8-11-2007

An Efficient Algorithm and Architecture for Network Processors

Shalini Batra

AN EFFICIENT ALGORITHM AND ARCHITECTURE FOR NETWORK PROCESSORS

By

Shalini Batra

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Electrical Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

August 2007

AN EFFICIENT ALGORITHM AND ARCHITECTURE FOR NETWORK PROCESSORS

By

Shalini Batra

Approved:

Yul Chu Assistant Professor of Electrical and Computer Engineering (Director of Thesis)	Raymond S. Winton Professor of Electrical and Computer Engineering (Committee Member)
YaroSlav Koshka Assistant Professor of Electrical and Computer Engineering (Committee Member)	Nicholas H. Younan Professor of Electrical and Computer Engineering (Graduate Program Director)

Dr. Roger King
Associate Dean of
College of Engineering
Name: Shalini Batra

Name: Shalini Batra

Date of Degree: 11th August 2007

Institution: Mississippi State University

Major Field: Electrical Engineering

Major Professor: Dr. Yul Chu

Title of Study: AN EFFICIENT ALGORITHM AND ARCHITECTURE FOR NETWORK PROCESSORS

Pages in Study: 84

Candidate for Degree of Master of Science

A Buffer management algorithm plays an important role in determining the packet loss ratio in a computer network. Two types of packet buffer management algorithms, static and dynamic, can be used in a Network Interface Card (NIC) of a network terminal. In general, dynamic algorithms have better efficiency than the static algorithms. However, once the allocated buffer space is filled for an application, further incoming packets for that application get rejected. We propose a history-based scheme called History Based Dynamic Algorithm (HBDA), which reduces packet loss ratio by monitoring whether or not the application is active.

For average network traffic loads [5], the HBDA improves the packet loss ratio by 15.9% and 11% (for load = 0.7) compared to DA and DADT, respectively. For heavy

traffic load, improvement is 16.2% and 11.7% (for load = 0.7) and for actual traffic load improvement is 12.7% and 7.1% (for load = 0.7) over DA and DADT respectively.

We also developed a new architecture named Multiprocessor Architecture for the Network Interface Card. The new architecture will support the multi-processor system and gives more consideration to the application with the highest priority. It has two control units for processing the incoming packets in parallel. For the traffic mix with average network traffic loads [5], the new architecture improves the packet loss ratio for priority application by a significant amount.

DEDICATION

I would like to dedicate this research to my parents and my brother.

ACKNOWLEDGMENTS

I would like to thank Dr. Yul Chu, my research advisor, for his valuable guidance, advice, constant encouragement, and continuing assistance throughout this work. I sincerely appreciate the effort that he has put forth in guiding me through the entire process, and providing invaluable advice when needed. I want to thank my committee members Dr. Raymond S. Winton and Dr. Yaroslav Koshka for their valuable comments and suggestion. I would also like to thank Mr. Lane for his help in data collection.

Finally, I would like to thank the faculty and staff of the Department of Electrical and Computer Engineering for providing me with an excellent education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Packets are the basic medium of communication in computer networks [1]. Each packet consists of necessary data for an application associated with headers. Processing of packets, based upon their host application, is done by a protocol processor in a wired or wireless network terminal [2]. Processed packets (payload data) are stored in a packet buffer until they are accessed by the host application. In a packet buffer, packets reside in FIFO queues, each of which is associated with an application [3]. A Network Interface Card (NIC) is used for receiving the packets, processing the packet, passing the packet to the host processor, and sending the packet to other computers in a network. To increase the efficiency of an NIC, embedded processor is used in parallel to the host processor [2].

A Packet buffer is a large shared, dual-ported memory [6]. Packets for each application are multiplexed into a single stream. Packet buffer management algorithm determines whether to accept or reject a packet. The accepted packets are then placed into logical FIFO queues; each application has its own queue in a packet buffer. [2, 4]. The accepted packet remains in the buffer until the application retrieves it from the buffer. Once the buffer gets full further incoming packets are rejected.

Figure 1.1 explains a packet buffer with an example. As seen from figure 1.1, port 1 has a space for four packets with two packets already buffered; therefore, packet buffer

can accept only two more packets for application 1. Application 4 has a space for 5 packets and all the packets are buffered; therefore, if a packet for the application 4 comes, it will be dropped since no buffer space is available.

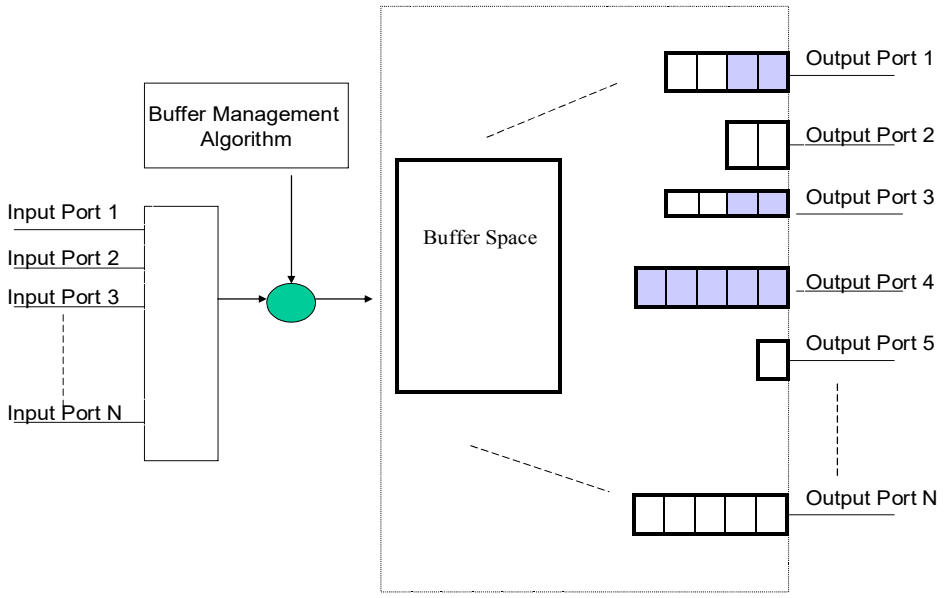


Figure 1.1: Packet Buffer

When the packet buffer is full, further incoming packets are dropped. This is called "packet loss." To achieve efficient end-to-end communication, reduction of packet loss is very important [5, 6]. Hence, to alleviate the ratio of packet loss, an efficient packet buffer management algorithm is needed. Efficient buffer space management can reduce the packet loss ratio. A buffer management algorithm determines how buffer space is distributed among different applications.

Therefore, a network processor, in general, should provide these basic functions:

1) Packet header parsing

2) Classification

3) Route Look Up

4) Packet header editing

5) Packet fragmentation

6) Packet storage

7) Packet scheduling and dequeue

The speed at which NIC can handle incoming data is a factor of number of packets that can be taken out (dequeued) of the buffer space at any time. If multiple packets can be dequeued from buffer space at the same time, then the number of packets in the buffer space will be reduced. Thus, NIC will be able to accept more number of packets in a given span of time, resulting in increase in the data rate that NIC can handle.

Most of the popular architectures for a NIC like traditional architecture and protocol processor architecture [2] support only uni-processor system, that is, the packet buffer in a NIC has a single output port. Hence, only one packet can be taken out of the packet buffer at any instant of time. Also, different applications may have different priorities. It becomes essential to further minimize the packet losses for priority applications. Of all the popular architectures for NIC (including traditional architecture and protocol processor architecture), none of them has special consideration for priority application packets.

1.1 Problem Statement and Motivation

Until now, research has primarily concentrated on making fast switches and routers, and less attention has been given to network terminals [3, 4]. It is essential to have an efficient buffer management algorithm that can reduce packet losses [16]. The buffer management algorithm should take application state (application is active or not) into consideration while allocating buffer space to different applications.

There have two types of packet buffer management algorithms appeared in the literature: Static Algorithms; Dynamic Algorithms [16].

In static algorithms, the limitation on the queue length always remains the same, while in dynamic algorithms; it can change according to the occupied buffer space. It has been shown that dynamic algorithms are more robust than static algorithms for uniform loads [14]. Therefore, in this thesis, we implemented two dynamic algorithms, DA (Dynamic Algorithm) and DADT (Dynamic Algorithm with Dynamic Threshold).

Two popular static algorithms are Completely Partitioned Algorithm (CP) and Completely Shared Algorithm (CS).

CP allows equal distribution of a buffer space on all queues in a packet buffer [23]. It is easy to implement CP in hardware [8], however, its adaptability to changing traffic is not good since CP can lead to an incomplete use of the buffer space; if a queue is active and space allocated to it is full, then it will not accept incoming packets even though there is unoccupied space in the packet buffer.

CS allows output queues to completely share all available space in a buffer. Implementation of CS in hardware is also easy [8]. However, each queue can occupy the whole buffer space since the limitation on the queue is the buffer space.

In DA, a threshold value is computed and used to determine the acceptance of incoming packets and is directly proportional to the unoccupied buffer space [8, 16]. DA proves to be effective in adapting to the changing traffic conditions and is also easy to be implemented in hardware.

DADT is similar to DA, but it has different threshold values for different queues [24]. If the length of a queue is smaller than its corresponding threshold value, an incoming packet is accepted. Otherwise it is dropped [24].

None of the above algorithms take application state into consideration while allocating buffer space to different applications. For example, suppose we have two applications say application 1 and application 2. Suppose application 1 is active from time ‘t’ to time ‘t+t'’. During this time packets for application 2 are coming at much slower rate than application1. Hence, if the state of the application is monitored then more buffer space can be allocated to application 1 from time ‘t’ to time ‘t+t'’.

This leads us to propose an efficient buffer management algorithm for Network Interface Card. All the popular architectures for an NIC support uni-processor systems [2]. Therefore, only one packet can be processed at any instant of time. As mentioned earlier, processing the packet to determine whether to accept it or reject it is the slowest process, hence this provides a bottleneck for further increase in input data rate in network terminals [5]. Also, data is transmitted at a very high rate across the network but the

speed of the processors on the computer limits the data speed. To overcome this, we proposed a new architecture named multiprocessor architecture in which a packet buffer on a NIC supports multiple processors.

This thesis proposes an efficient architecture which gives special consideration to priority packets and is capable of taking out multiple packets at any given instant of time from the packet buffer.

The main purpose of this research is to address the following issues:

1) Develop and simulate buffer management algorithm that can reduce the overall packet losses in network terminals, can take application state into consideration while allocating buffer space to any application.
2) Propose a new architecture for a NIC to reduce number of interrupts required to be sent to the host processor. The proposed architecture also considers priority application and multiple ports for a packet buffer.
3) Write a tutorial for power analysis of any circuit using Xilinx Xpower tool.

1.2 Summary of Main Contributions

The main contributions of this thesis work are as follows:

1) Proposal of a new buffer management algorithm called History Based Dynamic Algorithm (HBDA) for protocol processors in a NIC. HBDA takes applications state into consideration while allocation buffer space to different applications.
2) Development of a simulation model for the packet buffer in a protocol processor and performance comparison of the different algorithms.

3) Propose a new architecture for NIC that can support multiple processors and can minimize the packet losses for priority application.
4) Compute power consumption for comparing the traditional architecture with the proposed architecture.
5) Develop a tutorial which explains step by step procedure to do power analysis of any circuit using Xilinx Xpower tool.

1.3 Organization

The remainder of the thesis is organized as follows. Chapter II gives the background on existing architectures and popular buffer management algorithms. Chapter III introduces new algorithm History Based Dynamic Algorithm in detail. Chapter IV describes the new architecture (Multiprocessor Architecture) for a NIC in detail. Chapter V explains the simulation model used to compare the performances of different algorithms and different architectures. After that, Chapter VI shows and analyses the simulation results. Finally, Chapter VII concludes the thesis and discusses future work.

CHAPTER II

BACKGROUND

2.1 Traditional Architecture for Packet Reception

Packets coming in from the network are received on the NIC. The physical layer and the MAC layer present on the NIC process the packet for layer 1 and layer 2 protocols [1-4]. It is then buffered in the packet buffer before being sent to the main memory.

Once the packet is in the main memory, the host processor processes the TCP/IP or the UDP headers (layers 3-4 protocols) [1, 2]. Figure 2.1 gives the block diagram of packet reception in a network terminal [2]. Once the headers (layer 3-4 protocols) are processed by the host processor/OS, the necessary data (payload) is delivered to the corresponding applications [2, 6, and 7]. Since the amount of processing power spent by the host processor is around 20% - 60% when it is connected to a gigabit Ethernet [1, 5-7], Henrikkson et al [2] proposed the protocol processor to offload the host processor.

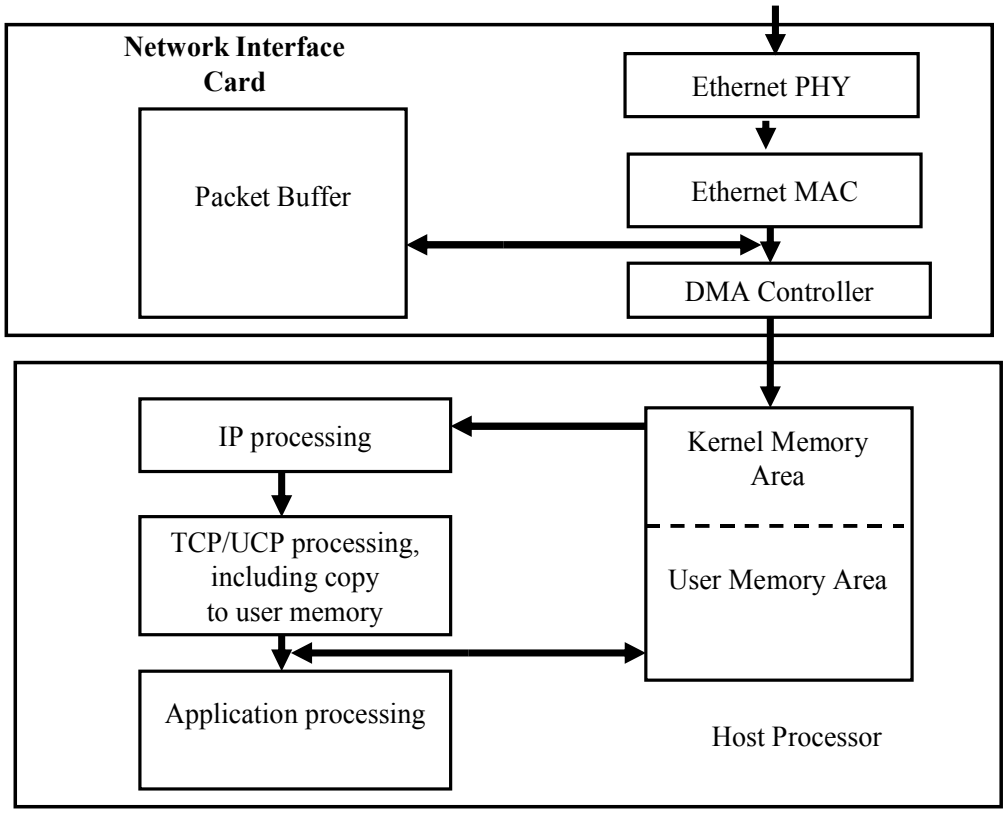


Figure 2.1: Traditional packet reception on the NIC and Host Processor [2]

2.2 Protocol Processor Architecture

Figure 2.2 shows the packet reception using a protocol processor. The new packet reception shown in figure 2.2 moves layer 3 and layer 4 processing onto the NIC [1, 5-7]. Packets coming in from the network are received on the NIC and are processed for layer 1-2 protocols. Instead of sending the packet over to the host processor for further processing, the protocol processor on the NIC handles the processing of layer 3 and layer

4 protocols. The main task of a protocol processor is to handle protocol processing at a wire speed [1, 7].

As shown in figure 2.2, packets coming in will stream through the protocol processor and the payload (application) data will be stored in the packet buffer until the host application retrieves it [7]. Packets are classified based on the application is they are destined for (per-flow). Once the packet is classified, it is stored in an output queue in the buffer. Each application has an output queue in the buffer. In general, the packet buffer has FIFO-based output queues for each application to store its application data [7].

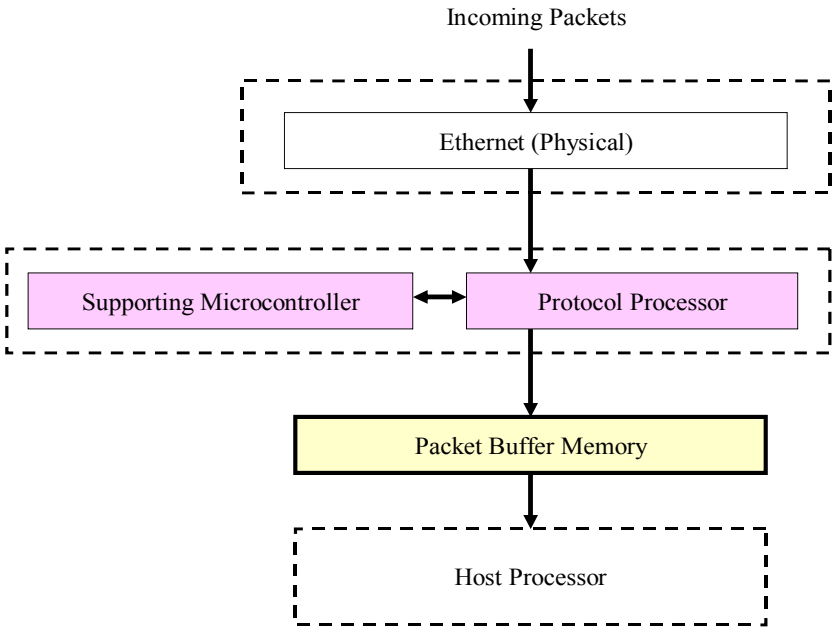


Figure 2.2: New architecture for packet reception using Protocol Processor [2].

2.3 Buffer Management Algorithms

Most of the network interface card architectures that have been proposed in the literature, use some buffering to accommodate packets whose service has been delayed due to contention. The buffer management algorithm directly affects the performance of a NIC. However, output-queued shared-memory packet switches with no buffer management procedures may not perform well under overload conditions [4]. The problem is that a single output port can take over most of the memory, preventing packets destined for less utilized ports from gaining access. This causes the total switch throughput to drop. One solution to this problem is to place restrictions on the amount of buffering a port can use. This makes buffers available to the less utilized ports and increases the total switch throughput.

Two styles of buffer sharing restrictions appear in the literature. One type places limits on the maximum or minimum amount of buffering that should be available to any individual queue. This is called the *Static Threshold* (ST) scheme. In this method, an arriving cell is admitted only if the queue length at its destination output port is smaller than a given threshold value. The ST strategy is very simple to implement in hardware. It requires only queue length counters, which are likely to be needed for network management purposes anyway, and a comparator. When so many queues are active at once such that the sum of their threshold values exceeds the buffer capacity, it is possible for the buffer to fill up completely even though all queues are obeying their threshold constraints. This allows some queues to become starved for space, which can lead to underutilization of the switch. At other times, when very few output queues are active;

these queues are needlessly denied access to the idle buffer space beyond the sum of their thresholds. This creates higher cell-loss rates and lower throughputs for these active queues than they would experience if they had access to extra buffer space.

The other style is called the *Dynamic Threshold* (DT) Scheme. In DT, threshold value for any application at an instant 't' is a function of unused buffer space. In DT, packets for any application are accepted as long as queue length for the application is less than the threshold value for that application.

2.4 Popular Buffer Management Algorithms

Buffer management algorithms determine how the packet buffer is shared among the various output queues. Four popular buffer management algorithms are reported in literature [8, 10-11, 13, 15]. They are

1) Completely Partitioned Algorithm (CP).

2) Completely Shared Algorithm (CS).

3) Dynamic Algorithm.

4) Dynamic Algorithm with Dynamic Threshold (DADT)

CP and CS come under ST scheme while the DA and DADT come under DT scheme.

2.5 Completely Partitioned Algorithm (CP)

Kamoun and Kleinrock [11] proposed Completely Partitioned algorithm. In this algorithm, the total buffer space 'M' is equally divided among all the applications. Packet loss for any application occurs when the buffer space allocated to it becomes full. If 'M'

is the total buffer space, 'n' is number of applications and $k_{i,,}$ i= 1….n, represents the size of queues i=1….n then:

$$k_1 + k_2 + \ldots. + k_n = M \quad (2.1)$$

$$\sum_{i=1}^{N} k_i = M \quad (2.2)$$

The advantage of this algorithm is that it works well when all the applications are active [6], that is, packets for all the applications are coming. In addition, CP is easy to implement in hardware. However, the algorithm has a disadvantage in that if one of the applications is not active, then the space allocated to it will be never utilized. Hence, CP is not adaptive to changes in traffic conditions.

2.5.1 *Example of Completely Partitioned Algorithm*

If the total buffer space is 400 packets, and if the number of output queues is 4, then each queue has 100 packets for it. When the queue length of any given queue exceeds 100 packets, it stops accepting further incoming packets. Now if packets are not coming for one of the queues then the space allocated to it is not utilized. This means that space for 100 packets is wasted in CP.

2.6 Completely Shared Algorithm (CS)

Unlike the Completely Partitioned algorithm, the individual queues do not have any static thresholds placed on them in CS. The incoming packet is accepted as long as there is space in the memory to accommodate it. Packet loss for an application occurs

only when there is no space in the buffer. If 'M' is the total buffer space, 'n' is the number of applications and $k_{i,,}$ i= 1….n, represents the size of queues i=1….n then:

$$k_i = M,\ i = 1, 2, \ldots, N \tag{2.3}$$

CS works well under balanced load conditions. In balanced load conditions, the incoming packets are almost equally distributed among all the applications; hence, this algorithm can provide fairness to all the applications under balanced load conditions. In addition, CS is easy to implement in hardware.

The drawback of this algorithm is that if only one application is active at any instant 't', it can fill the whole buffer space. Once the buffer is filled with this active application, the incoming packets for other applications are rejected. Hence, it does not guarantee the fairness to all the applications.

2.6.1 *Example of Completely Shared Algorithm*

If the total buffer space is 400 packets and there are 4 output queues, then any one queue can occupy the entire buffer space, leaving other output queues with no buffer space at all. Any of the given queues can occupy as much buffer space as possible. The only condition is that the cumulative sum of all the queues should not exceed the total buffer space [8]. Therefore, if initially only application1 is active then it may occupy whole space. Hence, by the time packets for other application arrive; whole buffer space might be occupied. Also, if one application has higher packet size than another then it gets more space for same number of packets.

2.7 Dynamic Algorithm (DA)

DA is more adaptive to changes in traffic conditions than CS and CP. In DA, threshold value for any application at an instant 't' is a function of unused buffer space. In DA, packets for any application are accepted as long as queue length for the application is less than the threshold value. Packet loss occurs only when queue length of an application exceeds its threshold value. If at any time 't', let T(t) be the controlling threshold and let $Q_i(t)$ be the length of queue 'i'. Let Q (t) be the sum of all the queue lengths. Then if 'M' is the total buffer space

$$T(t)=\alpha \times (M-Q(t)) \tag{2.4}$$

Where 'α' is some constant, which is taken as a power of two, so that shift registers can be used to implement in hardware. This algorithm is robust to changes in traffic conditions. In addition, it is easy to implement in the hardware.

In ATM switches, packet size is the same for all the applications. Hence, DA works efficiently in ATM switches. However, in network terminals, different applications may have different packet sizes, thus reducing the efficiency of DA.

2.8 Dynamic Algorithm with Dynamic Threshold (DADT)

The DADT [16] works similar to DA. In DADT, packet sizes are also considered while calculating the threshold value for different applications. In DADT the threshold value is calculated as shown in equation 2.5.

$$T(t)= \alpha_i \times (M-Q(t)) \tag{2.5}$$

Where 'α_i' is the proportionality constant and varies for each queue. Optimum 'α' value for each queue is calculated through simulations. By varying the threshold value, DADT does not allow queues with large packet sizes to fill the buffer at a faster rate.

It has been shown that dynamic threshold scheme (DT) is more efficient than static threshold scheme (ST) [16]. Among the dynamic algorithms, DADT achieves the lowest packet loss ratio in network terminals. However, it has a disadvantage that it is difficult to determine the optimum alpha value for each application [16]. In addition, simulation results of DADT have shown that the optimum alpha value for each application comes out to be different from the power of 2, which makes its hardware implementation difficult [16].

As mentioned above, in DA and DADT, threshold value for any application at an instant 't' is a function of unused buffer space. DA and DADT do not take application state, that is whether the application is active or not, into consideration while determining the threshold value for an application. Therefore, we propose a novel scheme called History Based Dynamic Algorithm (HBDA) which will take all the factors into consideration while calculating the threshold value for each application:

1) Unused buffer space.

2) Packet size.

3) Application state.

CHAPTER III

HISTORY BASED DYNAMIC ALGORITHM

The existing Dynamic algorithms do not consider application state (application is active or not) while calculating threshold value for any application [16]. The only consideration while calculating the threshold value is the amount of unused buffer space in the current existing algorithms. So, we proposed a History-Based Dynamic Algorithm (HBDA), which takes all the three factors into consideration: the amount of unused buffer space, packet size and the application state. If at any time 't', let T (t) and T'(t) be the controlling thresholds and Q_i (t) be the length of queue 'i'. If 'M' is the total buffer space, the algorithm works as shown in figure 3.1.

The idea for the HBDA is to optimize the use of buffer space by taking application state into consideration while calculating threshold value for the application.

3.1 Example for HBDA

The following example explains the working of HBDA in more detail. Let us consider three applications for our example. Say at any instant 't', application 1 has filled its allocated buffer space such that the queue length of application 1 is greater than the threshold value for that application. Application 2 at this instant is inactive such that queue length of application 2 is less than the threshold value for that application. Application 3 at this instant 't' has filled almost half of its allocated buffer space such

that queue length of application 3 is almost half of the threshold value for that application.

Now if the incoming packet is for application 3, it will be accepted since queue length for application 3 is less than the threshold value for application 3. On the other hand, if an incoming packet is for application 1, it will be rejected since queue length for application 1 is greater that its threshold value, though there is still free space available in the buffer.

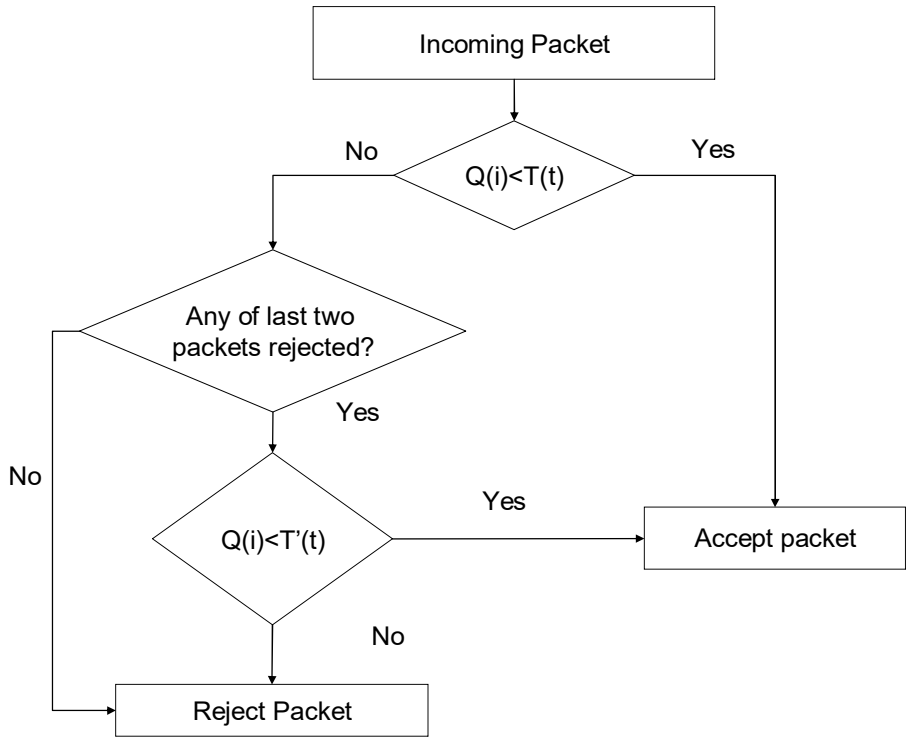


Figure 3.1: Algorithm for the History-Based Dynamic algorithm (HBDA).

Where

$$T(t) = (\alpha / psize_i) \times (M\text{-}Q(t)) \quad (3.1)$$

$$T'(t) = (\alpha / psize_i) \times (M\text{-}Q(t)) + History_i(1) \times M/2 + History_i(2) \times M/4 \quad (3.2)$$

History(1) is '1' if the last packet of 'i'th application is rejected and '0' if accepted;

History(2) is '1' if the second last packet of 'i'th application is rejected and '0' if Accepted;

and 'M' is the total buffer space.

Our simulation studies have shown that when an application fills the buffer space allocated to it, then the probability of being rejected for further few incoming packets for that application is high. In other words, by the time the packets for that application dequeue [6] themselves and increase the threshold value for that application above the queue length for that application, some packets have already been rejected for that application. Therefore, to minimize this packet loss, we keep track of last two packets for each application. If any of the last two packets has been rejected for an application which has queue length greater than the threshold value (calculated by equation 3.1), then the threshold value for that application is determined by equation 3.2. By increasing the threshold value for such an application, packet loss for that application can be minimized.

Though, by tracking the last three packets for an application, we can further increase the efficiency of buffer management algorithm, but increase in efficiency is not as significant as compared to increase in the hardware cost. Our simulation results have shown that increase in efficiency is only about 0.01% as compared to when we keep track of last two packets for an application.

3.2 Threshold Value Computation in HBDA

In DADT, the threshold value for an application is calculated as shown in equation 3.3.

$$T_i(t)=\alpha_i \times (M-Q(t)) \quad (3.3)$$

Different applications have different 'α' values in DADT. In general, the optimum alpha value comes out to be different than the power of two in DADT [16]. Also, in DADT, determining the optimum alpha value for each application is difficult. Therefore, equation for calculating the threshold value for an application has been modified as shown in equation 3.4.

$$T(t) = (\alpha / psize_i) \times (M-Q(t)) \quad (3.4)$$

In equation 3.4, 'α' value is same for all the applications and since different applications will have different packet size, factor of '$\alpha / psize_i$' in equation 3.4, achieves the same effect as different alpha values for different applications, in DADT. This eliminates the need to determine the optimum 'α' value for each application.

As mentioned above, if an application has a queue length greater than the threshold value, then the threshold value for such an application is determined using equation 3.5.

$$T'(t) = (\alpha / psize_i) \times M-Q(t)) + History_i(1) \times M/a + History_i(2) \times M/b \quad (3.5)$$

where 'psize' represents the packet size of the application with 'i' varying from 1 to n, 'a' and 'b' are constants which are determined through simulations . The 'α' value is generally taken as a power of two (either positive or negative), so that threshold computation is easy to implement in hardware.

Our simulation results as shown in table 3.1 shows that optimum value of ‘a’ and ‘b’ comes out to be 2 and 4 respectively. For our simulations, we have used six applications, bursty uniform traffic model, alpha value as 128 (from table 6.3), average traffic mix, average dequeue time of 14 clock cycles for the burst of 10 packets, buffer size as 600 packets and load of 70% on each of the queue. For simulations, the values of ‘a’ and ‘b’ are taken as power of 2 so that shift registers can be used to implement it in hardware.

Table 3.1

Optimum value of factors, ‘a’ and ,‘b’

Variation of (a, b)	**Packets Rejected / Total number of incoming**
2,2	0.093
2,4	0.081
2,8	0.085
4,4	0.088
4,8	0.090

3.3 Advantages of HBDA

The HBDA, DA and DADT have one major advantage over static threshold schemes; they are adaptive to changes in traffic conditions [14]. The HBDA has one distinct advantage over DA and DADT that it continuously monitors the state of the application and controls the threshold value dynamically. Another advantage of HBDA

over DADT is that, there is no need to determine the optimum ‘α’ value for each application.

CHAPTER IV

PROPOSED ARCHITECTURE

4.1 Need for New Architecture

Processing the packet on the Network Interface card (NIC) is the slowest process [5]. Whenever a packet is accepted and placed in the packet buffer, an interrupt is sent to the host processor by the NIC. It takes approximately 50 µs to process a single interrupt. If one interrupt per packet is sent, the result is one interrupt every 12µs. Hence, this results in slowing down the overall packet processing time in a NIC.

With multi-processor systems becoming so popular for high-speed networks, it becomes essential to design a new architecture for an NIC, which can support multi-processor systems. In general, the packet buffer in an NIC has a single output port, thus only one processor can communicate with the NIC at any given time. Hence, only one packet can be taken out (dequeue) of the packet buffer at any instant of time. To support multi-processor systems, the packet buffer must have multiple output ports.

Therefore, large number of interrupts and large dequeue time of packets provide a bottleneck for further increase of the capacity in the networks. Thus, processing multiple packets in parallel, supporting multi-processors systems and reducing the number of interrupts become essential to overcome the bottleneck. So, we propose a

new architecture which can process more than one packet at the same time and also reduce the number of interrupts sent to the host processors in multi-processors systems.

4.1.1 Priority applications

Different applications in a network may have different priorities. An application with the highest priority should have the minimum packet loss to avoid loosing important data. One way to reduce the packet losses of the priority application could be to use Push-Out algorithm as a buffer management algorithm [14]. In Push-Out algorithm, incoming packets are accepted as long as there is space in the buffer. Once the buffer is full, further incoming packets are allowed to enter by selectively pushing out another packet that is already in the queue. So, a high priority application packet can push out a low priority application packet. Push-Out algorithm pushes out a packet at the head of the longest queue.

This algorithm is highly adaptive to changing traffic conditions [14], as there is competition among the queues to keep their queue lengths short when all the queues are competing for buffer space. This algorithm is difficult to implement in hardware [16] as it involves keeping track of the longest queue, pushing out a cell from the longest queue and then finally writing a cell into a new queue. Also, the source will get no information of those packets that are pushed out. Hence, source will assume that the packets have been accepted by the receiver. Thus, better way of reducing the priority application packets could be to design a new architecture that is capable of reducing packet loss of high priority applications.

4.2 Proposed Architecture for a NIC

Figure 4.1 shows the diagram of the proposed architecture for a NIC. As seen in figure 4.1, an incoming packet in stored in an input buffer. The packet is then processed and the control unit uses a buffer management algorithm to determine whether to accept the packet or reject the packet. The buffer management algorithm compares the queue length of an application for which the packet is destined, with the threshold value of that application. The detailed working of the control unit is explained in section 4.4. The packet is rejected if the queue length of an application is greater than the threshold value for that application, otherwise the packet is accepted. The accepted packets are then, placed in the packet buffer. Each application has its own queue in the packet buffer. Different applications may have different priorities. Therefore, it becomes important to minimize the packet loss for the highest priority application. To achieve this, a small priority-based buffer (Section 4.3) has been placed in front of the packet buffer. If the incoming packet is for the highest priority application and the controller rejects that packet, as there is no space in the packet buffer for that application, then this high priority application packet is placed in the priority-based buffer. So, instead of rejecting this highest priority application packet, the packet is stored in the priority-based buffer and is injected into the packet buffer when there is enough space for this packet in application queue in the packet buffer. The priority controller is responsible for determining when the packet can be moved from the priority-based buffer to the packet buffer. To achieve this, the priority controller sets the signal ‘WritetoBuffer’ high.

As seen from figure 4.1, the packet buffer has multiple output ports. Thus, the processor1 and processor2 can take the packets out of packet buffer at the same time [11], which means more than one packet can be dequeued at one time. This leads to increase in the buffer space and hence more packets will be accepted.

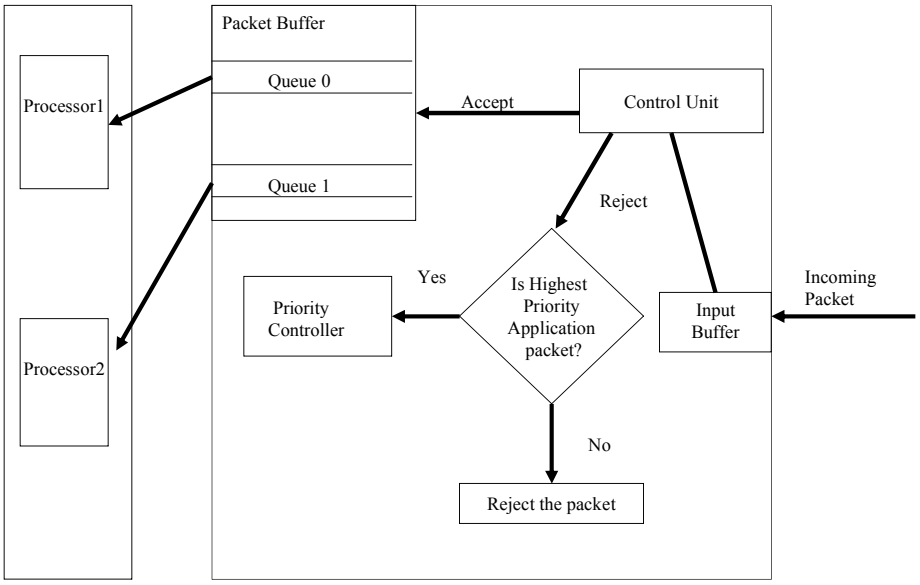


Figure 4.1: Proposed Architecture for a NIC.

Figure 4.2 shows the flowchart describing functionality of the proposed architecture. As seen from figure 4.2, the packets for the application with highest priority are rejected only if there is no space for that packet in the packet buffer as well as in the priority-based buffer. This way, packet loss for the highest priority application can be reduced.

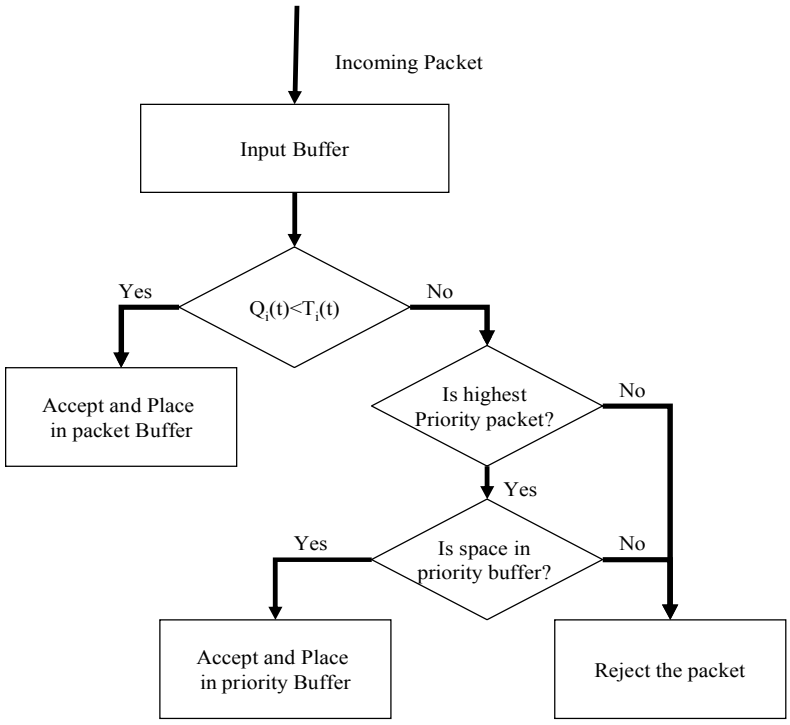


Figure 4.2: Flowchart for Multiprocessor Architecture.

4.3 Working of the Priority Controller

Figure 4.3 explains the working of the priority controller in the proposed architecture. As seen from figure 4.3, if the packet for the highest priority application is rejected by the controller, that is, the queue length for that application is greater than the threshold value of that application; the control is passed from the controller to the priority controller. The priority controller checks for the available space in the priority-based buffer. In case, there is space in the priority-based buffer, then the packet is accepted and placed in the priority-based buffer, otherwise, the packet is rejected.

The priority controller continuously monitors the packet buffer for the available space for the highest priority application. In case, there is sufficient space in the packet buffer for the highest priority application packet, then the 'WritetoBuffer' signal is set high and the packet is moved from the priority-based buffer to the packet buffer. The priority controller also sends a signal to controller to indicate the number of packets moved from the priority-based buffer to the packet buffer, so that controller can update its counters accordingly. The controller responds with 'DONE' bit to indicate that it has updated the counters and priority controller can now proceed with further moving of packets from the priority-based buffer to the packet buffer.

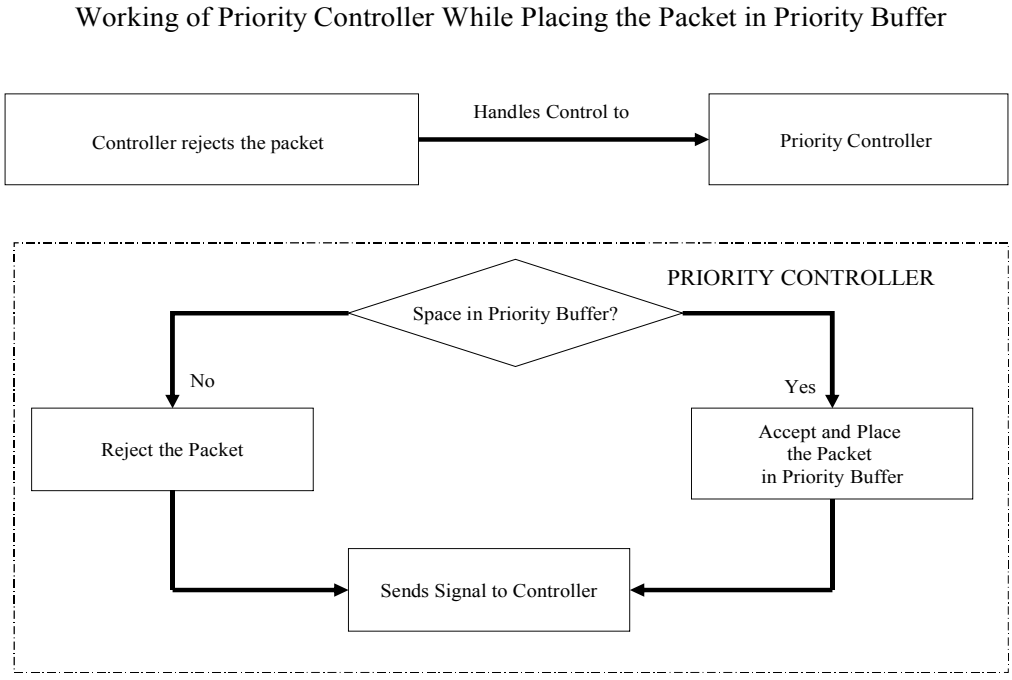


Figure 4.3: Working of Priority Controller.

Researchers have shown that interrupts are very costly, and generating an interrupt for each packet arrival can severely throttle a system [12]. If one interrupt per packet is received, the result is one interrupt every 12µs. It takes approximately 50 µs to process a single interrupt [13]. In the proposed architecture, multiple packets can be dequeued at the same time, thus, only one interrupt has to be send to the CPU for dequeue of multiple packets. Thus, the number of interrupts in the proposed architecture would be drastically reduced as the host processor would no longer need to be informed of the arrival of individual packets.

4.4 Working of the Control Unit

Figure 4.4 shows the inner details of the control unit. Incoming packets are placed in the input buffer. In figure 4.4, we have assumed that the first two packets in the input buffer are ‘p1’ and ‘p2’ respectively. ‘Control unit 1’ starts processing of the first packet in the input buffer which is packet ‘p1’ in our case.

Now, if we have only one control unit say ‘control unit 1’, then processing of packet ‘p2’ can start only after the processing of packet ‘p1’ is finished. Generally, processing the packet to determine whether to accept or reject is the slowest process [10]. To overcome this limitation, we have placed another control unit ‘control unit 2’ in parallel with ‘control unit 1.’ ‘Control unit 2’ takes the advantage of fact that more than 90% of the incoming packets are accepted by the NIC. After ‘control unit 1’ starts processing of packet ‘p1’, ‘control unit 2’ updates its variables like “Queue Length”, “Threshold Values” assuming that the packet ‘p1’ will be accepted by the NIC. Then,

'control unit 2' starts processing of packet 'p2'. This way, by processing multiple packets at the same time, the overall processing time of packets will be reduced significantly.

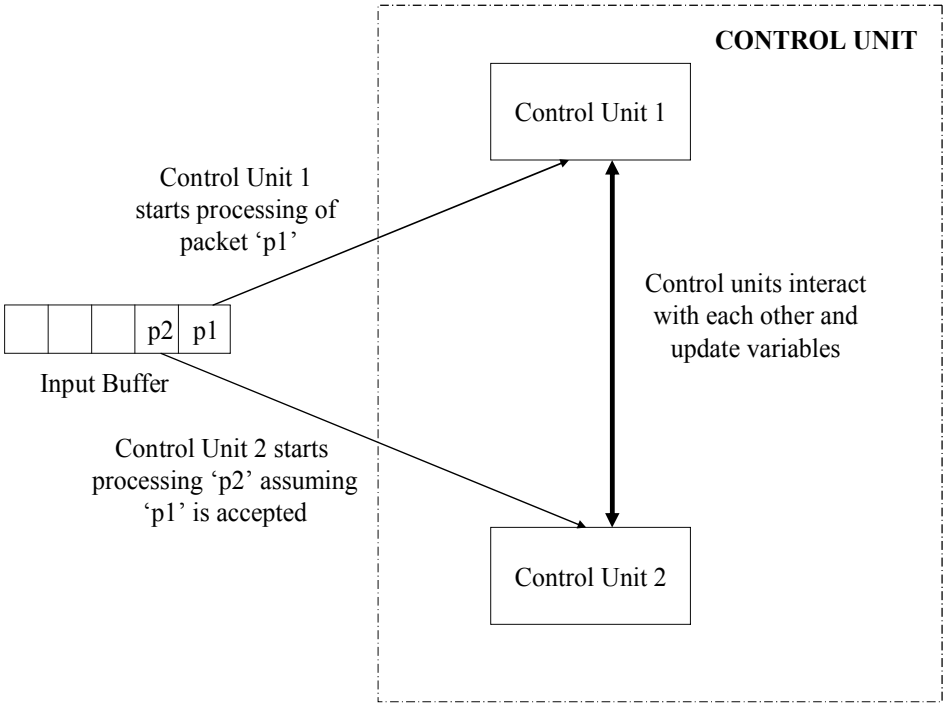


Figure 4.4: Working of Control Unit.

Figure 4.5 shows the flowchart describing working of the control unit. As discussed earlier, 'control unit 1' starts processing packet 'p1' and 'control unit 2' will start processing next packet 'p2', assuming that the packet 'p1' will be accepted. Now, there are can be two cases depending upon whether the packet 'p1' is accepted or rejected.

Case 1: If the packet 'p1' is accepted by 'control unit 1' and packet 'p2' is accepted by 'control unit 2', then 'control unit 2' updates the variables (Queue lengths, Number of

Packets accepted etc) of ‘control unit 1’ with the updated values. The ‘control unit 1’, then, starts the processing of next packet in the input buffer.

Case 2: In case, the packet ‘p1’ is rejected by ‘control unit 1’, then, ‘control unit 1’ updates the variables of ‘control unit 2’ with its values. This has to be done as ‘control unit 2’ updated its variables assuming that packet ‘p1’ will be accepted by the ‘control unit 1’. Then, ‘control unit 1’ starts processing of packet ‘p2’ again as we have to flush all the processing done by the ‘control unit 2’. Then, the ‘control unit 2’ starts the processing of next packet from the input buffer.

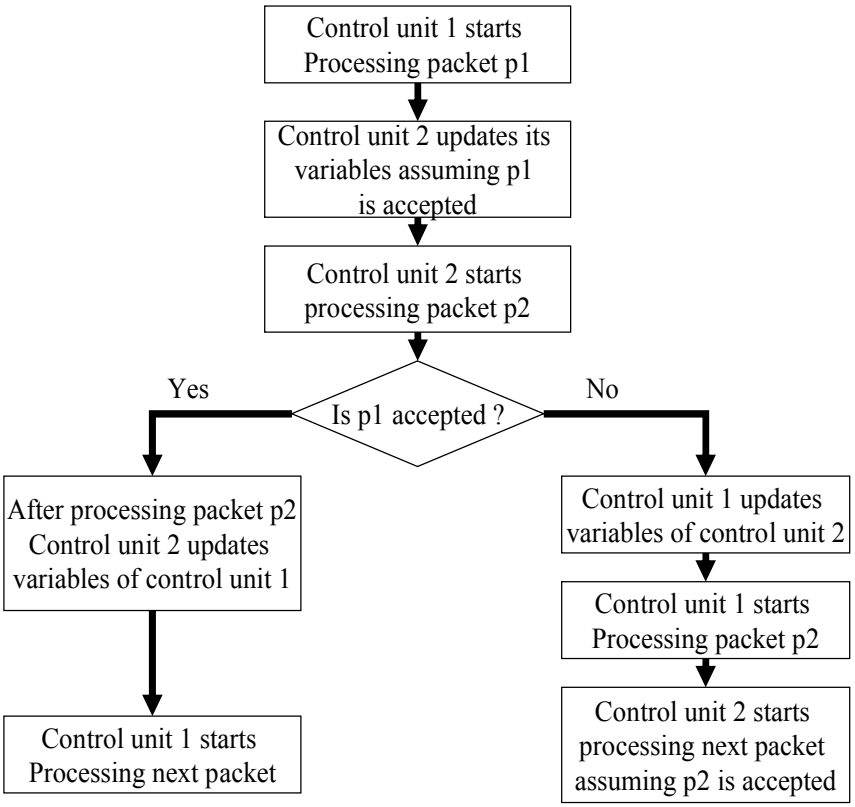


Figure 4.5: Flowchart explaining the working of Control Unit.

This way, by processing packets in parallel, the slowest path can be made to work faster, thus increasing the number of packets processed per unit time.

CHAPTER V

SIMULATION ENVIRONMENT

The entire simulation model is developed using a Hardware Description Language (HDL) simulator in MODELSIM [2]. VHDL, a Hardware Description Language was chosen to code the entire simulator.

5.1 Simulation model for the packet buffer

Figure 5.1 shows the diagram of Simulation model for the packet buffer in a NIC.

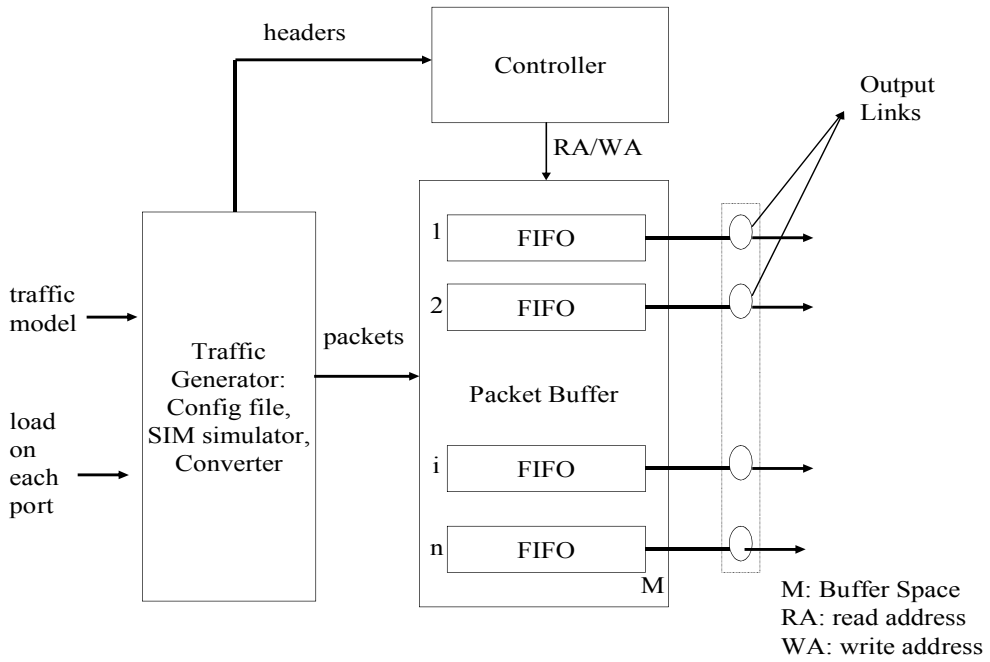


Figure5.1: Simulation model for the packet buffer.

The input parameters to the Simulation model are:

1) Traffic parameters: Traffic parameters define the incoming packet size of the traffic, nature of the traffic that is coming in (ex: bursty uniform), the inter-arrival time between each packet and the load on the input ports.
2) Memory (Buffer size): Buffer size defines the size of memory in terms of number of packets that can be placed in the memory [8].
3) Service time or Dequeue time: It is the amount of time spent by each packet in the memory.

The individual blocks works as follows:

5.1.1 *Traffic Generator*

A fixed length packet simulator called 'SIM' [17]; developed by Sundar Iyer et al generates the trace of serial packets. The packets produced by the 'SIM' simulator are destined randomly between all the output queues. The packets are produced with a specified mean inter-arrival time and mean burst length.

The traffic generator reads from the configuration file. This file contains packets whose output-destination requests are randomly distributed on all of the output queues. This file acts as an input to the 'SIM' simulator [17] and specifies the number of input and output destination ports, the traffic model to be used and the load for each of the ports.

There are three kinds of traffic model that are available [16]. These are:

- Bursty Uniform Traffic Model: Burst of packets in busy-idle periods with destinations uniformly distributed packet-by-packet or burst-by-burst over all the output ports. The number of packets in the busy and idle periods can be specified; and
- Bursty Non-Uniform Traffic Model: Burst of packets in busy-idle periods with destinations non-uniformly distributed packet-by-packet or burst-by-burst over all the output ports; and
- Bernoulli Uniform Traffic Model: Incoming packets are in the form of Bernoulli arrivals and distributions on all output ports

The "load on each port (ρ)" is determined by the ratio of the number of packets in the busy-idle periods [14] and is given by the equation:

$$\rho = \frac{L_b}{L_b + L_{idle}} \quad (5.1)$$

where L_b = *mean burst length* and L_{idle}= *mean idle length.*

A part of the configuration file is shown below in figure 5.2. This configuration file shows that there are 6 input and 6 output ports. It also mentions the traffic model connected to each of the input ports and the corresponding load on each of the port. The various mean burst lengths of each of the input port are also specified.

```
(port)  0   bursty -u 0.70 –b 240   #bursty traffic-load 70%, burst length=240
(port)  1   bursty -u 0.80 –b 240   #bursty traffic-load 80%, burst length=240
(port)  2   bursty -u 0.70 –b 120   #bursty traffic-load 70%, burst length=120
(port)  3   bursty -u 0.60 –b 120   #bursty traffic-load 60%, burst length=120
(port)  4   bursty -u 0.90 –b 120   #bursty traffic-load 90%, burst length=120
(port)  5   bursty -u 0.75 –b 160   #bursty traffic-load 75%, burst length=160
```

Figure 5.2: Part of the configuration file

5.1.2 *Controller*

This is a key part of the simulator written in VHDL, which decides acceptance and rejection of a packet, depending upon the type of the packet buffer management algorithm used. This component is responsible for generating all signals for proper functioning of the simulator. The VHDL simulator reads the packets from the file generated by the traffic generator. The controller then decides whether to accept the packet based on buffer management algorithm used. If the packet is accepted then the controller specifies the write address (WA) based on the output queue to which the packet is destined. Irrespective of whether the packet is accepted or dropped, the controller updates its state variables like “packetaccepted”, “tpacket”. Figure 5.3 shows the sample waveforms and state variables for the simulation model developed for NIC.

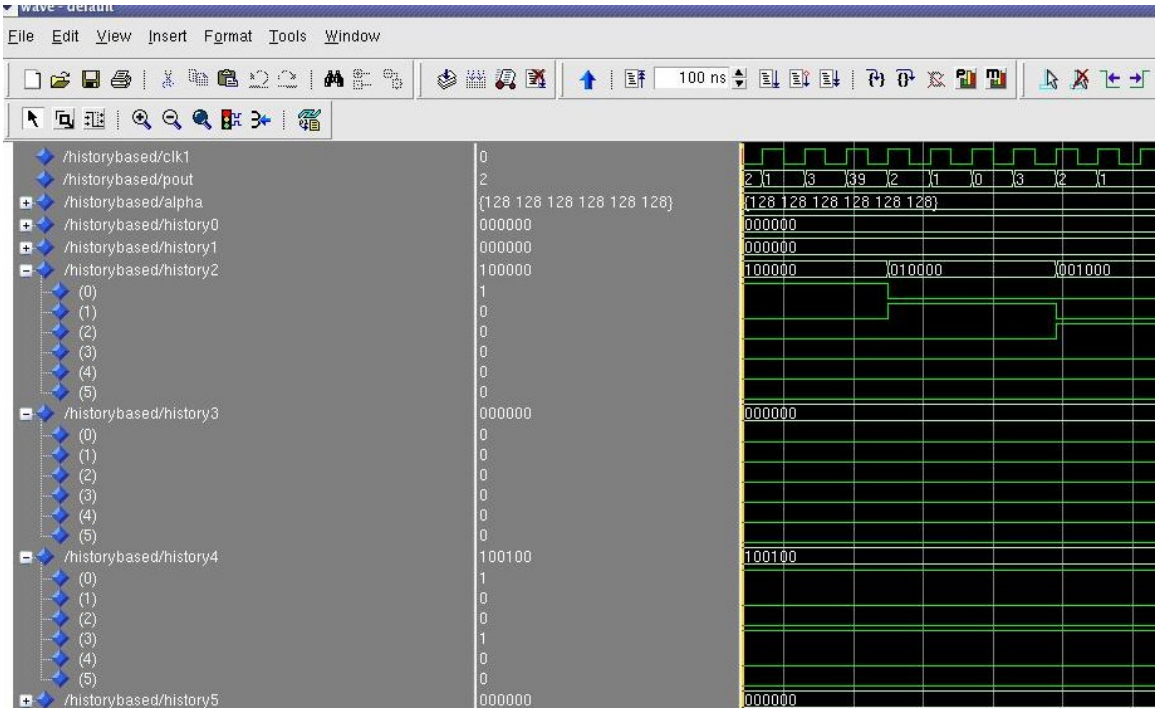


Figure 5.3: Sample Waveform for the simulation model developed for NIC.

As seen from the figure 5.3, 'psize' represents the size of the packets of different applications. 'Memsize' represent the size of the buffer memory in terms of packets. Variable 'pout' represents the destination of current incoming packet.

5.1.3 *Packet Buffer*

A packet buffer is a large shared dual-ported memory [6]. It has an arrangement in which a buffer space is distributed on output queues based on the total buffer size and the total number of queues. Packets for each application are multiplexed into a single stream. The accepted packet remains in a buffer until the application retrieves it from the buffer. The size of the memory is specified in terms

of the number of packets. The size can be varied and performances of different buffer management algorithms can be compared.

5.1.4 *Reading and writing from memory*

If the packet is accepted by the controller then controller enables a signal called ‘memwrite’ and specifies the write address (WA) based on the output queue for which the packet is destined. The packet is written into the memory at the negative edge of the clock. In addition, the queue length for that application is incremented. Once a packet is written to the memory, the controller signals the output link that a packet is received and is stored in a particular output queue. This initiates the “dequeue” process for the packet. Dequeue time has been taken as a Poisson random variable with a fixed mean.

5.1.5 *Converter*

The converter is a simple program written in C. The purpose of the converter is to converts the output of ‘SIM’ simulator to a format, which is compatible to the VHDL Simulator. Converter extracts the output destination for all the packets from the output of ‘SIM’ simulator. A sample output file from the converter is shown in figure 5.4.

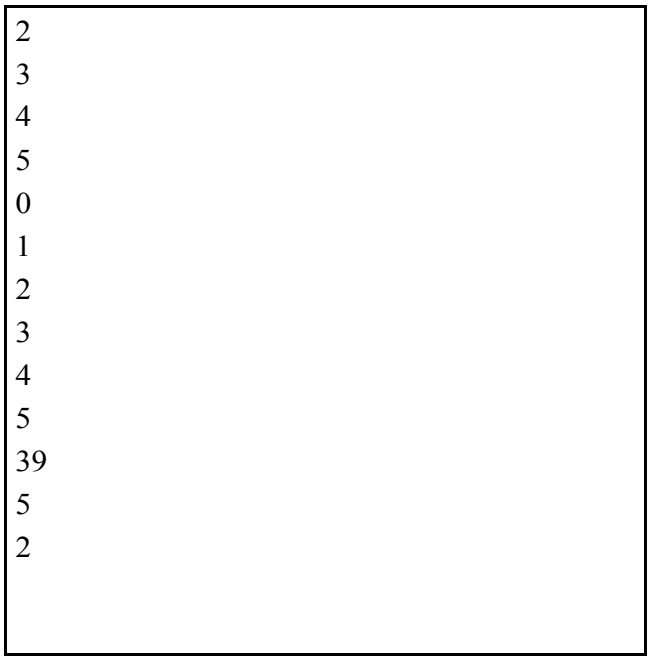


Figure 5.4: Input file after conversion

5.2 **Model for Power Analysis**

For power analysis of traditional and Multiprocessor Architecture architecture, we have used Xilinx Xpower tool. The Xilinx Xpower toll takes design file and the simulation file as an input and calculates the power consumption by that design. The design file is generated using Xilinx ISE and the simulation file is generated using ModelSim simulator. The detail of doing power analysis is explained in Appendix A.

CHAPTER VI

SIMULATION RESULTS AND ANALYSIS

The primary objective of this thesis is to reduce the packet loss ratio in network interface card. To prove the above hypothesis, simulations were done on three different traffic loads:

1) Average network traffic load

2) Heavy network traffic load

3) Actual network traffic load.

The simulations described in this chapter compare the performance of HBDA and DADT, DA for all the three traffic loads with varying loads and varying buffer size. It has been proved that dynamic threshold schemes are better than static threshold schemes. So, we have compared our proposed algorithm HBDA with DA and DADT. In the section 6.5, we have compared Multiprocessor Architecture results with the traditional architecture. We have also compared power consumption of proposed architecture with the traditional architecture. For all comparisons of proposed architecture with the traditional architecture, we have used HBDA as our buffer management algorithm.

6.1 Simulation results for HBDA

Three different network traffic loads are considered for our simulations as described above: average network traffic load, heavy network traffic load, and actual network traffic load. We have used Bursty Uniform Traffic Model for our simulations since this is the most commonly used model [16, 19]. For each traffic load, the following steps have been followed:

1) Optimum alpha value is determined for DA for different network traffic load.

2) Optimum combination of alpha values for different queues is determined for DADT. Optimum alpha values are the combination of alpha for different queues for which DADT gives minimum packet loss ratio.

3) Optimum alpha value is determined for HBDA for different network traffic load.

4) Packet loss ratio is plotted for DA, DADT and HBDA as the load is varied, keeping the buffer size constant.

5) Packet loss ratio is plotted for DA, DADT and HBDA as the buffer size is varied, keeping the load constant.

6) Improvement ratio is calculated for different values of load for the corresponding traffic mix. Improvement ratio is defined as the difference of packet loss in HBDA and the compared algorithm (DA or DADT) divided by packet loss in HBDA.

Section 6.2, 6.3 and 6.4 discusses our simulation results for average network traffic loads, heavy network traffic load and actual network traffic load respectively.

For all simulations, we have used the number of the applications as six, bursty uniform traffic model and average dequeue time of 14 clock cycles for the burst of 10 packets.

6.2 Simulation Results for average traffic load

We implemented a traffic mix with average network traffic load according to [5]. With this traffic mix, first, we determine the optimum 'α' (alpha) value for DA.

6.2.1 *Optimum alpha value for DA*

Table 6.1 shows the packet sizes of different applications in bytes based on the average network traffic load flow in [5]. For our simulation purpose, we have used these packet sizes for different applications.

Table 6.1

Queue properties for average traffic load

	Q0	Q1	Q2	Q3	Q4	Q5
Size in Bytes	256	64	256	32	128	512
packet unit # (32 bytes/unit)	8	2	8	1	4	16

Figure 6.1 shows the variation of packet loss ratio (number of dropped packets/ number of received packets) with alpha value varying from 4 to 20 for DA. In figure 6.1, size of buffer is 600 packets, and load on each queue is 70%. In figure 6.1, we can see that packet loss ratio decreases till α=14 and after that the packet loss ratio starts increasing because larger alpha values can increase the control threshold

of the queues with large packet sizes. This increase in control threshold for large packet size prevents them from being dropped even though they have taken significantly large space in the buffer. Therefore, we determine the optimum alpha value to be 14 for DA.

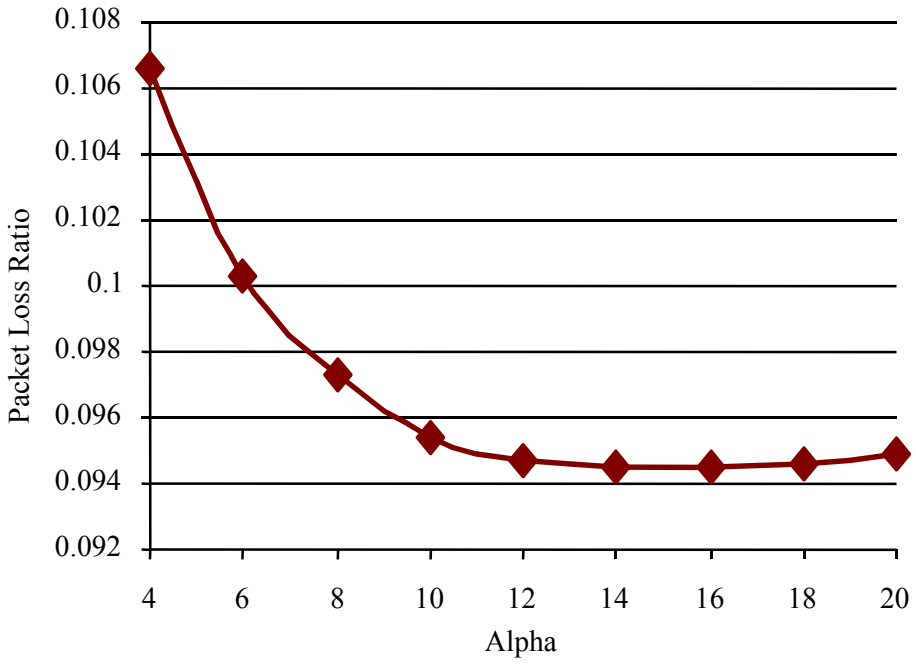


Figure 6.1: Packet loss ratio vs. Alpha for DA for the average traffic load.

6.2.2 *Optimum alpha value for DADT*

For DADT, each queue has different alpha value, hence, different threshold value. We will determine the optimum alpha values for different queues so that packet loss ratio is minimum for DADT. Table 6.2 shows the different combinations

of alpha that we have taken for our simulations and figure 6.2 shows the packet loss ratio corresponding to them.

Table 6.2

Variation of alpha for DADT for the average traffic load

Variation	Q0	Q1	Q2	Q3	Q4	Q5
1	12	10	12	10	10	8
2	14	10	14	10	10	7
3	14	12	14	12	12	8
4	16	14	16	14	14	6
5	16	14	16	14	16	8

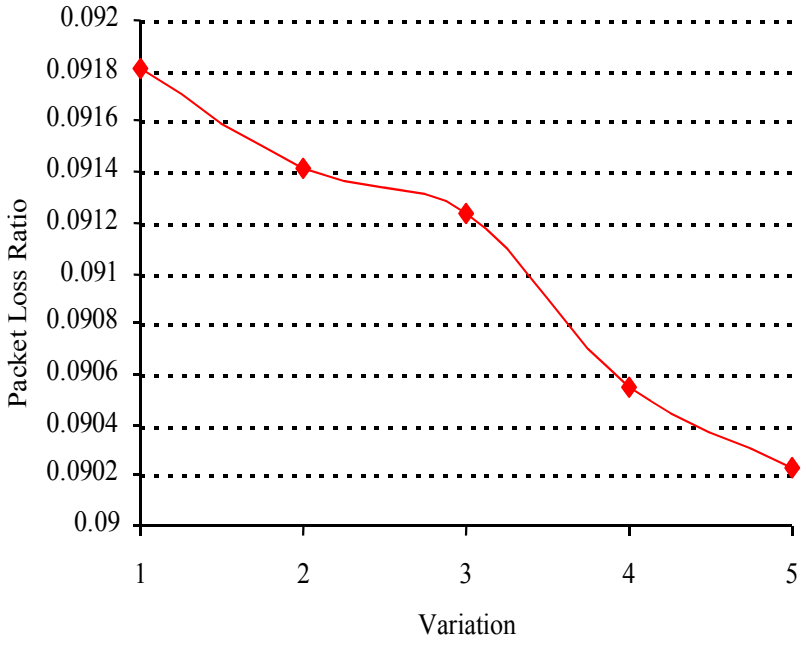


Figure 6.2**:** Packet loss ratio vs. Alpha for DADT for the average traffic load

6.2.3 *Optimum alpha value for HBDA*

Table 6.3 shows the packet loss ratio for HBDA as alpha value is varied between 16 and 256. As shown from table 6.3, optimum alpha value comes out to be 128. We will use alpha value as 128 for HBDA.

Table 6.3

Variation of alpha for HBDA for the average traffic load

Value of alpha	**Packet Loss Ratio**
16	0.095
32	0.093
64	0.087
128	0.081
256	0.083

6.2.4 *Comparison of HBDA, DA, DADT with varying load*

Figure 6.3 shows the performance of the three algorithms (HBDA, DA and DADT) for different loads, with buffer size of 600 packets. Load has been varied from 0.5 to 0.9. As seen in figure 6.3, HBDA has least packet loss ratio for all loads. As seen from the figure, DADT has less packet loss ratio as compared to DA since it takes packet size into consideration. Also we can see that HBDA outperforms DADT as it takes packet size as well as application state into consideration while allocating buffer space to different applications.

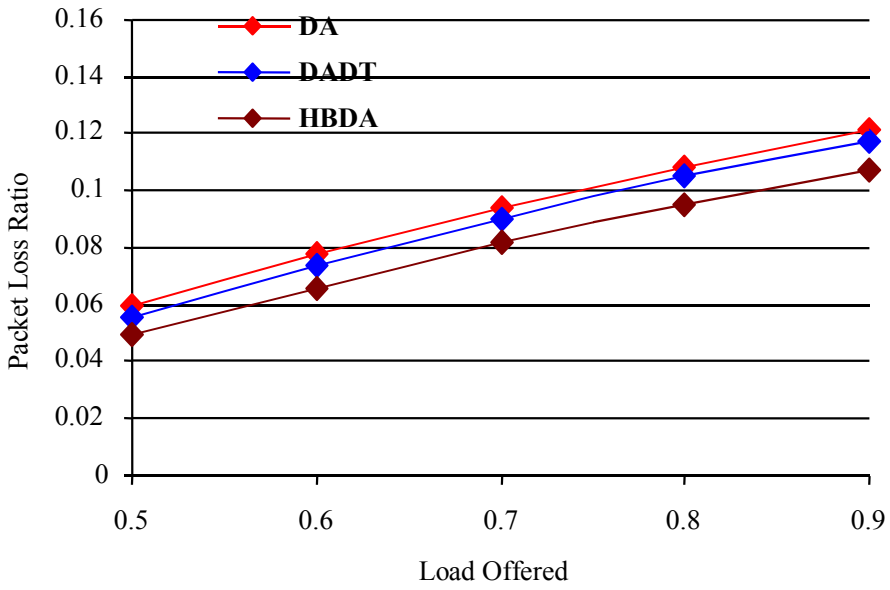


Figure 6.3**:** Packet Loss Ratio Vs Load for HBDA, DA, and DADT for the average traffic load.

6.2.5 *Comparison of HBDA, DA, DADT with varying buffer size*

Figure 6.4 shows the performance of the three algorithms (HBDA, DA and DADT) for different buffer size, with load of 70 percent on each queue. The buffer size has been varied from 500 packet size to 800 packet size. As seen from the figure 6.4, as the buffer size increases, packet loss ratio decreases for all the algorithms. This is due to the fact that all applications get more buffer space.

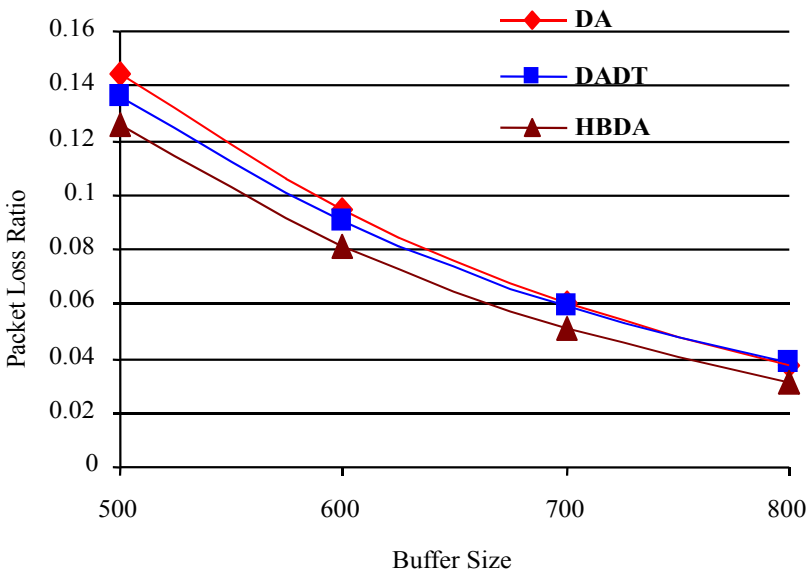


Figure 6.4**:** Packet Loss Ratio Vs Buffer size for HBDA, DA, and DADT for the average traffic load

6.2.6 *Improvement ratio of HBDA over DA and DADT*

Table 6.4 shows the improvement in packet loss ratio for HBDA, for different loads when compared with DA and DADT. The improvement ratio is defined as the difference of packet loss in HBDA and the compared algorithm (DA and DADT) divided by packet loss in HBDA.

Table 6.4

Improvement ratio of HBDA over DA and DADT for average traffic load

Load	Improvement ratio (%) (HBDA /DA)	Improvement ratio (%) (HBDA/DADT)
0.5	20	10.89
0.6	18.4	11.29
0.7	15.9	11.02
0.8	14.4	10.77
0.9	13	9.52

6.3 Simulation Results for Heavy traffic load

Table 6.5 shows the packet sizes of different applications in bytes based on the heavy network traffic load flow in [5]. For our simulation purpose of heavy traffic, we have used these packet sizes for different applications.

Table 6.5

Queue properties for Heavy traffic load

	Q0	Q1	Q2	Q3	Q4	Q5
Size in Bytes	128	64	128	32	256	512
packet unit # (32 bytes/unit)	4	2	4	1	8	16

6.3.1 *Optimum alpha value for DA*

Figure 6.5 shows packet loss ratio for DA as alpha value is varied from 4 to 20. In figure 6.5, the size of the buffer is 600 packets, the number of applications is six, bursty uniform traffic model is used with a load of 70% on each of the queues; and average dequeue time of 14 clock cycles for the burst of 10 packets. As seen from the figure 6.5, the optimum alpha value comes out to be 16.

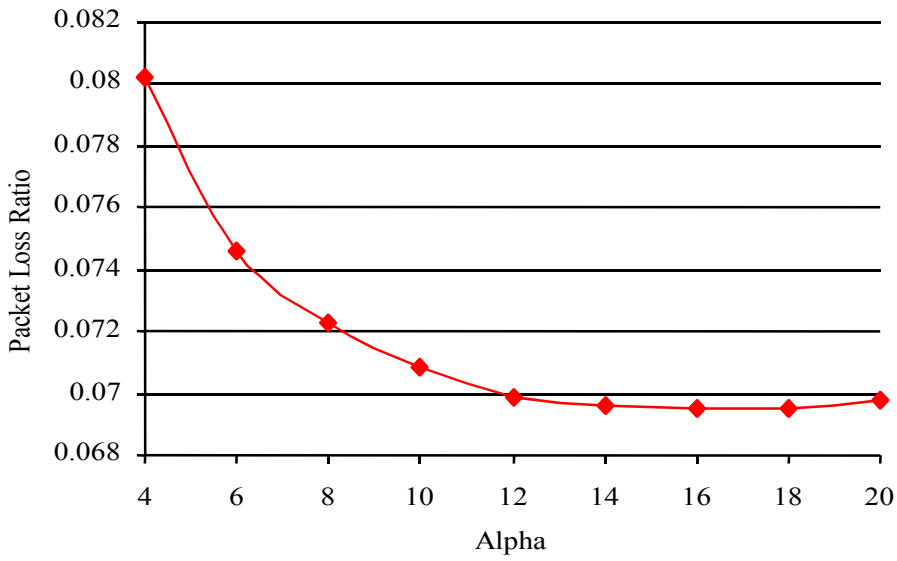


Figure 6.5**:** Packet loss ratio vs. Alpha for DA for the heavy traffic load

6.3.2 *Optimum alpha value for DADT*

Now we will determine the optimum values of alpha for DADT. Table 6.6 shows the different combinations of alpha that we have taken for our simulations and figure 6.6 shows the packet loss ratio corresponding to them. In figure 6.6, size of buffer is 600 packets, and load on each queue is 70%. Figure 6.6 shows that the optimum combination of alpha values comes out to be for variation 3.

Table 6.6

Variation of alpha for DADT for the heavy traffic load

Variation	Q0	Q1	Q2	Q3	Q4	Q5
1	18	18	18	18	18	6
2	14	10	14	10	10	7
3	**14**	**12**	**14**	**12**	**12**	**8**
4	16	14	16	14	14	6
5	16	14	16	14	16	8

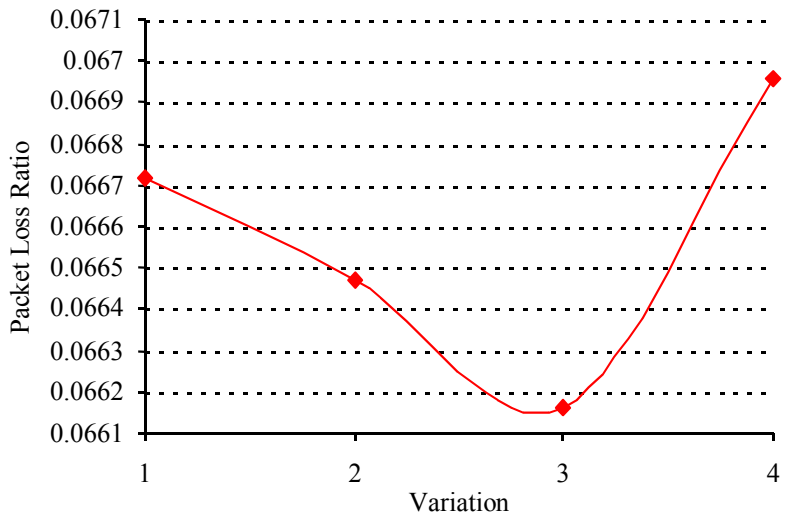


Figure 6.6**:** Packet loss ratio vs. Alpha for DADT for the heavy traffic load

6.3.3 Optimum alpha value for HBDA

Table 6.7 shows the packet loss ratio for HBDA as alpha value is varied between 16 and 256. As shown from table 6.7 optimum alpha vale comes out be 128. We will use alpha as 128 for HBDA for our comparison purpose.

Table 6.7

Variation of alpha for HBDA for the heavy traffic load

Value of alpha	**Packet Loss Ratio**
16	0.067
32	0.065
64	0.061
128	0.059
256	0.058

6.3.4 Comparison of HBDA, DA, DADT with varying load

Figure 6.7 shows the performance of the three algorithms (HBDA, DA and DADT) for heavy traffic loads with buffer size of 600 packets. Load has been varied from 0.5 to 0.9.

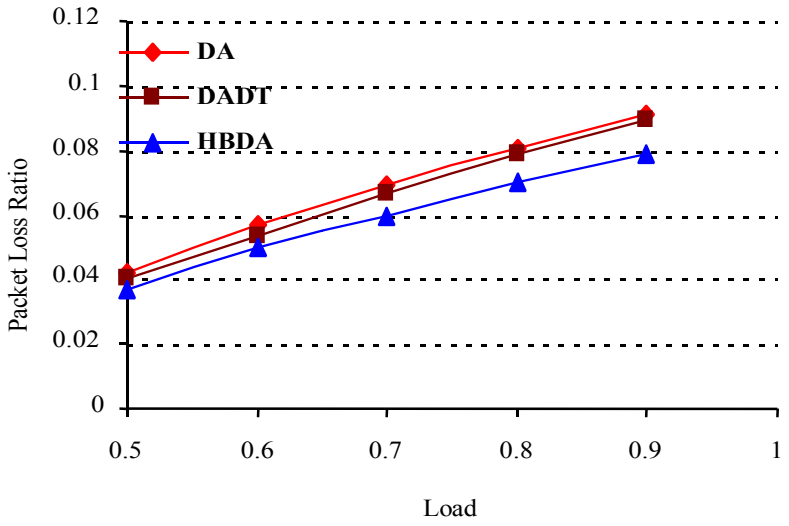


Figure 6.7**:** Packet Loss Ratio Vs Load for HBDA, DA, and DADT for the heavy traffic load

6.3.5 Comparison of HBDA, DA, DADT with varying buffer size

Figure 6.8 shows performance of three algorithms HBDA, DA, and DADT as the buffer size is varied from 500 packets to 800 packets. By monitoring the application state, HBDA reduces the overall packet loss ratio.

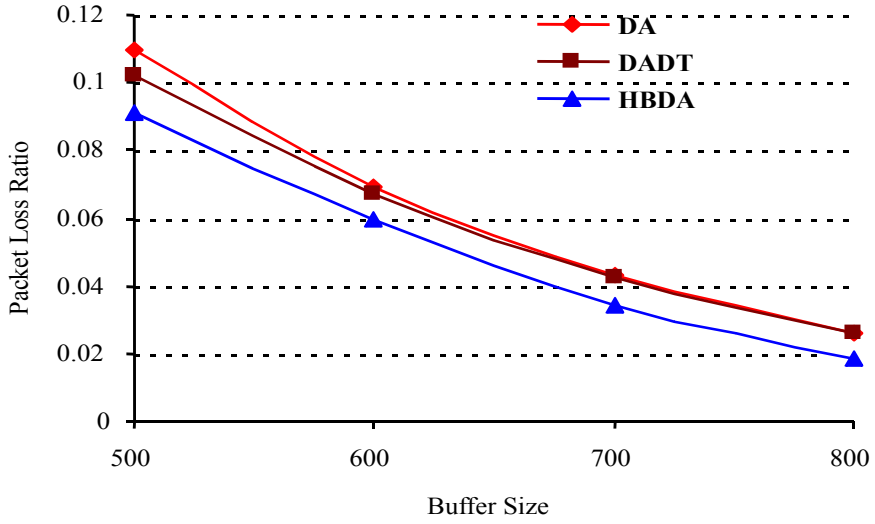


Figure 6.8: Packet Loss Ratio Vs Buffer size for HBDA, DA, and DADT for the heavy traffic load

6.3.6 Improvement ratio of HBDA over DA and DADT

Table 6.8 shows the improvement in packet loss ratio for HBDA, for different loads when compared with DA and DADT. For a load of 0.7 the improvement ratio is 16.2% over DA and 11.7% over DADT.

Table 6.8

Improvement ratio of HBDA over DA and DADT for heavy traffic load

Load	**Improvement ratio (%) (HBDA /DA)**	**Improvement ratio (%) (HBDA /DADT)**
0.5	15.2	8.5
0.6	15.8	9.8
0.7	16.2	11.7
0.8	15.4	13.0
0.9	13.2	11.4

6.4 Simulation Results for actual traffic load

Table 6.9 shows the packet sizes of different applications in bytes based on the actual network traffic load flow in [18].

Table 6.9

Queue properties for actual traffic load

	Q0	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**
Size in Bytes	32	32	32	64	512	1472
packet unit # (32 bytes/unit)	1	1	1	2	16	46

6.4.1 *Optimum alpha value for DA*

Figure 6.9 shows packet loss variation using DA as alpha value is varied from 4 to 16 for actual network traffic load. As seen from Figure 6.9 the optimum alpha value comes out to be 4. As the value of alpha is increased the packet loss ratio increases. This is due to the fact that packet size of queue5 (46 bytes) is very large.

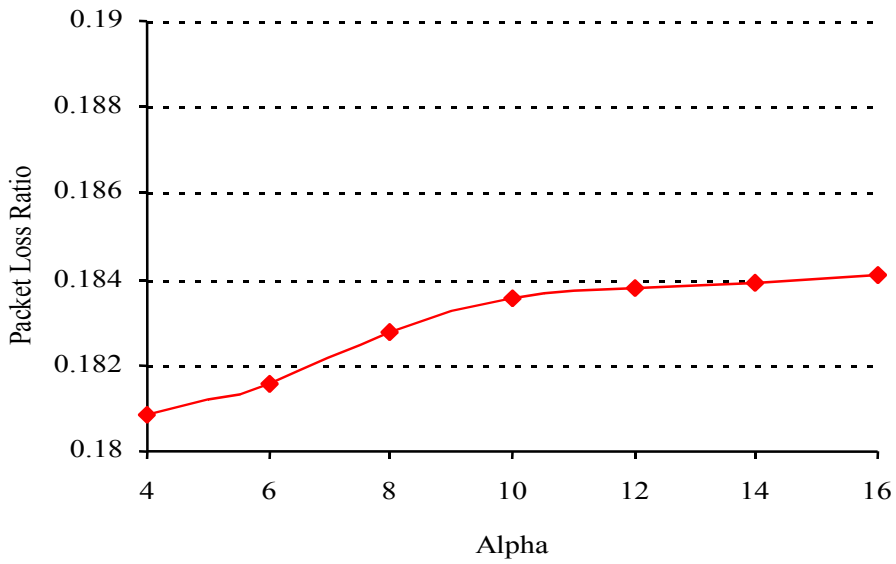


Figure 6.9**:** Packet loss ratio vs. Alpha for DA for the actual traffic load

6.4.2 *Optimum alpha value for DADT*

Table 6.10 shows the different combinations of alpha values that we have taken for our simulations and figure 6.10 shows the packet loss ratio corresponding to them. From figure 6.10, optimum value of alpha comes out to be for variation 5.

.

Table 6.10

Variation of alpha for DADT for the actual traffic load

Variation	**Q0**	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**
1	16	16	16	16	6	4
2	16	16	16	16	6	6
3	18	18	18	18	6	4
4	16	16	16	16	16	6
5	**16**	**16**	**16**	**16**	**16**	**4**

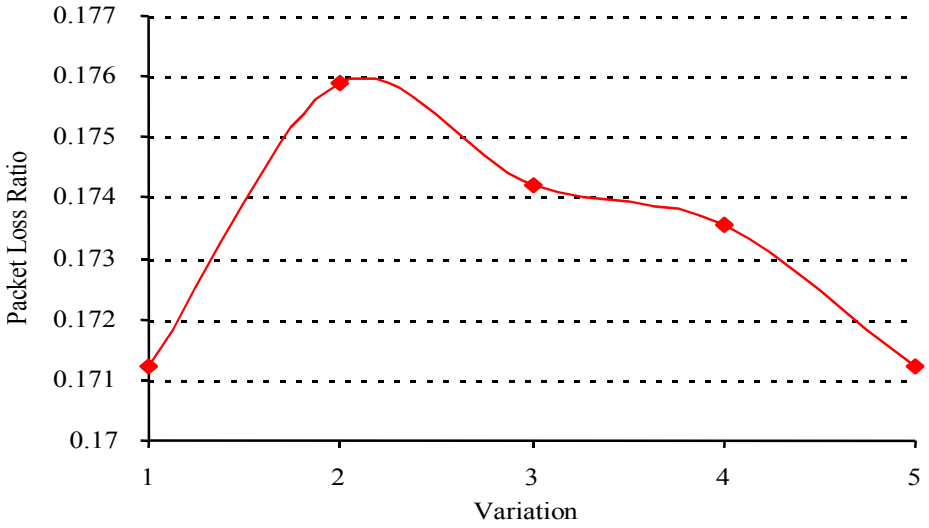


Figure 6.10**:** Packet loss ratio vs. Alpha for DADT for the actual traffic load

6.4.3 Optimum alpha value for HBDA

Table 6.11 shows the packet loss ratio for HBDA as alpha value is varied is between 16 and 256. As shown from table 6.10 optimum alpha value comes out to be 64. We will use alpha as 64 for our comparison purpose for HBDA.

Table 6.11

Variation of alpha for HBDA for the actual traffic load

Value of alpha	**Packet Loss Ratio**
16	0.167
32	0.165
64	0.162
128	0.168
256	0.171

6.4.4 Comparison of HBDA, DA, DADT with varying load

Figure 6.11 shows the performance of the three algorithms (HBDA, DA and DADT) for actual traffic loads with buffer size of 600 packets. Load has been varied from 0.5 to 0.9. As seen in figure 6.11, HBDA has least packet loss ratio for all loads.

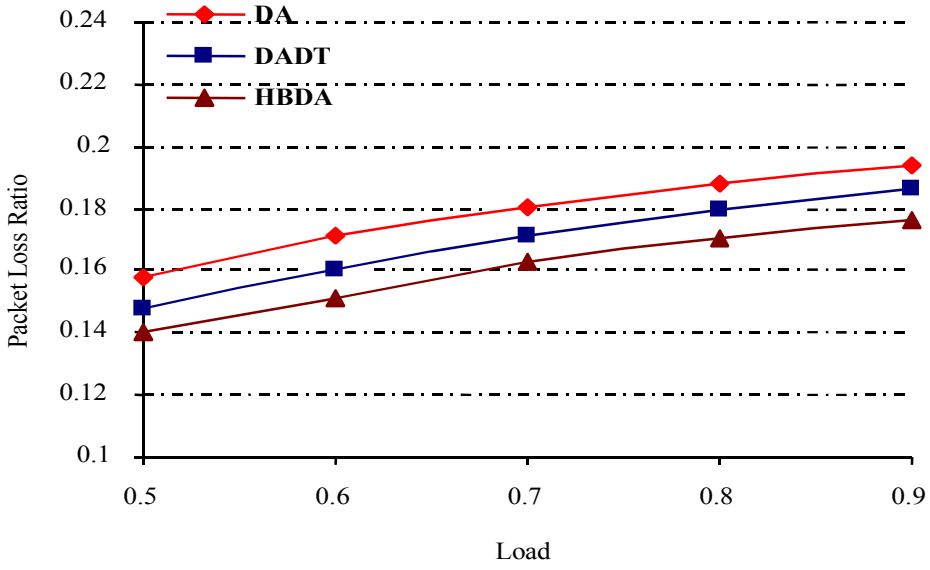


Figure 6.11**:** Packet Loss Ratio Vs Load for HBDA, DA, and DADT for the actual traffic load

6.4.5 Comparison of HBDA, DA, DADT with varying buffer size

Figure 6.12 shows performance of three algorithms HBDA, DA, and DADT as the buffer size is varied from 500 packets to 800 packets.

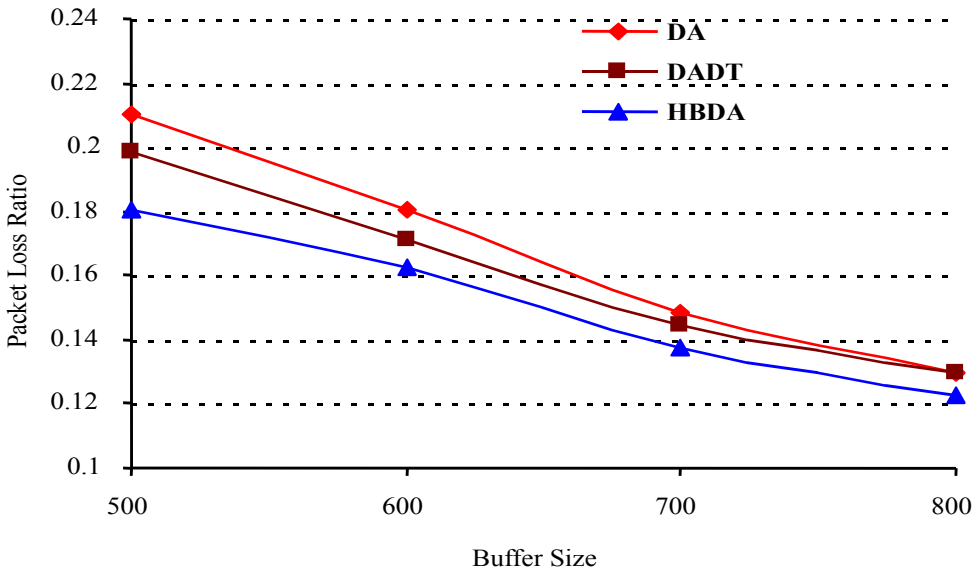


Figure 6.12**:** Packet Loss Ratio Vs Buffer size for HBDA, DA, and DADT for the heavy traffic load

6.4.6 Improvement ratio of HBDA over DA and DADT

Table 6.12 shows the improvement in packet loss ratio for HBDA, for different loads when compared with DA and DADT.

Table 6.12

Improvement ratio of HBDA over DA and DADT for actual traffic load

Load	**Improvement ratio (%) (HBDA /DA)**	**Improvement ratio (%) (HBDA /DADT)**
0.5	11.7	4.6
0.6	12.5	6.0
0.7	12.7	7.1
0.8	22	17.1
0.9	11.9	7.6

6.5 Proposed Architecture for NIC

Table 6.13 shows the packet sizes of different applications in bytes based on the average network traffic load flow [10]. For our simulation of the average traffic load, we used these packet sizes for different applications. For our simulations we have taken application 2 as application with highest priority. So, packets for application 2 will be placed in the priority controller in case they are rejected by the buffer management algorithm.

Table 6.13.

Queue properties for average traffic load

	Q0	Q1	**Q2** Priority Application	Q3	Q4	Q5
Size in bytes	256	64	256	32	128	512
packet unit # (32 bytes/unit)	8	2	8	1	4	16

Figure 6.13 compares the packet loss ratio of the HBDA for different loads for the traditional architecture and the proposed architecture. Load has been varied from 0.5 to 0.9. As seen from figure 6.13, the overall packet loss ratio has been reduced significantly. Table 6.14 show the packet loss ratio for priority application (Application 2) for traditional architecture and the proposed architecture. As seen from the table 6.14, packet loss ratio for application 2 has been reduced significantly.

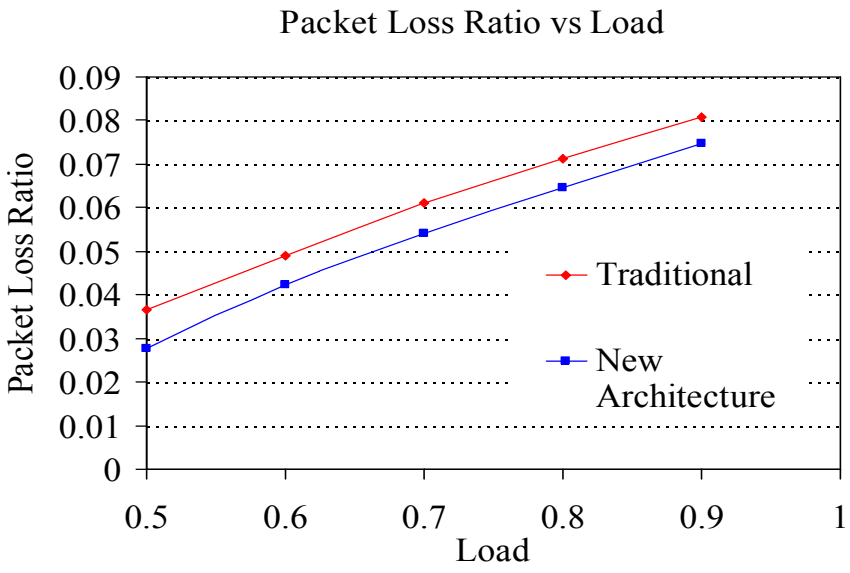


Figure 6.13: Packet loss ratio vs. Load for HBDA for the average traffic load in traditional architecture and the Multiprocessor architecture.

Table 6.14

Packet loss ratio of priority application

Load	**Packet loss ratio of priority application in traditional architecture**	**Packet loss ratio of priority application in Multiprocessor architecture**
0.5	202187	135918
0.6	349921	201981
0.7	526547	318768
0.8	730182	519076
0.9	883981	617632

Though, the main idea behind the new architecture is to reduce the number of interrupts needed to be sent to the host processor and also to process multiple packets at the same time. But, as shown above packet losses will also be reduced.

6.5.1 Power Analysis of Proposed Architecture

Table 6.15 shows the power comparison of transitional architecture and proposed architecture. In the proposed architecture, we have two control units for processing the incoming packets. This increase in the hardware increases the overall the dynamic power consumption.

Increase in power is calculated as:

$$\frac{\text{Dynamic power (new architecture) - Dynamic power (traditional architecture)}}{\text{Dynamic power (traditional architecture)}}$$

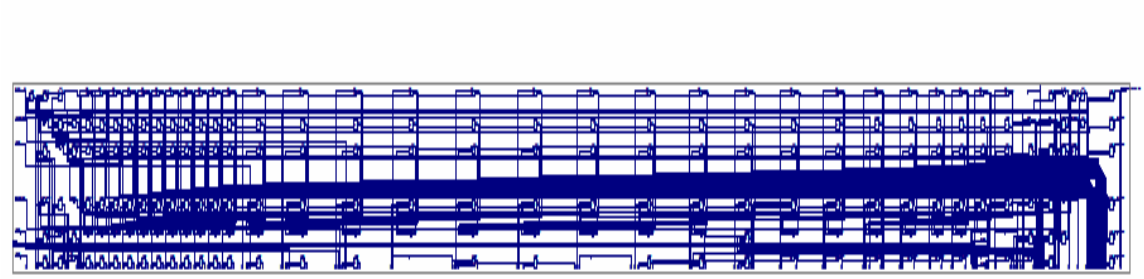


Figure 6.14: Snapshot from Logic Diagram for Multiprocessor Architecture

For calculating dynamic power for traditional and Multiprocessor architecture following steps were followed: Simulation code was first converted to synthesizable code. After converting the code, the design was then synthesized and implemented.

After implementation, Post-route simulation is done to generate the VCD file (Appendix A.1).

Figure 6.14 shows the logic diagram of the Multiprocessor architecture. Then VCD file was given as input to Xilinx Xpower tool to calculate power consumption. Appendix A describes all the steps involved in determining the power with the help of an example.

For our performance comparison, load has been varied from 0.5 to 0.9 for the average traffic load and buffer size is 600 packets.

Table 6.15

Power Comparison of traditional and proposed architecture

Load	**Dynamic Power Increase (%)**
0.5	27.7%
0.6	29.1%
0.7	29.9%
0.8	31.3%
0.9	32.6%

CHAPTER VII

CONCLUSION AND FUTURE WORK

This thesis proposes the History Based Dynamic Algorithm (HBDA) to reduce the number of packets being dropped at the packet buffer. The HBDA reduces that packet loss ratio by considering application state, packet sizes of applications while determining the threshold value for each application. Thus, an application which is active at any instant gets more threshold value than any other application which is not active at the same instant.

The buffer management algorithm decides the amount of space for each output queue in the packet buffer. Three buffer management algorithms Dynamic Algorithm (DA), Dynamic Algorithm with Dynamic Threshold (DADT) and History Based Dynamic Algorithm (HBDA) are implemented in this thesis. DA does not take packet sizes into consideration while allocation buffer space to any application. DADT outperforms DA by taking packet size into consideration. HBDA keep tracks of last two packets of any application. This way, HBDA determines whether the application is active or not at any given instant of time.

Of all the popular architectures including traditional and protocol processor architecture for NIC, none of them is designed for multi processors system and also none of the architecture takes priority packets into consideration.

So we also propose a new architecture that can work in multi-processors system and also reduce the packet losses of priority applications.

This chapter summarizes the advantages of HBDA algorithm over the conventional algorithms. This chapter also summarizes the benefits of new architecture for an NIC over the existing architectures for the NIC and discusses future research possibilities. Section 7.1 summarizes the results of the previous chapters and section 7.2 discusses the future work.

7.1 Summary of Results

7.1.1 *HBDA*

The Dynamic algorithm (DA) works well for ATM switches where packet size is same for all the applications. However in network terminals, different applications may have different packet sizes. So, if we use DA, application with large packet size tends to occupy more buffer space resulting in increase in packet loss of other applications. The Dynamic Algorithm with Dynamic Threshold (DADT) takes only the packet size into consideration and not the application state while calculating the threshold values. Also, it is difficult to determine the optimum alpha value for each application in DADT.

So we proposed a HBDA algorithm that takes both the application state and packet size into consideration. Also, it eliminates the need to calculate the optimum alpha value for each application. By taking alpha as power of two, hardware implementation has been made easier.

The simulations considered 6 output queues (0-5), bursty uniform traffic model, dequeue time of 14 clock cycles for a burst of 10 packets, and uniform load for all the output queues.

1) For the traffic mix with average network traffic loads [5], the HBDA improves the packet loss ratio by 15.9% and 11% (for load = 0.7) compared to DA and DADT, respectively.
2) For heavy traffic load improvement is 16.2% and 11.7% (for load = 0.7) compared to DA and DADT, respectively.
3) For actual traffic load improvement is 12.7% and 7.1% (for load = 0.7) over DA and DADT respectively.

7.1.2 *A New Architecture for a NIC*

Multi-processors systems are most commonly used now days. Data is transmitted at a very high rate across the network but the speed of the processors on the computer limits the data speed. To overcome this, we proposed a new architecture in which a packet buffer on a NIC supports multiple processors. A packet buffer has multiple output ports and thus multiple processors can demand for packets at the same time. A priority-based buffer has also been placed in front of packet buffer to minimize the packet loss ratio for the priority packets. Priority packets rejected by the controller are placed in the priority-based buffer. Packets in priority-based buffer are moved to the packet buffer when there is space for them in the packet buffer. Priority Controller determines when the packet can be moved from the priority-based buffer to the packet buffer and also informs the controller about the number of packets

moved. The controller updates its counters accordingly. The proposed architecture results in minimizing the overall packet loss ratio and increasing the capacity in the networks.

7.2 Future Work

The HBDA algorithm proposed in this thesis does not take 'time' factor into consideration while determining whether the application is active or not. It would be interesting to see how HBDA behaves if 'time' factor is taken into consideration. Thus, an application which has been inactive for a longer period of time should have 'History(1)' and 'History(2)' flags set to zero again.

REFERENCES

[1] A. Tanenbaum, *Computer Networks*, 4th ed., Prentice Hall, 2002.

[2] T. Henriksson, U. Nordqvist, D. Liu, "Embedded Protocol Processor for fast and efficient packet reception", *IEEE Proceedings on Computer Design: VLSI in Computers and Processors*, vol. 2, pp. 414-419, September 2002.

[3] V. Paxson, "End-to-End internet packet dynamics", *Proceedings of ACM SIG-COM*, vol. 27, pp. 13-52, October 1997.

[4] Tomas Henriksson, "Intra-Packet Data-Flow Protocol Processor", *PhD Dissertation*, Linkopings universitet, 2003.

[5] U. Nordqvist, D. Liu, "Power optimized packet buffering in a protocol processor", *Proceedings of the 2003 10th IEEE International Conference on Electronics, Circuits and Systems*, vol. 3, pp. 1026-1029, December 2003.

[6] M. Arpaci, J.A. Copeland, "Buffer Management for Shared Memory ATM Switches", *IEEE Communication Surveys*, First Quarter 2000.

[7] T. Henriksson, U. Nordqvist, D. Liu, "Specification of a configurable general-purpose protocol processor", *IEEE Proceedings on Circuits, Devices and Systems*, vol. 149, issue: 3, pp. 198-202, June 2002.

[8] A. Tobagi, "Fast Packet Switch Architectures for Broadband Integrated Services Digial Networks", *Proceedings of IEEE*, vol. 78, pp. 133-167, January 1990.

[9] M. Irland, "Buffer Management in a Packet Switch", *IEEE Transactions on Communications*, COM-26, no. 3, pp. 328-337, March 1978.

[10] G. J. Foschini, B. Gopinath, "Sharing Memory Optimally", *IEEE Transactions on Communications*, vol. COM-31, no. 3, pp. 352-360, March 1983.

[11] F. Kamoun, L. Kleinrock, "Analysis of Shared Finite Storage in a Computer Network Node Environment under General Traffic Conditions", *IEEE Transactions on Communications*, vol., COM-28, pp. 992-1003, July 1980.

[12] S. X. Wei, E.J. Coyle, M.T. Hsiao, "An Optimal Buffer Management Policy for High-Performance Packet Switching", *Proceedings of IEEE GLOBECOM'91*, vol. 2, pp. 924-928, December 1991.

[13] A. K. Thareja, A.K. Agarwal, "On the Design of Optimal Policy for Sharing Finite Buffers", *IEEE Transactions on Communications*, vol. COM—32, no. 6, pp 737-780, June 1984.

[14] A. K. Choudhury, E.L. Hahne, "Dynamic Queue Length Thresholds for Shared-Memory Packet Switches", *IEEE/ACM Transactions on Communications*, vol. 6, no. 2, pp. 130-140, April 1998.

[15] Sundar Iyer, " *SIM: A Fixed Length Packet Simulator*", http://klamath.stanford.edu/tools/SIM .*(Accessed : 8th January ,2006)*

[16] Yul Chu, Vinod Rajan "An Enhanced Dynamic Packet Buffer Management", In the proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC'05), Volume, Issue, 27-30 June 2005 Page(s): 869 – 874, Spain, June 2005

[17]Cisco Systems: http://www.cisco.com/warp/public/473/lan-switch-cisco.shtml.*(Accessed : 2nd March ,2006)*

[18] S. McCreary and K. Claffy, "*Trends in Wide Area IP Traffic Patterns: A View from Ames Internet Exchange,"* In ITC Specialist Seminar on IP Traffic Measurement, Modeling, and Management, Manterey, California, September 2000.

[19] Sam Manthorpe: http://lrcwww.epfl.ch/people/sam/research_protlevels.html. *(Accessed : 2nd October ,2005)*

APPENDIX

XPOWER ANALYSIS

Power consumption is critical in the designing process of a mobile device. With the rapidly advanced technology and the greatly increased integration density and clock frequency, power consumption is becoming more and more important. Higher power consumption has a negative effect on battery life, packaging, cooling costs, and reliability. There are many tools that can perform power analysis. However, the popular one is Xpower tool provided with Xilinx ISE. XPower tool is a post-route and post-fit analysis tool that enables to interactively and automatically analyze power consumption for Xilinx FPGAs and CPLDs.

VHDL, or VHSIC Hardware Description Language, is commonly used as a design-entry language for field-programmable gate arrays and application-specific integrated circuits in electronic design automation of embedded network digital circuits. Therefore, we have used VHDL for describing our design while performing the power analysis. There are four factors that determine the power dissipation in a circuit: 1) magnitude of supply voltage; 2) switching activity in the circuit; 3) switching capacitive loads; and 4) clock frequency. There are two main components to power consumption:

• Dynamic power, which is determined by the switching power of the core and the switching speed of the I/O. Dynamic power is affected by capacitive load, supply voltage, and switching frequency.

• Quiescent power, which is dominated by transistor leakage current and by DC current from a few specialized FPGA circuits.

A.1 Power Analysis

Figure A.1 show the steps involved in performing the power analysis. The behavioral code, written in VHDL, is simulated using Modelsim to test the functionality of the code. If required, necessary changes are made in the behavioral code to achieve the desired functionality. After simulating the behavioral code, the next step involved in power analysis is synthesis.

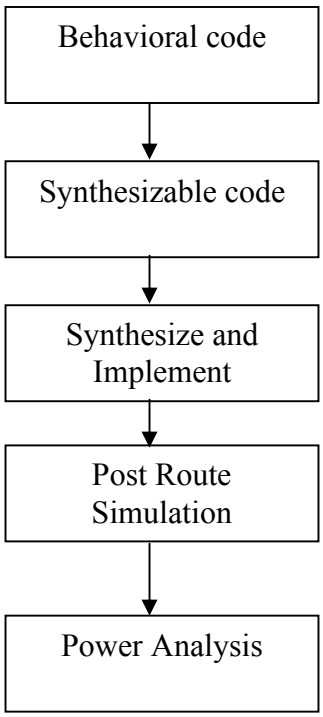


Figure A.1: Steps involved in Power Analysis

Behavioral code cannot be synthesized directly. It has to be converted to synthesizable code. After converting the code, the design is then synthesized and implemented. After implementation, Post-route simulation is done to generate the VCD file.

A.2 Converting Behavioral Code to Synthesizable Code

Directly synthesizing the behavioral code can result in errors and warnings. Synthesis tool does not understand all the statements that might have been used in behavioral code for behavioral simulation. Following are some points that should be taken into consideration while converting the behavioral code to the synthesizable code.

- Avoid using any of the signal attributes 'active, 'stable, 'quiet, 'last_value, 'last_event, 'delayed. Standard expressions clk'event and clk='1' or rising_edge(clk) can be used . This is the way to construct positively clocked flipflops .
- With clk'event, always add clk='1'. It is not possible to synthesize flip flops that are clocked on both the positive and negative clock edge .
- Combinational processes must have "complete sensitivity lists". That is, all signals that are read in the process must be listed in the sensitivity list.
- Don't use any arithmetic operations, like division or modulus division. Though, addition and multiplication can be done .
- Use std_logic_vector instead of integers. Though some synthesize tool can implicitly convert integers to 32 bit std_logic_vector, it is a better practice to convert them explicitly.
- Different synthesis tools may require certain programming "styles" to recognize, for example, state machines. Follow these rules to be on the safe side .

- Synthesis Tool does not support Operation on files.
- Synthesis Tool does not support “after statements”.
- Do not assign signals and variables initial values because initial values are ignored by most synthesis tools.
- All outputs should be defined in all branches of an If statement to prevent latches in the circuit.
- Synthesis Tool does not support “Transport statements”.
- Synthesizers infer latches from incomplete conditional expressions, such as: an If statement without an Else clause and an intended register without a rising edge or falling edge construct.

After converting the behavioral code to synthesizable code, the next step is to synthesize and implement the design.

A.3 Synthesizing and Implementing the Design

Once a design is entered and simulated, the next step in the design flow is synthesis. Synthesis is the process of converting behavioral HDL descriptions into a network of logic gates . The synthesis engine takes as input the HDL design files and a library of primitives. Primitives are not necessarily just simple logic gates like AND, OR gates and D-registers, but can also include more complicated things such as shift registers and arithmetic units. Xilinx Synthesis tool XST takes VHDL file as input and generates ‘.ngc’ file. A synthesis report file is also generated, which

describes the logic inferred for each part of the HDL file, and often includes helpful warning messages .

After synthesis, the next step is implementing the design. Figure A.2 show the steps (Translate → Map → Place and Route) involved in implementing the design. In the translate process, the design is cut into small pieces which are implemented in look-up tables (LUTs). The output of translate process is '.ngd' file.

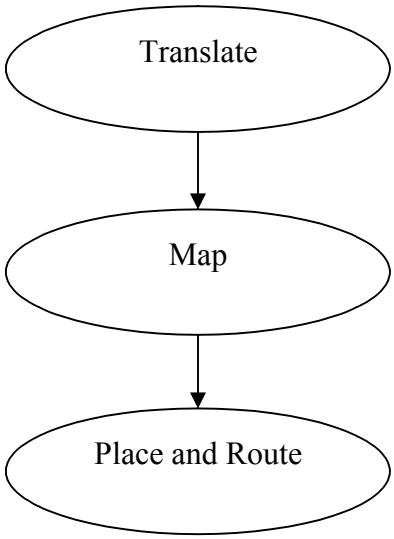


Figure A.2**:** Steps in implementing the design.

The .ngd file is a netlist of primitive gates, which could be implemented on any one of a number of types of FPGA devices Xilinx manufacturers. The next step is to map the primitives onto the types of resources (logic cells, I/O cells, etc.) available in the specific FPGA being targeted . The output of the Xilinx map process tool is an '.ncd' file.

The design is then placed and routed, meaning that the resources described in the '.ncd' file are assigned specific locations on the FPGA, and the connections between the resources are mapped into the FPGAs interconnect network . The delays associated with interconnect on a large FPGA can be quite significant, so the place and route process has a large impact on the speed of the design. The output of the place and route engine is an updated '.ncd' file, which contains all the information necessary to implement the design on the chosen FPGA .

6.5 Writing a Testbench

The entity/test-bench pair can form the basis for executable specifications and documentation in a top-down design methodology . In other words, the test-bench will generate the input signals for the design and, if necessary, it will also give the appropriate response from an output signal of the design. Figure A.3 below shows sample test bench code for 4-input multiplexer for a case with select line "=00".Similarly input test vectors can be applied for different select lines to check the functionality of multiplexer. Note, that the entity declaration is empty.

```
entity Mux_TB is          -- empty entity
end Mux_TB;
architecture TB of Mux_TB is
    signal T_I3: std_logic_vector(2 downto 0):="000";
    signal T_I2: std_logic_vector(2 downto 0):="000";
    signal T_I1: std_logic_vector(2 downto 0):="000";
    signal T_I0: std_logic_vector(2 downto 0):="000";
    signal T_O: std_logic_vector(2 downto 0);
    signal T_S: std_logic_vector(1 downto 0);
    component Mux
    port(   I3:   in std_logic_vector(2 downto 0);
      I2:   in std_logic_vector(2 downto 0);
      I1:   in std_logic_vector(2 downto 0);
      I0:   in std_logic_vector(2 downto 0);
      S: in std_logic_vector(1 downto 0);
      O: out std_logic_vector(2 downto 0)
    );
    end component;
begin
    U_Mux: Mux port map (T_I3, T_I2, T_I1, T_I0, T_S, T_O);
      process
       begin
   T_I3 <= "001";
   T_I2 <= "010";
   T_I1 <= "101";
   T_I0 <= "111";
   wait for 10 ns;
   T_S <= "00";
   wait for 1 ns;
   assert (T_O="111") report "Error Case 0" severity error;
```

Figure A.3: Testbench for Multiplexer

A.5 Generating the VCD File

Using ModelSim, we can create a VCD file containing transition data noted during the simulation. This VCD file is imported into XPower which then converts the data to activity rate data and is matched to the appropriate net. It is important to note that a simulation must be of sufficient length that is all signals should toggle in that simulation period. . Otherwise, signals that change states very infrequently will

be misrepresented in the VCD file. The following paragraph describes how to generate a VCD file using XILINX ISE.

Create a new project or open an existing project. Add the test bench, project files in the project. Synthesize and implement the project. Then, select post route simulation from the "Source for" combo box above project files as shown in figure A.4. Then, select the testbench in the sources window and expand Modelsim-Simulator in processes window and right click on Simulate Post-place & Route Model and click properties. This is shown in figure A.5 below. Then, check the Generate VCD file checkbox to generate VCD file and specify the Simulation run time. The simulation time should be such that most of the signal should toggle.

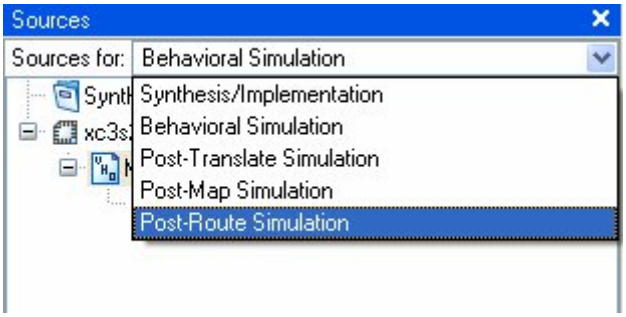


Figure A.4: Selecting Post Route Simulation.

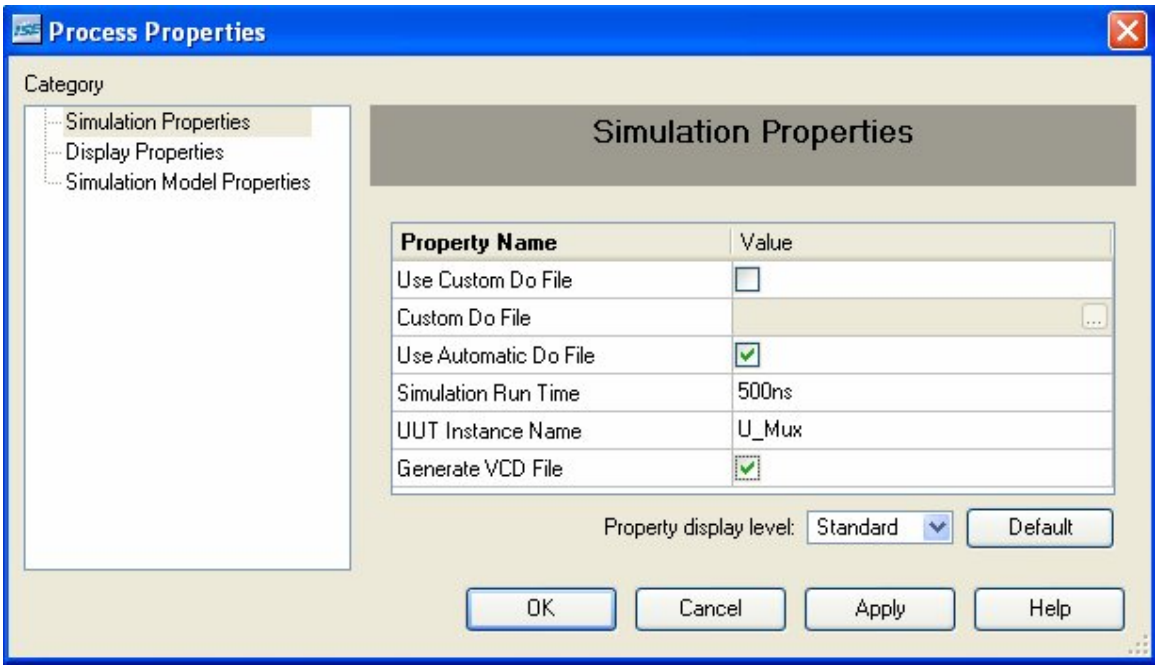


Figure A.5: Selecting Simulation Properties.

After setting the simulation time, click the apply button shown in figure A.5 above. Then, right click on Simulate Post-place & Route and click run. This will open the Modelsim and run it till the time specified in the simulation run time. This will also generate VCD file. This VCD file will be used by Xpower tool. The signals that are changing on clock must be there to get accurate power.

A.6 VCD FILE Format

The format of generated VCD files adheres to IEEE Std 1364–2001 . The following table describes the format

Table A.1

VCD File Format

File Content	Description
$date 23-Sep-2003 14:38:11 $end	Data and time the file was generated.
$version Link for ModelSim version 1.0 $ end	Version of the VCD block that generated the file.
$timescale 1 ns $ end	The time scale that was used during the simulation.
$scope module manchestermodel $end	The scope of the module being dumped.
$var wire 1 ! Original Data [0] $end $var wire 1 " Recovered Clock [0] $end $var wire 1 # Recovered Data [0] $end $var wire 1 $ Data Validity [0] $end	Variable definitions. Each definition associates a signal with character identification code (symbol). The symbols are derived from printable characters in the ASCII character set from ! to ~. Variable definitions also include the variable type (wire) and size in bits.
$upscope $end	Marks a change to the next higher level in the HDL design hierarchy.
$enddefinitions $end	Marks the end of the header and definitions section.
#0	Simulation start time.
$dumpvars 0! 0" 0# 0$ $end	Lists the values of all defined variables at time equals 0.

Table A.1 (continued)

#630 1!	The starting point of logged value changes. Variable values are checked at each simulation time increment and are logged if a change occurs. This entry indicates that at 63 nanoseconds, the value of signal Original Data changed from 0 to 1.
. . . #1160 1# 1$	At 116 nanoseconds the values of signals Recovered Data and Data Validity changed from 0 to 1.
$dumpoff x! x" x# x$ $end	Marks the end of the file by dumping the values of all variables as

Figure A.6 below shows the part of sample VCD file for multiplexer.

```
$date
        Sat Jul 22 23:57:54 2006
$end
$version
        ModelSim Version 6.1e
$end
$timescale
        1ps
$end
$scope module mux_tb $end
$scope module u_mux $end
                        $var wire 1 ! i0 [2] $end
$var wire 1 " i0 [1] $end
$upscope $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
1!
```

Figure A.6: Part of VCD file for Multiplexer.

A.7 Xpower tool

XPower is the first power-analysis software available for programmable logic design. The designer supplies estimates of parameters like logic, memory, and I/O utilization, clock frequencies, toggle rates, and operating temperatures to the XPower tool. The Xpower tool then produces an estimate of power consumption for those conditions. XPower calculates the power as a summation of the power consumed by each element in the design. XPower calculates an estimate of power to within +/- 10% . Inputs to this tool are :

1. Placed and Routed NCD file (output of PAR)
2. Physical constraints file (PCF) (output of Map)

3. Simulation file (VCD file)

A.7.1 *Running the Xpower tool*

After synthesizing, implementing the design and doing post route simulation, the last step in determining the estimated power consumption is running the Xilinx XPower tool. Following are the steps for running the XPower tool and determining power:

1) Go to Start → programs and run xpower.exe

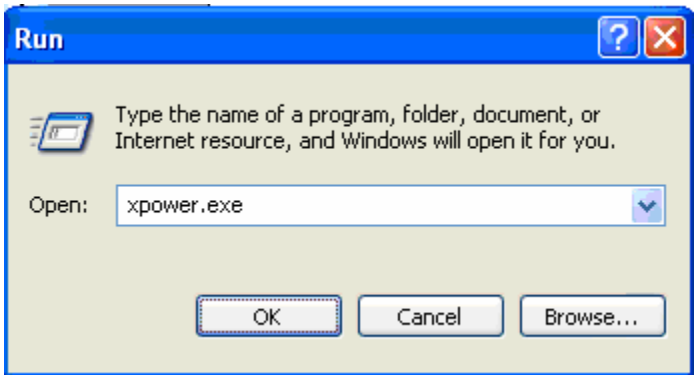


Figure A.7: Running the Xpower tool

2) This will open Xpower tool. Go to file → open.

3) Enter the design file, Constraint file and Simulation file as shown in figure A.8. Click 'ok' to run the tool. This will generate power report as shown in figure A.9 and figure A.10.

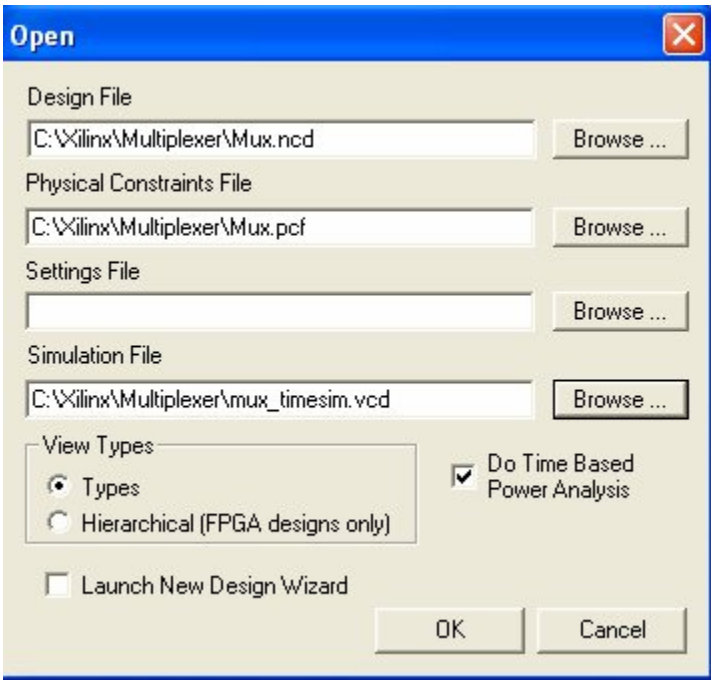
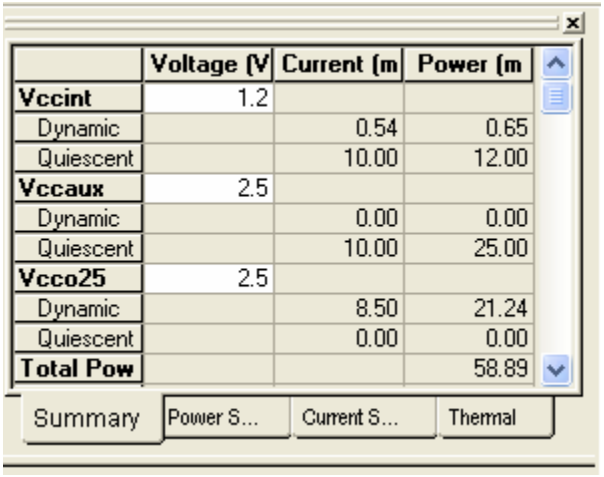


Figure A.8: Input files for XPower tool



	Voltage (V	Current (m	Power (m
Vccint	1.2		
Dynamic		0.54	0.65
Quiescent		10.00	12.00
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		10.00	25.00
Vcco25	2.5		
Dynamic		8.50	21.24
Quiescent		0.00	0.00
Total Pow			58.89

Summary | Power S... | Current S... | Thermal

Figure A.9: Power summary for the design

XPower and Datasheet may have some Quiescent Current differences. This is due to the fact that the quiescent numbers in XPower are based on measurements of real designs with active functional elements reflecting real world design scenarios.

Power summary:	I(mA)	P(mW)
Total estimated power consumption:		59
Peak Power consumption::		898
Vccint 1.20V:	11	13
Vccaux 2.50V:	10	25
Vcco25 2.50V:	8	21
Inputs:	0	0
Logic:	0	0
Outputs:		
Vcco25	8	21
Signals:	0	0
Quiescent Vccint 1.20V:	10	12
Quiescent Vccaux 2.50V:	10	25

Figure A.10: XPower Report