A SCHEDULER ARCHITECTURE FOR INTEL'S IXP2400 NETWORK PROCESSOR

Fariza Sabrina and Sanjay Jha
School of Computer Science and Engineering
The University of New South Wales, Sydney2052, Australia.
Email: {farizas,sjha}@cse.unsw.edu.au

UNSW-CSE-TR-0505
February, 2005

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA

School of Computer Science and Engineering
The University of New South Wales
Sydney, Australia.

Abstract

This report describes the design and implementation of a scheduling algorithm called CBCS on network processors. The CBCS algorithm is a fair and efficient resource scheduling algorithm that can fairly allocate multiple resources among contending flows. We have implemented this algorithm on Intel's IXP2400 network processor. The main objectives of implementing our scheduling algorithms on a real world Attached Network Processor (ANP) such as Intel IXP2400 Network processor were (1) to validate that CBCS can be implemented on a highly scalable real world programmable node or ANP for managing the CPU and bandwidth resources, (2) to provide scheduling solutions that could be opted or used by the ANP vendors like Intel to build and deliver efficient building block(s) (as a part of their Software Development Kit (SDK) or development framework) for resource scheduling purposes. The experimental results from IXP2400 implementation demonstrate the effectiveness and high performance of this algorithm in real world system.

1 Introduction

Parallel and Distributed Computing (PDC) enables sharing distributed network resources for solving large-scale resource intensive problems in multiple domains such as science, engineering, and commerce. In this paradigm, parallelism scales from a single machine to massive systems distributed across the whole Internet. Moreover, to provide flexibility in deploying new protocols and services, in today's internet, packet processing may be accomplished in the routers within the data path. To effectively schedule resources in programmable router, there must be an effective resource scheduling algorithm. Network processors are an attractive media for implementing smart network interfaces for cluster computing applications. Several companies have introduced powerful network processors (NPs) that can be placed in programmable routers to execute application level tasks in the network. An NP consists of a number of on-chip processors to carry out packet level parallel processing operations. The extensive multithreading on the IXP2400 and other network processors offers new design opportunities and challenges as well [5], [6], [7]. Intel's IXP2400 PCI card is a programmable network processor that implements a parallel processing architecture on a single chip and is designed for processing complex algorithms, deep packet inspection, traffic management, and packet forwarding tasks at high speed. It can be used for a wide range of access and edge applications including multi-service switches, routers, broadband access devices, and wireless infrastructure systems [9], [10], [11], [12].

Previously we proposed a fair resource scheduling algorithm for programmable networks called CBCS (Composite Bandwidth and CPU Scheduling Algorithm). The analysis and simulation work for CBCS could be found in [2], [1] and [3]. Later we implemented this algorithm on Intel's IXP2400 Network processor. This report provides an overview of the implementation hardware and software and presents the details of the implementation and experimental works that we have carried out. We also discuss the challenges faced during implementations and present the schedulers implementation architecture and experimental results.

We have developed a data plane application for IXP2400 network processor and have implemented the CBCS scheduler on the fast path processing i.e. on the microengines. We used microengine C for developing the code. We have also implemented separate CPU and bandwidth schedulers (based on DRR) on the microengines in order to compare experimental results of CBCS with the separate schedulers respectively.

We have performed extensive experiments and have collected results by running the compiled microengine codes against the Workbench Transactor. The workbench's port logging facilities were used to log the packets received and transmitted through the incoming and out going ports and the log files generated by the port logger were used by another tool (a custom Win32 application which we have developed for this purpose) to generate delay results.

The rest of the document is organized as follows: Section 2 briefly describes the implementation hardware and software. Section 3 presents the implementation challenges. Section 4 presents the details of the implementation work. Section 5 presents the experimental results and Section 6 concludes.

2 Implementation Hardware and Software

The implementation platform consists of a dual boot workstation, an IXP2400 PCI card, and Intel IXA (Internet Exchange Architecture) 3.1 SDK and framework. IXA 3.1 framework also includes a developer workbench or Integrated Development Environment (IDE). The development workstation is a Linux workstation configured to allow the use of Windows 2000 hosted tools. This functionality is enabled by the use of VMware (a software that allows PCs to support multiple operating systems simultaneously) to provide a virtual machine environment. The VMware allows running the IXA SDK developers Workbench under Microsoft Windows 2000 while running Linux 7.3 as the host operating system. The workstation has Pentium 4, 1.5 GHZ CPU and 512 MB of RAM. IXA 3.1 SDK and framework provide the IXP API libraries and some application building blocks that can be used for developing applications for IXP 2400 network processor.

3 Implementation Challenges

Our objective was to implement the schedulers on the fast-path i.e., on the microengines. We discovered problems and limitations with some of the application building blocks as provided in the SDK and had to overcome the problems. This section describes the challenges we faced and also describes the sequence of tasks we carried out in order to implementing the CBCS scheduler on the IXP 2400 microengines.

3.1 IXA SDK Applications and Incompatibility with IXA Education Platform

The IXA SDK 3.1 includes some example application codes that demonstrate network processor features and data flows. However, all the applications provided in the SDK are designed for IXP Advanced Development platform (which contains 2 separate IXP boards for handling packets in ingress and egress sides) and cannot be directly used to jump-start customer application development on an IXA Education platform which contains a single IXP2400 processor attached to Linux 7.3 host on a dual boot machine.

For example, figure 1 shows the software components in an Ethernet IPv4 forwarding application as given in the SDK example code for IXP2400 Advanced Development Platform. In this example, two half-duplex IXP2400 processors are used to implement an Ethernet line card that interfaces to a CSIX switch fabric. Figure 1 depicts the high-level design overview and shows the different software components used to build this application. The figure focuses only on the fast path or microengine components of the design the application. The description of the XScale Core Components for this application can be found in IXA Portability Framework Developer's Manual.

As shown in figure 1, the ingress IXP2400 receives Ethernet frames that carry IPv4 datagrams. The frames are assembled into IPv4 packets and the Layer-2 (Ethernet) headers are removed. Based on the IPv4 header, a Longest Prefix Match

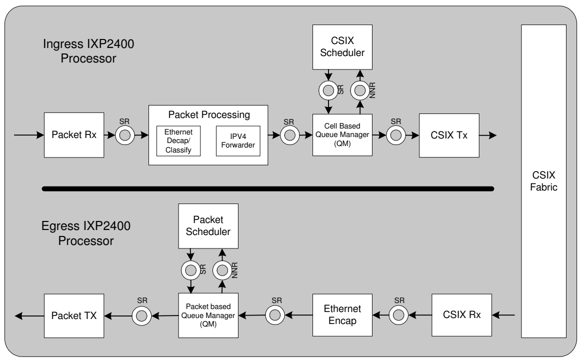


Figure 1: Ethernet IPV4 forwarding application in IXA SDK 3.1.

(LPM) lookup is performed and the packets are segmented into CSIX C-Frames and transmitted to the CSIX fabric. The result of the LPM lookup determines which IXP2400 connected to the fabric receives the packet, and which port on that IXP2400 the packet is transmitted on.

The egress IXP2400 receives CSIX C-Frames from the fabric and reassembles these into IPv4 datagrams. The Layer-2 (Ethernet) headers are added and the packets are transmitted over the appropriate port. The driver microblocks (Receive, Transmit, Scheduler and QM) run on different microengines from the packet processing code. In this design, each driver block occupies an entire microengine. The packet processing blocks on the ingress IXP2400 include the IPv4 Forwarder and the Ethernet decapsulation/classify microblock. There are four microengines that run in parallel and execute the packet processing code. On the egress side, the only packet processing code is the Ethernet Encapsulation/ARP block, which runs on a single microengine. The microengines communicates with each other via messages sent through scratch ring (SR) and next neighbor ring (NNR). The queue managers and schedulers implemented in the ingress and egress sides uses different queue manager and scheduler building blocks. Cell based queue manager uses c-frames for enqueuing and de-queuing whereas the packet based queue manager uses an entire packet for these purposes.

3.2 Building an IPv4 Application for IXA Education Platform

As mentioned earlier, the applications (such as the IPv4 forwarding application described above) designed for IXP Advanced development platform cannot be directly used for our development platform as it has only one IXP board. Instead of separate ingress and egress applications for the IPV4 forwarding application, our application is a single application consisting of modules for Packet Rx, Processing, Packet Tx, Queue Manager, and the Scheduler. Also the Ethernet layer 2 encapsulation is included in the packet-processing block. We used all the 8 microengines for the application as depicted in the figure. The reconfigurations were performed through some new compiler switches so that the updated building blocks do not affect the other SDK applications. Also we had to create few new scratchpad rings for the

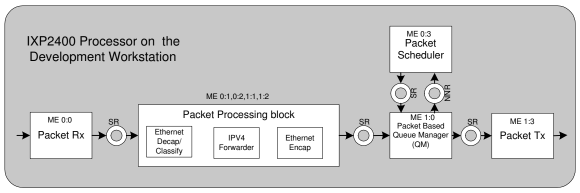


Figure 2: Developed ethernet IPV4 forwarding application.

communications. We used this application as a starter application for our further development and implementation tasks.

3.3 Problems Encountered with SDK 3.1 Building Blocks

Problems with Scheduler and Queue Manager:

Schedulers and Queue Manger are designed to run on separate microengines and the scheduler implements a DRR algorithm. During a DRR round, the scheduler schedules a packet every beat (88 cycles). This means that for large packets, the scheduler may be running faster than the transmit block can handle. After dequeuing a packet, the queue manager is expected to send the packet size to the scheduler so that the scheduler can use this delayed information to adjust the available credit. This design of the queue manager and schedulers has the following limitations:
1. The scheduler does not have any information regarding the queues and their depth and packet size during issuing a dequeue command to the queue manager and therefore it will be limited to implementing any efficient scheduling algorithm (e.g., even a simple DRR can not be implemented in real sense).
2. Even if the scheduler is extended to read the queue information, the synchronisation to protect the data may make the design inefficient compared to implementing scheduler and queue manager within a single microengine.
The implemented code of the packet-based queue manager and the scheduler building blocks in the IXA SDK 3.1 does not reflect or follow the design of the packet based queue manager and scheduler as specified in IXA SDK 3.1 Building Block design document. We noticed the following problems:

- The scheduler does not use the packet size at all that is supposed to come from the queue manager. Which means it is not doing any round robin based on packet size.
- Queue manager does not send the packet size to the scheduler for every valid packet it dequeued, rather it sends the information to the scheduler only if it was a transition i.e., when a queue becomes empty during a dequeue operation.

Problem with Packet Tx Block:

The packet transmission driver block has problem with transmitting packets with size over 127 bytes. We have experimented with larger packet size within the Developer Workbench and found that the logged packets (using the port logging option on the transmit port) contain some invalid trailing data (showing a stream of 0x5a5a5a5a) after each packet and transmission gets slower over a longer period of time.

3.4 CPU scheduling issues

Following are the limitations of the CPU scheduler:

- No building blocks are provided for CPU scheduling purposes.
- We found through experiments that measured CPU time consumed for processing a packet of the same size and for same processing task (e.g. IPv4 forwarding) varies in different executions. In our experiments the CPU requirement for IPv4 forwarding varied from 130 ns to 223 ns microengine cycles (where microengine clock frequency is 600 MHZ). The results support the case for applying a prediction algorithm for estimating CPU requirements for a packet within a CPU scheduler.

4 Implementations of CBCS on IXP2400

This section presents implementation details of CBCS on IXP2400. We have developed a data plane application for IXP2400 network processor and have implemented both the CBCS and also two sets of separate CPU and bandwidth schedulers (based on DRR) on the fast path processing i.e. on the microengines.

4.1 CBCS Implementation Architecture

We have implemented the CBCS scheduler on a single microengine, because it will not be efficient to run the enqueue and dequeue method of the same scheduler in different microengines. The implementation architecture of the schedulers is shown in Figure 3. The scheduler is implemented before the packet-processing block. The packet Rx microengine receives the packets and sends an enqueue message to the scheduler via scratchpad ring 1(SR-1). The scheduler microengine continually reads the enqueue request from SR-1, estimates the CPU requirements of the packet using the SES estimations technique, and enqueues the packet info in the SRAM queue. After dequeuing a packet, the scheduler sends a message to the processor microengines via a scratchpad ring (SR-2). Packet processing code runs on four microengines and all the microengines read the processing requests from SR-2 and process the packets. After processing the packet, the packet-processing microengines send a message specifying the CPU consumed and the flow id to the scheduler via another scratchpad ring (SR-3). After processing the packet, packet processor microengines send a transmission message to the transmitter microengine via a scratchpad ring (SR-4).

4.2 Separate CPU and Bandwidth schedulers

As mentioned earlier, we have also implemented a set of separate DRR schedulers for scheduling CPU and bandwidth separately on the IXP2400 processor in order to

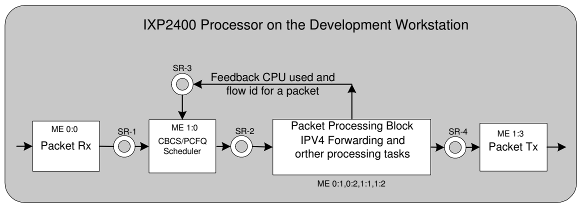

IXP2400 Processor on the Development Workstation
SR-3
Feedback CPU used and flow id for a packet
ME 0:0
Packet Rx
SR-1
ME 1:0
CBCS/PCFQ Scheduler
SR-2
Packet Processing Block IPV4 Forwarding and orther processing tasks
ME 0:1,0:2,1:1,1:2
SR-4
ME 1:3
Packet Tx

Figure 3: CBCS implementation architecture.

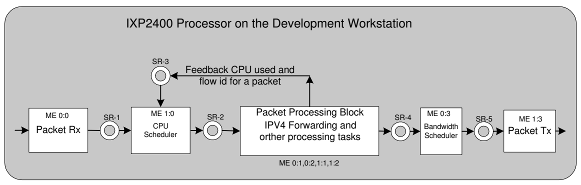

IXP2400 Processor on the Development Workstation
SR-3
Feedback CPU used and flow id for a packet
ME 0:0
Packet Rx
SR-1
ME 1:0
CPU Scheduler
SR-2
Packet Processing Block IPV4 Forwarding and orther processing tasks
ME 0:1,0:2,1:1,1:2
SR-4
ME 0:3
Bandwidth Scheduler
SR-5
ME 1:3
Packet Tx

Figure 4: Separate CPU and Bandwidth scheduler implementation architecture.

evaluate the performance of the CBCS scheduler compared to using separate CPU and bandwidth schedulers. Figure 4 shows the implementation architecture of the separate schedulers.

The messages that pass through the SR-1, SR-2, and SR-3 are same as that of Figure 7.12. Here, after processing the packet, the processor microengines send an enqueue request to the bandwidth scheduler via SR-4. After dequeuing a packet, the bandwidth scheduler sends a transmission message to the Packet TX microengine via SR-5.

4.3 Data Structures and Inter-microengines Messages

The main data structures for CBCS implementations and the format of the inter-microengine messages are briefly described in this section. As mentioned in the earlier section, for each packet received, packet data are kept in DRAM and packet metadata (i.e., information about the packet) is kept in the SRAM. The packet metadata structure has 8 long word members. IXP library provides macros and functions called dispatch loop functions to read packet metadata from SRAM and to write back the metadata into the SRAM. A dispatch loop combines microblocks on a microengine and implements the data flow between them. The dispatch loop also caches commonly used variables in registers or local memory. These variables can be accessed by microblocks using the dispatch loop macros and functions. We have used dispatch loop functions to write some data like total resource requirement for a packet (for CBCS scheduler) into a member of the packet meta data in the SRAM during packet enqueue operation and to retrieve the data back from the SRAM during packet dequeue operation.

Dispatch Loop / Packet Buffer Metadata Structure

This data structure defines the metadata for the packet buffers. The data fields contained in its 8 long words (LW) are shown in table 1.

Table 1: Packet buffer meta data structure.

LW	Bits	Size	Data Field	Description
0	31:0	32	bufferNext	Buffer handle for the next buffer in the chain. Refer to table 2 for the buffer handle data structure.
1	31:16	16	bufferSize	The amount of data currently in the buffer.
1	15:0	16	offset	The offset in DRAM where the data begins.
2	31:16	16	packetSize	The amount of data in the buffer chain.
2	15:12	4	freeListId	The free list to which this buffer belongs.
2	11:8	4	rxStat	The receive status.
2	7:0	8	headerType	The header type-IPv4, IPv6, and so on.
3	31:16	16	inputPort	The input port on which this packet was received.
3	15:0	16	outputPort	The output port on which this packet is to be transmitted.
4	31:16	16	nextHopId	The next hop ID.
4	15:8	8	fabricPort	The blade port.
4	7:4	4	reserved	Reserved for future use.
4	3:0	4	nhidType	The next hop ID type.
5	31:28	4	colorId	-
5	27:24	4	reserved1	-
5	23:0	24	flowId	The flow ID.
6	31:16	16	classId	The class ID.
6	15:0	16	reserved2	Reserved for future use.
7	31:0	32	packetNext	The next packet in the chain.

Table 2: Buffer handle data structure.

LW	Bits	Size	Data Field	Description
0	31:31	1	eop	End of packet flag.
0	30:30	1	sop	Start of packet flag.
0	29:24	6	seg_count	Segment count.
0	23:0	24	lw_offset	Long word offset of buffer address in the SRAM.

In our implementations, we used the *7th* long word (i.e., the class_id and reserved_2) field to store the scheduler data.

Packet Rx to CBCS Scheduler Message Structure

In our implementations, the enqueue message that the packet Rx microengine sends to the CBCS scheduler microengine via SR-1 contains three long words of data as shown in table 3.

It may be mentioned that Packet Rx to CPU scheduler message structure (in figure 4) also uses this data structure.

CBCS Scheduler to Packet Processor Message Structure

The CBCS scheduler to the packet processor messages (via SR-2) contains 2 long words of data as shown in table 4. The packet processor microengines cache (i.e., store) the first long word data in local memory and uses the same

Table 3: CBCS enqueue message structure

LW	Bits	Size	Data Field	Description
0	31:0	32	sopHandle	Start of packet buffer handle, which uses the data structure defined in table 2.
1	31:0	32	eopHandle	End of packet buffer handle, which uses the data structure defined in table 2
2	31:0	32	flow_id	Flow number or flow id.

Table 4: CBCS scheduler to packet processor message structure.

LW	Bits	Size	Data Field	Description
0	31:31	1	valid	Valid bit set to 1 for a valid message.
0	30:28	3	reserved	Reserved for future usage.
0	27:24	4	portNumber	Port Number (for transmitter).
0	23:0	24	sopBufferOffset	Start of packet buffer offset in long words.
1	31:0	32	flow_id	Flow number or flow id.

data to generate transmit message to the packet transmitter microengine. Also the sopBufferOffset field value is used to access the packet metadata from the SRAM memory using the dispatch loop functions. The second long word value is used later as a part of Packet Processor to Scheduler feedback message.

Note that CPU scheduler to packet processor message structure (in Figure 4) also uses this data structure.

Packet Processor to CBCS Scheduler Feedback Message Structure

After processing of a packet is completed, the processor microengine sends a feedback message to the scheduler (via SR-3) that contains two long words of data as shown in table 5. Note that packet processor to CPU scheduler message structure (in Figure 4) also uses this data structure.

Packet Processor to Packet Tx Message Structure

Also in our implementations (Figure 3), after processing of a packet is completed, the processor microengine sends a packet transmission message to the Packet Tx microengine (via SR-4) that contain just one long word of data as shown in table 6.

Also in the separate scheduler implementations (Figure 7.13), the bandwidth scheduler uses this message structure to generate the transmission messages that are sent to the packet Tx microengine via SR-5.

Table 5: Packet processor to scheduler feedback message structure

LW	Bits	Size	Data Field	Description
0	31:0	32	usedCpuCycles	Number of CPU cycles used for processing the packet.
1	31:0	32	flow_id	Flow number or flow id

Table 6: Packet processor to packet Tx message structure.

LW	Bits	Size	Data Field	Description
0	31:31	1	valid	Valid bit set to 1 for the transmitter to consider
				the message as a valid transmission message.
0	30:28	3	reserved	Reserved for future usage.
0	27:24	4	portNumber	Port Number (for transmitter).
0	23:0	24	sopBufferOffset	Start of packet buffer offset in long words.

It may be noted that in separate scheduler implementations (Figure 4), the processor block sends another enqueue message to the bandwidth scheduler block where the message contains two long words. The first long word is the start of packet buffer handle and the second long word is the flow id.

4.4 CBCS Implementation Details

We have used microengine local memory for keeping CBCS scheduler variable such as Quantum (or credit increment), packet counts for the flows or queues, credit counter per flow, estimated CPU requirements (per packet per flow) etc. We used the local memory as it's the fastest to access and it was enough to accommodate our variables for our experiments (with 16 flows). However, SRAM can be used for allocating the variables when number of flows is extremely high.

The CBCS scheduler is implemented using 4 threads e.g., initialization thread, enqueue thread, dequeue thread, and CPU prediction thread. After initialization is completed, the initialization thread sends signals to the enqueue, dequeue, and CPU prediction threads to begin their tasks as they wait on the initialization thread's completion signal.

Initialization Thread

Initialization thread sets the SRAM channel CSR to indicate that packet based enqueue and dequeue would be done, i.e., we enqueue and dequeue a full packet every time. The thread also initializes SRAM queue descriptors (and queue array) and the scheduler variables (e.g., it initialises the value of quantum, credit counter for the flows, estimated CPU requirements per flow etc). After initializing the scheduler variables, the thread terminates itself so that the microengine thread arbiter excludes this thread from its list.

Enqueue Thread

Figure 5 shows a simplified flow diagram of works performed within the CBCS enqueue thread. The enqueue thread waits for the signal from the initialization thread before starting its infinite loop. In each turn, the thread calls an SRAM API (e.g. scratch_get_ring) to read an enqueue message from SR-1 and specifies a signal number (as a parameter to the API call). The thread then swaps out to allow other threads to run as the SRAM read operation would take some time. After receiving the control back, the thread checks the presence of the signal (i.e., checks whether the enqueue message read operation is completed or not. Once the enqueue message is read, it checks the validity of the enqueue message as there may not be any message in the ring.

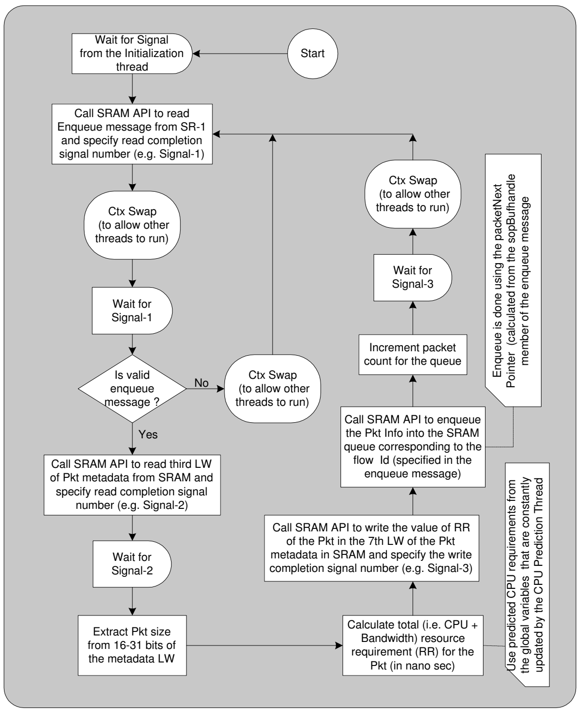

Start
Wait for Signal from the Initialization thread
Call SRAM API to read Enqueue message from SR-1 and specify read completion signal number (e.g. Signal-1)
Ctx Swap (to allow other threads to run)
Wait for Signal-1
Is valid enqueue message ?
No
Ctx Swap (to allow other threads to run)
Yes
Call SRAM API to read third LW of Pkt metadata from SRAM and specify read completion signal number (e.g. Signal-2)
Wait for Signal-2
Extract Pkt size from 16-31 bits of the metadata LW
Calculate total (i.e. CPU + Bandwidth) resource requirement (RR) for the Pkt (in nano sec)
Use predicted CPU requirements from the global variables that are constantly updated by the CPU Prediction Thread
Call SRAM API to write the value of RR of the Pkt in the 7th LW of the Pkt metadata in SRAM and specify the write completion signal number (e.g. Signal-3)
Call SRAM API to enqueue the Pkt Info into the SRAM queue corresponding to the flow Id (specified in the enqueue message)
Enqueue is done using the packetNext Pointer (calculated from the sopBufhandle member of the enqueue message
Increment packet count for the queue
Wait for Signal-3
Ctx Swap (to allow other threads to run)

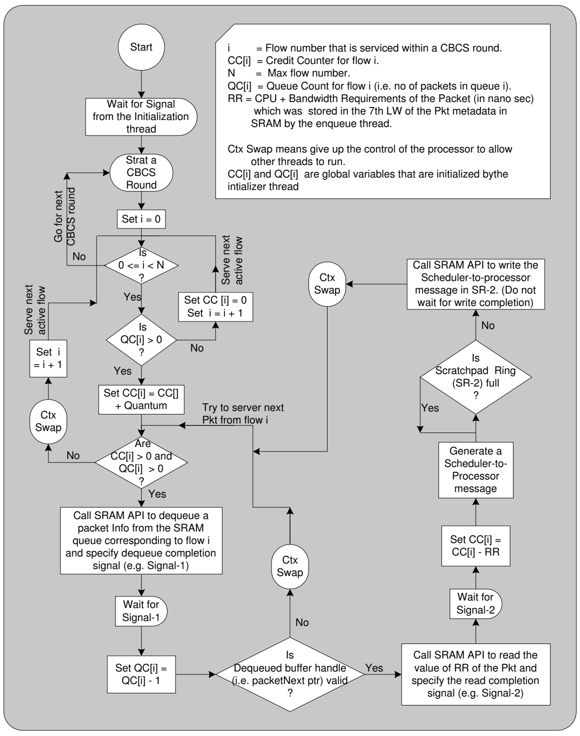
Figure 5: Flow diagram of the CBCS enqueue thread.

If the thread receives an invalid message, it does context swap and then goes for the next turn. As shown earlier in table 1, the third LW of packet metadata contains the packet size field. So, if the enqueue message is a valid message, the thread reads the third LW of the packet metadata from the SRAM using another API (e.g. sram_read) and extracts the packet size for calculating the total resource requirement (i.e. both the CPU and bandwidth) for the packet. The CPU requirement data is taken from the global variable (per flow), which is constantly updated by the CPU prediction thread. The calculated total resource requirement is used by the dequeue thread for scheduling purposes, and therefore it needs to be stored. We decided to use 7^{th} LW of the packet metadata to store this scheduler data.

The enqueue thread calls an SRAM API (e.g., sram_write) to write back the resource requirement data to the SRAM and specifies a signal number. While the write operation is in progress, the thread calls another API to enqueue the packet info in the SRAM queue corresponding to the flow-id. It may be mentioned that the enqueue is done using the packetNext pointer (calculated using the sopBufHandle member of the enqueue message). The thread increments the packet count for the queue and waits for the SRAM write operation to be completed. The thread then does a context swap and goes for the next round.

RR Calculations

We calculate the total resource requirement (RR) for the incoming packets in


Start
Wait for Signal from the Initialization thread
Strat a CBCS Round
Set i = 0
Go for next CBCS round
No
Is 0 <= i < N ?
Serve next active flow
Set CC [i] = 0 Set i = i + 1
Yes
Is QC[i] > 0 ?
No
Serve next active flow
Set i = i + 1
Yes
Set CC[i] = CC[] + Quantum
Try to server next Pkt from flow i
Ctx Swap
No
Are CC[i] > 0 and QC[i] > 0 ?
Yes
Call SRAM API to dequeue a packet Info from the SRAM queue corresponding to flow i and specify dequeue completion signal (e.g. Signal-1)
Wait for Signal-1
Set QC[i] = QC[i] - 1
Is Dequeued buffer handle (i.e. packetNext ptr) valid ?
No
Ctx Swap
Yes
Call SRAM API to read the value of RR of the Pkt and specify the read completion signal (e.g. Signal-2)
Wait for Signal-2
Set CC[i] = CC[i] - RR
Generate a Scheduler-to-Processor message
Is Scratchpad Ring (SR-2) full ?
Yes
No
Call SRAM API to write the Scheduler-to-processor message in SR-2. (Do not wait for write completion)
Ctx Swap

i = Flow number that is serviced within a CBCS round.
CC[i] = Credit Counter for flow i.
N = Max flow number.
QC[i] = Queue Count for flow i (i.e. no of packets in queue i).
RR = CPU + Bandwidth Requirements of the Packet (in nano sec) which was stored in the 7th LW of the Pkt metadata in SRAM by the enqueue thread.

Ctx Swap means give up the control of the processor to allow other threads to run.
CC[i] and QC[i] are global variables that are initialized bythe intializer thread

Figure 6: Flow diagram of the CBCS dequeue thread.

nano seconds (ns) using the following equation.

RR= CPU Cost of the packet (ns) + Transmission cost of the packet (ns)
= CPU cost (ns) per CPU Cycle $*$ Estimated CPU Cycles Requirement +
Transmission cost per byte (ns) $*$ Packet size in Bytes
It should be mentioned that, each microengine has clock frequency of 600 MHZ i.e., 600 millions cycles per sec. Therefore, CPU cost (ns) per $CPUCycle = \frac{5}{3}ns$. For a 100 Mbits network interface, the transmission cost per byte would be = 80 ns.
Since the microengines do not support the floating-point calculations, the CPU cost calculation for a packet is approximated, where the calculation error is less than or equal to $\frac{2}{3}$ ns. This calculation error or approximation is quite acceptable as it is tiny compare to the value of RR and it happens for some of the packets for all flows.

Dequeue Thread

Figure 6 shows the simplified flow diagram of the activities performed within the CBCS dequeue thread. As shown in the figure, dequeue thread waits for signal from initialization thread before starting its infinite loop. In each CBCS round, the algorithm serves all the active or backlogged flows (i.e., the flows having one or more packets in the queue). So for each flow i, the algorithm checks whether the Queue Count i.e., $QC[i]$ (stored in global variables) is positive or not. If $QC[i]$ is positive, it adds quantum to the value of the

Credit Counter of the flow i (i.e. $CC[i]$), otherwise it resets the $CC[i]$ to 0 and tries to serve the next active flow.

While serving flow I within each CBCS round, the algorithm checks whether both the $CC[i]$ and the $QC[i]$ are positive or not. If either of them is 0 or negative, the algorithm does a context swap (so that other threads get a chance to run) and then tries to serve the next active flow. Otherwise, the algorithm calls an SRAM API (e.g., sram_dequeue) to dequeue a packet info from the SRAM queue corresponding to flow i and it waits for the dequeue completion signal. After the dequeue, it decrements the queue count for flow i and then it checks the validity of the dequeued buffer handle (i.e., the packetNext ptr as enqueued in the enqueue operation). If the buffer handle is invalid, it does a context swap and then tries to serve the next packet from the same flow i.

For a valid dequeue of a packet, the code calls another SRAM API to read the resource requirement (RR, which is the CPU requirement plus bandwidth requirement in nano seconds) from the 7th LW of the packet metadata in SRAM (as it was stored there during enqueue operation) and waits for the read operation to complete. On completion of the SRAM read, the system signals the thread and the code then decrements the $CC[i]$ by the value of RR. The thread then generates a scheduler-to-processor message using the data structure defined in table 4 and enqueues the message to the scratchpad ring 2 (SR-2). However, before enqueuing the message in SR-2, it checks the fullness of the ring using IXP library API and waits if the ring is full. After sending the message to the processor, the thread swaps out and tries to serve the next packet from the same flow i.

CPU Prediction Thread

This thread waits for the signal from the initialization thread before it starts its infinite loop. In each turn, the thread calls an SRAM API to read the processor-to-scheduler message (as defined in table 5) from scratchpad ring 3 (SR-3) and specifies a signal number to wait on and then swaps out so that other threads can work while it is waiting for the read to complete.

After reading the message, the thread validates the message and if it's a valid message, then it updates the estimated CPU requirement of the specified flow using SES estimation technique. The estimated CPU requirements (per packet) per flow are kept in global variables. Again, due to unavailability of the floating-point calculations, the estimations are approximated and the approximations or error of calculation is less than or equal to cycles while using alpha value of 0.5 for SES equation.

5 Experiments and Results

we have tested the performance of the CBCS scheduler against the performance of the implemented separate schedulers. The experiments were performed by running the code on IXA workbench's "Cycle Accurate" transactor. The port logging options were turned on to log the packets received and transmitted at the media interfaces.

Table 7: CPU requirements and packet sizes for the experimental traffic flows.

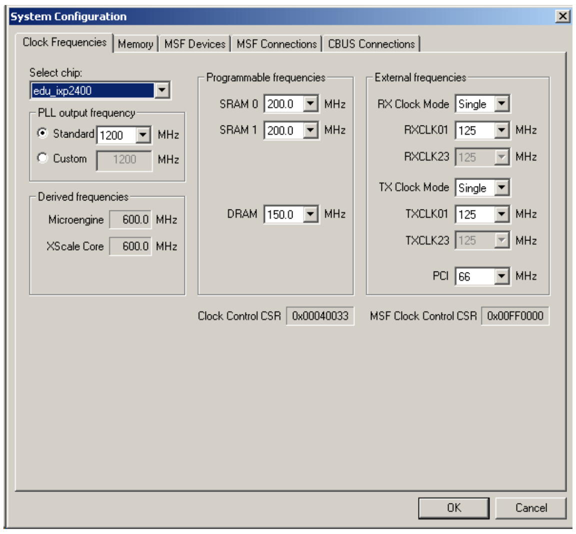
Flow Number	CPU Req. Category	Bandwidth Req. Category	CPU Req. (Cycles)	Packet Size (Bytes)
1, 5, 9, 13	High	Low	2400 - 3600	42 - 48
2, 6, 10, 14	Low	High	78 - 134	120 - 127
7, 16	Medium	Medium	1200 - 1800	80 - 88
3, 8, 12	Low	Low	78 - 134	42 - 48
4, 11, 15	High	High	2400 - 3600	120 - 127

The logs files produced were used by a custom software tool (that we have also developed under this project) to analyze the packet logs and produce the delay results for the individual flows.

5.1 Design of experiments

We used 16 flows with varying packet sizes and different CPU requirements. Four of the flows (e.g., flow 2, 6, 10, and, 14) required IPv4 forwarding and other flows required some other processing code. Table 7 shows the CPU requirements and packet sizes for each individual flows.

For all the experiments, receive and transmission rates on the media interfaces were set to 50 Mbps. For system settings, workbench simulator's default settings (as shown in Figure 7) were used.



Select chip: edu_ixp2400

PLL output frequency
(•) Standard 1200 MHz
( ) Custom 1200 MHz

Derived frequencies
Microengine 600.0 MHz
XScale Core 600.0 MHz

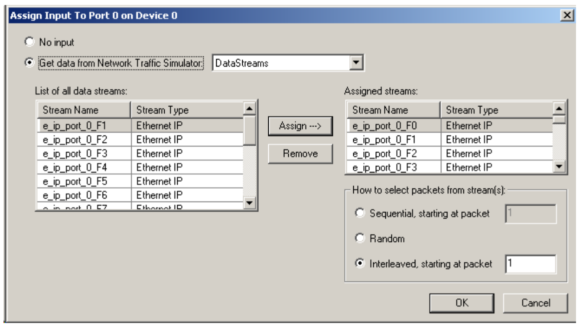
Programmable frequencies
SRAM 0 200.0 MHz
SRAM 1 200.0 MHz
DRAM 150.0 MHz

External frequencies
RX Clock Mode Single
RXCLK01 125 MHz
RXCLK23 125 MHz
TX Clock Mode Single
TXCLK01 125 MHz
TXCLK23 125 MHz
PCI 66 MHz

Clock Control CSR 0x00040033
MSF Clock Control CSR 0x00FF0000

Figure 7: Experimental system configurations.

Each microengine has a clock frequency of 600 MHz. The SRAM clock frequency was set to 200 MHz and DRAM frequency was set to 150 MHz. The PLL output frequency used was 1200MHZ. The receive and transmission rates on the media interfaces were set to 50 Mbps. We created 16 data stream files containing Ethernet frames and used the Workbench Simulator's Network traffic assignment functionality (as shown in Figure 8) to inject the data frames during experiments.


Assign Input To Port 0 on Device 0

No input [✓ Get data from Network Traffic Simulator:] DataStreams

List of all data streams:

Stream Name	Stream Type
e_ip_port_0_F1	Ethernet IP
e_ip_port_0_F2	Ethernet IP
e_ip_port_0_F3	Ethernet IP
e_ip_port_0_F4	Ethernet IP
e_ip_port_0_F5	Ethernet IP
e_ip_port_0_F6	Ethernet IP

Assigned streams:

Stream Name	Stream Type
e_ip_port_0_F0	Ethernet IP
e_ip_port_0_F1	Ethernet IP
e_ip_port_0_F2	Ethernet IP
e_ip_port_0_F3	Ethernet IP

How to select packets from stream(s):

Sequential, starting at packet 1 / Random / [✓ Interleaved, starting at packet 1]

Figure 8: Assigning experimental data streams using workbench simulator.

5.2 Experimental Results: CBCS vs Separate DRR schedulers

Experiments were performed and packet logs were collected while using both the CBCS scheduler and the separate DRR schedulers for 16 flows. Then we used our tool to analyze the logs and produce the delay results. The delay graphs and the delay summaries for each type of packet flow are shown below. The results show that the CBCS provided superior delay performance. We could not provide any other kind of performance comparison because of the limitations of the workbench simulator (which only provides the input and output port logging options).

Table 8: Delay statistics for flow with high CPU and high BW requirements.

Scheduler	Max.	Avg.	Std. Dev.
CBCS	1.057624	0.664084	0.149557
Sep. DRR	1.498264	0.915469	0.194695

Table 9: Delay statistics for flow with high CPU and how BW requirements.

Scheduler	Max.	Avg.	Std. Dev.
CBCS	1.109144	0.708829	0.144204
Sep. DRR	1.315864	0.888282	0.187556

Figures 9– 13 show the delays measured for data flows using CBCS and DRR. The maximum and average delays (measured in ms) for these flows are shown in table 8 – 12. Delay results show that CBCS achieved superior delay guarantees compared to DRR (when used individually for CPU and bandwidth scheduling) for all the flows.

We run our experiments for five scenarios (as shown in 7). For all the cases the maximum is worse for separate DRR than CBCS. The results show that using CBCS, the worst case delay is reduced by 29% for flows with high CPU and high BW, 15% for high CPU and low BW scenario, 26% for low CPU and high BW scenario,

Table 10: Delay statistics for flow with low CPU and high BW requirements.

Scheduler	Max.	Avg.	Std. Dev.
CBCS	0.956184	0.598476	0.13783
Sep. DRR	1.283544	0.823126	0.205549

Table 11: Delay statistics for flow with low CPU and low BW requirements.

Scheduler	Max.	Avg.	Std. Dev.
CBCS	0.996664	0.58697	0.142346
Sep. DRR	1.098904	0.771186	0.19177

9% for low CPU and low BW scenario, and 15% for data flow with medium CPU and medium BW requirement compared to the delays achieved using separate DRR. Also the average delay was reduced by 28% for flows with high CPU and high BW, 21% for high CPU and low BW scenario, 28% for low CPU and high BW scenario, 25% for low CPU and low BW scenario, and 19% for data flow with medium CPU and medium BW requirement compared to the delays achieved using DRR.

Table 8 – 12 show the standard deviation of the delays measured. It shows that using CBCS, standard deviation is reduced by 25% for flows with high CPU and high BW, 22% for high CPU and low BW scenario, 35% for low CPU and high BW scenario, 26% for low CPU and low BW scenario, and 35% for data flow with medium CPU and medium BW requirement compared to the delays achieved using DRR.

The delay graphs (Figure 9 - 13) show that CBCS provides much more consistent delay guarantees compared to that achieved using separate DRR for CPU and BW scheduling.

6 Conclusion

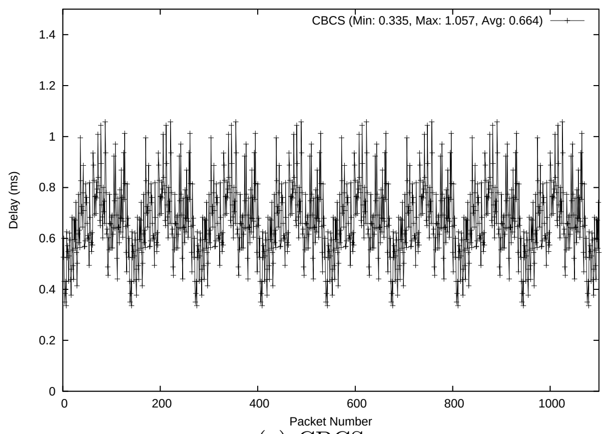
In addition to measuring the performance of the developed CBCS scheduling algorithms through extensive simulations on Berkley NS2 network simulator, we have also implemented the scheduling algorithms on a real world Attached Network Processor (ANP) such as Intel IXP2400 Network processor. This report presented our implementation work.

Acknowledgment

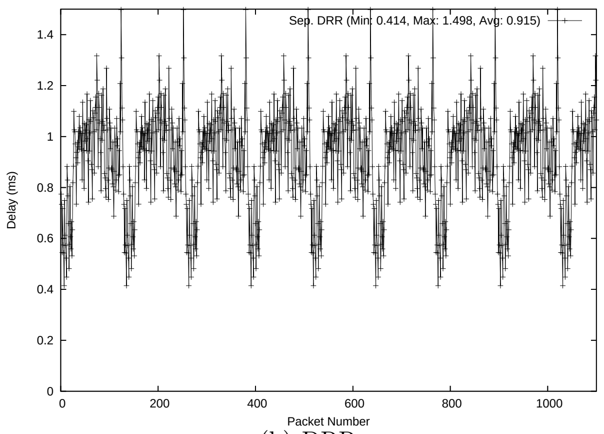
This research work is supported by Smart Internet Technology Co-operative Research Center (CRC), Australia.

References

[1] F. Sabrina and S. Jha, "A novel Architecture for resource management in Active Networks using a directory service", ICT 2003, Tahiti, French Polynesia, February 23 -1 March, 2003, pp: 45-52.

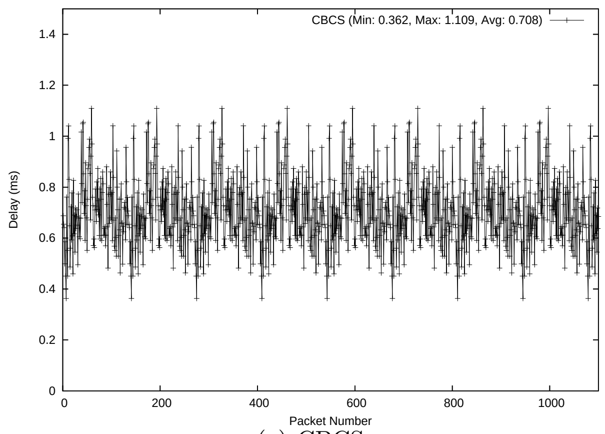


(a) CBCS

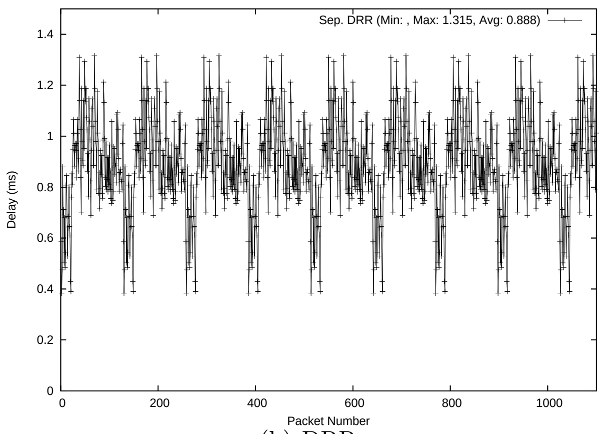


(b) DRR

Figure 9: Delay for Flows with High CPU and High BW Requirements

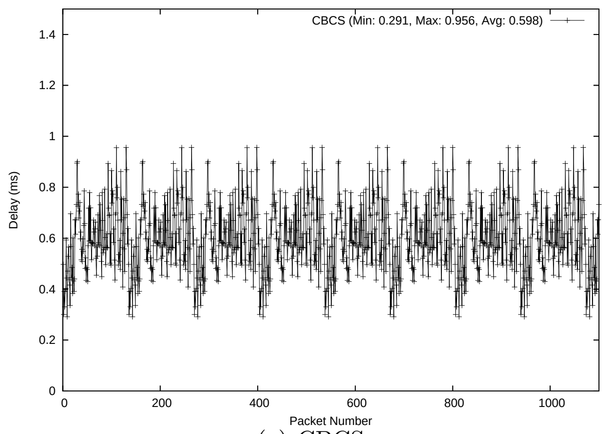


(a) CBCS

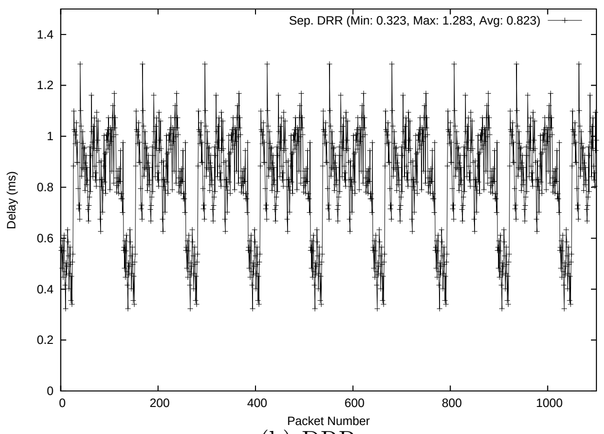


(b) DRR

Figure 10: Delay for Flows with High CPU and Low BW Requirements

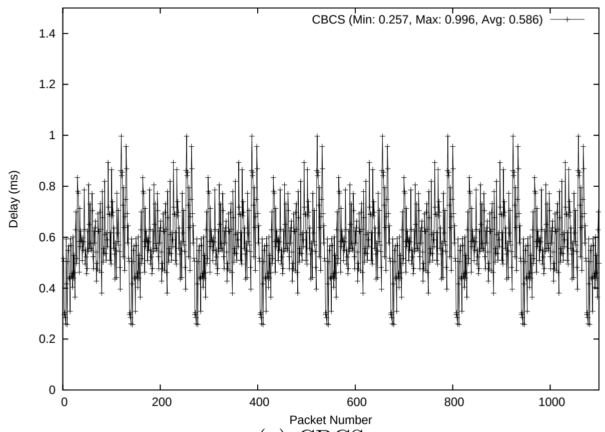


(a) CBCS

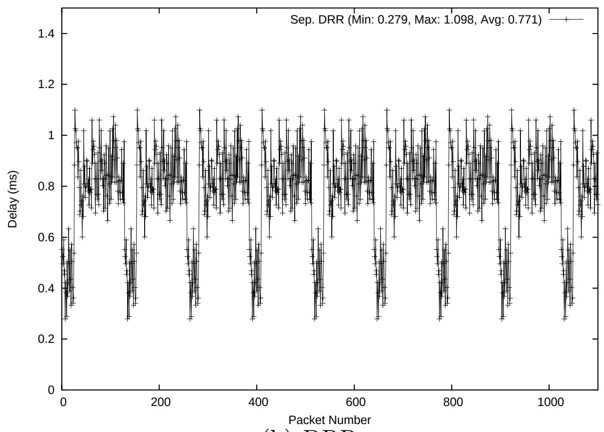


(b) DRR

Figure 11: Delay for Flows with Low CPU and High BW Requirements

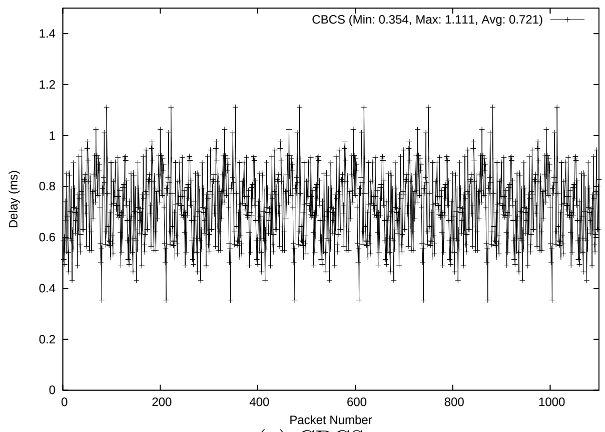


(a) CBCS

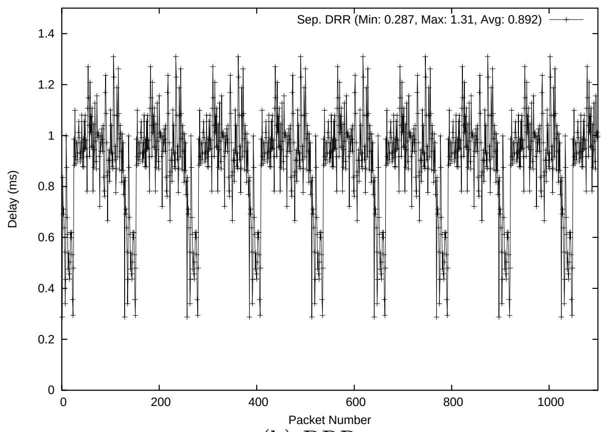


(b) DRR

Figure 12: Delay for Flows with Low CPU and Low BW Requirements



(a) CBCS



(b) DRR

Figure 13: Delay for flows with medium CPU and medium BW requirements.

Table 12: Delay statistics for flow with medium CPU and medium BW requirements.

Scheduler	Max.	Avg.	Std. Dev.
CBCS	1.111224	0.72162	0.138434
Sep. DRR	1.310104	0.892562	0.209577

[2] F. Sabrina and S. Jha, "Fair Resource Allocation in Programmable and Active Networks Using a Composite Scheduler", 7th International Symposiam on Digital Signal Processing and communication system, DSPCS'03, Coolangatta-Goldcoast, Australia, pp: 217-222, 2003.

[3] F. Sabrina, S. Jha, "Fair Queuing in Active and Programmable Networks". in Advanced Wired and Wireless Networks Series: Multimedia Systems and Applications, Vol. 26, Springer, New York, 2004. ISBN: 0-387-22781-4, Editors: Wysocki, Tadeusz A.; Dadej, Arek; Wysocki, Beata J.

[4] F. Sabrina, S. Jha, "Scheduling resources in Programmable and Active networks based on Adaptive Estimations", The 28th Annual IEEE Conference on Local Computer Networks (LCN), Bonn Germany, 20-24 Oct. 2003, IEEE, Germany, pp: 2-11.

[5] Kenneth Mackenzie, Weidong Shi, Austen McDonald and Ivan Ganev, "An Intel IXP1200-based Network Interface," in 2nd Annual Workshop on Novel Uses of Systems Area Networks (SAN-2), February, 2003.

[6] Tammo Spalink, Scott Karlin, Larry Peterson. Evaluating Network Processors in IP Forwarding. Technical Report TR-626-00, Department of Computer Science, Princeton University, November 2000.

[7] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a Robust Software-Based Router Using Network Processors. In Eighteenth ACM Symposium on Operating Systems Principles, December 2001.

[8] IXP 2400 Hardware Reference Manual, as provided with the Intel IXA SDK 3.1.

[9] IXA_EDU_Workstation_manual.pdf.

[10] Erik J Johnson and Aaron R. Kunze, IXP2400/2800 Programming - The Complete Microengine Coding Guide, Intel Press, 2003.

[11] Bill Carlson, Intel Internet Exchange Architecture and Applications - Apractical Guide to IXP2XXXX Network Processors, Intel Press, 2003.

[12] IXP2400 Framework developer manual, as provided with the Intel IXA SDK 3.1.

[13] IXA Portability Framework Reference Manual, as provided with the Intel IXA SDK 3.1.

[14] Microengine C Compiler Language Support Reference Manual, as provided with the Intel IXA SDK 3.1.

[15] Erik J Johnson and Aaron R. Kunze, IXP2400/2800 Programming - The Complete Microengine Coding Guide, Intel Press, 2003.