Implementation and Performance Analysis of a Packet Scheduler on a Programmable Network Processor

Fariza Sabrina, Salil S. Kanhere and Sanjay Jha
School of Computer Science and Engineering
The University of New South Wales, Sydney, NSW 2052, Australia
Email: {farizas, salilk, sjha}@cse.unsw.edu.au

Abstract—The problem of achieving fairness in the allocation of the bandwidth resource on a link shared by multiple flows of traffic has been extensively researched over the last decade. However, as these flows traverse a computer network, they share many different kinds of resources such as links, processor cycles, buffers and battery power, a critical resource in mobile devices. The ultimate goal should hence be overall fairness in the allocation of multiple resources rather than a single specific resource such as link bandwidth. In our earlier work we have presented a novel scheduler called Prediction-based Composite Fair Queueing (PCFQ), which jointly allocates the fair share of the link bandwidth as well as processing resource to all competing flows. Our scheme also uses a simple and adaptive online prediction scheme for reliably estimating the execution times of the incoming data packets. We have demonstrated via simulation experiments that PCFQ can provide much improved Quality of Service (QoS) guarantees as compared to separate bandwidth and processor schedulers. With the rapid increase in the capacity of transmission links, the ease with which a scheduler can be implemented in real hardware systems gains paramount importance. In this paper we concentrate on the design and implementation of PCFQ scheduler in a programmable router. We demonstrate that our scheduler can be easily implemented on an off-the-shelf network processor such as the Intel IXP 2400 board. We also validate our design by carrying out extensive experiments and demonstrate the improved performance achieved by the PCFQ scheduler. The experimental results from the IXP2400 implementation highlight the effectiveness and high performance of this algorithm in a real world system.

I. INTRODUCTION

Programmable Networks facilitate fast deployment of new services and protocols by allowing customized operations on packets in routers and switches. Although there is significant effort in deploying network programmability, the traditional network devices which are custom logic based or pure software based, could hardly satisfy both performance and flexibility requirements. So, there has been a growing interest in usage of network processor within programmable routers. Network processors are being targeted at a wide range of applications with varying packet processing and high-throughput requirements. The extensive multithreading on the IXP2400 and other network processors offers new design opportunities and challenges as well [11], [12]. In this scenario, resource scheduling becomes a more complicated issue. During the past several years, a variety of algorithms have been proposed for achieving fair allocation of the link bandwidth amongst multiple flows that share the link [2], [13]. However, most of these schemes cannot be readily adapted for processor scheduling because they require precise knowledge of the execution times for

the incoming packets at time of their arrival in the node. Start-Time Fair Queueing (SFQ) [9] (which is a bandwidth scheduling algorithm) does not use packet lengths for updating virtual time, seems to be suitable for scheduling computational resources (since it would not need prior knowledge of the execution times of packets) [9]. However, the worst-case delay under SFQ increases with the number of flows. Further, it tends to favor flows that have a higher average ratio of processing time per packet to reserved processing rate [14]. Over the years, several researchers working in the field of computer architecture have proposed schemes for scheduling access to the CPU processing cycles [8], [7], [6], but most of these proposals focus on CPU scheduling for end systems and work on task level (not at the packet level).

Pappu et al.[14] presented a processor scheduling algorithm for programmable routers called Estimation-based Fair Queueing (EFQ) that estimated the execution times of various applications on packets of given lengths offline and then scheduled the processing resources based on the estimations. Fixed values of the estimation parameters measured off-line may not always produce good estimations due to variation in server load and operating system scheduling. Gallier et al. [4] proposed a scheme to predict the CPU requirements of executing a specific code on a variety of platforms. However, the scheme seems too complicated to be implemented with programmable routers.

A few scheduling schemes have also been proposed for multiprocessors, but these are mostly simple static policies [12], [1]. [15] describes the design and implementation of the Dynamic Window-Constrained Scheduling (DWCS) algorithm for scheduling packets on network processors. However, DWCS was developed only for bandwidth scheduling. Researchers in [16] developed another scheduler called the Packet Timed Token Service Discipline (PTTSD), which allows the coexistence of RG flows (e.g.real-time flows) and classic BE (best-effort) flows and schedules bandwidth resources among the competing flows.

All the scheduling schemes discussed above are designed to schedule only a single resource, i.e. either bandwidth or processing resource. Although the determination of execution times for packets in advance on a programmable node has been identified as a major obstacle in managing processing resources [3], [4], [14], none of the previous studies provided a generalized online solution to the problem. Significance of considering the fair allocation of more than just link
bandwidth is increasingly becoming apparent today, since the link bandwidth is often not the critical resource. A router’s processor is often also a critical resource to which, ideally speaking, all competing flows should have fair access. Given the fact that processing requirements of different packets vary widely, the issue of fairness in the allocation of the processing resources gains significance. In addition, besides the fact that packet lengths can vary widely due to the presence of optional headers, there is a wide variation in the ratio of a packet’s demand for bandwidth and its demand for processing cycles. Thus, one cannot achieve overall fairness merely by fair sharing of the link bandwidth alone, or merely through fair allocation of the processing resource alone. Moreover, allocation of bandwidth and CPU resources are interdependent and maintaining fairness in one resource allocation does not necessarily entail fairness in other resource allocation. Therefore, for better maintenance of QoS guarantees and overall fairness in resource allocations for the contending flows, the processor and bandwidth scheduling schemes should be integrated.

To resolve the issues of efficiently and fairly managing multiple resources in programmable nodes our new composite scheduling algorithm, PCFQ uses a simple and adaptive online prediction scheme (SES) for determining packet execution times. In [2], we have shown through simulation that our novel idea of integrating the CPU and bandwidth scheduling functionalities within a single scheduling scheme can provide significantly better delay guarantees than those achievable through separate resource schedulers. The prior knowledge of processing requirements helps the scheduling algorithm to schedule resources adaptively, fairly and efficiently among all the competing flows. The main contributions of this paper are: (1) designing an integrated processor and bandwidth scheduling framework for a programmable router using Intel’s IXP2400 network processor and (2) demonstrating the performance benefits achieved by implementing our composite scheduler on to the Network Processors using an Intel IXP2400 network processor on a Radisys ENP2611 PCI board.

The rest of the paper is organized as follows. Section II briefly describes the PCFQ scheduling algorithm. Section III presents the details of the implementation on the Intel IXP processor. Experimental results and validation of our implementation are discussed in Section IV. Finally, Section V concludes the paper.

II. PCFQ - PREDICTION BASED FAIR QUEUING ALGORITHM

In this section we present a brief overview of the PCFQ scheduler. A more detailed description along with NS2 simulation results can be found in [2], [10]. PCFQ is designed to schedule both bandwidth and CPU resources adaptively, fairly and efficiently among all the competing flows. PCFQ is based on the notion of max-min fairness [17] which is employed in most popular resource schedulers such as Weighted Fair Queueing (WFQ) [13]. In traditional WFQ (for bandwidth scheduling) packets from different sessions are stamped with virtual finishing times and selected for scheduling based on increasing order of virtual finishing times. The virtual finish time of a packet indicates a virtual time by which the last bit of the packet must be transmitted through the link. The algorithm uses the packet size and a system virtual time (which is updated each time a new packet arrives) to compute the finishing times of the incoming packets. The virtual time is updated by using a system virtual function and its role is to compute finishing times of packets in newly backlogged sessions in order to equalize the normalized services of these sessions with current backlogged sessions. A flow is backlogged during the time interval \((t2 - t1)\) if the queue for the flow is never empty during the interval.

However, the WFQ algorithm cannot be directly used for CPU scheduling (as this would require precise knowledge of execution times for packets in advance). Further, WFQ cannot be employed for jointly allocating multiple resources. On the contrary, in PCFQ, we formulate a new system virtual function and finish time computation mechanism to incorporate scheduling of multiple resources such as bandwidth and CPU. We now present the equations used in the PCFQ scheduler:

\[
\text{sum}_{i}(t) = \sum_{i \in B(t)} (\phi_{i,cpu}^{1} + \phi_{i,bw}^{1}) \tag{1}
\]

\[
V(t + \tau) = V(t) + \frac{\tau}{\text{sum}_{i}(t)} \tag{2}
\]

\[
PP_{i}(t + \tau) = \alpha P_{i}^{1} + (1 - \alpha) PP_{i}(t) \tag{3}
\]

\[
S_{i}^{k} = \max\{F_{i}^{k-1}, V(a_{i}^{k})\} \tag{4}
\]

\[
EP_{i}^{k} = SF \cdot PP_{i}(a_{i}^{k}) \tag{5}
\]

\[
F_{i}^{k} = S_{i}^{k} + \frac{EP_{i}^{k}}{\phi_{cpu}^{i}} + \frac{8L_{i}^{k} \gamma_{i}^{1}}{10^{6}(\phi_{bw}^{i} \cdot BW)} \tag{6}
\]

where,

- \(\phi_{cpu}^{i}, \phi_{bw}^{i}\) = Weights for CPU and Bandwidth for flow \(i\).
- \(B(t)\) = Set of backlogged flows at time \(t\).
- \(\text{sum}_{i}(t)\) = Summation of the weights (both CPU bandwidth) of all the active flows at time \(t\).
- \(V(t)\) = System virtual time at time \(t\).
- \(\tau\) = Time difference between two virtual time updates, i.e., inter-packet arrival time.
- \(S_{i}^{k}\) = Virtual start time of packet \(k\) of flow \(i\).
- \(V(a_{i}^{k})\) = System virtual time at the arrival of pkt \(k\) of flow \(i\).
- \(F_{i}^{k}, F_{i}^{k-1}\) = Virtual finish time of pkt \(k\) and \((k - 1)\) of flow \(i\).
- \(EP_{i}^{k}\) = Estimated processing cost of packet \(k\) of flow \(i\) in sec.
- \(L_{i}^{k}\) = Length of packet \(k\) of flow \(i\) in bytes.
- \(\gamma_{i}^{1}\) = Resize factor of the packets in flow \(i\).
- \(BW\) = Transmission link bandwidth in Mbps.
- \(P_{i}^{1}\) = Actual processing cost of the packet in flow \(i\) that is
processed at time $t$.

$PP_i(t) = \text{Predicted processing cost of a packet in flow } i \text{ at time } t.$

$PP_i(a^k_i) = \text{Predicted processing cost of a packet in flow } i \text{ at the arrival of packet } k \text{ in the flow}.$

$\alpha = \text{SES parameter, which was set to 0.4.}$

$SF = \text{Scaling Factor}.$

$SF = 1$ and $\frac{L_{i+1}}{T_{i+1}}$ for header and payload processing applications respectively.

$L_{i+1} = \text{Length of a packet arriving at time } t \text{ and } t+1 \text{ respectively}.$

The scheduler maintains per-flow variables for storing the values of $\gamma_i, PP_i(t)$ and so on. When the scheduler is initialized, $V(t)$ and $PP_i(t)$ (for all flows) are set to zero and $\gamma_i$ for all flows is set to one. With the arrival of every new packet, the scheduler updates $V(t)$, and calculates the virtual finish time of the packet ($F^k_i$) using Eq. (1 - 6).

The scheduler then stores the packet in the corresponding flow queue. While any packet exists within the queues, the dequeue method of the scheduler checks the finish times of the packets at the head of all active flow queues and dequeues the packet (from the head of the queue) having the minimum finish time. It then hands the packet to the processor handler object, which in turn processes the packet and enqueues it in the FIFO transmission queue. After each packet for a flow $i$ is processed, the scheduler updates the predicted processing time for the next packet for the same flow $i$ using Eq. (3). Also, $\gamma_i$ is updated after a packet is processed.

III. IMPLEMENTATIONS OF PCFQ ON IXP2400

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

A. 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.

B. PCFQ Implementation Architecture

We have implemented the PCFQ scheduler on a single microengine, because it will not be efficient to run the enqueue and dequeue method of the same scheduler in different micro-engines. The implementation architecture of the scheduler is shown in Figure 1. 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). It may be mentioned that as the microengine C compiler does not support floating point calculations, the calculations of the PCFQ schedulers are approximated.

C. Separate CPU and Bandwidth schedulers

As mentioned earlier, we have also implemented a set of separate WFQ schedulers for scheduling the CPU and bandwidth separately on the IXP2400 processor. This allows us to evaluate the performance of the PCFQ scheduler compared to using separate CPU and bandwidth schedulers. Figure 2 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 1. 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.

D. Data Structures and Inter-microengines Messages

The communications between different microengines are done through some pre-defined messages. 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.

E. PCFQ Implementation Details

We have also used microengine local memory for keeping PCFQ scheduler variables such as System Virtual Time (SVT), Last Virtual Time Update Time (LVTUT), virtual finish time of the last packet of each flow (lastVFT[i]), CPU weight per flow (CPUwt[i]), bandwidth weight per flow (BWwt[i]), B variable (i.e., active flow indicator) per flow (B[i]), sum of weights for all the active flows (sum) etc. Local memory was used because it is the fastest to access and it was sufficient 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.

PCFQ serves a packet from the head of a queue where the packet’s virtual finish time is minimum compared to the virtual times of all the other head of queue packets (of the other flow queues). In our implementation of PCFQ, virtual finish time is calculated during arrival of a packet and it is stored in the 7th long word of the metadata of the packet in the SRAM. During dequeue, PCFQ scheduler needs to peek the virtual finish time of the head of queue packets for all the active flows to decide which packet to serve. But SRAM queues and the available SRAM API have some constraints regarding peeking the head of queue packet information. The SRAM library does not provide any API to peek the head of queue, rather it simply provide API to dequeue the head of queue from a specified queue. Meaning, we need to dequeue packets from head of all queues to decide which packet to serve and after serving a packet among the dequeued packets we need to store somehow the other dequeued packets information as the dequeued (but not served) packets cannot be put back in the flow queues (as it will break the packet orders).

Head of Queues Packet Cache: In order to resolve the above mentioned constraints of the SRAM hardware queue on implementing the PCFQ scheduler, we have created a head of queues packet cache in the local memory. The head of queues packet cache is an array of structure where each structure element (HOQCache[i]) contains two members: dequeued buffer handle from the head of a queue, and scheduler data (i.e. the virtual finish time). During the process of serving a packet, the PCFQ scheduler first updates the head of queues packet cache with all the active flow’s packet data and then serves the packet from the cache.

The PCFQ scheduler is also implemented using 4 threads e.g., initialization thread, CPU prediction thread, enqueue thread, and dequeue thread. After initialization is completed, the initialization thread sends a signal 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 SVT, LVTUT, sum, and the per flow variables (e.g., lastVFT[i], CPUwt[i], BWwt[i], B[i], HOQCache[i]) etc.). After initializing the scheduler variables, the thread terminates itself so that the microengine thread arbiter excludes this thread from its list.

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 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.

Enqueue Thread: Figure 3 shows a simplified flow diagram of works performed within the PCFQ enqueue thread. During enqueue of each packet, the scheduler updates the SVT and LVTUT, and it checks whether the flow is just becoming active by checking the B variable (i.e. B[i]). If B[i] were zero, it set B[i] to 1 (indicating the flow is now active) and then updates sum (i.e. it adds the CPUwt[i] and BWwt[i] to sum).

Dequeue Thread: Figure 4 shows the simplified flow diagram of the activities performed within the PCFQ dequeue thread. As shown in the figure, dequeue thread waits for the signal from initialization thread before starting its infinite loop. In each PCFQ loop, the algorithm first updates the HOQCache for all the active flows, and then serves a packet from the HOQCache and the loop continues.

During updating the HOQCache for each individual active flow i, the algorithm first checks whether the HOQCache[i].bufHandle is 0 or not. If its not 0, it means
that it does not need to dequeue a new packet from the SRAM queue for flow i as it was done before in one of the previous cache update operation (but it was not served by the dequeuer loop as its Virtual Finish Time or VFT was not minimum). If HOQCache[i].bufHandle is 0, the algorithm calls a SRAM API (e.g., sram_dequeue) to dequeue a packet handle from SRAM queue corresponding to flow i and waits for the read to complete. Once it receives the read completion signal, it checks the validity of the dequeued buffer handle and if its valid then it calls another SRAM API (e.g., sram_read) to read the VFT of the packet from SRAM and waits again for reading completion. After reading is completed, it sets the value of the members of HOQCache[i] and then goes for updating the cache for next active flow (i.e., i + 1).

While serving a packet from the HOQCache within each dequeuer thread loop, the algorithm looks up the cache and tries to locate the flow i where the HOQCache[i].bufHandle is not 0 and also the HOQCache[i].schedData value is minimum among the schedData values of all the valid members in the cache. If the algorithm does not find any valid member in the cache then it swaps out and then continues for the next dequeuer loop (i.e., goes to update the HOQCache). But if it finds a valid member i in the cache, then it generates a scheduler-to processor message using the values of the flow id and the buffer handle. It then resets the valid member in the cache and also the HOQCache[i].bufHandle and HOQCache[i].schedData to 0, and decrements the total packet count. If it finds that the packet count in the flow i is zero then it resets other scheduler variables (e.g. it sets as B[i] = 0, lastVFT[i] = 0, and sum = sum − CPUwt[i] − BWwt[i]). The algorithm then sends the message to the processor using scratchpad ring 2 (SR-2), swaps out, and then continues the dequeue loop.

IV. Experiments and Results

In this section we present the results of our comparisons of the PCFQ scheduler with the separate WFQ schedulers. The experiments were performed by running the code on IXA workbench's "Cycle Accurate" transistor. The port logging options were turned on to log the packets received and transmitted at the media interfaces.

A. 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 I shows the CPU requirements and packet sizes for each individual flows. For all
In this paper, we present the implementation and performance analysis of a composite scheduler for dynamically scheduling multiple resources in network processor-based routers. The performance results strongly support the benefits of our composite scheduler to better support computing along the forwarding path without sacrificing efficiency. It should also be noted that implementing the composite schedulers (which are able to schedule both the CPU and bandwidth resources) in the IXP2400 environment is much simpler than implementing two separate schedulers for CPU and bandwidth resources because of the IXP2400 hardware architecture and the design of the IXA portability framework. The delay improvements achieved by PCFQ in IXP2400 implementations are quite similar to that achieved in our NS2 simulators (due to space limitation we could not present those simulation results here). Interested readers are referred to [2]). In designing PCFQ scheduler, we mainly focused on single output port. In 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 5) were used.

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 to inject the data frames.

### B. Experimental Results: PCFQ vs Separate WFQ schedulers

Experiments were performed and packet logs were collected while using both the PCFQ scheduler and the separate WFQ 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. 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). The delay results obtained are presented in this section.

The results show that PCFQ provides better delay guarantees as compared to the separate WFQ schedulers. The maximum and average delays (in msec) for these flows are shown in table II. The results show that by using PCFQ, the worst case delay is reduced to 4% for flows with high CPU and high BW, 9% for high CPU low BW scenario, 24% for low CPU high BW scenario, 20% for low CPU and low BW scenario, and 12% for data flow with medium CPU and medium BW requirement compared to the delays achieved using WFQ. The average delay is reduced to 13% for flows with high CPU and high BW, 12% for high CPU low BW scenario, 17% for low CPU high BW scenario, 21% 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 WFQ. Table II also shows that the PCFQ provides smaller standard deviation of the delay compared to that using WFQ in most cases. Smaller standard deviations provided by PCFQ means it provides much more consistent delay guarantees compared to that achieved using WFQ.

Figures 7 – 10 also show that PCFQ provided better delay guarantees than WFQ because under this dynamic scenario (where packet sizes varied significantly in all the flows and the processing requirement also varied significantly), the maximum total cost of a packet within all the flows is lower than the summation of the maximum processing cost of a packet and maximum transmission cost of a packet in all the flows. This was because of the maximum normalized total cost of a packet within a flow is less than or equal to the summation of the maximum normalized processing cost and maximum normalized transmission cost of a packet in the same flow.

### V. Conclusion

In this paper, we present the implementation and performance analysis of a composite scheduler for dynamically scheduling multiple resources in network processor-based routers. The performance results strongly support the benefits of our composite scheduler to better support computing along the forwarding path without sacrificing efficiency. It should also be noted that implementing the composite schedulers (which are able to schedule both the CPU and bandwidth resources) in the IXP2400 environment is much simpler than implementing two separate schedulers for CPU and bandwidth resources because of the IXP2400 hardware architecture and the design of the IXA portability framework. The delay improvements achieved by PCFQ in IXP2400 implementations are quite similar to that achieved in our NS2 simulators (due to space limitation we could not present those simulation results here). Interested readers are referred to [2]). In designing PCFQ scheduler, we mainly focused on single output port.
Fig. 6. Delay for Flows with High CPU and High BW Requirements

Fig. 7. Delay for Flows with High CPU and Low BW Requirements

Fig. 8. Delay for Flows with Low CPU and High BW Requirements

our future work, we intend to extend our implementation for multiple ports.

Acknowledgement

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

REFERENCES


