Fair Resource Scheduling for QoS Aware Collaborative Services on the Internet

Fariza Sabrina
Networking Research Lab
ICT Centre, Commonwealth Scientific and Industrial Research Organisation (CSIRO)
Sydney, Australia
Email: Fariza.Sabrina@csiro.au

***Abstract*— The popularity and availability of Internet connection have accelarated the emergence of new idea for network-centric collaborative works. Contending traffic flows in this collaborative scenario share different kinds of resources such as network links, buffers, and router CPU. The goal should hence be overall fairness in the allocation of multiple resources rather than a specific resource. Moreover, conventional resource scheduling algorithms depend strongly upon the assumption of prior knowledge of network parameters and cannot handle variations or lack of information about these parameters. In this paper, *firstly*, we present a novel QoS-aware resource scheduling algorithm called *Weighted Composite Bandwidth and CPU Scheduler (WCBCS)*, which jointly allocates the fair share of the link bandwidth as well as processing resource to all competing flows. WCBCS also uses a simple and adaptive online prediction scheme for reliably estimating the processing times of the incoming data packets. *Secondly*, we present complexity analysis, extensive NS-2 simulation works, and experimental results from our implementation on Intel IXP2400 network processor. The simulation and implementation results show that our low complexity scheduling algorithm can efficiently maximise the CPU and bandwidth utilisation while maintaining guaranteed Quality of Servie (QoS) for each individual flow.**

I. Introduction

The popularity and availability of Internet connection has opened up the opportunity for network-centric collaborative work that was impossible a few years ago. Different network centric entertainment applications such as networked online games, multimedia streaming (video and audio) to heterogenous system in collaborative environment, Network-centric Music Performance or NMP where Internet is used as rehearsal room by a number participants [1] etc. are becoming very popular. Recent development of Internet technology has opened up new opportunities for enterprises too. Dynamic collaboration service in virtual enterprise scenario where multiple participants join in a secured audio/video conferencing session and record and store the whole session for future reference is one example of collorative work in Enterprise scenario.

Provisioning smart, efficient, dynamic collaborative service has drawn huge interest from industries and as a result has become a challenging research issue. Many of these applications in collaborative environment need processing the packet upon arrival, before transmitting it to the clients ([1], [2]). Many applications have strict delay bound where as others are intolerable of packet loss [2]. The best effort Internet, with no guarantee of network capacity or packet delivery, is a challenge for the real time interaction required for most of these collaborative services. Efficient resource allocation in such a system is an important and fundamentally complicated problem. In order to satisfy QoS requirements of various applications the node must control the use of network and processing resources by properly scheduling them. The system must ensure that all the flows receive their reserved resources while QoS is also maintained. To ensure this, there must be mechanisms to give guaranteed bandwidth and computational resources to incoming flows. However, 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.

A significant amount of work has been done in bandwidth resource scheduling for traditional network. Packet Fair Queueing (PFQ) disciplines such as WFQ and WF^{2}Q [3] provide perfect fairness among contending network flows. However, WFQ and WF^{2}Q cannot readily be used for processor scheduling because they require precise knowledge of the execution times for the incoming packets at time of their arrival in the node. Moreover, the work complexity of these algorithms are $O(N)$, where N is the number of flows sharing the link. Another PFQ algorithm for bandwidth scheduling is Start-Time Fair Queueing (SFQ) [4], which does not use packet lengths for updating virtual time, and therefore seems suitable for scheduling computational resources (since it would not need prior knowledge of the execution times of packets) [4]. However, the worst-case delay under SFQ increases with the number of flows and it tends to favor flows that have a higher average ratio of processing time per packet to reserved processing rate [5]. One algorithm called deficit round robin (DRR) achieves fair scheduling with $O(1)$ complexity. But it is used only for link bandwidth scheduling.

A large amount of work has also been done on CPU scheduling [6], [7], but most of them are on CPU scheduling for end systems and work on task level (not on packet level). Moreover, the execution times of various applications on packets are not known in advance, thus constraining efficient and

fair processor scheduling algorithms, which in turn limits the applicability of well-known bandwidth scheduling algorithms and also makes explicit or implicit admission control at the flow level more difficult.

Pappu et al. [5] 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 off-line 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. Galtier et al. [8] proposed a scheme to predict the CPU requirements of executing a specific code on a variety of platforms. Doulamis et. al. [9] used least square algorithm to predict task work load and used this information for resource scheduling in grid computing. However, all these schemes seem too complicated to be implemented in routers.

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 or active node has been identified as a major obstacle in managing processing resources [8], [5], none of the previous studies provided a generalized online solution to the problem.

Unlike all the previous works, our work takes an integrated approach and provides a composite scheduler for both bandwidth and CPU scheduling in order to provide better QoS guarantees to the contending data flows. In our previous work [10], [11], we presented a composite bandwidth and processor scheduler called *Composite Bandwidth and CPU Scheduler (CBCS)*, which can schedule multiple resources adaptively, fairly and efficiently among all the competing flows. Detailed simulation, analytical and experimental work presented in [10] proves 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.

Although, CBCS has high efficiency but it was developed only for best effort flows and does not ensure flow differentiation. In this paper, we present our new composite scheduling algorithm called Weighted Composite Bandwidth and CPU Scheduler (WCBCS), which is the extended version of CBCS to make it suitable for QoS flows. The novelty of this algorithm is, (1) it schedules multiple resources in a single algorithm, (2) it employs a simple and adaptive online prediction scheme called modified Single Exponential Smoothing for determining the packet execution times. (3) it is suitable for QoS flows where flows with different reserved rate are weighted differently, but unlike other scheduling algorithm of similar capability WCBCS has very low work complexity ($O(1)$), making it attractive for implementating in high-speed routers.

The paper is organised as follows: Section II presents the proposed algorithm. Section III describes details of the simulation set-up and analyses the performance of the scheduler through simulation. Section IV presents some experimental results and conclusions are drawn in Section VI.

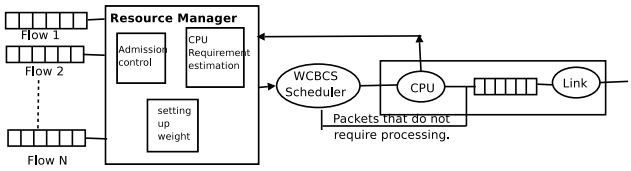


Fig. 1. System Model for WCBCS Scheduler.

II. WCBCS - Weighted Composite Bandwidth and CPU Scheduler

This section describes the WCBCS scheduler and prediction technique used to estimate the packet processing duration.

A. Online Prediction Process

Since the processing requirement of each packet is not known apriori, the WCBCS scheduler needs to estimate the processing duration for each arriving packet. We have investigated several smoothing methods and their suitability for predicting the processing requirements of the packets. A detailed analysis, investigations and a comparative performance analysis of the alternatives are discussed in [11]. Our investigations show that the Single Exponential Smoothing (SES) technique is well-suited to estimate the execution times of the packets. SES is computationally simple and an attractive method of forecasting. SES uses the following equation to calculate a new predicted value.

$$F_{t+1} = \alpha X_t + (1-\alpha)F_t \qquad where, 0 \leq \alpha \leq 1 \qquad (1)$$

where, F_t and F_{t+1} are the predicted value at t^{th} and $(t+1)^{th}$ period respectively. X_t is the actual duration required to process the packet that arrived at time t, and α is the SES coefficient which determines the relative weight allocated to the history and the current estimated sample. In our work, the SES coefficient, α, was set to 0.4, based on earlier experimentation, which indicates that a value of 0.4 provides the most accurate results.

Most of the packets that are processed by today's routers can be broadly classified into two categories based on their processing needs: (a) header processing and (b) payload processing. Header processing application (i.e., IP forwarding) only requires read and write operations in the header of the packet and so the processing complexity is independent of the size of the packets. In contrast, payload processing application (such as IPSec Encryption, packet compression and packet content transcoding etc.) involves read and write operations on all the data in the packet, and therefore the processing complexity strongly correlates to the packet size [5]. In order to count for the correlation between the processing costs and packet sizes, we define a parameter called *Scaling Factor (SF)*, as: $SF = 1$ for header processing packets and $\frac{L_{t+1}}{L_t}$ for payload processing packets. Here, L_t, L_{t+1} = Length of

the packet arriving at time t & $(t+1)$ respectively. The scaling factor is incorporated in the SES estimation as follows:

$$F_{t+1} = SF\{\alpha X_t + (1-\alpha)F_t\} \quad (2)$$

B. Setting up the flow weights

WCBCS schedules guaranteed rate connection by assigning weight to each flow. Here we describe how the weight was set for each flow. Let ϕ_c^i and ϕ_b^i are the weights for CPU and bandwidth for flow i. If r_c^i and r_b^i are the reserved CPU and bandwidth for flow i respectively, and r_c^m and r_b^m are the minimum reserved CPU and bandwidth among all flows respectively. The weights are assigned as below: $\phi_c^i = \frac{r_c^i}{r_c^m}$ and $\phi_b^i = \frac{r_b^i}{r_b^m}$.

C. Overview of the WCBCS algorithm

We first begin by briefly describing the system model used in our work. The Resource Manager (as shown in Figure 1) controls the flow registration and setup (including setting up weights for any reserved flows based on the reserved rates of the bandwidth and processing resources) and admission of each individual packet. The resource manager also estimates the CPU requirements of an individual flow based on the feedback received from the processor handler. The scheduler enqueues them in the corresponding flow queues and dequeues packets using its composite scheduling algorithm WCBCS, which takes both the estimated processing time and transmission time of packets into account to decide which packet to dequeue. After dequeueing a packet, the scheduler hands the packet to the processor handler object for processing if required. The processor handler object notifies the Resource Manager after processing each packet so that the scheduler can re-estimate the processing times for the new incoming packets. The processing and transmissions of different packets happen in parallel in the system, i.e., after processing, the packets enter into a FIFO queue for transmission to their next destinations, which is served by a separate thread. The packets that do not require processing enter directly into the transmission queue after the scheduler dequeues them from their flow queues. Packets from each flow are first processed by the processor and then transmitted to the output link. The joint allocation of the processing and bandwidth resource is accomplished by the composite scheduler which selects a packet from the input buffers and passes it onto the CPU for processing. No scheduling action takes place after the processing; the packets processed by the CPU are stored in the buffer between the processor and the link, and are transmitted in a first-come-first serve order. The WCBCS scheduler is based on the principles used in DRR [12] . Further contrary to the single resource schedulers, WCBCS is designed to schedule both bandwidth and CPU resources adaptively, fairly and efficiently among all the competing flows. It succeeds in eliminating the unfairness of pure packet-based round-robin by maintaining a *Credit Counter* to measure the past unfairness. *Credit Counter* is similar to the variables *Deficit counter* used by single resource schedulers such as DRR [12]. All backlogged flows are stored in a linked list and the flows are served in a round-robin order by the WCBCS scheduler. The algorithm uses the following parameters and equations:

- BW = Bandwidth of the transmission link in Mbps.
- 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 bits.
- γ^i = Resize factor of the packets in flow i.
- c_t^{UL}, c_t^{LL}= Upper and lower limit (respectively) of the total CPU queue in terms of CPU processing time requirement for all the packets in all the flows.
- Ptc_i^k= the combined processing and transmission cost for the k^{th} packet of flow i.
- Ptc^{Max} = The maximum allowable time slice that a packet requires to cover both CPU processing and network transmission, amongst all flows (total cost). Therefore,

$$Ptc_i^k = \frac{10^6 * \gamma^i * L_i^k}{BW} + EP_i^k \quad (3)$$

- $Quantum$ = A variable that represents a time slice used to serve packets from each flow queue, which includes both CPU processing time and network transmission time (in msec). If w^i is the summation of the weights (both CPU & bandwidth) of flow i and w^m is the smallest summation of the weights among all flows. The *Quantum* of each flow is calculated as $\frac{w^i}{w^m} * Ptc^{Max}$. Let $Q(r)$ denote the quantum in round r.
- $CC[i]$ = Credit Counter, a state variable that represents a time slice for which flow i deserves to be served within a specific round of scheduling (in msec). Let $CC_r[i](r)$ represent the Credit Counter for flow i in round r.

On receiving a new packet, the scheduler examines the header to determine the flow-id, calculates the CPU processing time using the online prediction scheme discussed earlier and the resize factor, and then stores the packet in the corresponding flow queue. It may be worth noting that processing of packets can also affect the sizes of the packets after processing is completed. To take this packet size change into account for bandwidth consumption, we define and calculate a resize factor that is the average packet size after processing for an individual flow divided by the average packet size before processing for that flow. The composite scheduler maintains a resize factor for each flow and its value is updated online.

The WCBCS scheduler continues to monitor the queue length for all individual flows in terms of the CPU time requirement and stops accepting packets from a flow if its queue length becomes greater than $c_{t[i]}^{UL}$. In this case the scheduler continues to refuse new packets from flow i until its queue length becomes smaller than $c_{t[i]}^{LL}$.

Upon initialisation, the $Quantum$ is set to $\frac{w^i}{w^m} * Ptc^{Max}$ and the Credit Counter $(CC[i])$ for all flows are set to zero. The scheduler continues to serve all non-empty queues within each round of processing. When it starts to serve a queue within a round, the Credit Counter is set to $Quantum$ plus the Credit Counter of the previous round. The scheduler then dequeues

a packet from the head of the queue and calculates the Ptc_i^k of the packet according to the Eq.(3). It sets the $CC[i]$ to ($CC[i]$ - Ptc_i^k) and hands the packet to the processor Handler object for execution. The packet is sent to its next destination after processing. The scheduler stops serving a queue once the queue is empty or the credit counter becomes zero or negative. It may be noted that the $CC[i]$ for a non-active flow (i.e., a flow having no packets in the queue) is reset to zero.

D. Work Complexity

The work complexity of a scheduler is defined as the order of time complexity with respect to enqueuing and then dequeuing a packet for transmission.

Theorem 1: The worst-case work complexity of the WCBCS scheduler is $O(1)$.

Proof: The enqueue operation consists of determining the flow at which the packet arrives and adding the flow to the linked list if it is not already in the list. Both of these operations are $O(1)$. The dequeue procedure involves determining the next flow to be served, calculating the credit counter and removing the flow from the active list. All of these can be done in constant time, so we can say that dequeue operation is of time complexity $O(1)$. As the complexity of both the enqueuing and dequeuing tasks is $O(1)$. it follows that the work complexity of the WCBCS scheduler is $O(1)$. ■

III. Simulation Results

This section presents simulation results on the delay characteristics and the fairness properties of the WCBCS scheduler. We compared the performance of WCBCS with WFQ, which is well known for its good delay and fairness behaviour and currently used in the routers.

A. Simulation settings

The simulations were performed using the NS2 network simulator [13] on a PC with 1.7 GHz Pentium M processor and 1012 MB memory running the Linux operating system (Ubuntu 5.10). Our simulation model consists of 30 UDP flows sharing a single processor and a link. The simulation settings of the individual flows are given in Table I. The output link capacity was set to 10 Mbps. The simulation was run for 300 seconds and samples were collected at 1-second intervals. The packet generation rates for all the flows were adjusted such that the cumulative demand for the CPU and bandwidth resources were 92% and 94% respectively. This ensures that the measured delays reflect the performance of the scheduler and are not affected by large queuing delays. We compare the performance of WCBCS with an implementation consisting of separate WFQ schedulers for CPU and bandwidth scheduling. We assume that the WFQ CPU scheduler uses the same online prediction scheme SES, used by WCBCS for estimating the processing duration of each packet.

TABLE I

Settings for Individual Flows

Flow Number	1-10	11-20	21-30
Referenced Application	MPEG2 Encoder	RC2 Decryption	RC2 Encryption
Data Size	1.5-24 KB	16 KB	4 KB
CPU Requirements per data block	10-40msec	1-3 msec	1-3 msec
Resize Factor	0.11 -0.36	0.25	4.0

B. Delay Measurements

Fig. 2 shows the delays experienced by the packets of MPEG2 flows using WCBCS and separate WFQ schedulers. The maximum and average delays and the standard deviation of the delays for all three flows are shown in Table II. Our results show that WCBCS achieves better delay characteristics compared to separate WFQ schedulers for CPU and bandwidth scheduling for flows with variable packet size and CPU requirements. With WCBCS, the worst case delay is reduced to 16% for MPEG2 and was more or less same for both RC2 decryption and encryption data flow compared to the delays achieved using WFQ. Also the average delay was reduced to 33% for MPEG2 data flow, and was same for both RC2 decryption and RC2 encryption data flows compared to the average delays achieved using WFQ. Similar improvements are observed in the standard deviation achieved with WCBCS, implying that it provides much more consistent delay guarantees than that with separate WFQ schedulers. Note that in our simulation scenario the packet sizes and the processing requirements varied significantly for MPEG2 flows, which is typical of real network traffic. The improved delay performance of WCBCS can be attributed to the fact that the maximum total cost of scheduling a packet with WCBCS (for CPU and bandwidth) is lower than the maximum processing cost of a packet and maximum transmission cost of a packet with separate WFQ schedulers.

TABLE II

Delay Measurements

Data flow	Delay using WCBCS (in sec)			Delay using WFQ (in sec)		
	Max delay	Avg delay	Standard deviation	Max delay	Avg delay	Standard deviation
MPEG2	1.6	0.2	0.2	1.92	0.3	0.3
Decryption	0.6	0.1	0.1	0.4	0.1	0.29
Encryption	0.5	0.1	0.1	0.5	0.1	0.1

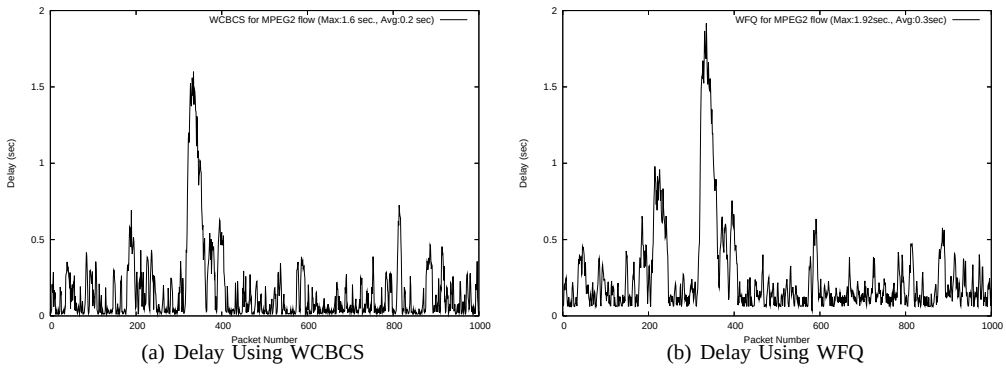
TABLE III

Fairness Measurements

Data flow	w_{cpu}^i	w_{bw}^i	Utilized CPU Rates		Utilized BW Rates	
			WCBCS	WFQ	WCBCS	WFQ
MPEG2	0.068	0.002	0.0689	0.0689	0.002	0.002
RC2 Decryption	0.016	0.02	0.0156	0.0156	0.02	0.019
RC2 Encryption	0.016	0.078	0.0157	0.0157	0.078	0.079

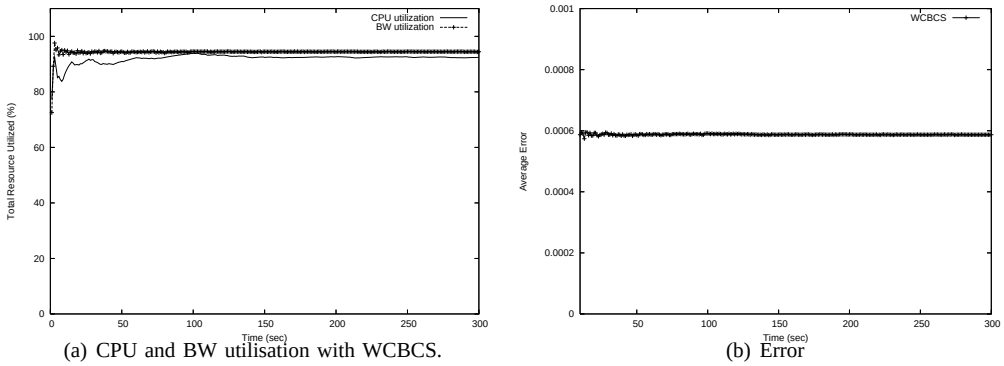
C. Fairness Measurements

To evaluate the fairness characteristics, we measure the CPU and bandwidth utilized by all the flows and compare them with the reserved rates. The results are summarized in Table III, it shows that the fairness achieved by WCBCS is better than that achieved by separate WFQ schedulers.



(a) Delay Using WCBCS (b) Delay Using WFQ

Fig. 2. Delay for MPEG2 flows using WCBCS and WFQ.



(a) CPU and BW utilisation with WCBCS. (b) Error

Fig. 3. Resource utilisation and Error in Fairness

D. Resource utilisation

We measured the CPU and bandwidth utilisations for all the flows for the entire simulation period. Resource utilisation results for WCBCS and WFQ were comparable (CPU utilisation was 92% and BW utilisation was 94%). Figure 3(a) shows the resource utilisation while using WCBCS.

E. Error Calculation

Here we present the analysis of error in maintaining fairness in resource allocation among contending flows. We measured the error in maintaining fairness in resource allocation for any backlogged flow i for the time period $(t2-t1)$ as:

$$Error_i^{(t2-t1)} = \{Dev_i^{(t2-t1)}\}^2 \tag{4}$$

Here, $Dev_i^{(t2-t1)}$ is the deviation from the ideal resource allocation for any flow i. The error in maintaining fairness in resource allocation for any backlogged flow i for the time period $(t2-t1)$ is measure as

$$Error_{avg}^{(t2-t1)} = \sum_{i=1}^{N} \frac{Error_i^{(t2-t1)}}{N} \tag{5}$$

Here, N is total number of flows. We measured the error (i.e., deviation from the ideal situation) as defined in Eq. (5) for all the flows and the results are shown in Figure 3(b). The average error for each flow was recorded as 0.0006 using WCBCS. It proves that using WCBCS the achieved results did not deviate noticeably from the expected ideal situation.

IV. Implementations of WCBCS on IXP2400

This section presents implementation details of WCBCS on IXP2400. We have developed a data plane application for IXP2400 network processor and have implemented both the WCBCS 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 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

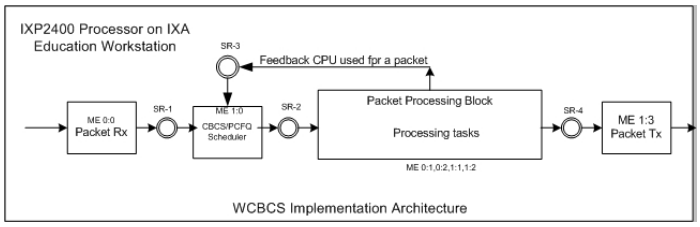


Fig. 4. WCBCS implementation architecture.

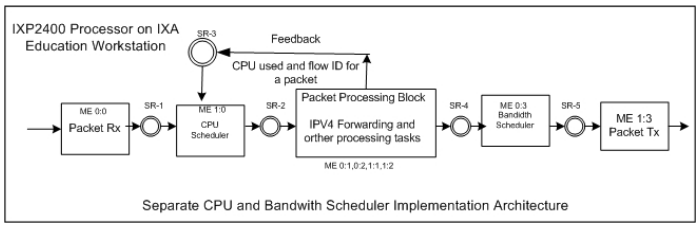


Fig. 5. Separate CPU and Bandwidth scheduler implementation architecture.

application building blocks that can be used for developing applications for IXP 2400 network processor.

B. WCBCS Implementation Architecture

The implementation architecture of the schedulers is shown in Figure 4. 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).

C. Separate CPU and Bandwidth schedulers

As mentioned earlier, we have also implemented a set of separate WFQ schedulers for scheduling CPU and bandwidth separately on the IXP2400 processor in order to evaluate the performance of the WCBCS scheduler compared to using separate CPU and bandwidth schedulers. Figure 5 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 4. 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. We have used dispatch loop functions to write some data like total resource requirement for a packet (for WCBCS 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.

E. WCBCS Implementation Details

We have used microengine local memory for keeping WCBCS 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.

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

1) Initialisation: Initialisation 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.

2) Enqueue: Figure 6 shows a simplified flow diagram of works performed within the WCBCS 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.

If the thread receives an invalid message, it does context

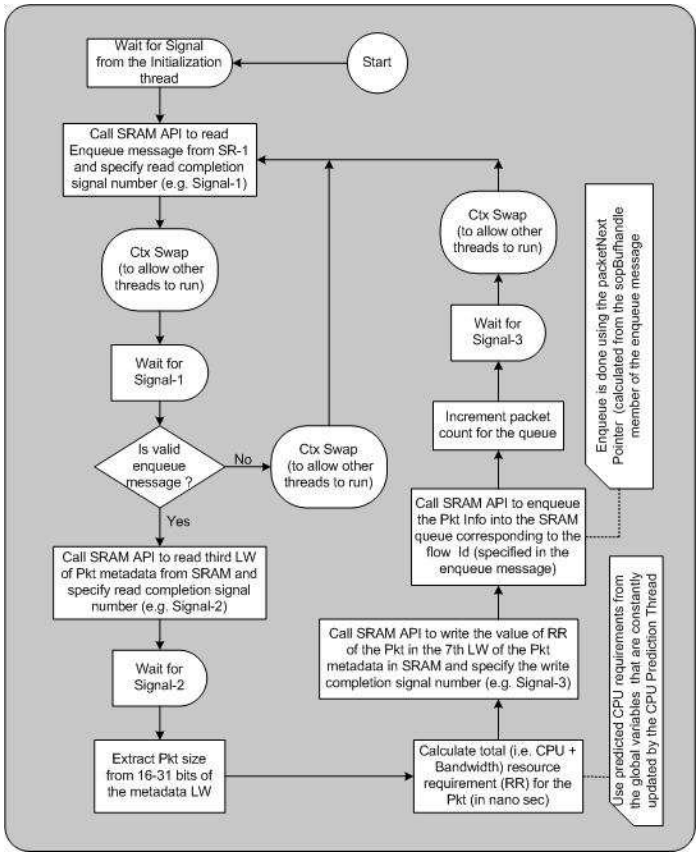


Fig. 6. Flow diagram of the WCBCS enqueue thread.

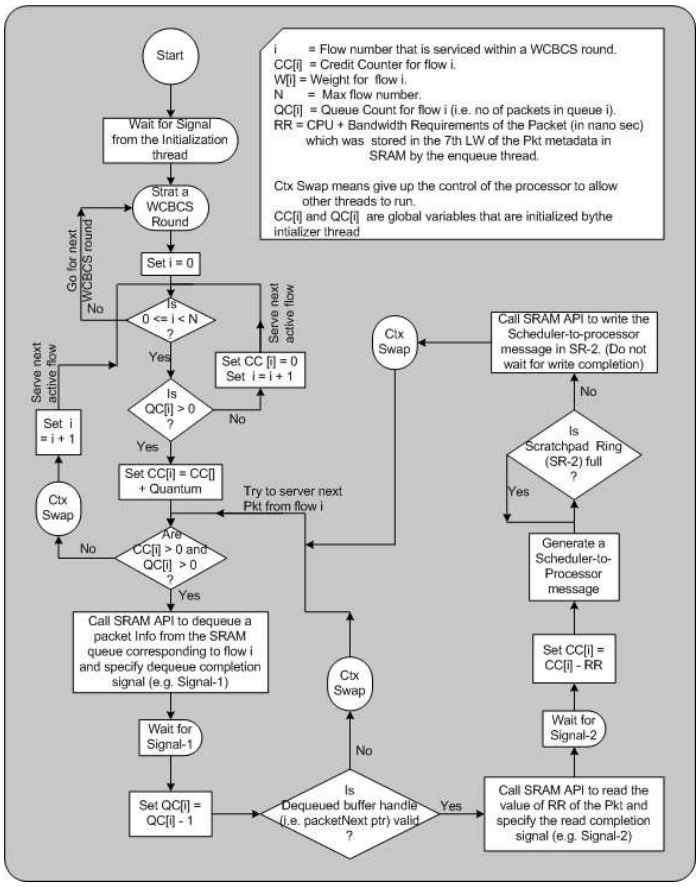


Fig. 7. Flow diagram of the WCBCS dequeue thread.

swap and then goes for the next turn. 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.

The total resource requirement (RR) for the incoming packets is calculated in nano seconds (ns) using the following equation.

RR= CPU Cost (ns) + Transmission cost (ns) of the packet $=$ 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 compared to the value of RR and it happens for some of the packets for all flows.

3) Dequeue Thread: Figure 7 shows the simplified flow diagram of the activities performed within the WCBCS dequeue thread. As shown in the figure, dequeue thread waits for signal from initialization thread before starting its infinite loop. In each WCBCS 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 WCBCS 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 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.

The enqueue operation consists of determining the flow at which the packet arrives and adding the flow to the linked list if it is not already in the list. Both of the operations are O(1) operation. The Dequeue procedure includes determining the next flow to be served, calculating the credit counter, removing the flow from the active list. All of these can be done in constant time, so we can say that dequeue operation is of time complexity O(1). As the time complexity of enqueuing and dequeuing is O(1)the overall time complexity of $WCBCS$ scheduler is O(1).

4) 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 $\frac{1}{2}$ cycles while using alpha value of 0.4 for SES equation.

TABLE IV
EXPERIMENTAL SETUP

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

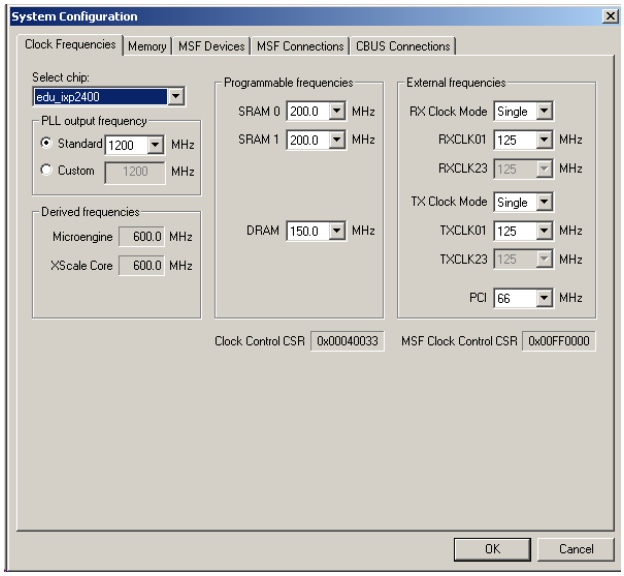


Fig. 8. Experimental system configurations.

V. EXPERIMENTS AND RESULTS

As mentioned earlier, we have tested the performance of the WCBCS 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. 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.

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 IV 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 8) were used.

We created 16 data stream files containing Ethernet frames and used the Workbench Simulator's Network traffic assignment functionality (as shown in Figure 9) to inject the data

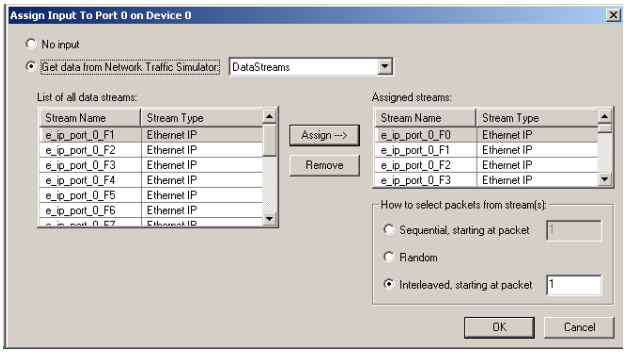


Fig. 9. Assigning experimental data streams using workbench simulator.

frames during experiments.

B. Experimental Results: WCBCS vs Separate WFQ schedulers

Experiments were performed and packet logs were collected while using both the WCBCS 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. The results show that the WCBCS 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 V
DELAY STATISTICS UNDER DIFFERENT SCENARIOS(RESOURCE REQUIREMENTS).

Scenarios	Delay using WCBCS (sec)			Delay using WFQ (sec)		
	Max. delay	Min. delay	Avg delay	Max. delay	Min. delay	Avg delay
High CPU, High BW	1.01	0.3	0.61	1.1	0.31	0.79
High CPU, Low BW	1.09	0.38	0.68	1.10	0.4	0.77
Low CPU, High BW	0.86	0.27	0.55	1.09	0.4	0.70
Low CPU, Low BW	0.96	0.3	0.58	1.0	0.28	0.68
Med. CPU, Med. BW	1.06	0.28	0.61	1.08	0.5	0.79

Figures 10 and 11 show the delays measured for data flows using WCBCS and WFQ. Due to space limitation we present graphs only for two scenarios (for flows with high CPU and low bandwidth requirement, and flows with low CPU and high bandwidth requirement), delay behaviour is similar for other scenarios too. The maximum, minimum and average delays (measured in ms) for these flows are shown in Table V. Delay results show that WCBCS achieved superior delay guarantees compared to WFQ (when used individually for CPU and bandwidth scheduling) for all the flows.

We ran our experiments for five scenarios (as shown in Table IV). For all the cases the maximum is worse for separate WFQ than WCBCS. The results show that using WCBCS, the average delay was reduced by 10% for flows with high CPU and high BW, 12% for high CPU and low BW scenario, 21% for low CPU and high BW scenario, 15% for low CPU and low BW scenario, and 23% for data flow with medium CPU and medium BW requirement compared to the delays achieved using WFQ.

VI. CONCLUSION

In this paper, we present our new composite scheduling algorithm called Weighted Composite Bandwidth and CPU Scheduler (WCBCS) which has the following properties: (1) unlike any other scheduling algorithms presented by other researchers it schedules multiple resources in a single algorithm, (2) WCBCS employs a simple and adaptive online prediction scheme for determining the packet execution times. (3) it is suitable for QoS flows where flows with different reserved rate are weighted differently, but unlike other scheduling algorithm of similar capability WCBCS has very low work complexity ($O(1)$), making it attractive for implementation in high-speed routers.

Our simulation and experimental results show that the delay behaviour of WCBCS is better than WFQ algorithm when used separately for CPU and bandwidth scheduling specially in the scenario when flows come with variable packet size and CPU reuirements. Here it should be noted WFQ can not be directly used for CPU scheduling (here we used our prediction scheme), moreover the work complexity of WFQ is much higher than the work complexity of WCBCS. The WCBCS algorithm would be very attractive for scenarios where flows are competing for both CPU and bandwidth resources. In particular, WCBCS can provide superior delay guarantees in highly dynamic environments where some or all flows can carry packets with varying sizes and varying CPU requirements.

REFERENCES

[1] Z. Kurtisi, X. Gu, L.Wolf, Enabling network-centric music performance in wide-area networks, Special issue on Entertainment networking, Communications of the ACM, Volume 49, Issue 11, PP: 52 - 54, ISSN:0001-0782, November 2006

[2] K. Chen, P. Huang, C. Lei, How sensitive are online games to network quality, Communication of the A
CM, Vol. 49, Numbers 11, PP: 34-38, November 2006.

[3] J.C.R. Bennett and H. Zhang. “WFQ: Worst-case Fair Weighted Fair Queueing.” In Proceedings of the IEEE INFOCOM, San Francisco, March 1996.

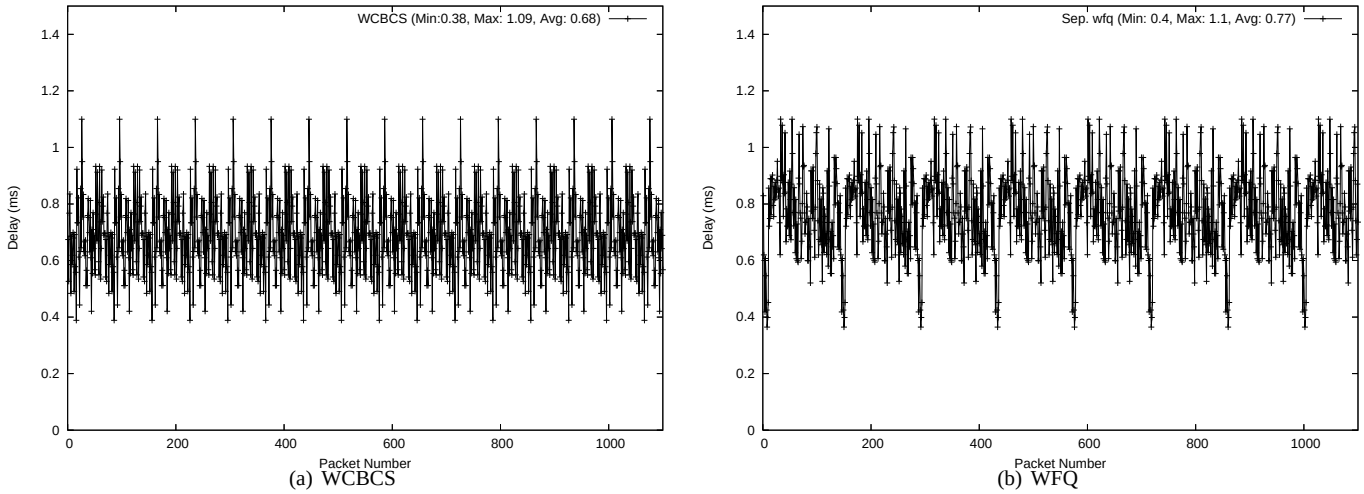
[4] P. Goyal, H.M. Vin and H. Cheng, “Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks.” IEEE/ACM Transactions on Networking, vol. 5, no. 5, pp. 690–704, October 1997.

[5] P. Pappu and T. Wolf, ”Scheduling Processing Resources in Programmable Routers”, in Proc. of IEEE INFOCOM’02, NY, June 2002.

[6] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda.”Processor Capacity Reserves: Operating System Support for Multimedia Applications”. In Proc. of the IEEE International Conference on Multimedia Computing and Systems, May 1994. pp. 90-99.

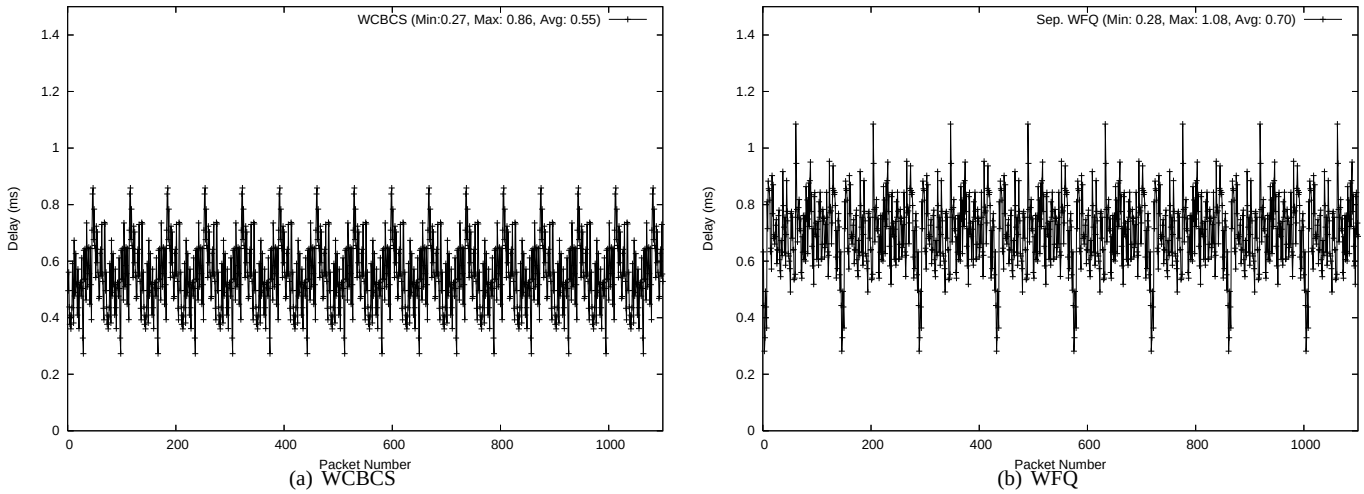
[7] K. Lakshman, R. Yavatkar and R. Finkel, ”Integrated CPU and Network I/O QoS Management in an Endsystem”, in Proc. of the 5th International Workshop on Quality of Service (IWQOS’97), New York, USA, pp 167-178.

[8] V. Galtier, K. Mills and Y. Carlinet, “Predicting and Controlling Resource Usage in a Heterogeneous Active Network (2001).” National Institute of Standards 2001.



(a) WCBCS (b) WFQ

Fig. 10. Delay for Flows with High CPU and Low BW Requirements



(a) WCBCS (b) WFQ

Fig. 11. Delay for Flows with Low CPU and High BW Requirements

[9] N. Doulamis, A. Doulamis, A. Litke, A. Panagakis, T. Varvarigou, E. Varvarigos, Adjusted fair scheduling and non-linear workload prediction for QoS guarantees in grid computing, Volume 30, pp: 499-515, Computer Communication, Elsevier, 2007

[10] F. Sabrina, Salil S. Kanhere, Sanjay K. Jha, Design, Analysis and Implementation of a Novel Low Complexity Scheduler for Joint Resource Allocation, Accepted for IEEE Transaction of Parallel and Distributed Systems, Volume 18, No 6, June 2007.

[11] F. Sabrina, S. Jha, "Scheduling Resources in Programmable and Active Networks Based on Adaptive Estimations", in Proceedings of the 28th Annual IEEE Conference on Local Computer Networks (LCN), Bonn, Germany, Oct. 2003, pp. 2-11.

[12] M. Shreedhar and George Varghese, "Efficient Fair queueing using Deficit round robin." IEEE/ACM Trans. Networking, vol 9, n0 3, pp. 375 - 385, 1996.

[13] The Network Simulator- ns-2, [Online], Available: http://www.isi.edu/nsnam/ns/.