

ABSTRACT

SURYANARAYANAN, DEEPAK. A Methodology for Study of Network Processing

Architectures. (Under the direction of Dr.Gregory T Byrd.)

A new class of processors has recently emerged that encompasses programmable ASICs and microprocessors that can implement adaptive network services. This class of devices is collectively known as Network Processors (NP). NPs leverage the flexibility of software solutions with the high performance of custom hardware. With the development of such sophisticated hardware, there is a need for a holistic methodology that can facilitate study of Network Processors and their performance with different networking applications and traffic conditions. This thesis describes the development of Component Network Simulator (ComNetSim) that is based on such a technique. The simulator demonstrates the implementation of Diffserv applications on a Network Processor architecture and the performance of the system under different network traffic conditions.

**A METHODOLOGY FOR STUDY OF NETWORK PROCESSING
ARCHITECTURES**

By

Deepak Suryanarayanan

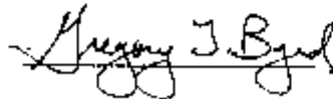
**A thesis submitted to the Graduate Faculty of
North Carolina State University in partial fulfillment of the
requirements for the Degree of Master of Science**

Computer Engineering

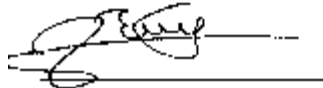
Raleigh

2001

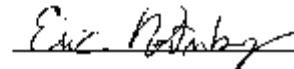
APPROVED BY:



Dr. Gregory T. Byrd (Chair)



Dr. Ioannis Viniotis



Dr. Eric Rotenberg

Acknowledgement

The past year has been one of the most challenging, yet fulfilling periods of my academic career. During this time, I have had the privilege of working with many talented individuals at NC State and Cisco Systems. The collective input from all these people helped me architect this complete work. I owe many thanks to them.

Dr. Greg Byrd, for allowing me the opportunity to work with him, probe his intellect and learn from his experience. Mr. Ken Key, for his immense patience and constant encouragement. Without his involvement and effort this work would have remained but a possibility. Dr. Ioannis Viniotis for steering me in the right direction and Dr. Eric Rotenberg for introducing me to microprocessor simulator design. Mr. John Marshall for providing the initial impetus and highlighting the methodology used in this project and Mr. Robert Jeter, who helped me understand the Toaster memory system better. Ms. Lisa Huang for her many pointers that helped shed light on the working of Toaster and Mr. Jeff Brown, whose insights in software design helped me make the simulator more modular. Mr. Vinayak Parameshwara for being my white board and directing me to solutions. And the executive team of Cisco Systems who funded this project and permitted the use of Toaster material.

Dedicated to my parents, whose dreams I seek to fulfill. I thank them for providing me with the education, background and support that made it all possible.

Biography

Deepak Suryanarayanan was born on August 1, 1978 in the city of Bangalore, India. He attended high school at Vidya Mandir Sr. Sec. School, Mylapore, Madras and went on to obtain his Bachelor's degree in Electrical and Electronics Engineering from Sri Venkateswara College of Engineering, University of Madras in April 1999. Deepak has been a graduate student in the Electrical Engineering and Computer Engineering department at North Carolina State University since August, 1999. During this time he was a member of the Architecture Research Group at NC State University and was a Co-op Hardware Engineer at Cisco Systems, RTP from May 2000 to May 2001.

Contents

LIST OF TABLES	VI
LIST OF FIGURES	VII
CHAPTER 1 INTRODUCTION	1
1.1 NETWORK PROCESSING	1
1.2 NETWORK PROCESSORS	3
1.2.1 <i>The Need for Network Processors</i>	3
1.2.2 <i>The IBM Network Processor</i>	4
1.2.3 <i>Intel Network Processor – Internet Exchange Architecture</i>	7
1.2.4 <i>Cisco Systems Toaster Network Processor</i>	9
1.2.4 <i>Summary</i>	12
1.3 DIFFERENTIATED SERVICES (DIFFSERV)	12
CHAPTER 2 SIMULATION STUDIES	16
2.1 PREVIOUS WORK	16
2.2 OBJECTIVE	18
2.3 DESIGN RATIONALE	18
2.4 ORGANIZATION	19
2.4.1 <i>Trace Generator</i>	19
2.4.3 <i>Input Interface</i>	23
2.4.4 <i>Network Processing Component</i>	23
2.4.5 <i>Output Interface and Meter</i>	24
CHAPTER 3 IMPLEMENTATION OF COMNETSIM	25
3.1 INPUT INTERFACE	25
3.2 THE DMA COMPONENT	27
3.3 THE NETWORK PROCESSOR	28
3.3.1 <i>Software Design of Toaster Framework</i>	28
3.3.2 <i>Toaster Memory Hierarchy:</i>	32
3.3.3 <i>Memory Address Handling and Translation</i>	34
3.3.4 <i>Memory Request Handling</i>	34
3.4 OUTPUT INTERFACE	38
3.5 TOP LEVEL OF THE SIMULATOR	39
3.6 APPLICATION DEVELOPMENT	39
3.6.1 <i>Classifier</i>	41
3.6.2 <i>Conditioner</i>	41
3.6.3 <i>Scheduler</i>	43
CHAPTER 4 OPERATION, EXPERIMENTS AND RESULTS	49
4.1 SIMULATOR DIRECTORY STRUCTURE	49
4.3 SIMULATOR SETUP	51
4.2 EXPERIMENTS AND RESULTS	52
EXPERIMENT 1	52
EXPERIMENT 2	53
EXPERIMENT 3	55
EXPERIMENT 4	55
CHAPTER 5 SUMMARY AND FUTURE SCOPE	56

5.1 NEXT GENERATION OF COMNETSIM.....	56
5.1.1 DMA System	56
5.1.2 Scaling the System.....	56
5.2 METHODOLOGY	57
REFERENCES	60
APPENDIX.....	62
HEADER FILES	62
<i>Data Structures</i>	62
<i>Simulator Parameters</i>	68
<i>Toaster Parameters</i>	70

List of Tables

1.1 Trace Specification.....	21
2.1 Conditioner table element.....	43
4.1 Simulator Configurable Parameters.....	51
4.2 Queue performance metrics.....	53
4.3 Low rate queue test.....	54
4.4 High rate queue test.....	54

List of Figures

1.1 Simplifying the network.....	2
1.2 IBM Network Processor.....	5
1.3 Intel IX1200 Network Processor.....	7
1.4 Cisco Systems Toaster Network Processor.....	9
1.5 Diffserv architecture.....	14
2.1 Component Network Simulator.....	18
2.2 Packet trace.....	20
2.3 Stream library file.....	22
2.4 Input interface.....	23
3.1 Pipeline function snippet.....	29
3.2 Diffserv on Toaster.....	40
3.3 Two Rate Three Color Marker (TRTC).....	42
3.4 Weighted Round Robin (WRR).....	44
3.5 Weighted Round Robin control structures.....	47
5.1 Router structure	58

Chapter 1 Introduction

1.1 Network Processing

The Internet continues to grow, changing the way we live, work and play everyday in more ways than we could imagine possible. From its genesis as a small network used for exchanging data within the university systems [1], the Internet has grown into a massive worldwide network that links together a significant percentage of the world's populations.

The continued evolution of the Internet has posed great challenges to its designers. Even as the Internet grows wider and reaches more people, the pipes that carry the data within the network have scaled. Large amounts of data have to be processed at very high speeds to prevent the nodes from becoming bottlenecks. The development of sophisticated network hardware that operates at the nodes has kept up with this requirement through innovations in microarchitecture, hardware design and semiconductor fabrication processes. These devices are architected to be parallel and pipelined to maximize throughput necessary for network processing. They are referred to as being wirespeed capable and herald a general class of architectures for network processing.

With the maturing of the Internet, focus on providing advanced services has increased. Networks can no longer just provide a conduit for data, they need sufficient intelligence to provide guarantees to its users, on issues such as reliability, latency and sequential delivery. Reliability in a network is a measure of the assurance that a datagram will reach its destination. Latency is the delay such a datagram may encounter from its source to destination and a related property jitter, is a measure of its variation. Sequential delivery relates to receipt of datagrams in the order they were transmitted. These guarantees are a prerequisite for advanced applications to be practical; for example voice calls on the Internet would not become widespread unless the service

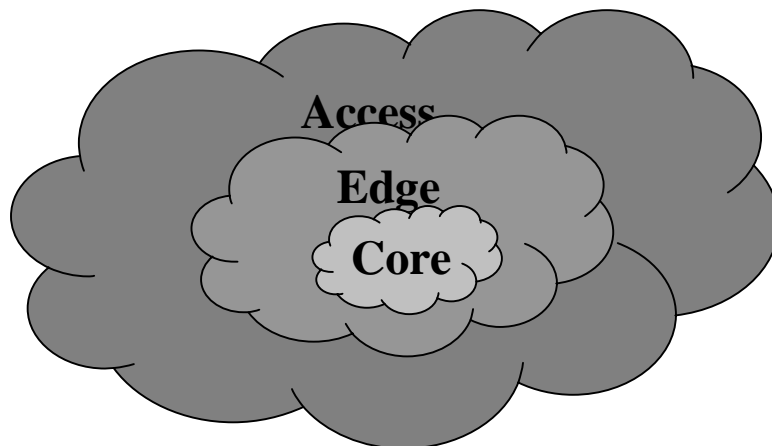


Figure 1 Simplifying the network

quality of such calls approached that provided by a traditional telephone network. Such service quality cannot be achieved without a network that is capable of providing the service guarantees cited. Quality of Service (QoS) summarizes the issues of reliability and delay guarantees that a network provides to its users.

Architectures for Network Processing and their use in solving QoS issues are the focus of this thesis. Network processors implement applications including those that provide QoS. Several frameworks have been proposed as initiatives for providing QoS in the Internet and the Differentiated Services (Diffserv) [2] is one such scheme. This thesis delves in to the concept and application of Network Processors. A simulator built for this purpose, ComNetSim, implements a set of Diffserv applications on a network processing framework. This simulator introduces a methodol-

ogy for study that explores the interaction between traffic vectors, applications and the platforms used for implementation.

The remainder of this thesis is organized as follows. The remainder of this chapter describes the concept of a network processor and a few representative architectures are summarized. The Diff-serv architecture is introduced and its relevance to this thesis is discussed. In Chapter 2, the motives for development of the simulator are discussed and in Chapter 3 the design and functionality of the simulator are presented. The use of the simulator in studying network hardware is demonstrated in Chapter 4 and results obtained are discussed. In Chapter 5, future scope and extensions to this work are proposed.

1.2 Network Processors

1.2.1 The Need for Network Processors

A simplified picture of the Internet divides the network into core, access and edge [figure 1.1]. At the core are high-speed pipes carrying large amounts of data with current generation equipment having processing speeds of OC192 (10 Gbps). The nodes linking up to form the core implement rudimentary services – routing, tag-switching and access control. The edge segment of the network forms the ingress and egress to the core. Services at the edge are complex and run at medium to high speeds. Services at this point include routing, switching, netflow, access control and QoS features. The access portion of the network covers all the delivery points of the Internet. The end user accesses the Internet through campus networks, broadband connections and dialup lines. In this portion of the Internet, there are several different protocols and base technologies interoperating with one another at relatively low speeds.

Given that the Internet is such a tangled web with its nodes having complex and extreme performance requirements, advanced hardware that fulfills these needs has forever been a neces-

sity. Conventionally, custom-designed hardware fulfilled this need, but this approach has its demerits, including a costly development process, long design pipelines and inflexibility once produced. Evolving services demand flexibility and scalability that only general purpose processors can offer. These off the shelf processors designed for an entirely different space are too generic to implement networking services at high speeds. Network Processors have emerged from this quagmire as a new category of programmable hardware devices that leverages the best of both of these approaches for implementing high performance networking services.

There are many approaches to the design of network processors, but the goals are very similar. The common aim is to provide a programmable platform with a feature set that is sensitive to network specific operations and has high memory bandwidth. These also try to provide a high degree of parallelism to increase processing throughput.

In the following sections, a few network-processing architectures are briefly surveyed. The processors discussed have several common features. The most significant, is the use of a multiprocessor framework. In addition, each of the processors has a modified Instruction Set Architecture (ISA) and implement atomic memory operations. While the IBM NP[3] and Intel IX1200[4] Processors leverage processor cores available off the shelf with custom designed processing elements, the Cisco Toaster NP[5] takes an entirely customized approach throughout. Between the IBM NP and Intel IX1200, the difference in their overall approach to the problem is significant. While the IBM NP provides hardwired queuing functions, protocol specific units, the IX1200 is more generic like the Toaster.

1.2.2 The IBM Network Processor

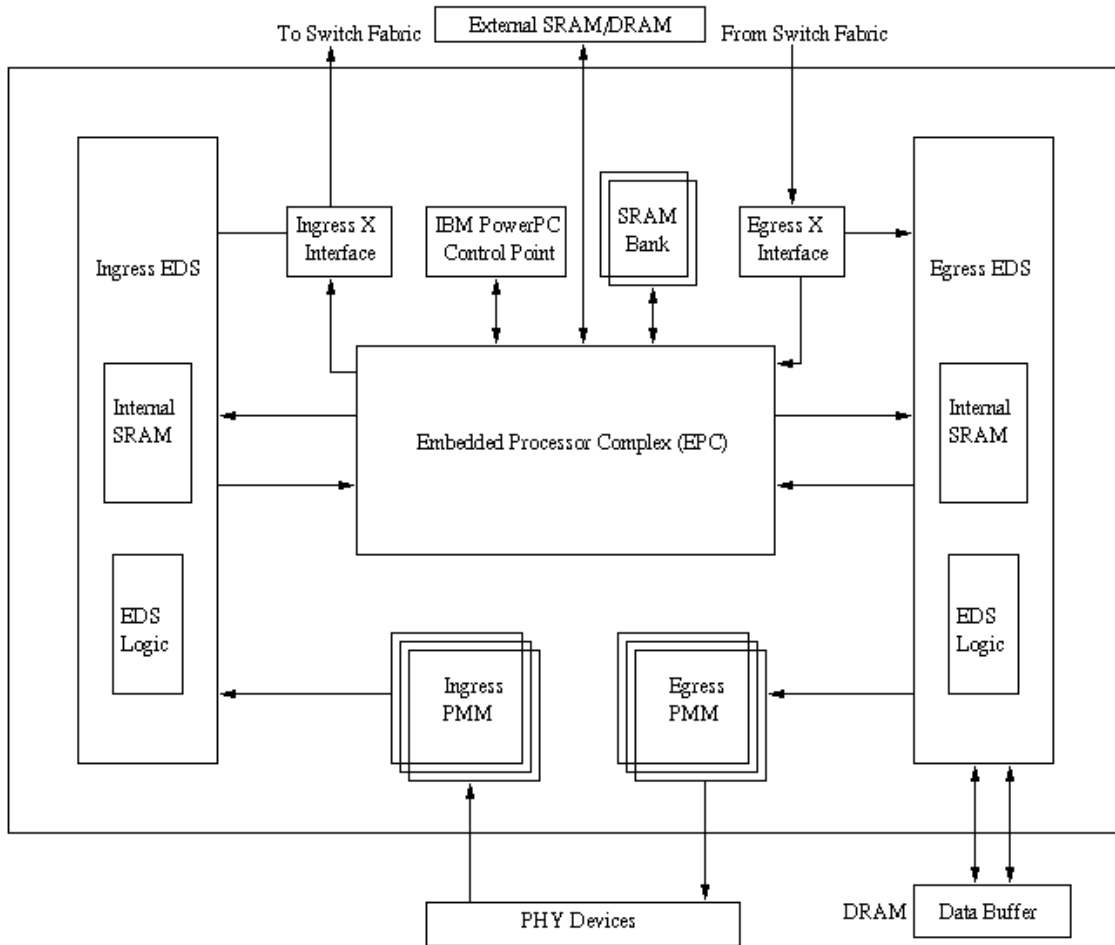


Figure 1.2 IBM Network Processor

The IBM Network Processor [figure 1.2] consists of 16 internal protocol processors managed by an IBM PowerPC core. It has two high speed Data Aligned Serial Links (DASL) that allows inter-processor communication in a multi-processor environment. The core of the NP is a complex of 16 Protocol Processors. The Protocol Processors are paired and each pair utilizes a set of seven assisting coprocessors. Each of the protocol processors makes use of a three stage fetch, decode and execute pipeline, general purpose registers, special purpose registers, eight entry 16 word instruction cache, dedicated ALU and coprocessors.

The coprocessors are special purpose units whose functionality is hardwired, unlike the programmable Protocol Processors. The data store coprocessor allows DMA functionality by interfacing between the Protocol Processors and the Enqueue/Dequeue/Scheduling (EDS) units on both sides. The checksum coprocessor is used to calculate checksums. The interface coprocessor interfaces between the internal registers, counters and memory for debugging and information gathering. Movement of data within the processor complex is made more efficient by the String Copy coprocessor. The Counter coprocessor manages counter updates. The Policy coprocessor implements policing and shaping mechanisms. The Enqueue coprocessor controls transfer of parsed frames to the EDS units. Hardware accelerators speed up operations such as tree searches, frame filtering, alteration and forwarding.

The Control Point (CP) for the system is a single embedded IBM PowerPC processor. The NP can also be configured and initialized by an external processor connected to one of the four Ethernet ports. In a scaled system, a single CP can be used to control a system of NPs. In addition, in a scaled system, communication between the processors occurs in the form of guided frames on the DASL. Guided frames are also used to communicate between the CP and the various processors in the subsystem.

The IBM NP provides a variety of internal and external options for buffering data and control information. The internal control memory consists of 1024x72 bit, 2048x128 bit and a 1024x64 bit RAM. Up to 18 External control memory modules of a maximum size of 256 Mb can be attached. Internal data buffers consist of 2048x64 bytes of space at the ingress and the egress. External data buffers can be connected to provide up to 512 Mb storage, in various configurations. These are DDR RAMs operating at 133 MHz. The instruction cache in the processor complex consists of eight 512x128 bit RAM banks that provide parallel access to the various processors.

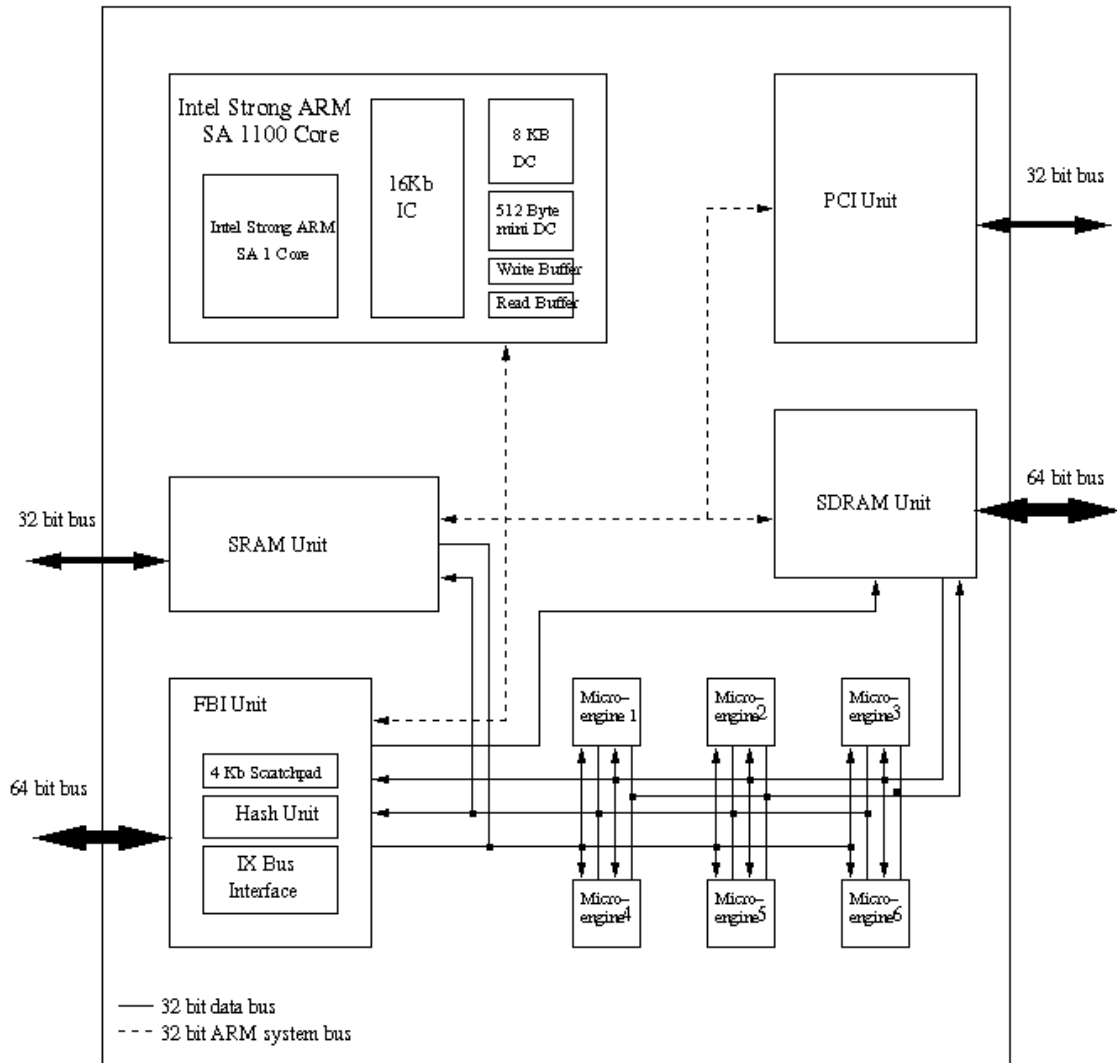


Figure 1.3 Intel IX1200 Network Processor

1.2.3 Intel Network Processor – Internet Exchange Architecture

The Intel IX1200 Network Processor [figure 1.3] combines an embedded StrongArm core with six custom 32 bit RISC microengines. While the 'microengines' perform tasks such as packet forwarding, the RISC core takes over management functions like configuration, maintenance and operation. The microengines operate in a multiprocessor, multithreaded environment, with a special instruction set that allows context switching of threads. Context switching increases the effi-

ciency of a microengine by using the latency of memory lookups for useful computation on a different thread. Every microengine has an ALU and shifter and can perform an operation on both within a single cycle.

The control point of the system is the Intel StrongARM SA 1100 with an Intel StrongARM 32 bit SA1 processing core operating at 166 Mhz. It has a data-oriented instruction set that allows byte-manipulation, data throughput and pattern matching.

Each microengine on the IX1200 has four independent instruction caches of four kilobits each. There are 128 32-bit general-purpose registers and 128 transfer registers on each microengine. SRAM and SDRAM operations are always interfaced using the transfer registers. The SA100 core has a 16 Kb instruction cache, eight kilobit data cache, memory management units, read and write buffers and a 512 byte mini data cache. The mini data cache can be used to improve cache performance while handling frequently used data structures. The core has an advanced memory controller operating at 100 MHz that supports a SDRAM and an SRAM bank using a 32-bit system bus. The SDRAM would be the packet store memory in the system and would provide low cost, high bandwidth access. A total address space of 256 Mb is possible with a peak throughput of 666 Mbps. The SDRAM unit can also be accessed using the 32-bit data bus from the microengines and a dedicated bus from the PCI unit. The SA1100 can access a byte, word or long word and perform read-modify-write operations. The SRAM unit provides a fast access memory for route lookups, filtering rules etc. The data is used by the microengines but is configured by the SA1100 core. The SRAM unit provides access to an eight megabit SRAM device, two megabits of Boot ROM for booting and a two megabit bank called the SlowPort region for access by peripheral devices by means of a 32-bit bus. The unit provides a read lock feature using an eight entry CAM. There is also an eight-entry push/pop register list for fast queue operations and bit test, set and clear instructions for atomic operations.

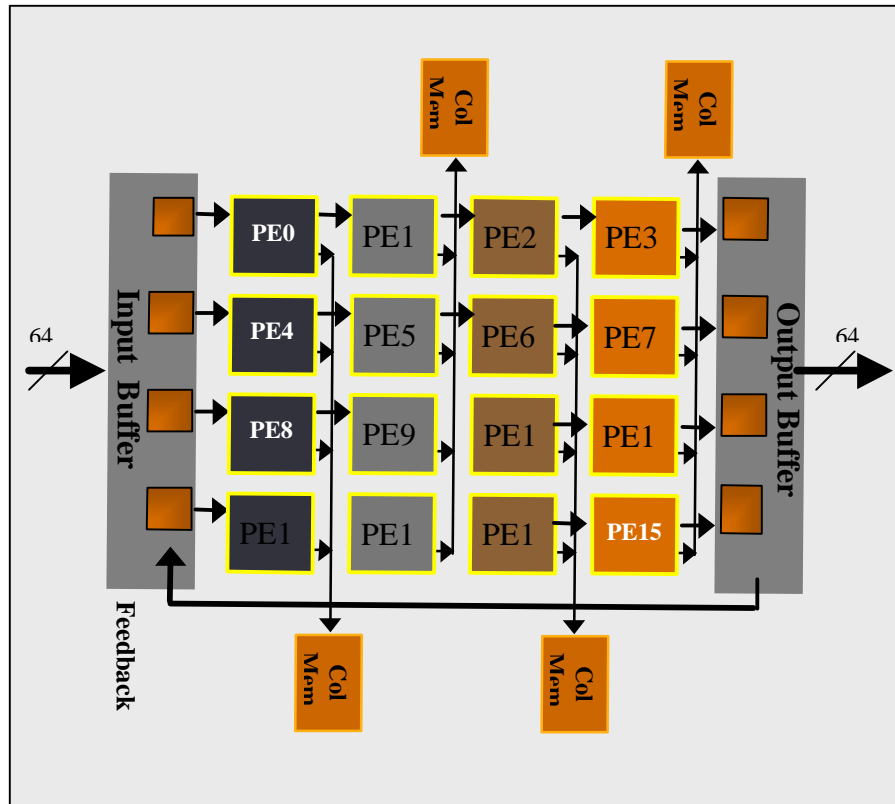


Figure 1.4 Cisco Systems Toaster Network Processor

1.2.4 Cisco Systems Toaster Network Processor

The Toaster Network Processor [figure 1.4] developed at Cisco Systems® Inc., is an example of a parallel, pipelined multiprocessor system. The processor is a 4x4 matrix of custom designed Processing Elements (PE). The PEs are laid out to form four similar parallel data flow pipelines. Each pipeline has four stages and corresponds to a row in the PE matrix. Buffers on either side of the pipeline provide a high bandwidth exterior interface. The PEs in each column share a hierarchical memory system.

A Toaster processor works in conjunction with a Direct Memory Access (DMA) device. Packets are received at the DMA device and are translated into contexts, the Toaster unit of data. A context is of 128 bytes length and may contain the packet header and additional fields for control information transfer. The portion of the packet that is not a part of the context is buffered at the DMA device while the context is processed. Once generated, contexts enter the Toaster processor and are queued in a buffer on the entry side of the data flow pipelines. At periodic intervals, each pipeline accepts a data unit for processing. As a context traverses the data flow pipeline, it is operated on by every PE in its path. In order to provide uniform processing each data flow pipeline implements the same set of applications. To illustrate, the code corresponding to an application may be divided in to four sequential segments with each part residing on a Toaster PE in its relative position. The code is reproduced on each of the pipelines and the treatment of a context is independent of the pipeline that it enters. Contexts leave the Toaster processor through an independent high speed interface and re-enter the DMA device. Processed contexts are stripped off the control segment to leave just the packet headers that are reattached to their respective payloads.

The time allowed to each PE is constrained and is called a phase. The length of a phase is programmable, but is applied uniformly to all the PEs on the processor. In order to reduce contention for memory amongst the PEs in a column, the code being executed on each of them is skewed by an amount called a phaselet. The minimum length of the phaselet is guided by the number of cycles taken to transfer a context from an upstream stage to a downstream stage. Toaster provides a feedback path from the Output Buffer to the Input Buffer. This path can be used to pass packets through the pipeline for additional processing.

The Toaster PEs are Very Long Instruction Word (VLIW) cores with independent cache, 12 KB Instruction RAM (IRAM), and two four-stage instruction pipelines. The Instruction Set Architecture (ISA) is native to Toaster and is optimized for network processing. This includes fast lookups,

atomic memory operations, preset masks and bit level operations. A Toaster PE's instruction is 64 bits in length and is segmented, with each part being tied to a pipeline.

The memory system in Toaster is a multi-level hierarchy with data memory devices being shared by each column. Local memory on each core is in the form of a register file, 64-byte data cache and the space for context storage. Large data structures are stored in the column memory devices that are present on and off the chip. Typically, on chip memory is a 16KB SRAM device and the off chip memory is a 256Mb SDRAM device. A memory controller for each of these devices manages accesses. The memory controller for the SDRAM device is partitioned in to two parts, each of which can support up to eight banks.

Memory accesses take the form of requests that travel from each core to independent FIFO queues in the memory controllers. Requests are processed based on bus and device availability, deadlocks being broken based on the row number of the originating core. Data is returned to the originating cores on 32-bit result buses available to each memory controller. The two external memory controller splits the result bus equally between its two sections.

There are three basic memory operations – Read, Write and Prefetch. The selection of the memory device is determined by the address. Memory requests operate on 32 or 64 bits of data. The prefetch operation is like a load in the conventional sense. It is used to fetch data from the column memory and place it in the local cache, helping to hide the long latency of a load. Read operations never spawn a request to memory as long as data is available in the local cache. Given the constraints imposed by the skewed execution of code, the best processing throughput is achieved by deterministically executing memory operations. They are hence laid out in application code, so that there is minimal interference between the requests from the cores in each column. A lock controller is available that can be used to protect data which is in use and avoid its corruption.

The control point for the Toaster NP is located off chip and could be any off the shelf microprocessor. It is connected to Toaster on a dedicated interface. The Toaster NP operates with a 125 Mhz clock. A phase length of 64 cycles would allow a context to enter the pipeline ever 16 cycles (phaselet). This translates to a context throughput of 7.8 million per second.

1.2.4 Summary

Each of the network processors briefly discussed show a lot of common ground, yet are unique. The IBM NP and the Intel IX1200 combine packet processing and storage tasks while the Cisco Toaster offloads the packet storage to an external device. The usable memory space is similar for all the processors, and so is the availability of control memory. The programmable cores in the IX1200 and the Toaster use independent instruction caches while the IBM NP uses a shared instruction cache. In terms of the ISA, all the processors allow bit level operations, atomic operations, fast lookups and easy mask and shift operations. The Intel Processor implements context switches to hide long latency operations. The availability of the prefetch instruction on Toaster is for the same purpose. A difference in the approach to configuring and maintaining the processor is apparent. Cisco's Toaster alone does not have an on chip controller. It instead uses a dedicated interface to an external processor. All three of the processors provide high-speed interfaces that can be used to connect similar devices to build scaled systems.

1.3 Differentiated Services (Diffserv)

Differentiated Services is a framework for providing QoS guarantees in a network [2]. The basic tenet of Diffserv (DS) is to treat each packet in an aggregate stream according to an agreed level of service. In a system implementing Diffserv, packets from aggregate traffic are classified and matched with a flow. They are then treated according to this classification in the rest of the system. Routers that implement the Diffserv system complying with the associated IETF RFCs are referred to as Diffserv compliant routers.

When architected as an application on a router, Diffserv can be viewed as a set of components [figure 1.5] as follows [3]:

1. Classifier: A packet entering a system needs to be classified to determine the policy used to service it. The classification can be based on two different techniques. A Behavioral Aggregate (BA) classifier uses the Diffserv field [4] alone to establish the classification. The Diffserv field is defined as the Type of Service (TOS) byte of an Internet Protocol version-4 packet. It has two segments – one of which is currently unused. The useful segment of the field is of six bits in length and starts from bit seven of the TOS byte. This is called the Diffserv Code Point (DSCP). A multi-field (MF) classifier [2] uses a number of different characteristics of a packet to determine the flow to which the packet belongs. This can include the source address, destination address, DS field, protocol identifier, source port and destination port. A large number of fields allow greater flexibility in configuration and provision of service to a customer. To illustrate, the packets originating or terminating at a particular IP address and carrying data associated with a particular protocol can be treated according to a specific policy.

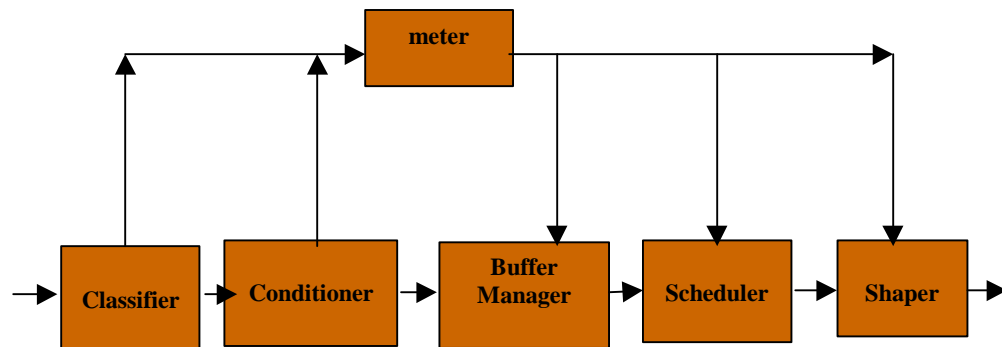


Figure 1.5 Diffserv Architecture

2. **Conditioner:** A conditioner is a component that is used to measure the adherence of a flow to its configured service rates and mark the packets belonging to it, appropriately. Conditioners are inherently based on the leaky bucket algorithm [5]. The conditioners defined in Diffserv literature are the Single Rate Two-Color Marker (SRTCM) [6] and the Two-Rate Three-Color Marker (TRTCM) [7]. A conditioner can take the form of a marker, dropper or shaper [3]. A marker merely marks a packet based on adherence while a dropper discards a packet based on the same property. The shaper delays the passage of a packet through the system to compel the flow to which it belongs to adhere to the configured rate.

3. **Scheduler:** A scheduler is an algorithm that is used to distribute the available bandwidth amongst all the flows configured on a router. In a Diffserv system, the packets may be stored in individual queues based on their classification. The scheduling algorithm services the packets from each queue in a certain service order allowing the packets to exit the system. A scheduler can be categorized as work-conserving or non-work-conserving. A work-

conserving scheduler services queues as long as they are non-empty, while a non-work-conserving scheduler services queues only when the algorithm dictates.

4. Buffer Management: A router has finite amount of memory space to store packets before forwarding them. This space can be actively managed and prevented from filling up completely using a buffer management algorithm [8]. A Tail-drop algorithm drops packets once the space reserved for the queue to which the packet belongs is exhausted. A more active algorithm is the Random Early Detection (RED) and its several variants.
5. Metering: Diffserv systems are managed by measuring and responding to statistics collected within the system. A Meter is a generic component that performs this function. The various statistics gathered might include buffer occupancy, number of dropped packets, system throughput, latency and jitter metrics.

The Diffserv framework provides a highly configurable system to establish QoS guarantees when extended on network wide basis. The scope of this thesis precludes the description of the entire Diffserv architecture and its network wide implications. The goal instead is to concentrate on the implementation a few Diffserv components on a network processor simulator.

Chapter 2 Simulation Studies

2.1 Previous Work

With the advancement of network equipment, system-level modeling to explore design space and verify concepts has gained importance. Past research has covered different aspects of communication system design such as algorithms, protocols and benchmark applications.

A methodology for studying performance of network hardware is Commbench [12]. Its purpose is to develop benchmark applications that can be used in the design and evaluation of Network Processors. Commbench's focus is on implementing networking applications that, provide a distribution of various instructions and their frequency. These can have a role in designing the ISA of a network processor. Independent of any particular architecture, it resembles the microarchitect's approach of design and evaluation using a standard set of benchmark applications.

The work of Patrick Crowley, et al [13] has been to combine a system of benchmarks with micro-processor simulators. This has the twin benefit of arriving at a system for evaluating network processors and using these to explore the network processor design space. The initial experiments in this body of work evaluated the benchmarks on several conventional microarchitecture paradigms.

A simulator that has a very broad development community is the Network Simulator (NS)[14]. This system is used to study behavior of algorithms and protocols at a network level with interaction between different nodes. NS incorporates code developed by its several users and contains a rich set of libraries for traffic generation and algorithm studies. While the researchers working with NS study various issues related to the operation of the Internet, the simulator is not built to address hardware architecture issues.

The PAcKet Lookup And Classification (PALAC) [15] simulator developed at Stanford University provides a framework for studying behavior of packet classifying algorithms. The system includes a traffic generator, packet classifier algorithms and statistics collection mechanisms. The goal of this simulator seems to have been to design and implement efficient packet classification algorithms.

The design space explored by methods discussed so far can be organized in to three segments – applications, hardware/software architecture and traffic generation. It is apparent that none of the above methods considers combining all three of these segments to study their interaction. The conventional microarchitect's approach has been to study hardware architectures using benchmarks. This technique neglects the role that network traffic can play in shaping the performance of a particular architecture. The approach of researchers in the networking field has been to simulate a network and study the interaction between nodes implementing protocols and algorithms and passing simulated traffic through them.

There is a space that remains to be explored that combines the three different segments. This space, which would bridge the gap between design, implementation and deployment from a network systems perspective, should address the interaction between traffic vectors, applications and the platforms used to implement these applications. The purpose of this thesis is to address this space and implement a methodology that demonstrates the benefit of such a multi-pronged initiative. The Component Network Simulator (ComNetSim) [figure 2.1] has been borne of this effort.

The following sections describe the organization of the simulator and the rationale on which the design of its components is based.

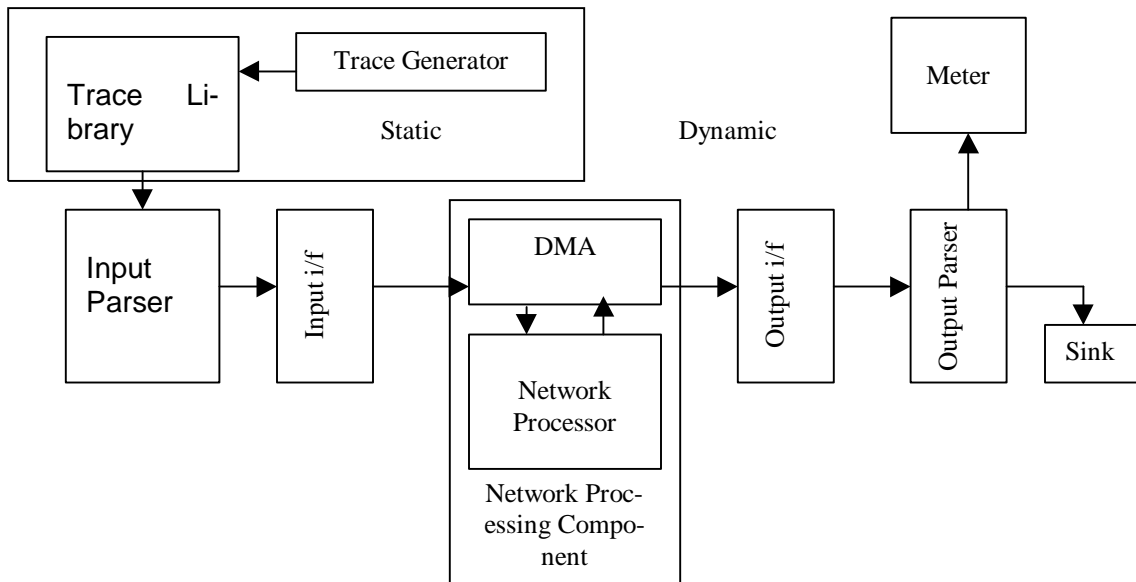


Figure 2.1 Component Network Simulator

2.2 Objective

The goal of the simulator is to model the behavior of network processing components and facilitate study of their interaction with network traffic traces. This is achieved by using a library of traffic traces as input vectors to a time driven simulator. Metrics derived from the operation of the simulator can be used to study the working of the various components.

2.3 Design Rationale

The simulator is modeled from the perspective of a line card on a router. The entire simulator can be construed as a simplex path on a line card – with input interfaces or ports on which packets

are received and buffered. The packets are then multiplexed to form input queue to the Network Processing Component (NPC). The DMA ASIC handles the packet buffering function. A network processor processes the packet headers and returns them to the DMA ASIC. The headers are reattached to their respective payloads and stored in the packet memory. Packets are allowed to exit the packet memory at an appropriate time, determined by the NP. In the case of the simulator, the packets are sinked once they exit the NPC, but in a realistic scenario, these would be bound for an output port or a switch fabric.

Each of the components in the packet flow pipeline is configurable and allows scope for study. The NPC is a model of a real network processor complete with its memory system that permits implementation of applications. Thus the design goal of the simulator is achieved wherein there is scope for study of both the individual and combined aspects of traffic, algorithms and hardware architecture.

2.4 Organization

At the top level the simulator can be divided into two types of components – static and dynamic. The static portion of the simulator comprises the traffic generator and the trace library. The Trace generator uses at its core a self-similar [16] traffic generator [17] developed at University of California, Davis. Traces are generated and stored as a library. The files in the trace library form the input to the dynamic segment that comprises the network processing components.

2.4.1 Trace Generator

A critical requirement for network simulation is the availability of realistic traffic traces. Network traffic traces can be obtained by several methods. A popular scheme is to collect real traces from routers for extended periods of time [18]. These traces represent the mix of traffic flowing through a router and are collected on one of the input or output links. While providing a picture of aggregate traffic, these traces may not be flexible for simulation studies. They contain a mix of traffic

from different sources that is very generic and have a preset arrival pattern that cannot be changed. These traces can instead be used to derive characteristics such as packet sizes and packet types to produce packet traces. These traces can be generated based on statistical models that can be parameterized for rate, type, length, average packet size etc.

The prevalent idea in the networking research community is that self-similar processes best model network traffic [19]. The basis for this theory is that network traffic has been observed to be

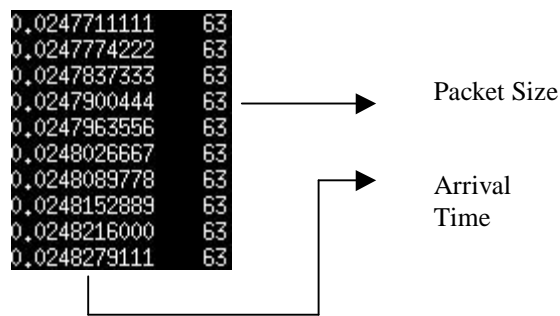


Figure 2.2 Packet Trace

bursty across different timescales. The trace generator uses a Self-Similar Traffic Generator (SSTG) available in the public domain [17]. The traffic generator can generate traffic trace files that take the form shown below [figure 2.2]. The fields correspond to packet arrival time and packet size. The parameters for the traffic generator are – rate (Megabits per second), minimum packet size (bytes), maximum packet size (bytes), number of packets and number of sources.

The Trace Generator parses the packet traces produced by the SSTG and generates formatted packet traces [table 1.1].

Type of trace	Input I/f	Output I/f	Packet Arrival time	Size	TOS	Protocol
SSTG	3	3	0.0536451556	149	240	6

Table 1.1 : Trace Specification

The traces are stored in the **Trace Library** along with a descriptor file. The descriptor file (strlib.dat) [figure 2.3] contains information about each trace in the library. Each line in the descriptor file corresponds to a trace. The characteristic information consists of the following fields: trace number, rate (Mbps), average packet size, number of packets in trace, output interface number for packets in the trace, Type of Service (TOS) byte of packets in the trace and trace file-name.

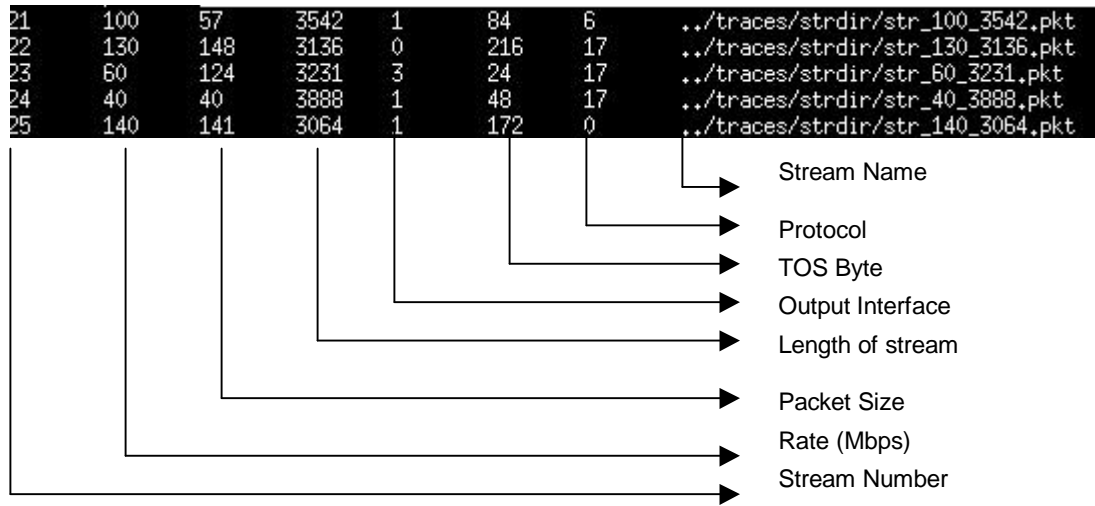


Figure 2.3 Stream Library File

2.4.2 Input Parser

The Input Parser [figure 2.4] selects the traces to read and present as input to the Input Interface unit. Entries are read from the trace file and are buffered. Each entry or line in the trace file corresponds to a packet. The arrival of each packet is simulated at the Input Interface based on the arrival time recorded in the trace file. The Agent makes the association of streams with the input interface. This association is based on the configuration of the interface and the trace data file [figure 2.3]. The Stream Library comprises all the files containing packet traces. The File Reader performs the function of reading packets from the Stream Library. It determines if the arrival time of a packet has crossed the simulator's real time. A packet whose arrival time has passed the simulator's real time is passed to the Queue Writer. The Queue Writer places the packet it receives in one of the input queues (Q1 to Q4).

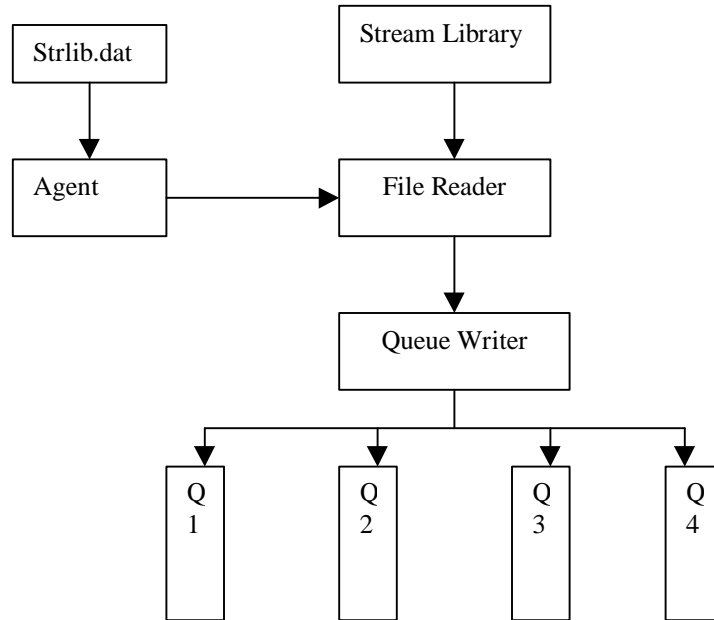


Figure 3 Input Interface

2.4.3 Input Interface

The input interface performs the function of multiplexing the traffic from the different input queues. It receives traffic from the Input Parser and forms an aggregate queue. The aggregate queue is the feed for the Network Processing Component.

2.4.4 Network Processing Component

The Network Processing Component (NPC) has two parts – the DMA component and the Network Processor.

The DMA component receives packets from the input interface and holds them in a temporary memory. The packet headers are separated from the packet body and directed to a network processor. The packet headers when returned are attached to their respective payloads and are queued in an external memory unit until they are ready to exit.

The Network Processing Component (NPC) receives packet headers from the DMA component and buffers them for processing. The packet headers are processed based on the applications configured on the NPC. On completion, they are returned to the DMA component.

The NP in the simulator is modeled based on the Toaster Network Processor [5] architecture. The presence of an external DMA component is peculiar to the choice of NP. The Intel or IBM NPs would not need an additional DMA component. Diffserv components are implemented as applications on Toaster. They are constrained by hardware parameters such as cycle times, memory availability and memory latencies. Theoretically, any NP can be modeled within the simulator, but Toaster offers itself as a flexible, robust and independent platform that has been voted best in its class. In addition, Toaster's shared memory system offers an interesting opportunity for study.

2.4.5 Output Interface and Meter

The Output Interface models the reverse functionality of the Input Interface. Packets are received from the DMA and are parsed to measure characteristics such as latency and jitter. The packets are then sent to the sink where they are discarded.

Chapter 3 Implementation of ComNetSim

The design and implementation of the simulator has been with a focus on modularity and scalability. Hence, the simulator is partitioned into separate components by way of an object-oriented approach such that they can be modified individually. The source code for the simulator comprises C++, PERL and shell scripts. The simulator has been compiled on SunOS 5.6 using gcc version 2.9.

The previous chapter discussed the organization of the simulator. The following sections elaborate on the implementation details. In this chapter, the components that are functionally inter-linked are combined. The Input Parser and Input Interface are combined to form a single unit that is called the Input Interface. The output interface has the sink and meter built in. The NPC is treated as two separate components – DMA and NP. Each component is sufficiently complex to be treated as a separate structure within the simulator. Hence the dynamic portion of the simulator is reduced to four components – the input interface, DMA, network processor and output interface.

3.1 Input Interface

The input interface brings to the simulator the functionality of the link buffers. At startup, the physical interface's data processing rates and the size of the various link buffers are configured. The data processing rates correspond to the speeds of the incoming links. Based on this configuration the trace data file is parsed to associate different trace files with each of the physical interfaces. The number of traces associated with each interface depends on the sum of the rates of all the traces associated with the interface. The design of the simulator is such that a physical interface at any instant during the simulation has enough trace files associated with itself to fulfill its configured rate. As packets in the trace files are exhausted, new files are added to maintain the

configured rate. This functionality is achieved by maintaining a linked list of streams for each physical interface. The *stream_element* structure is used for this purpose.

Whenever the aggregate rate of the physical interface drops below configured rate the *file_agent()* function is called to make new trace file associations. The structure *link_desc* used for this purpose corresponds to each interface and has information on the buffer and statistics for the traffic passed through it.

The *get_packet()* function is used to simulate the packet arrival process at the ports. Within this function, the trace file corresponding to each port is parsed to find all eligible packets. Pointers to the trace files corresponding to ports are available from the stream-linked list. Eligible packets are those packets with arrival time that is past the simulator's system time. The arrival time for a packet is computed as sum of the arrival time obtained from the trace file and the start time for the trace. The start time for the trace is one of the control variables recorded within the stream element list. This correction in arrival time is necessary, as trace files are liable to be reused. Once an eligible packet is found, a *link_element* is created and its fields set based on the trace file data. The *link_element* corresponds to a packet and is added to the appropriate port buffer queue.

Once the packets enter the individual port buffers, they have to be funneled into a single link buffer that is hooked up to the DMA component. A simple Round Robin (RR) approach is inappropriate for ports operating at different speeds. Such an approach does not allow more packets from the higher speed link to be delivered. Applying Weighted Round Robin (WRR) solves this problem. The weights for each of the port buffer queues are set in proportion to their rates.

Bus constraints are in operation throughout the simulator and this applies to the input interface. When packets are funneled into the multiplexed link queue, the number of bytes transferred per cycle is limited by bus width parameters.

3.2 The DMA Component

The DMA interface rids the network processor of the complex task of memory management. The DMA in the case of the simulator resembles a traffic light where data is directed from one section to another. It is the composite of many different queues with associated functions operating on these to effect movement of data.

The Link Input Queue (LIQ) is the buffer for packets entering the DMA system. Packets are queued here until they can be transferred to the Internal Packet Memory (IPM). As packets enter the IPM, their header is stripped off and used to form a Toaster context. The toaster context consisting of a control and data segment. The control segment contains fields that are used for communicating information between the DMA and Toaster. The data segment contains the entire packet header. The structure used is the *t2_context*.

Every packet entering the DMA system causes a Toaster context to be generated. These contexts are queued in the To Toaster Context (TTC) buffer, until they are ready to be transferred to the Toaster NP's input buffer. Since packet payloads are not handled by the simulator, the IPM and TTC queue are abstracted to form a composite queue that is referred to as the IPM.

Contexts are returned from the Toaster NP after processing. The control segments contain information that is used for enqueue and dequeue purposes. The control segment contains the queue number of the packet in the current context. Based on this information, that packet is stored in the eXternal Packet Memory (XPM). In a conventional line card, the packet header would be attached with its payload residing in the IPM before being enqueued. The control segment also contains the queue number that has to receive outbound service. A packet is read from the corresponding XPM queue and placed in the Fabric Output Queue (FOQ).

Bus constraints are applicable between the DMA and the simulator components to which it interfaces. This limits the number of bytes that can be transmitted per clock cycle. The limits are configurable from a common parameter file.

3.3 The Network Processor

The Network Processor is the heart of the simulator and is modeled based on the Cisco Toaster Network Processor. A brief introduction to Toaster was provided in the Section 1.2.4. This section provides the details of implementation. The basic structures within the Toaster framework are the input buffer, Core matrix and the output buffer. The input buffer collects Toaster contexts from the DMA. The queued contexts are directed to one of the free Toaster rows when it becomes available. As described in Chapter1 the rows form a context pipeline and the columns share control memory amongst them. The output buffer collects contexts that exit from any of the rows. The queued contexts are sent to the DMA component for queuing.

3.3.1 Software Design of Toaster Framework

Generic buffer:

The functionality of the input and output buffers are similar to one another. A single buffer class is used of which the Input and output buffer are instantiations. The buffer is primarily a queue structure. It has no intelligence built in. Contexts are added or deleted based on external commands.

Toaster Core:

The Toaster core is specified as an abstract base class with a virtual pipeline() [figure 4.1] function. The specification of the pipeline() function within a derived class determines the application running on top of the core. If the pipeline() function is not defined, a context will pass through the core unaffected. This function is defined to simulate cycle accurate behavior of the Toaster core.

```
Pipeline( cycle, context) {
Switch(cycle)
.....
case 20: c_dif = c_tokens - pkt_size;
        break
case 21: p_dif = p_tokens - pkt_size;
        break;
case 22: if (p_dif < 0)
        packet_color = RED;
        break;
.....
}
```

Figure 4.1 Pipeline function snippet

On a real system, each core in the Toaster framework executes applications in the form of microcode. The applications are time constrained and consume cycles for each instruction that is executed. In order to incorporate this in the simulator, the pipeline() function is made to resemble a sequence of instructions, each of which is executed in finite time. This is accomplished by organizing the source code within a case statement whose switch parameter is a cycle number. As a core steps through the same code in every phase, the cycle number corresponds to the temporal position within a phase. The code selected for execution is based only on the cycle number passed to the pipeline() function as a parameter. A stall results when the code corresponding to a particular cycle fails to complete. This is accounted for by failing to update the cycle number for

that core. Doing so results in code belonging to the stall cycle being re-executed until it becomes successful.

The Toaster core class contains several control variables and the local cache. In addition, several functions operate as part of the toaster memory hierarchy. A virtual function is included that can be used to perform data checks.

Toaster Column:

The major components within this structure are the Toaster cores and the column level memory system. The toaster cores are objects of the base abstract class (tmc). The number of cores in a column is a configurable parameter. The column memory system consists of the internal and external memory structures. Many functions perform the role of processing memory requests and play a support role in the memory state machine. The memory system is discussed later in this chapter. Other simulator related functions perform time updates, print statistics and make function calls to the Toaster cores.

The Toaster column structure also contains a lock mechanism for use by the cores. There are four column-level lock elements. Each element has two fields – lock identifier and lock busy fields. A Toaster core can request a lock for a specific identifier. This identifier could be a memory address or any value computed, such as a queue number. A lock is granted if the specified identifier is not already locked by another core and the lock tied to the requesting core is free.

Toaster:

This is the top level Toaster structure. The Toaster columns, the input and output buffers are instantiated as objects in this structure. The configuration of the applications being executed on the framework takes place at this level. As mentioned previously each application class is derived from the Toaster core class. The specification of the pipeline() function within the derived class determines the functionality of the core e.g. a scheduling application would be coded within the

pipeline virtual function of a derived class called scheduler. Objects of this class would be instantiated within the Toaster structure. A pointer to each such application object would be instantiated and passed as a parameter to the column of that type. Pointers to base and derived class objects are type compatible. Hence, the pointers to the generic core declared at the column level can be associated with the derived class cores instantiated at the framework level. The advantage of this approach is that the base functionality of the Toaster core and column can be made generic allowing for reduced code size, greater flexibility and modularity. The Toaster structure contains functions that configure the internal and external memories for each column (`columnX_memory_init()`).

Function calls are made to the column level execute functions (`exec_column()`) every cycle. The parameters for this call include two important array structures:

1. Cycle map: This is a two dimensional array that contains the current cycle number for each of the cores in the Toaster framework. The cycle map overlaps the Toaster framework. Within the simulator, only a segment of the cycle map is passed to each column. This segment is a single dimensional structure that contains the current cycle number for each of the cores in that column.
2. Context map: This is a two dimensional array of pointers to Toaster contexts. This structure, like the cycle map discussed above, overlaps the Toaster framework. By maintaining a context map, movement of contexts between cores is facilitated, as a "context switch" is merely a transitioning of pointer variables. This is accomplished at the Toaster framework level and the cores do not have any role to play except for informing the top level when it is finished with a context. Information flow for this purpose begins at the core as a flag (`pdone` – processor done) that is set at the end of a phase. At the end of each cycle, the flag within each core is sampled. If the logical AND of all the `pdone` flags in a row is detected to be TRUE, a context

switch is carried out, although an extra condition applies. This condition accounts for the phase difference that needs to be maintained between the processor rows.

3.3.2 Toaster Memory Hierarchy:

The Toaster memory hierarchy is composed of three sections:

1. **Tag Buffer:** The tag buffer is local to the Toaster cores and is akin to a software managed register file with eight 64-bit wide entries. The tag buffer is the only memory unit to which the core has direct access. All memory requests pass through the tag buffer. Memory reads to the tag buffer are zero-latency operations and are similar to reading local registers. Write and prefetch operations cause requests to be spawned to the column-level memory structures. The tag buffer is maintained as a structure within the Toaster core class (tmc). The tag buffer element has the following fields:
 - a. **Address:** Memory address of the data stored in the tag element.
 - b. **Valid bits 1 & 2:** Indicates validity of the two 32-bit data segments.
 - c. **Pending:** Indicates if the tag element is waiting on a request that has been spawned to column level memory.
 - d. **Pending timer:** A simulator variable to compute latencies.
 - e. **Data 1 & 2:** Data is stored as two 32-bit elements.
2. **Internal Column Memory (ICM):** The ICM is a column level memory structure and is physically located on chip as an SRAM device. The ICM size is limited to 16kB and is organized as 32-bit elements. Memory requests are received from each core in a column and serviced in the order of arrival. The ICM exists as a column-level structure in the t_column class.
3. **eXternal Column Memory(XCM):** The XCM is also a column-level memory but is located off chip as a SDRAM device. The XCM size is upwards of 256 Mb and is composed of eight

banks, each line being 16 bits in length. A set of four banks shares a 32-bit request bus and all eight banks share a 32-bit result bus. There are two possible addressing schemes for the banks – block and stream modes. In block mode, the address map is split into eight segments and each segment corresponds to a bank. In stream mode, addresses stride banks continuously. While stream mode is preferred for storing and reading datagram structures, the bank mode would be preferred when operated as a control memory. Memory requests to the XCM experience a higher latency. The order of service of the requests is the same as in the case of the ICM. The XCM exists as a column level structure in the `t_column` class.

With respect to the simulator, memory operations start out as requests within the application code. Requests are for data of four or eight bytes in length. Requests are of three different types:

1. **READ:** This is a request to read data from memory. The request can be for either 32 bits or 64 bits of data. The request contains the tag buffer element to be used and the location of the data in memory. Data is read from the tag buffer from the specified memory element if the element is marked as valid and the pending flag is not set. Thus, the management of data is up to the application code. If a read request is not fulfilled, the processor stalls until the situation is remedied. A read operation in the simulator does not spawn a request to main memory under any circumstance.
2. **PREFETCH:** A prefetch request fetches data from column memory and writes the data to the specified tag buffer element. These are non-speculative requests for data and can be viewed as a **LOAD** on a microprocessor. A prefetch request is spawned as a memory request to the column memory. The traversal of the request within the Toaster framework is modeled as a state machine. This is described in the next section.
3. **WRITE:** A write operation is spawned as a request to column memory. The request contains new data, the target address and data width. A request has to include a reference to a tag

buffer element. It is accepted only if the tag buffer element is not already in a pending state. WRITE requests take the form of messages that traverse a state machine like in the case of a PREFETCH. WRITE and PREFETCH requests have access to separate buses to the column-level memory controllers.

3.3.3 Memory Address Handling and Translation

Requests to memory are implemented by macro calls that translate a control variable's relative location to a physical memory address. Doing so hides the physical organization details of the memory from the application. The downside of providing this translation is that a separate macro is needed for each control variable structure that is used. The input parameters for such macros are predefined variable identifiers and the sequential reference for the structure. To illustrate, if the application were using a queue structure holding the rate and priority, the predefined identifier would be Q_STRUCT and the sequential reference would be the queue number. The Q_STRUCT identifier translates to a base address for the structure e.g. 0x01800000. The macro computes the real physical address as:

$$\text{Address} = \text{Q_STRUCT} + (\text{size_of_element} \times \text{q_number})$$

The physical memory address is thus available to the memory state machine to make decisions on the location of the data – XCM, ICM and bank locations.

3.3.4 Memory Request Handling

The software design of the Toaster framework realizes its shared memory design. As described earlier, memory operations result in requests being spawned to the column level memory structures. By handling all the memory requests outside the cores from which they originate, they can

be pooled and serviced at the column level. This brings to the fore the shared memory characteristic.

In a physical system, a memory request would pass between the originator and the memory in either or both directions depending on its type. During this passage, the request travels on several buses from one location to another and is decoded and serviced by a sequence of state machines. The software design models both of these aspects. Each memory request transitions through a sequence of states until its objective is met. These transitions are guided by:

1. Type and length: The request can be a PREFETCH or a WRITE operation. The latency of the operation is also affected by the length of the request – 32/64 bits.
2. Tag Buffer Availability: All requests have to target a tag buffer element. They are accepted by the tag buffer only if the target element is not already in a pending state. A pending state indicates that a previous request using the same element is incomplete.
3. Memory Location: The request may be to the internal or external column memories. The latency of the request and the operations that need to be performed vary. The internal memory is SRAM and external memory is SDRAM.
4. Bank Availability: If the request is to the external memory, the target bank needs to accept the request.
5. Bus Availability: As requests have to pass from one device to another, they compete for usage of the buses.
6. Resource prioritization: Requests are prioritized based on the originating core and on a FCFS basis.

Requests to memory are made using explicit function calls `read()`, `write()` and `prefetch()`. The write and prefetch requests cause a memory request element to be generated (`request_gen()`). The fields of the memory request structure are set based on the parameters of the function call. The structure is then added to a request queue at the core.

The following are the states that a memory request encounters:

1. **ISSUE:** This state corresponds to a request that has been issued and is awaiting transmission to the tag buffer.
2. **LOCAL_BUS1:** On a physical device, once an instruction references a memory location, a request is issued that traverses the local bus to the tag buffer. The write and prefetch requests have a dedicated bus on the Toaster core and requests travel on this bus. In the case of the simulator, the memory request is placed in this state before being added to the request queue.
3. **LOCAL_BUS2:** This state corresponds to the second cycle spent by the write or prefetch request on the channel to the tag buffer.
4. **TAG_BUFFER:** This state corresponds to a request that has been accepted by the tag buffer and is being communicated to the column memory controllers. The request is directed to the XCRAM or ICRAM controller based on the target address. Within the simulator, once a request transitions to this state it is popped from the core level queue and added to the ICM composite column level queue or the XCM composite column level queue depending upon the target device. Requests are popped by calling the `req_to_column()` function.

5. XCRAM_REQ: A request enters this state if it has been issued to the XCM. A request to access the XCM is made if the bus to the XCM is found free. Once a request enters this state, the bus is locked as long as it takes to communicate the request.
6. XCRAM_ACC: A request to XCM transitions to this state while the XCM is being accessed. The time spent in this state is dependent on the SDRAM's configuration (CAS latency and the time between the active and read/write commands and the width of the request).
7. XCRAM_RES: Only prefetch requests transition to this state. This state accounts for the cycles consumed in communicating the data to the originating core.
8. XCRAM_COMPLETE: This is an artificial state to extract statistical data and is of relevance only within the simulator. The actual transfer of data takes place when the request is in this state. The target address is used to determine the index of the element and the bank in which it resides. A prefetch request results in data being read and written to the data elements in the memory request structure. A write request causes contents of the data elements in the request to be written to the XCM.
9. ICRAM_REQ: This is analogous to the XCRAM_REQ state.
10. ICRAM_ACC: This is analogous to the XCRAM_ACC state, though the latency of the operation is different because the ICM is an SRAM device.
11. ICRAM_RES: This is analogous to the XCRAM_RES state.
12. ICRAM_COMPLETE: This is analogous to the XCRAM_COMPLETE state.

Prefetch requests are returned to the originating core using the `req_to_tb()` function call. Data is transferred from the request element to the tag buffer and the request is discarded.

The various memory states are of a composite type called `mem_state` within the simulator. This is an enumerated data type and is listed in the appendix.

When a request transitions to any of the memory states, a timer is set that corresponds to the number of cycles the request is supposed to spend in that state on a real device. On expiry of this timer, the request is ready to transition to its next state. Similar timers exist for the various communications buses and XCM banks. These resources are marked as free in the cycle that the corresponding timers expire. The structures corresponding to this functionality are `bus_status` and `bank_status`. Several “update” functions operate on these structures.

The functions corresponding to the memory state machines are the `icm_pipeline()`, `xcm_pipeline()` and `tmc_memory_pipeline()`.

3.4 Output Interface

The output interface drains packets from the FOQ in the DMA module and processes them for information. The information recorded corresponds to the latency that the packet experienced within the system. The information is recorded on a per queue basis. The queue information is the same as that used by the XPM. The latency metric also allows the jitter to be computed. Other statistics collected are maximum and minimum delay and delay jitter in the system. Once the statistics are updated, the packet is destroyed and the count of the number of packets passed through the system is updated. The statistics are maintained in a text file (`resultfile`).

3.5 Top Level of the Simulator

The top level of the simulator is the simulator class. The simulator's constructor function instantiates the Input Interface (IPP), DMA (DMA), Network Processor (TOASTER) and Output Interface (OPP) as objects. Global time is set and updated at this level and passed down to all the component objects. Configuration parameters are obtained from the parameter header file (parameter.h). The entire simulator is instantiated as a Line Card (LC) object in cnetsim.cc. At run time, the various objects are executed in the reverse order of their position in the pipeline to simulate parallel operation of all the components.

3.6 Application Development

One of the goals of the simulator is to implement a networking application on the Toaster platform. Doing so would demonstrate how the last component required for completing the picture is included. The application chosen for implementation is a segment of a Differentiated router system.

Why Diffserv?

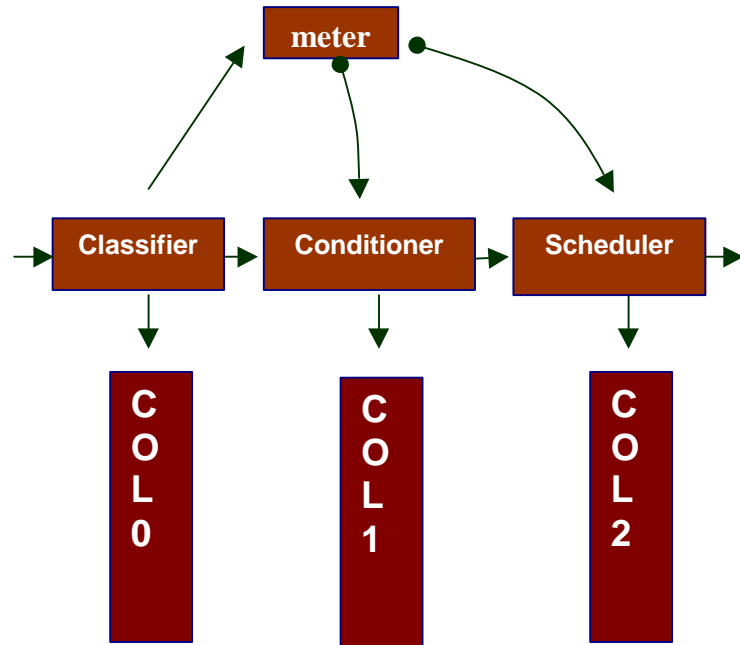


Figure 4.2 Diffserv on Toaster

The basis for Diffserv has found acceptance in the industry and the components that form the framework are widely used in routing systems manufactured by different vendors [20]. In looking for a sample set of network applications that can be used for the purpose of study, Diffserv components offer themselves as generic applications that will find use in any routing system independent of its position in a network.

A baseline implementation of a Diffserv router can be found in [6] - an implementation of Diffserv router components on a Linux system. The basic components in the system implemented in ComNetSim are Classifier, Conditioner and Scheduler. While it is expected that a shaper be in-

cluded in the system, the design of the remainder of the system does not allow for a shaper. The reason for this is discussed at a later point. The mapping of Diffserv components is illustrated in figure [figure 4.2].

The implementation of Diffserv on Toaster does not include a shaper. A shaper is usually implemented to control the outflow of packets from a system using a leaky bucket. In this system packets are dequeued from the XPM and directed to the output port. If a shaper were to be implemented on Toaster, packet information would have to be passed through it again when packets leave the DMA ASIC. Implementing the shaper as part of the DMA ASIC would be more efficient, allowing Toaster to operate on packets on the inflow path alone.

3.6.1 Classifier

The classifier is implemented on Column 0 of the Toaster framework. The function of the classifier is to match the packet to one of the pre-configured flows. A multi-field classifier has been implemented. The fields used to find the flow identifier are the Diffserv codepoint, the input interface, output interface and the upper level protocol. Once a flow identifier is computed, a lookup is made to the ICM. The flow identifier hashes into the ICM to give a queue number for the packet. The queue number is stored as the enqueue queue number in the control segment of the Toaster context and is used by the subsequent components.

3.6.2 Conditioner

The conditioner is implemented on Column 1 of the Toaster framework and takes the form of a Two Rate Three Color (TRTC) marker algorithm [10]. The algorithm operates in color aware mode and is as shown in figure [figure4.3].

Every queue configured has an entry in the conditioner table [table 3.1].

```
C_TOKENS = MIN(C_TOKENS, CBS) + (GLOBAL_TICKS - OLD_TICS) * CIR / 8;
C_DIF = C_TOKENS - PKT_LEN;

P_TOKENS = MIN(P_TOKENS, CBS) + (GLOBAL_TICKS - OLD_TICS) * PIR / 8;
P_DIF = P_TOKENS - PKT_LEN;

IF ((P_DIF < 0) AND (PACKET_COLOR == YELLOW)) OR (PACKET_COLOR == RED)
    PACKET_COLOR = RED;

ELSE IF ((C_DIF < 0) OR (PACKET_COLOR = YELLOW))
    BEGIN
        PACKET_COLOR = YELLOW;
        IF (P_TOKENS - PACKET_SIZE > 0)
            P_TOKENS = P_TOKENS - PACKET_SIZE;
        ELSE
            P_TOKENS = 0;
    END

ELSE
    BEGIN
        PACKET_COLOR = GREEN;
        IF (C_TOKENS - PACKET_SIZE > 0)
            C_TOKENS = C_TOKENS - PACKET_SIZE;
        ELSE
            C_TOKENS = 0;
    END

END
```

Figure 4.3 Two Rate Three Color Marker

Control Variable	Abbreviation	Width (bits)	Comment
Peak Information Rate	PIR	20	Maximum configured rate for the packets belonging to flow/ queue
Committed Information Rate	CIR	20	Average configured rate
Peak Burst Size	PBS	16	Maximum credit accumulated by the peak rate bucket
Committed Burst Size	CBS	16	Maximum credit accumulated by the committed rate bucket
Peak Tokens	P_token	16	Last peak token count
Committed tokens	C_token	16	Last committed token count
Old Tics	-	32	Last time queue was serviced

Table 3.1 Conditioner table element

The conditioner merely marks the degree of compliance to configured parameters. The action performed based on this check is left to the user. In this system the color code is marked in bits 5 and 6 of the DSCP and can be used by downstream Diffserv-aware routers.

3.6.3 Scheduler

The scheduler is a critical piece of the Diffserv system. A scheduler is an algorithm that distributes resources between different consumers. In a router, the resources correspond to limited buffer space and output bandwidth. The resource consumers are the various queues that need to

be serviced. There are several different classes of scheduling algorithms [21]. The goal is to provide guaranteed delay and bandwidth to every queue being serviced.

A Weighted Round Robin (WRR) Scheduler is implemented on the simulator's Toaster frame-

```
FOREVER {
  FOR (I =0 TO MAX_SERVICE_ROUNDS) {
    Q_NOT_FOUND = TRUE;
    Q_PTR = FIRST_Q;
    WHILE(Q_NOT_FOUND) {
      READ Q_WEIGHTQ_PTR;
      READ Q_EMPTYQ_PTR;
      IF ( !Q_EMPTY AND Q_WEIGHT > 0) {
        SERVICE Q(Q_PTR);
        Q_WEIGHT -= 1;
        Q_NOT_FOUND = FALSE;
      }
      Q_PTR = Q_PTR->NEXT;
    }
  }
  RESET ALL Q_WEIGHT;
}

FOREVER {
  IF LAST PACKET IN QUEUE SET Q_EMPTY = TRUE.
}
```

Figure 4.4 Weighted Round Robin Algorithm

work. A WRR scheduler is a variant of the simple Round Robin (RR) scheduler. WRR scheduler can prioritize service for a number of queues so that each queue may be serviced differently. At startup, a weight is associated with every active queue in the system. This weight is determined based on the mean packet size for the queue, the mean flow rate for packets belonging to the queue and the maximum mean rate configured. A weight counter is associated with every queue. In every scheduling interval, all non-empty queues with non-zero weight counters receive service. The weight counter for a queue is decremented at every instance that the queue is serviced. Once the weight counter for the queue with the maximum weight reaches zero or the number of rounds of service touches the maximum weight, all the weight counters are reset to their original values. The round number is a count of the number of times a check for serviceability has passed any queue in the system. Its reset value is the maximum of the weights of all currently active queues.

When implemented on a Toaster framework, several constraints play a role in the design. The implementation of the algorithm needs to be scalable and completely deterministic. While operating on a constrained cycle budget, it is not possible to query non-deterministically a number of queues until a serviceable queue is found. A serviceable queue is one that has packets in waiting and a weight counter that is non-zero. The baseline requirement is that the accesses to the queue structure holding the weight counter be optimized.

In order to meet the requirement the Toaster memory hierarchy is leveraged to implement a two-level system to deterministically compute queue serviceability. The implementation of WRR in the simulator can support 1024 queues.

The implementation of the WRR algorithm requires three basic structures:

1. Serviceability: This control variable indicates if a queue is eligible for service.

2. Non-emptiness: This control variable indicates if a queue has packets to send. A queue that is serviceable, yet empty, does not need to be served.
3. Weight Counter: For every instance a queue is served, the corresponding weight counter needs to be decremented. When this counter reaches zero, the queue is no longer serviceable.

The serviceability and non-empty states are stored in the ICM, which has a low read latency compared to the XCM. These are organized as two parallel structures. Each queue has representative bits to indicate serviceability and non-empty states. These are organized as 32 elements, each of 32 bits width. These structures are called tags. A consolidated structure is maintained that aggregates the information in the 32 tags. This structure is called a block. The layout thus formed has the following arrangement: Two block-level structures – to record aggregate serviceability and non-empty states. Each bit in these structures corresponds to a tag element of the same type i.e. a bit in the serviceability block references a serviceability tag.

The weight counter is placed along with other queue variables (original weight and queue statistics) in a queue structure. There is a queue structure associated with every queue. Memory constraints and the need for scalability direct that the queue structures be stored in the XCM.

When a Toaster context is processed by the core implementing WRR, the serviceability and non-empty blocks are read from ICM. The logical AND of these structures provides a 32-bit computed block that indicates the presence of tag elements that are serviceable and nonempty. The bit location in the computed block corresponds to the location of tag elements referencing serviceable queues. The first tag element in the order of service is read from the ICM. The logical AND of serviceable and nonempty tag elements is used to determine the queue number to be serviced. The queue number corresponds to the first bit position of a serviceable and non-empty queue in the order of service. The corresponding queue structure is fetched from XCM and its weight counter updated. If the weight counter is reduced to zero the queue is marked as non-

serviceable. The service order is recorded as an integer quantity called next_tag or next_q as the case might be. When a queue is serviced, the next_q variable is incremented and stored along with the corresponding tag. Thus, in the next service round the queue just serviced is ignored, as the next_q quantity would cause it to be ignored. The next_tag quantity is used in an analogous manner but is associated with the block structure.

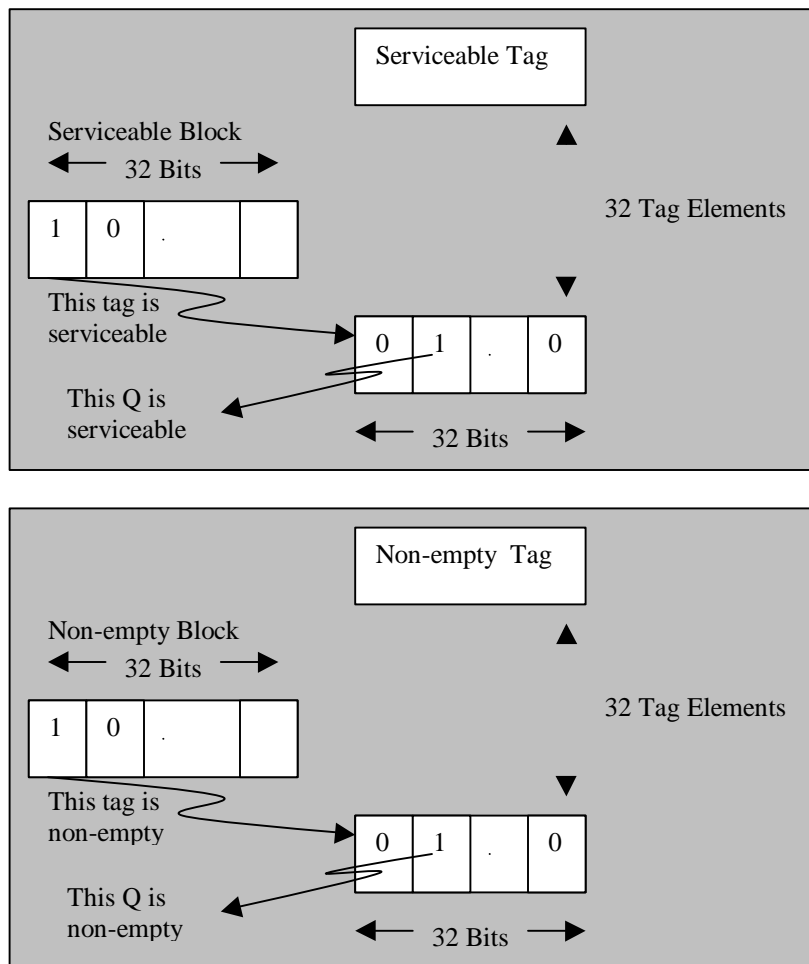


Figure 5 Weighted Round Robin Control Structures

At the end of a scheduling interval, the weight counters are not reset explicitly. Instead, a weight counter is reset if a referenced queue is found to have been serviced in a prior scheduling interval. The same method is applied to the tag elements, as it is inefficient to read every tag element and reset its serviceability bits. The serviceability block alone is reset at the end of the scheduling interval. The nonempty bit structures are controlled in parallel with the algorithm. It must be noted that the algorithm merely references the nonempty structure for information and does not manipulate its contents. The bits in the nonempty structure are set when a context corresponding to the queue referenced by the same bits passes through the framework. These bits are reset based on feedback information provided by the DMA component. This information is part of the control segment in the Toaster context.

A forced optimization that increases the efficiency of dequeue processing results in consecutive dequeues from the same queue. The optimization is forced, as the update to the tag variable targeting a queue can occur only after the queue structure has been processed. The immediately adjacent core accesses the same tag structure but the changes to the tag information are not visible to it. However, the queue structure itself is protected as it is read by the adjacent core late enough in a phase. In addition, the queue structure is protected by a lock. The new tag information will be visible to the third core in phase order. This ensures that the code does not spin on the same queue. The tag structure will reflect correct state at the end of the second core's phase. In this situation, performance of high rate queues will improve while that of low rate queues may degrade. Low rate queues that are dequeued consecutively are more likely to oscillate between empty and non-empty state. Dequeue requests to these are likely to be wasted as they are likely to occur when in an empty state.

Chapter 4 Operation, Experiments and Results

4.1 Simulator Directory Structure

The organization of the simulator is intuitive. The directory structure is organized in to three logical sections:

1. Traces: Packet sequences are generated using the “streamgen” shell script in the “pktgen” directory. The configurable parameters are minimum and maximum values for packet size, number of packets and arrival rate (Mbps). Packet traces are generated and stored in the “strdir” directory. A summary file (strlib.dat) is also created that is used by the dynamic component of the simulator. Files that are used in the setup of the dynamic component’s applications are also generated during stream generation.
2. Source: The “source” directory contains all the source files for the simulator. Running “make” from this directory creates an executable called “sim”.
3. Test: The simulator is invoked from this directory. A logical link needs to be established for the configuration files and executable. Output statistics are recorded in various files.

4.2 Simulator Configuration:

The simulator can be configured by modifying the *define* statements in the parameter files. The configurable parameters are listed in table [table 4.1].

Parameter Name	Parameter file	Purpose
CLOCK_PD	Parameter.h	Defines the clock period for the system
MAX_BW	Parameter.h	Maximum bandwidth allowed
NUM_IP_LINKS	Parameter.h	Number of input links
IP_LINK_OCCUPANCY	Parameter.h	Percentage occupancy of each link
IP_Q_SIZE	Parameter.h	Size of input queue buffer
PORT_TO_IQ	Parameter.h	Number of bytes that can be transferred from port to input buffer
OP_BUFFER_SIZE	Parameter.h	Size of output/switch port buffer
OP_DRAIN_RATE	Parameter.h	Rate at which the output buffer is drained
BYPASS_NPC	Parameter.h	Specifies if the network processor is to be bypassed
IPM_SIZE	Parameter.h	The size of Internal Packet Memory (IPM)
INFINITE_BUS	Parameter.h	Specifies if buses are constrained
IPIF_DMA_BYTE_RATE	Parameter.h	Multiplexed input buffer to DMA transfer rate
DMA_NPC_BYTE_RATE	Parameter.h	DMA to network processor transfer rate

Table 4.1 Simulator Configurable Parameters

NPC_DMA_BYTE_RATE	Parameter.h	Network processor to DMA transfer rate
T2_CONTEXT_SIZE	Parameter.h	Size of the toaster context
NUM_QUEUES	Parameter.h	Number of queues handled by the scheduler
NUM_ROWS	T_parameter.h	Number of toaster rows
NUM_COLUMNS	T_parameter.h	Number of toaster columns
TB_SIZE	T_parameter.h	Number of tag buffer elements
NUM_BANKS	T_parameter.h	Number of banks in external memory
XCM_MEMORY	T_parameter.h	Size of XCM memory
XCM_BANK_SIZE	T_parameter.h	Size of each bank
XCM_LINE_SIZE	T_parameter.h	Number of bytes per line
NUM_XCM_LINE	T_paramter.h	Number of lines for each bank
ICM_MEMORY	T_parameter.h	Size of XCM memory
ICM_LINE_SIZE	T_parameter.h	Number of bytes per line
NUM_ICM_LINE	T_paramter.h	Number of lines

Table 4.1 (Continued)

4.3 Simulator Setup

The following steps need to be taken to perform experiments:

1. Switch to “pktgen” directory and set the parameters in the streamgen script.
2. Run the “streamgen” script to generate packet streams, application configuration files and summary file.

3. Edit the parameter files in the source directory to set the physical configuration for the simulator.
4. Run “make” to generate the executable “sim.”
5. Create logical file links to “sim” and configuration files in the “strdir” directory.
6. Run “sim” to invoke the simulator.

4.2 Experiments and Results

The experiments performed need to demonstrate the working of the system and validate its purpose.

Experiment 1

The first experiment performed is a generic test with small number of flows and high speed interfaces. The simulator is configured with four input interfaces, each at 622 Mbps (OC12). The results [table 4.2] show the queue performance metrics. The queues have uniform delay and jitter characteristics.

Q#	Rate Mbps	Packets	Bytes	Avg Delay Seconds	Min Delay Seconds	Max Delay Seconds	Delay Jitter Seconds	Max Jitter Seconds
44	260	4854	315510	0.064144	5.78E-06	0.128299	2.84E-05	0.000352
113	290	4502	3124388	0.026071	0.000008	0.05217	3.99E-05	0.000335
144	290	31130	3735600	0.0146	5.77E-06	0.029276	4.15E-06	0.00018
211	250	36324	3232836	0.013185	5.88E-06	0.026448	3.48E-06	0.000152
695	240	6235	804315	0.055313	5.93E-06	0.110731	2.2E-05	0.000337
844	250	9341	1952269	0.037144	5.66E-06	0.074364	1.51E-05	0.000308
981	230	7615	1416390	0.043876	6.04E-06	0.087842	1.83E-05	0.000324

Table 4.2 Queue performance metrics

Experiment 2

Performance degradation with specific traffic patterns: This test demonstrates that the performance of the system is affected by the configuration of input traffic vectors. The test has two parts with only the input traffic configuration being changed. The input ports are configured as OC12 interfaces. In the first test, the input traffic flows are of a high rate above 500 Mbps. The second test uses several low rate flows whose average rates are about 100 Mbps.

The test shows that the average latencies for the high rate flows [table 4.3] are lesser than for the low rate flows [table 4.4]. The total number of contexts that needed to be passed through the Toaster cores is higher in case of the latter.

Q#	Rate Mbps	Packets	Avg Delay Seconds
137	100	1519	4.72E-03
201	100	1520	4.72E-03
523	100	176	7.94E-03
561	100	424	1.17E-05
562	100	160	8.11E-03
583	100	207	3.65E-03
625	100	424	1.23E-05
626	100	158	8.14E-03
651	100	177	7.89E-03
711	100	206	3.64E-03
715	100	178	7.81E-03
753	100	424	1.33E-05
754	100	161	8.03E-03
781	100	159	7.64E-03
802	100	304	7.34E-03
833	100	569	3.98E-03
860	100	224	7.28E-03
874	100	879	4.06E-03
897	100	572	3.94E-03
909	100	159	7.52E-03
930	100	301	7.39E-03
938	100	880	4.03E-03
988	100	220	7.51E-03
Contexts Passed		26508	

Table 4.3 Low Rate Queues Test

Q#	Rate	Packets	Avg Delay
48	580	538	3.50E-04
156	590	510	2.05E-05
176	580	64	1.63E-04
240	580	1423	1.91E-04
522	590	527	8.62E-06
561	550	457	1.00E-03
569	590	85	5.24E-04
588	570	1165	3.96E-05
625	550	525	5.05E-04
633	590	509	2.51E-03
652	570	1026	2.96E-04
689	550	1050	6.67E-04
697	590	545	1.63E-03
876	570	539	5.92E-04
940	570	856	7.93E-04
1004	570	182	9.86E-04
Contexts Passed		11577	

Table 4.4 High Rate Queues Test

Experiment 3

Packet re-order test: This test shows that the order of packets within a flow is not changed as they pass through the system. Packet reordering is a potential flaw in a parallel packet processing system. Four OC12 interfaces are used and 80 different flows are available. Results are collected on a per flow basis to determine if a packet's sequence number is lesser than that of the previous packet in the flow. The test results showed that no flow encountered reordering.

Experiment 4

Operation of low latency queues: Each input port has an associated low latency flow. Packets belonging to this flow bypass the queuing system. Low latency queues are available on routers as a channel for delay-sensitive traffic such as voice or video. Packets belonging to this flow are policed aggressively. This test shows that a low-latency flow experiences significantly lesser latency and jitter than an ordinary flow.

Chapter 5 Summary and Future Scope

This thesis presented a holistic methodology to study architectures for network processing. A cycle accurate simulator was implemented that will allow the study of the interactive nature of network traffic, algorithms and the hardware platforms. The implementation of a Network Processor architecture, such as Toaster for use in the simulator was discussed. Diffserv applications were developed for the Toaster platform in the simulator and an innovative implementation of Weighted Round Robin scheduling algorithm with support for 1024 queues was proposed. The development of the applications accounted for control memory design and optimization of memory references. The use of the simulator in studying the combined effects of network traffic, application and architecture was shown in Experiment 2, in Chapter 4.

Future work based on this simulator can progress along many different paths.

5.1 Next Generation of ComNetSim

ComNetSim is modular in design and can be scaled to build systems that are more complex. A more complete system would allow more opportunities for study.

5.1.1 DMA System

An initial step would be to improve the abstracted DMA system and add a more realistic memory system. Packet memory design and management is a complex task. Adding this capability to the simulator would increase the simulator's value tremendously. A real memory system would introduce details such as memory allocation and read/write latency. These would have to be recognized by the applications implemented on the Network Processor.

5.1.2 Scaling the System

As the simulator is modular in design, a line card with a duplex path can be built by using the components available. Traffic can be passed from the inflow direction to the outflow direction i.e. between the two simulator components. Different queue configurations and physical layout parameters can be used for each path. By introducing a switching fabric component, a model of a router can be developed [figure 5.1].

5.2 Methodology

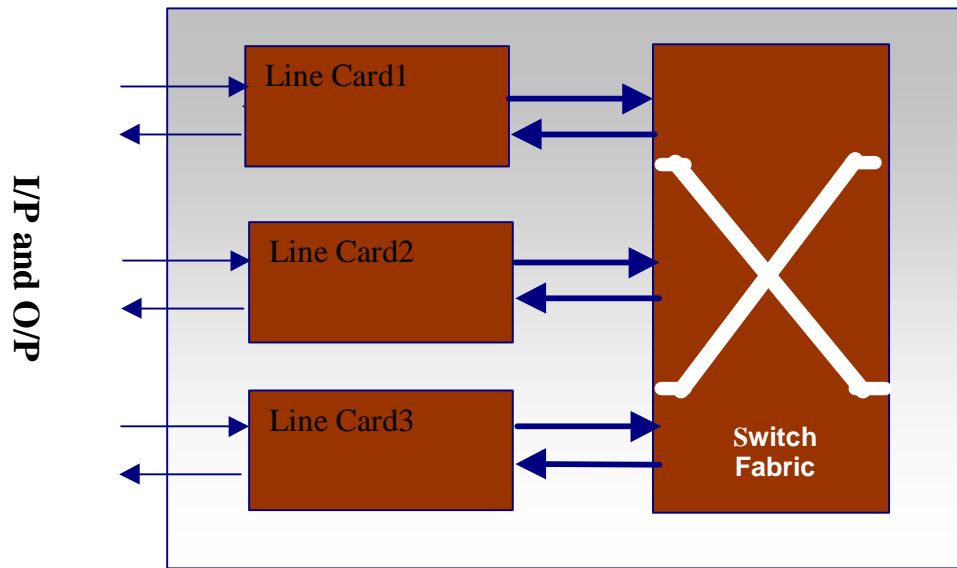


Figure 5.1 Router Structure

The holistic methodology used in this thesis can find a role in the design of network hardware architectures. As demonstrated, the performance of a network processing system depends on the applications and the input traffic vectors. It is important to acknowledge this in the development of network processors and their benchmarking schemes. A possible scenario would be one wherein application code may be compiled and executed on an execution based processor simulator. The processor simulator could be combined with a system that can retrieve packets from traces and

simulate their arrival. The design space in this case would include the application code, compiler, processor paradigm, memory system and traffic generator.

The simulator can be used to verify and study performance of a Network Processor architecture using load experiments. Two types of experiments can be performed. The system can be tested for a specific traffic configuration to verify performance for a network Processor system. The point at which the internal system buffers begin to overflow would indicate the maximum traffic load that can be handled by the system. The effect on a system's performance with incremental load can be studied by varying the traffic configuration.

References

1. Technical History of the ARPANET-Bibliography of ARPANET Papers and Articles. www.cs.utexas.edu/users/dragon/nph/ARPANET/ScottR/arpnet/bibliography.htm
2. Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z. and W. Weiss, "An Architecture for Differentiated Services", Internet Engineering Task Force (IETF), Request For Comment (RFC) 2475, December 1998.
3. IBM Technical Library, Networking Technology White Papers. www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/Networking_Technology
4. Intel(R) Internet Exchange Architecture. <http://developer.intel.com/design/ixa/whitepapers/ixapi.htm>
5. Various Toaster Documents, Cisco Systems, 2001.
6. Narasimhan, K., "An Implementation of Differentiated Services In A Linux Environment", Masters Thesis, North Carolina State University, December 2000.
7. Nichols, K., Blake, S., Baker, F. and Black, D., "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", IETF RFC 2474, December 1998.
8. Keshav, S., "An Engineering Approach to Computer Networking: ATM Networks, the Internet and the Telephone Network ", Addison-Wesley, 1997.
9. Heinanen, J. and R. Guerin, "A Single Rate Three Color Marker", IETF RFC 2697, September 1999.
10. Heinanen, J. and R. Guerin, "A Two Rate Three Color Marker", IETF RFC 2698, September 1999.
11. Kilkki, K. , "Differentiated Services for the Internet", Macmillan ISBN: 1578701325, July 1999.
12. Wolf, T., Franklin, M., "CommBench - A Telecommunications Benchmark for Network Processors", IEEE International Symposium on Performance Analysis of Systems and Software in Austin, TX, April 2000.
13. Crowley, P., Fiuczynski, M., Baer, J., and Bershada, B., "Characterizing Processor Architectures for Programmable Network Interfaces", Proceedings of the 2000 International Conference on Supercomputing, Santa Fe, N.M., May, 2000.
14. UCB/LBNL/VINT Network Simulator - ns (version 2), <http://www-mash.cs.berkeley.edu/ns>
15. Balkman, J. , Gupta, P., McKeown, N., "PAcket Lookup and Classification Simulator (PALAC)", <http://klamath.stanford.edu/tools/PALAC/SRC/>

16. Park, K., and Willinger, W., "Self-similar network traffic: An overview", In K. Park and W. Willinger, editors, *Self-Similar Network Traffic and Performance Evaluation*. Wiley Interscience, 2000.
17. Kramer, G., "Generator of Self-Similar Network Traffic", Networks Research Lab Dept. of Computer Science, University of California, Davis
http://wwwcsif.cs.ucdavis.edu/~kramer/code/trf_gen2.html
18. Various Documents, The Cooperative Association for Internet Data Analysis. www.caida.org
19. Floyd, S., Paxson, S., "Difficulties in Simulating the Internet", To appear in *IEEE/ACM Transactions on Networking*, February 2001.
20. Cisco 7600 Optical Services Router (OSR)
http://www.cisco.com/warp/public/3/ca/press/us_opticalrouter.html
21. Suryanarayanan, D., "An Analysis of Contemporary Link Schedulers",
<http://www4.ncsu.edu:8030/~dsuryan/ece633/scheduler.html>, March 2000.

Appendix

Header files

Data Structures

Data_type.h

```
#include<stdio.h>
#include"parameter.h"

////////////////////////////////////
typedef struct se {
    unsigned int num;           //Stream Number
    unsigned int rate;         //Stream rate
    unsigned int length;       //Stream length
    double start_time;         //Global start time for stream
    FILE *fileptr;             //File pointer
    //char *stream;             //File associated with the stream
    struct se *next;           //Pointer to next stream
} stream_element;

////////////////////////////////////
typedef struct le {
    unsigned int q_num;         //Q number for stats
    unsigned int ip_if;         //input interface
    unsigned int op_if;         //output interface
    double arrival;             //Arrival time
    unsigned int pkt_len;       //Packet Length
    unsigned int tos;           //Type of Service
    unsigned int protocol;      //Layer 4 protocol
    unsigned int tcp_in;        //TCP input port
    unsigned int tcp_out;       //TCP output port
    unsigned packet_num;        //Packet number
    unsigned int codepoint;     //DSCP
    bool drop;                  //Internal drop bit
    struct le *next;
} link_element;

////////////////////////////////////
typedef struct {
    unsigned int config_rate;   //Configured Rate for Link
    unsigned int curr_rate;     //Current Used Rate
    unsigned int max_bytes;     //Link Buffer Size (bytes)
    unsigned int curr_bytes;    //Link
    unsigned int num_streams;   //streams active on link
    unsigned int num_pkts;     //of pkts on the link
}
```

```

stream_element *stream_head;//Pointer to stream head
stream_element *stream_tail;//Pointer to stream tail
link_element *buffer_head;    //buffer Head pointer
link_element *buffer_tail;    //buffer Tail pointer
} link_desc;

```

```

/////////////////////////////////Stats Structure/////////////////////////////////

```

```

typedef struct {
    unsigned int configured_rate;//Configured rate
    double delay_sum;           //Sum of all delay for each packet
    double prev_delay;         //Delay for last packet
    double jitter;             //Average Delay jitter
    unsigned int bytes;        //Total # bytes
    unsigned int packets;     //Total # Packet
    double min_delay;
    double max_delay;
    double max_jitter;        //Max jitter for q
    double min_jitter;       //Min jitter for q
    double start_time;
    double last_time;
    double instant_rate;
    bool ef_q;                //Q Status
    unsigned int dropped;     //Number of packets dropped
} q_stat_element;

```

toaster_data_type.h

////////Tag buffer element//////////

```
typedef struct tbe {
    unsigned int address;           //Address Tag //Unused
    bool valid1;                   //Valid flag //Unused
    bool valid2;                   //valid Flag //Unused
    bool pending;                  //Pending flag
    unsigned int pending_timer;    //Pending timer
    unsigned int data1;            //first 32 bits of data
    unsigned int data2;            //Second 32 bits of data
} tag_element;
```

////////Memory States definition//////////

```
typedef enum mem_state { ISSUE,
                        LOCAL_BUS1,
                        LOCAL_BUS2,
                        TAG_BUFFER1,
                        XCRAM_REQ,
                        XCRAM_ACC,
                        XCRAM_RES,
                        XCRAM_COMPLETE,
                        ICRAM_REQ,
                        ICRAM_RES,
                        ICRAM_ACC,
                        ICRAM_COMPLETE };
```

////////Memory level//////////

```
typedef enum mem_level { TB,
                        ICRAM,
                        XCRAM };
```

////////Request Type//////////

```
typedef enum req_type { WRITE,
                        READ,
                        PREFETCH };
```

////////Request Len//////////

```
typedef enum req_len { _32BIT,
                       _64BIT };
```

////////Address struct//////////

```
typedef struct {
    unsigned int address;
    unsigned int base;           //Base address for struct
    unsigned int offset;        //Offset
} mem_address;
```

////////Data payload for address//////////

```

typedef struct {
    bool data1_valid;           //Data1 field is valid
    unsigned int data1;        //Data1 32 Bits
    bool data2_valid;           //Data2 field valid
    unsigned int data2;        //Data2
}data_element;

////////Memory request element////////

typedef struct mr {

    unsigned int CPU;           //CPU number in column
    mem_level LEVEL;           //TB/ICRAM/XCRAM
    req_type TYPE;             //write,read,pf;
    req_len LEN;               //32/64 bit access
    unsigned int BANK;         //Bank number - valid only for XCRAM requests
    mem_state STATE;          //State of request
    unsigned int timer;        //State timer
    unsigned int start_cycle;  //Start cycle
    unsigned int end_cycle;    //End cycle
    unsigned int index;        //Tag Buffer index
    mem_address address;       //Address element
    data_element data;         //data fields
    mr *next;                  //next pointer
} mem_request;

//Column Bus Status data type

typedef struct {
    unsigned int cpu;          //originating CPU
    bool busy;                 //Bus Busy
    unsigned int timer;        //Bus busy timer
} bus_status;

typedef struct {
    unsigned int cpu;          //originating cpu
    bool busy;                 //Busy state
    unsigned int req_timer;    //Next request can be accepted
    bool active;               //Active state
    unsigned int active_timer; //Next active command accepted
} bank_status;

typedef struct {
    unsigned int id;           //lock id
    bool busy;                 //busy flag
} lock_element;

typedef struct {
    //for statistics at core level
    unsigned int num_packets;  //Number of packets
    unsigned int num_stalls;   //Number of stall cycles
    unsigned int pf_icm32_latency; //Total pf latency for 32 bit access to ICM
    unsigned int pf_xcm32_latency; //pf latency - 32-bit access to XCM

    unsigned int pf_icm64_latency; //pf latency - 64 bit access to ICM

```

```

unsigned int pf_xcm64_latency;//pf latency - 64 bit access to XCM

unsigned write_icm32_latency;//write latency-32 bit access to ICM
unsigned write_xcm32_latency;//write latency-32 bit access to XCM
unsigned write_icm64_latency;//write latency-64 bit access to ICM
unsigned write_xcm64_latency;//write latency-64 bit access to XCM

unsigned int num_icm32_pf;//Number of 32 bit prefetches to ICM
unsigned int num_xcm32_pf;//Number of 32 bit prefetches to XCM
unsigned int num_icm64_pf;//Number of 64 bit prefetches to ICM
unsigned int num_xcm64_pf;//Number of 64 bit prefetches to XCM

unsigned int num_icm32_writes;//Number of 32 bit writes to ICM
unsigned int num_xcm32_writes;//Number of 32 bit writes to XCM
unsigned int num_icm64_writes;//Number of 64 bit writes to ICM
unsigned int num_xcm64_writes;//Number of 64 bit writes to XCM

unsigned int num_icm32_reads; //Number of 32 bit reads to ICM
unsigned int num_xcm32_reads; //Number of 32 bit reads to XCM
unsigned int num_icm64_reads; //Number of 64 bit reads to ICM
unsigned int num_xcm64_reads; //Number of 64 bit reads to XCM

unsigned int *pending_cycles; //Total pending/active cycles
unsigned int *num_req; //Requests per tag element

} cpu_data;

//////////Defining the Color Codes//////////
typedef enum COLOR_CODE {RED,
                        YELLOW,
                        GREEN,
                        WHITE };

//////ToToasterContext control segment definition//////
typedef struct ttc_control { //Control Context fields
    COLOR_CODE color; //Color (for Diffserv)
    unsigned int enq_qid; //Q# for current packet
    unsigned int deq_qid; //Q# to dequeue from
    unsigned int empty_q; //empty Q feedback info for toaster
    bool drop; //drop control bit
    bool ef; //Expedited Forwarding - dont enqueue
    bool empty; //empty context flag
};

//////To record queues that are empty//////////
typedef struct empty_list_element {
    unsigned int qid; //q that is empty
    empty_list_element *next;//next pointer
};

//////////TTC definition-context thro toaster//////////
typedef struct ttc {
    link_element *packet; //Pointer to Packet Header
    ttc_control *control; //Pointer to control segment
    ttc *next; //Next Pointer
} t2_context;

```

```
typedef unsigned int RAM_ELEMENT;

//////////Stats Structure for RAMs//////////
typedef struct RAM_STATS {
    unsigned int active_cycles; // # of cycles the bank was active
    unsigned int RD_32;        // # of 32 bit read/pf requests
    unsigned int RD_64;        // # of 64 bit read/pf requests
    unsigned int WR_32;        // # of 32 bit write/pf requests
    unsigned int WR_64;        // # of 64 bit write/pf requests
    unsigned int atomic;       // # of Atomic requests
};
```

Simulator Parameters

Parameter.h

```
#include <stdio.h>
#include <stdlib.h>

//////////global CONSTANT RATE defines//////////

#define OC192 10000 //10 x 1000 Mbps
#define OC48 2500 //2.5 x 1000 Mbps
#define OC12 622 //622 Mbps
#define OC3 155 //155 Mbps

//////////global clock define//////////

#define CLOCK_PD 10e-9 //10 ns

//////////Constant definitions//////////

#define BW_BITS 20
#define MAX_BW 2.5e+9
#define GRANULARITY (MAX_BW/(2^BW_BITS))
#define BW_FACTOR (GRANULARITY*CLOCK_PD)
#define AVERAGE_PACKET_SIZE 200

//////////System Configuration//////////

//////////IP Parser/ IP Interface//////////

//////////Number of Input links/Ports//////////
#define NUM_IP_LINKS 4 // Count from 1

//Occupancy factor for the link//////////
//The link has enough flows to fulfill this factor//////////
#define IP_LINK_OCCUPANCY 7e-1

//////////Size of the Input Interface queue//////////
#define IP_Q_SIZE 2000

//////////Transfer limit in bytes//////////
#define PORT_TO_IQ 6

//////////Output Parser/OP Interface //////////

//////////Number of Output links/ports//////////
#define NUM_OP_LINKS 4

#define OP_BUFFER_SIZE 1000

#define OP_DRAIN_RATE 4
#define INFINITE_DRAIN_RATE true

//////////Bypass Option//////////

//Enable options for components
```

```
//If true, NPC is bypassed and
#define BYPASS_NPC false

////////////////////////////////DMA Config////////////////////////////////

#define IPM_SIZE 128000

////////////////////////////////BUS CONSTRAINTS////////////////////////////////

#define INFINITE_BUS false
#define DMA_OPIF_BYTE_RATE 64
#define T2_CONTEXT_SIZE 128
#define NPC_DMA_BYTE_RATE 64
#define DMA_NPC_BYTE_RATE 64
#define IPIF_DMA_BYTE_RATE 64

////////////////////////////////Q CONFIG////////////////////////////////
#define NUM_QUEUES 1024

////////////////////////////////Q Metrics Transient Compensation////////
#define TRANSIENT_COMPENSATE_NUM 100
```

Toaster Parameters

T_parameter.h

```
/////////////////////////////////Toaster Parameters////////////////////////////////
/////////////////////////////////Layout Parameters////////////////////////////////

/////////////////////////////////Number of Rows////////////////////////////////
#define NUM_ROWS 4

/////////////////////////////////Number of Columns////////////////////////////////
#define NUM_COLS 4

////////////////////////////////\B////////////////////////////////
#define IB_LIMIT 1

/////////////////////////////////Tag buffer////////////////////////////////
#define TB_SIZE 8

/////////////////////////////////XCM Config////////////////////////////////

#define NUM_BANKS 8 //Number of Banks

#define XCM_MEMORY 262144 //256 KB Total XCM Memory

#define XCM_BANK_SIZE (XCM_MEMORY/NUM_BANKS)

#define XCM_LINE_SIZE 4 //Line size = 4 Bytes

#define NUM_XCM_LINES (XCM_BANK_SIZE/XCM_LINE_SIZE)

#define SPLIT_XCRAM_REQ_BUS true

/////////////////////////////////ICM Config////////////////////////////////

#define ICM_MEMORY 16384

#define ICM_LINE_SIZE 4 //Line size = 4 Bytes

#define NUM_ICM_LINES (ICM_MEMORY/ICM_LINE_SIZE)

#define SPLIT_ICRAM_REQ_BUS true
```