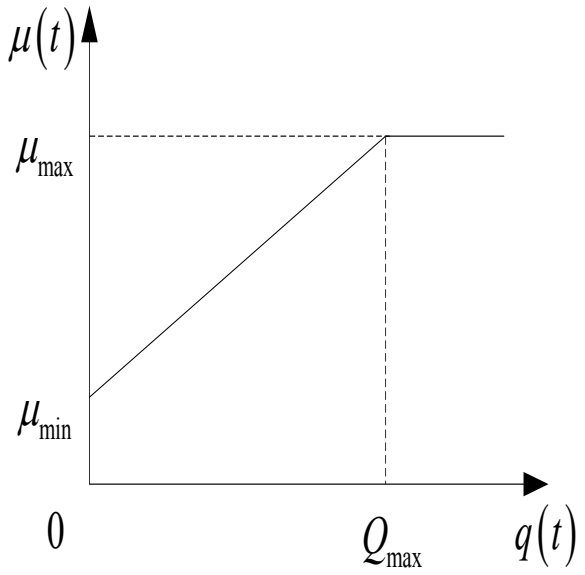


Figure 4.3. Pacing Rate of QLBP (from [3])

From the quoted parts of the QLBP paper, where the delay and the rate are calculated, it is necessary that we translate this scheme into an algorithm that can efficiently implement it. Again, since this pacing scheme and algorithm came from the QLBP paper, we quote verbatim the algorithm.

The algorithm works in the following manner:

1. The queue and the last transmission time are initialized to zero.

2. As evident from the algorithm, it consists of two main functions, *handle_packets* and *send_packets*.

3. Every time a packet arrives into the system, *handle_packets* is called. It enqueues the packet in the delay queue (except the first packet) and calculates

Algorithm 1 QLBP Algorithm

```
1: q ← empty_queue()
2: t_last ← 0
3:
4: function handle_packet(p)
5:    enqueue(q,p)
6:    t_next ← t_last + S_p/((μ_max − μ_min)/max_length(q) · length(q) + μ_min)
7:    if t_next > system_time() then
8:       callback(t_next,send_packets())
9:    else
10:      send_packets()
11:   end if
12: end function
13:
14: function send_packets()
15:   while (system_time() > t_next) ∧ (length(q) > 0)
16:      p ← dequeue(q)
17:      transmit_packet(p)
18:      t_last ← system_time()
19:      t_next ← t_last + S_p/((μ_max − μ_min)/max_length(q) · length(q) + μ_min)
20:   end while
21:   if length(q) > 0 then
22:      callback(t_next,send_packets())
23:   end if
24: end function
```

Figure 4.4. The QLBP Algorithm

the time at which the packet is supposed to be sent out as indicated by lines 5 and 6.

4. Lines 7 and 8 show that if the calculated transmission time is greater than the system time, i.e. it is in the future, the packet transmission event that is scheduled is canceled and a callback to the NIC is initiated. A new transmission event at the calculated time is scheduled.

5. If its calculated transmission time is greater than the system time (only two things could have caused this: the queue is heavily loaded or the packet rate is really high and hence every recalculation of the next transmission time has taking it closer to the actual system time), we start transmitting packets by calling *send_packets* immediately because we do not want to encounter or generate any packet losses.

6. *send_packets* continues to send packets out of the queue as long as the calculated transmission time is below the system time, which is indicative of the fact that the system time has already moved ahead and we should not buffer the packets any longer. Before doing any of this, we first update the last transmission time with whatever time we are sending out the packet and then dequeue the packet.

7. We recalculate the transmission time for the next packet at the head of the queue.

8. If the transmission time becomes greater than the system time, we then schedule a transmission event for that packet at that time with the kernel or NIC.

CHAPTER 5

DESIGN AND ARCHITECTURE OF THE HARDWARE IMPLEMENTATION

This chapter discusses the design and architecture of the hardware implementation of the pacer and what it aims to accomplish. We will first list out the goals and objectives, then move on to the platform being used for the implementation and finally talk about the results in simulation. These are the implementation goals:

1. To implement the QLBP algorithm on the NetFPGA hardware as an equivalent working prototype of the one implemented on the network simulator NS2.

2. Achieving the above listed objective would mean changing the existing IPv4 reference router design to include the new module that would provide the functionality of the algorithm.

3. Once we modify the reference router design to incorporate the pacing module, it is to be simulated structurally and behaviorally in Modelsim.

4. After ensuring correct functionality in simulation, the goal is to test is on hardware with a simple two node network topology for correctness of operation.

5. Once it is deemed to be operational on the FPGA, various experiments are to be performed with respect to throughput, packet drop rate, average/instantaneous queue length etc.

5.1 The NetFPGA Platform

The NetFPGA is an open platform for gigabit Ethernet switching and routing that has been developed at Stanford University [13]. It is a complete network hardware platform implemented on a FPGA. It is mostly used to build and test out new protocols and networking devices. Going a bit deeper into what NetFPGA is all about and its specifications, it comprises primarily of a PCI card containing a large Xilinx Virtex-II Pro FPGA, a smaller Spartan FPGA, 4 gigabit Ethernet ports, SRAM and DRAM. It enables researchers to build working prototypes of high-speed, hardware accelerated networking systems.

5.2 The IPv4 Reference Router

The IPv4 reference router that is part of the NetFPGA base package consists of various library modules that are all plugged in together in a pipelined fashion to enable the design to operate at 125 MHz [13]. It is essential to understand the modular design of the NetFPGA to make optimal use of the existing design. Figure 5.1 shows the pipelined structure of the NetFPGA reference design. We first take a look at that design and then go into the details of our design. There are primarily 4 Gigabit Ethernet ports on the FPGA. Each of these can of course act as an ingress or egress port for the router. In addition to these ports, we have 4 other logical DMA ports, corresponding to each physical port that act as the conduit between the physical port and the host computer on which the PCI card resides. If the reference router is not able to process or handle any packet, it is transferred onto the respective DMA port and sent across to the CPU for processing.

Apart from the 8 ports, the NetFPGA has 3 other modules that comprise the reference design. Before we go into the details of each of these, it would be useful to note that in between all the modules, there exists a corresponding input FIFO into which the previous module writes its processed output. This input FIFO is used for

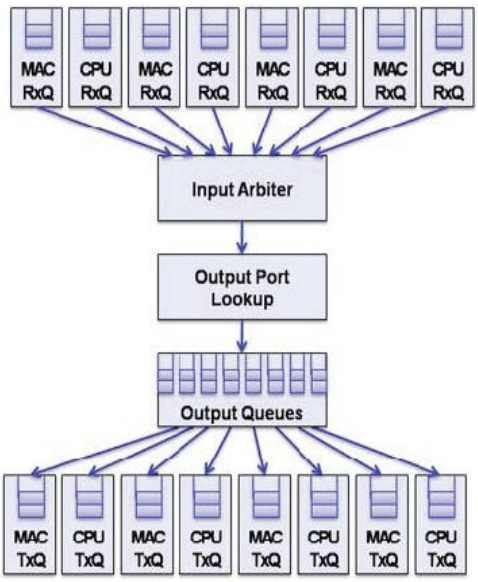


Figure 5.1. The IPv4 Reference Router

pipelining the design. The current module reads the data from the input FIFO as and when required and processes it. The first module is the input arbiter. It is a round robin arbiter that picks up packets from the input queues and queues them in the Output Port Lookup modules input FIFO. The O/P Lookup module does the packet processing, namely, parsing the Ethernet header, the IP/LPM lookup, and destination filtering and finally, modifying the Ethernet headers. It then hands the packet to the Output Queues module that initially stores the packet in the SRAM and then hands it to the corresponding physical port when it is available for transmission.

5.3 Architecture and Design of the Pacer Module

Now that we have a brief idea about the reference router that we are going to modify, we can take a look at our design and the methodology behind the implementation. Since this is the initial prototype implementation for proof of concept, we are going to implement a single module, evaluate it for functional and behavioral correctness, along with conformance to the theoretical standard and then on the basis of the

results move forward to the full implementation. Pacing is going to be accomplished using a delay queue and figure 5.2 is what the module is going to look like. It is going to exist in between the Output Port Lookup module and the Output Queues module.

The pacer module takes packets from the Output Port Lookup module after they have been switched/routed and then puts them into the delay queue till they are ready to be transmitted. The operation of the various sub-blocks are explained in the coming sections. For now, let us just list the blocks for introductory purposes. They are the Input State Machine, the Signal Control Block, the Delay Lookup Block and the Output State Machine.

5.4 Pacer Module Sub-Blocks

In this section, we will examine the working of the pacer as split into functions of its sub-blocks. Each section below clearly outlines the responsibility of each block and how it interacts with the other blocks.

A top level overview of the pacer state transition is provided in figure 5.3. This graphically depicts the individual sub-block functions mentioned above.

The sub-blocks have their individual state machines and state transition controls. They are not discussed here for the sake of brevity. As discussed above, there already exists an implementation of the pacer on the NS-2 software simulator. We are trying to obtain an event model that is as close to the software event model. Since it is not possible to completely model it in the same way, a few allowances have been made for the hardware design.

In the software implementation, the first packet is never delayed. The packet size is used to calculate the next transmission time and the queue is then blocked from transmitting any packets till the timer expires. Since this is not entirely feasible in the hardware implementation, this behavior is implicitly handled by the hardware.

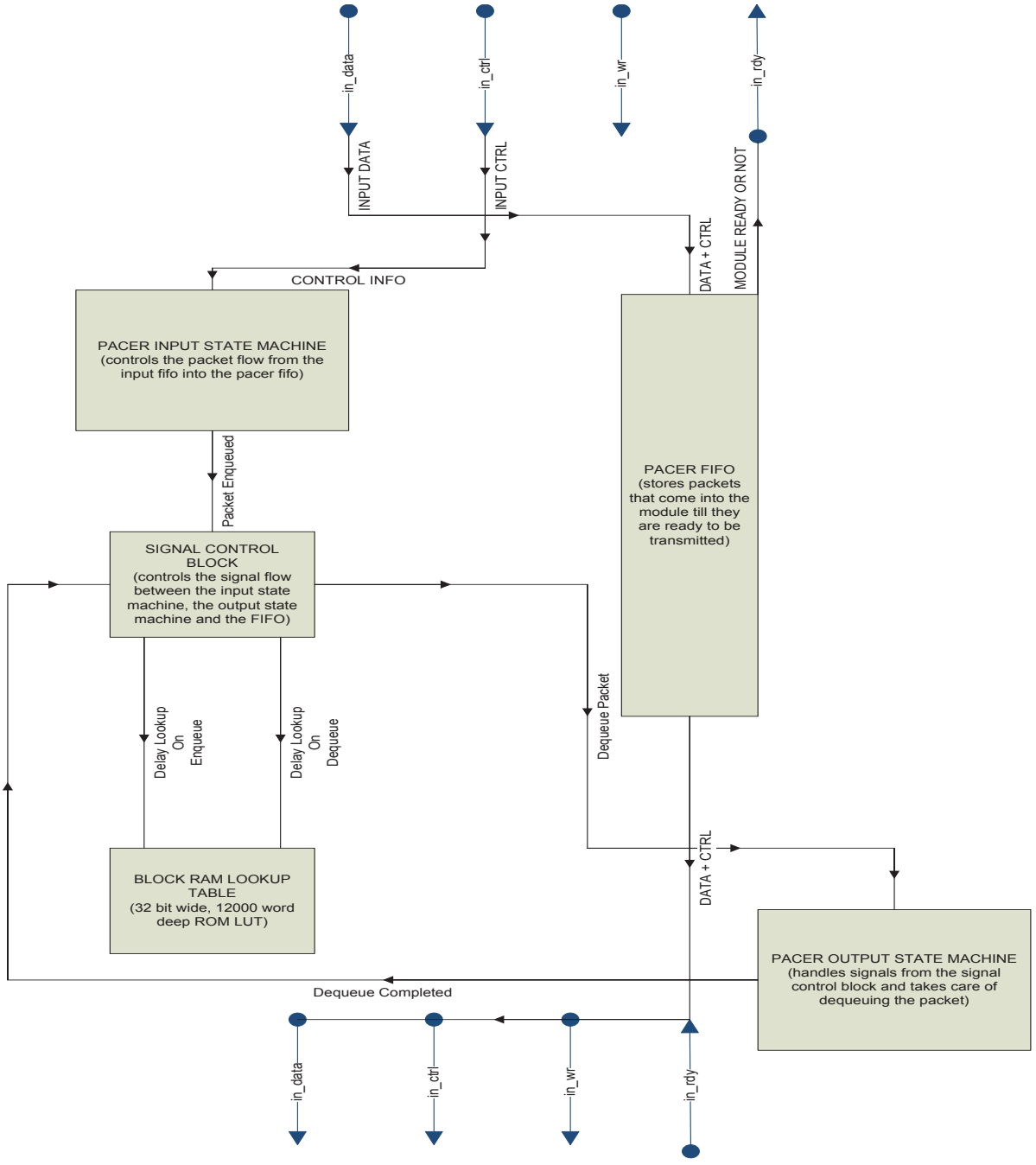


Figure 5.2. The Pacer Module Block Diagram

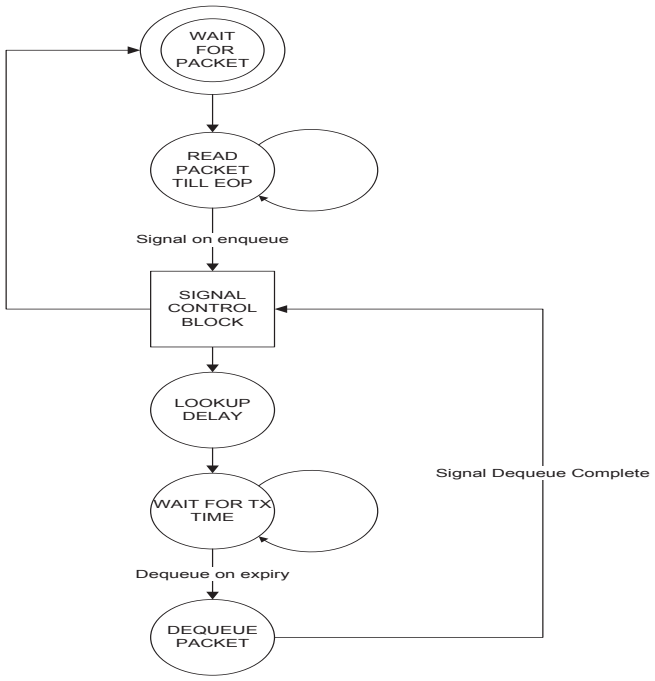


Figure 5.3. The Pacer State Machine

For isolated packets, the next transmission time is almost always exceeded when the delay is looked up and hence they are transmitted instantaneously.

5.4.1 The Input State Machine

The Input State Machine or the IPSM handles data transfer from the input FIFO to the delay queue. Whenever data is available in the input FIFO, the IPSM is activated and does the data transfer. At the same time, it is responsible for obtaining the size of the incoming packet and providing it to the Signal Control Block which maintains the queue size information and does the post-enqueue processing. Once done with one whole cycle of enqueuing, it goes back to waiting for new data to arrive.

5.4.2 The Signal Control Block

The Signal Control Block or SCB handles the communication between the Input State Machine(IPSM), the Output State Machine(OPSM) and also the delay lookup

on either enqueue or dequeue. It is the fabric that holds together all the other sub-blocks. It also maintains a lot of state information for the pacer, like the queue size, the next transmission time and the previous transmission time. The SCB also maintains the timer state to signal expiry and packet removal times to the Output State Machine.

It is initially in a wait state waiting for a signal from either the IPSM or the timer control for indications of a packet enqueue or dequeue event. Once the Input State Machine is done enqueuing a packet, it sends a signal to the SCB asking it to do a delay lookup for the respective packet and queue sizes. On obtaining the respective delay time, it updates its next transmission time register.

The timer block keeps checking against this next transmission time register on every clock cycle. Once it hits the specified clock cycle count, it activates a signal that tells the Output State Machine or OPSM to dequeue the packet at the head of the queue. In the meantime, the SCB holds its state till the OPSM completes dequeuing the packet. After the OPSM indicates that is has finished dequeuing the packet, the SCB updates its queue size and then does a delay lookup once again to update the next transmission time for the new packet at the head of the queue.

5.4.3 The Delay Lookup Block

The Delay Lookup Block or the DLUT is a single ported ROM implementation that is pre-loaded with a set of values that correspond to the delay produced for a certain queue and packet size combination. For the purposes of our implementation, the queue size is bounded by the size of the delay queue. The delay queue in our case can hold 512 words, each word being 64 bits wide which is the equivalent of a 4KB buffer. We consider packet sizes ranging from 1 word or 8 bytes (highly improbable) to 187 words or 1500 bytes (the MTU over the Internet) and as a result, we can have a lot of permutations and combinations. The ROM is pre-loaded with delay values

for all the possible combinations of queue length and packet size. This is in the order of around 96,500 entries.

The advantage of using a ROM as a lookup table as opposed to doing the whole delay calculation as proposed in the algorithm is a trade-off between the time taken for the calculation and the precision obtained. If the delay module had been implemented as a precision module modeled on floating point operations, it would take up as much as 30-40 cycles per delay calculation (on every packet enqueue or dequeue). With its current implementation, the delay lookup only takes one cycle with two more cycles taking up the post processing time. This is just 10% of the time taken but at the cost of some stepping in the precision of the pacing. Consequently, we are not going to get a smooth curve when we plot the instantaneous queue length against the delay, but we are going to get a sharper curve.

5.4.4 The Output State Machine

The Output State Machine or OPSM handles the packet dequeue process. It is activated by a signal from the SCB when the timer expires (i.e. when the next transmission time has been reached). It then dequeues the packet at the head of the delay queue. Once it completes dequeuing, it sends an acknowledgement signal back to the SCB so that the next transmission time can be calculated for the new packet at the head of the queue. The OPSM has to also handles situations like ones arising from scenarios where the next transmission time has been reached, but the next module is not ready to receive packets etc.

CHAPTER 6

EVALUATION AND DISCUSSION OF RESULTS

In this chapter, we will discuss the results from the prototype implementation of the pacer. In the results section, we will be looking at these points specifically:

1. The first and last packets of the burst getting spaced out.
2. The packets in the center of the burst getting transmitted at full rate.
3. The system adapting to the delay as new packets arrive into the queue.
4. We will also be looking at a graph of $q(t)$ Vs $d(t)$, where $q(t)$ is the instantaneous queue length and $d(t)$ is the delay introduced by the pacer at that instant of time.

6.1 Evaluation

In this section we evaluate the functional correctness and hardware overhead of implemented pacer. We will compare our implemented pacing capable router against the the base reference router implementation that comes with NetFPGA. This will give us some insight on the area overhead of deploying pacers inside the reference design.

6.1.1 Hardware Overhead

The table 6.1 shows the resource usage in the reference router design that is part of the NetFPGA base package. In addition to that, it shows the overhead that the lookup table implementation adds to the reference design which is around 3%. It is

Table 6.1. Resource Utilization

	Reference Router		Packet Pacer		
Resource Type	Count	Utilization	Count	Utilization	Overhead
Flip-Flops	16,431	34%	17,920	37%	3%
4 Input LUTs	22,961	48%	24,552	51%	3%
Block RAMs	106	45%	133	57%	12%

evident that the pacer implementation is not very heavy on hardware resources and constitutes a minimal increase on the existing design.

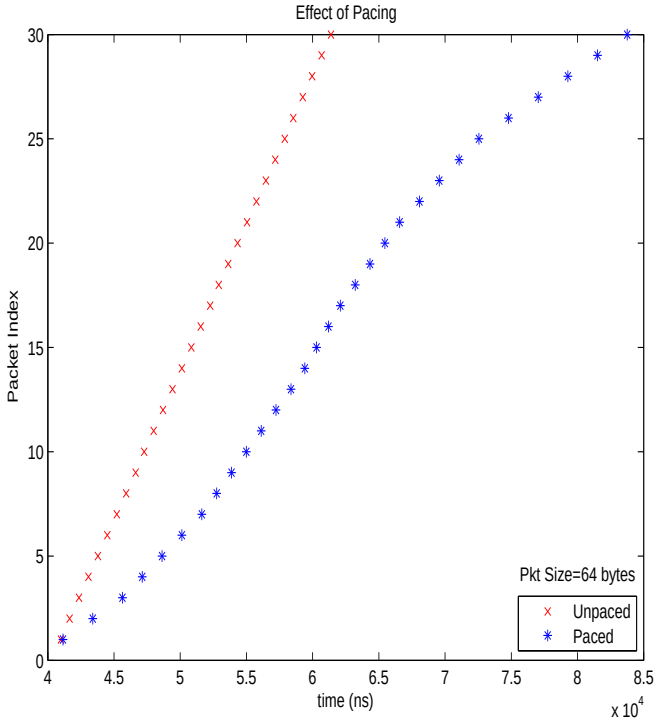
The Block RAM usage might be a bit higher than the one noted and documented here because of some changes necessitated to the design of the lookup table.

6.2 Discussion of Results

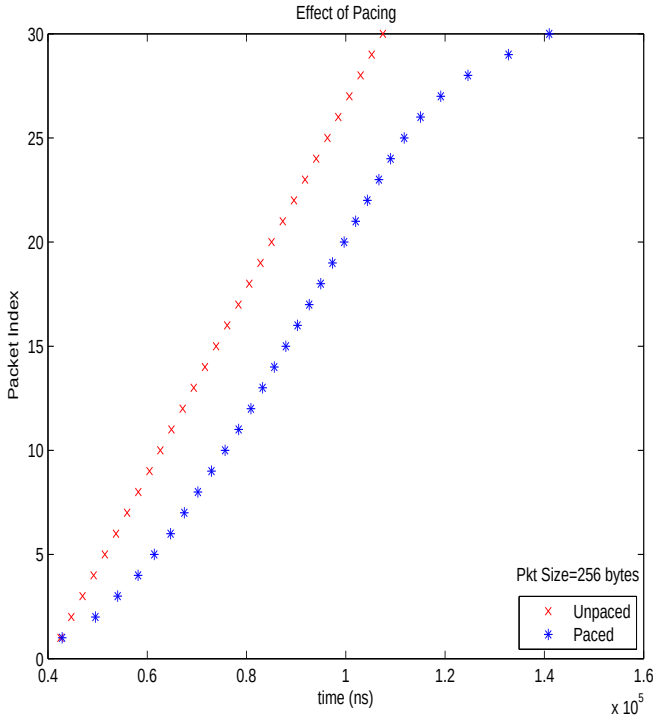
6.2.1 The Effect of Pacing

Since we are attempting to model an ideal system, the results or plots reflect the ideal situation and may not be encountered in normal operation. Let us have a look at the results being produced to evaluate the pacer for correctness of operation. We will look at plots that compare the reference router without pacing against the one with pacing applied for various packet sizes.

If we look at the plot for the pacing effects of 64 byte packets in figure 6.1(a), we notice that the packets are delayed as per the algorithmic expectations. Also, one other important thing is that in the middle of the graph, the line straightens out towards the Y-axis. This is indicative of the fact that when the queue begins to fill up, or rather the load on the queue increases, the delay decreases. This behavior translates to a cluster of packets being sent out at around the same time. If the load on the queue is pushed up towards its maximum, the line will become linear like the non-paced one indicating that packets are being sent out at line rate without any delay.



(a) Effect of pacing for 64 byte packets



(b) Effect of pacing for 256 byte packets

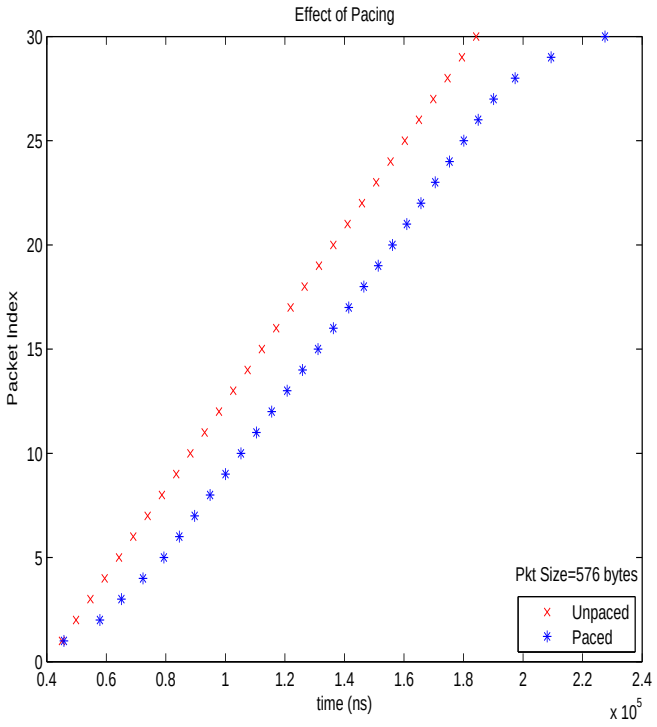
Figure 6.1. Effect of pacing on 64 and 256 byte packets

The plot for the 256 byte packets in figure 6.1(b) shows almost the same behavior as the 64 byte plot. The two remaining plots show the pacing effect for 576 bytes and 1300 bytes. One of the main points to be taken into account here is that as the packet sizes increase, the effect of pacing is not that dominant. This could be explained with respect to queue occupancy for certain packet sizes. If we consider a certain maximum queue size, as the packet size increases, the queue occupancy at any given point of time varies accordingly. Take for example the 4KB queue size that we have in our hardware. Since memory is costly, the queue size cannot be increased without limit. If we have 1300 byte packets, then it would take 3 packets at the most to increase the queue length to such a level that the delay starts becoming constant. By constant, we mean that the performance will come to a point where one packet ingresses, the delay falls to a low value and because of that, one packet egresses. As a result of this, the queue occupancy fluctuates between two or three distinct values and the effect of pacing diminishes.

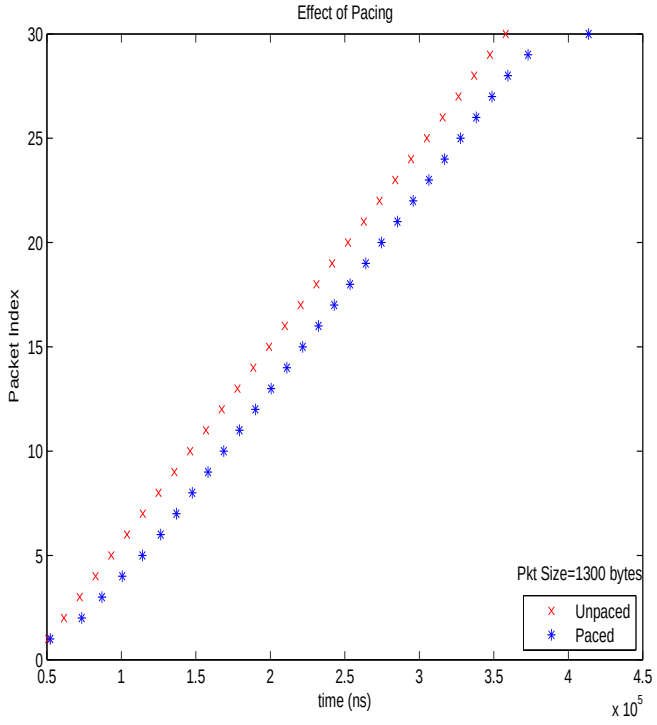
If we want to observe the effect of pacing with respect to big packet sizes, it would be more advisable to have them interspersed with smaller packet sizes as depicted in figure 6.3. The following plot shows us the pacing effect with multiple packet sizes. It is also observable that in the middle, where packets seem to be clustered together, it can be construed that the queue occupancy has hit a level where the pacer calculates that packets need to be released at a faster rate. Once the queue length is again down to a lesser value, the pacing increases and the next packet is delayed for a much greater time.

If we want to consider how the delay is adapted according to changes in queue size and packet size, we can take a look at table 6.2.

We consider the pacer at some instant in time 't'. The pacer was initialized in the beginning with the queue size, delay and next transmission time all as 0. The measurement of queue size is words (where 1 word=8 bytes), delay is in clock cycles



(a) Effect of pacing for 576 byte packets



(b) Effect of pacing for 1300 byte packets

Figure 6.2. Effect of pacing on 576 and 1300 byte packets

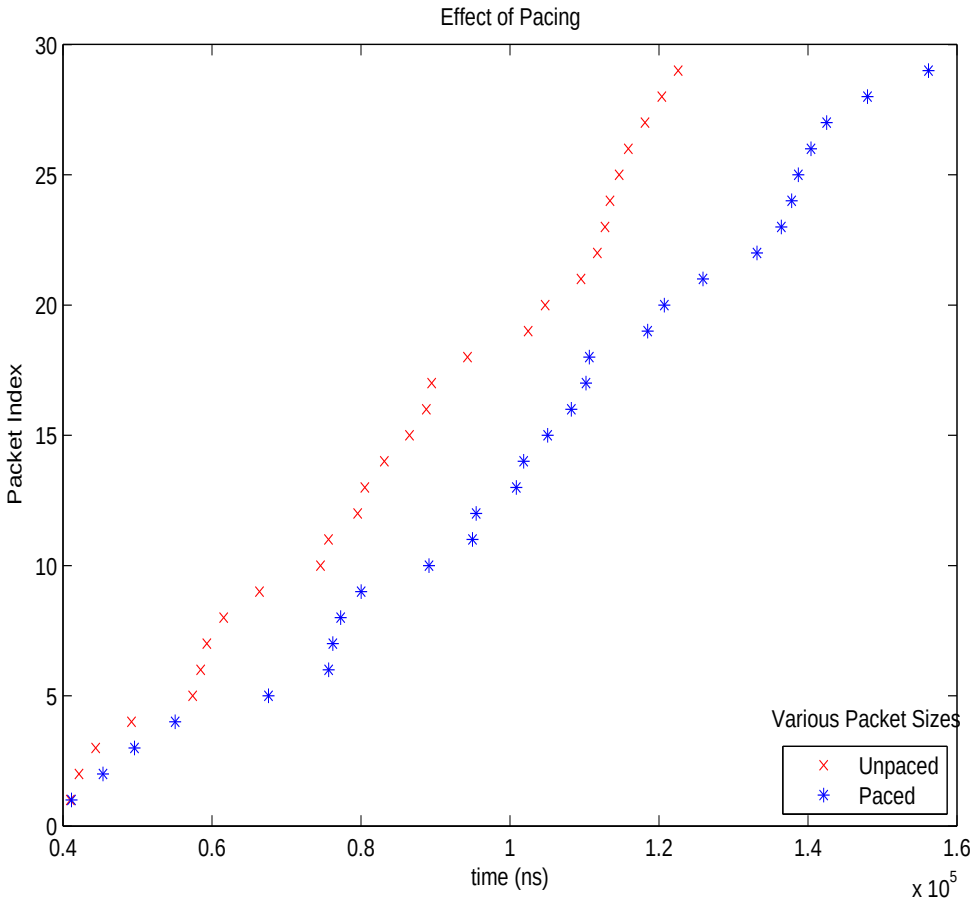


Figure 6.3. Effect of pacing random packet sizes

Table 6.2. Adaptive Delay for Different Queue Sizes

Queue Size	Delay	Next Transmission Time
17	527	5199
50	527	5199
33	527	5726
106 (73)	682	6408

and the next transmission time is the corresponding i_{th} clock cycle. We assume that one packet has arrived and and has already departed (analogous to the software implementation of the first packet never being delayed). At this point, the queue size is 0 words again and but the last transmission time was updated to the time the previous packet left the queue, in our case, the 4672nd clock cycle. We will show a sub-section of a 30 packet burst with its respective queue sizes, delays and adjusted transmission times.

Here is the sequence of events to explain the contents of table 6.2.

1. The first packet has arrived and has been sent out, thus updating the last transmission time to the 4672nd clock cycle. At this point the queue size is 0 words.

2. The next packet of size 17 words arrives. Queue size becomes 17 words, delay for this packet and queue size is 527 clock cycles and hence the next transmission time gets updated to the 5199th clock cycle.

3. 33 word packet arrives. The queue size increases to 50 words now. But for this packet size and queue size permutation, the delay still stays 527 clock cycles. Since no packet has still be sent out, the last transmitted time is still the 4672nd clock cycle and hence the adjusted time remains as the 5199th clock cycle.

4. At this point, the 17 word packet is sent out and the queue size is down to 33 words now. The packet at the head of the queue is 33 words now. The delay for this new combination is still 527 clocks. But since the 17 word packet

was sent out, the last transmission time was updated to some value and the adjusted next transmission time value is now the 5726th clock cycle. The next transmission time increases on decrease in queue size.

5. The last part is a little complicated. A 73 word packet was enqueued. In hardware, this takes some time to happen, 73 clock cycles to be precise. Before the next and last transmission times could be updated, the 33 word packet had to be sent out. As a result of this, we do an aggregate update of both the changes together once the queue is stable after the dequeue of the 33 word packet. Due to this aggregate update, the last transmission time was updated to the 5726th clock cycle (as per the previous update) and now the new delay is 682 clock cycles. Subsequently the adjusted next transmission time becomes 6408 because of an increase in queue size.

This is in accordance with the theoretic definition where the delay has to decrease with increase in queue size. There are other sub-sections of the burst where the sizes and delay vary accordingly, but have been left out for the sake of brevity.

6.2.2 Delay Vs Instantaneous Queue Length

Another way of assessing the correctness of operation is to plot the delay at any given instant for the queue length at the same instant keeping the packet size constant. In the following plot of delay $d(t)$ against instantaneous queue length $q(t)$ in figure 6.4, we are given to understand that the delay is maximum (but still bounded) when the queue length is 0 and is equal to the transmission time (minimal or no delay) of the packet when the queue is full.

The line is not too smooth for the 1300 byte packet size since there are only 3 plot points. The fewer number of plot points are because of the reasons touched upon in the previous section. The plots for 64 byte and 256 byte packet sizes are a bit stepped because of the delay values that are associated with them. A lot of queue lengths for

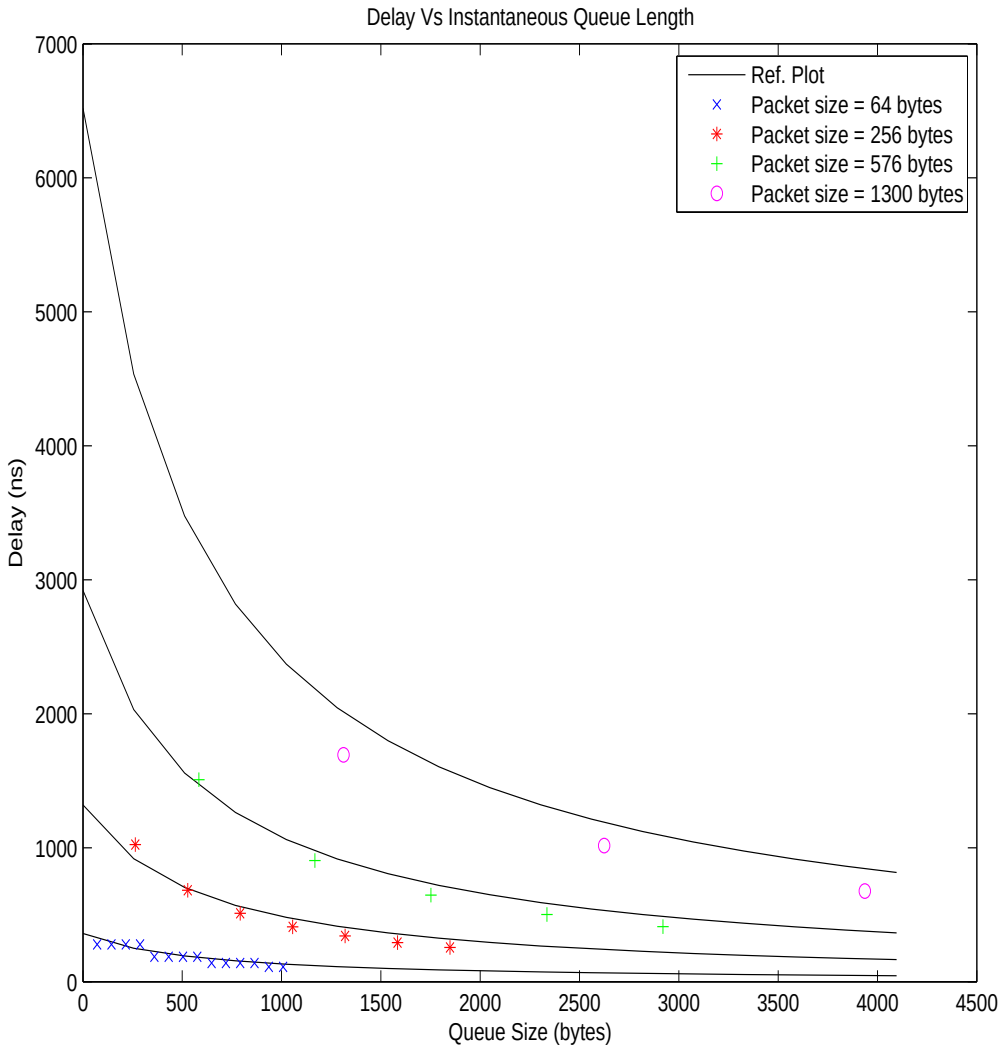


Figure 6.4. Delay Vs Instantaneous Queue Length

that particular packet size correspond to the same delay and hence the stepping in the plot.

It can also be seen from the graph that the plots for each of the different packet sizes are close to their ideal case counterparts. The minimal difference that can be noticed can be attributed to the precision calculations. The Delay LUT in hardware is loaded with values that were calculated using integer arithmetic. The ideal case delay plots were generated with values calculated using floating point arithmetic and then rounded off.

CHAPTER 7

SUMMARY, CONCLUSIONS AND FUTURE WORK

7.1 Summary and Conclusions

High performance optical networks with small packet buffers may well be a real possibility in the next generation Internet core. While a lot of emphasis is laid on speed and performance improvement and many other metrics, ensuring proper operation of existing protocols and traffic characteristics at the same time is of paramount importance. As explored and explained in previous chapters, widely used protocols like TCP have some traffic characteristics that might prove to be a bottleneck for small-buffer networks.

Prior research, publications and theses [3] [7] have shown pacing can potentially eliminate these issues and more significantly, ensure continual high performance operation of existing protocols over next generation networks.

This thesis has focused on the implementation of a high performance hardware pacer that can be implemented with low hardware overhead. It has also showed that the implemented pacer conforms to the algorithmic design to its permissible entirety and works as intended without degrading the performance of the router.

All in all, this thesis might contribute in its own minute way to the efficient operation of small-buffer networks and the widespread deployment of high speed networks including optical packet switched networks.

7.2 Future Work

As with almost all areas of research, there exist areas of continual improvement in design and implementation. Here are some aspects that might/have to be looked into in the near future.

1. Currently, the pacer is limited to run in the center of the reference router design. This means that all packets entering on all ports of the router are paced, irrespective of whether they need to be paced or not. This issue can be solved with minimal work since it requires moving the pacer to a different location in the reference router pipeline.

2. The current design can run at a maximum of 80 Mhz according to the synthesis reports. This can be optimized for better performance and made to run at 125 Mhz like the reference router.

3. Extended testing can be done in a real world environment like Emulab where real network traffic can be mimicked.

4. The current design is implemented with a lookup table for delay calculations. This reduces the precision of the system by some amount. If more precision is required, then a calculation module can be implemented for this (already present). The only downside of precision is that floating point operations take a lot of hardware and the latencies are pretty high. If low latencies are required, then a huge amount of hardware is needed to make this possible.

BIBLIOGRAPHY

[1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of tcp pacing. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1157 –1165 vol.3, March 2000.

[2] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 281–292, New York, NY, USA, 2004. ACM.

[3] Yan Cai, Y.S. Hanay, and T. Wolf. Practical packet pacing in small-buffer networks. In *Communications, 2009. ICC '09. IEEE International Conference on*, pages 1 –6, June 2009.

[4] Mihaela Enachescu, Yashar Ganjali, Ashish Goel, Nick McKeown, and Tim Roughgarden. Part iii: routers with very small buffers. *SIGCOMM Comput. Commun. Rev.*, 35:83–90, July 2005.

[5] Yu Gu, D. Towsley, C.V. Hollot, and Honggang Zhang. Congestion control for small buffer high speed networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1037 –1045, May 2007.

[6] Matt Mathis, John Heffner, and Raghu Reddy. Web100: extended tcp instrumentation for research, education and diagnosis. *SIGCOMM Comput. Commun. Rev.*, 33:69–79, July 2003.

[7] Anindya Misra. *Analysis of Queue Length Based Pacing*. PhD thesis, University of Massachusetts, Amherst, 2009.

[8] R. Morris. Tcp behavior with many flows. In *Network Protocols, 1997. Proceedings., 1997 International Conference on*, pages 205 –211, October 1997.

[9] J.D. Salehi, Zhi-Li Zhang, J. Kurose, and D. Towsley. Supporting stored video: reducing rate variability and end-to-end resource requirements through optimal smoothing. *Networking, IEEE/ACM Transactions on*, 6(4):397 –410, August 1998.

[10] S. Sen, J.L. Rexford, J.K. Dey, J.F. Kurose, and D.F. Towsley. Online smoothing of variable-bit-rate streaming video. *Multimedia, IEEE Transactions on*, 2(1):37 –48, March 2000.

[11] V. Sivaraman, D. Moreland, and D. Ostry. Ingress traffic conditioning in slotted optical packet switched networks. *Proceedings of ATNAC*, 2004.

[12] V. Sivaraman, D. Moreland, and D. Ostry. A novel delay-bounded traffic conditioner for optical edge switches. In *High Performance Switching and Routing, 2005. Workshop on*, pages 182 – 186, May 2005.

[13] Stanford University. The netfpga platform for networking development. A networking research platform developed by researchers at Stanford University, 2010.

[14] Curtis Villamizar and Cheng Song. High performance tcp in ansnet. *SIGCOMM Comput. Commun. Rev.*, 24:45–60, October 1994.

[15] Lixia Zhang, Scott Shenker, and Daivd D. Clark. Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. In *Proceedings of the conference on Communications architecture & protocols*, SIGCOMM '91, pages 133–147, New York, NY, USA, 1991. ACM.