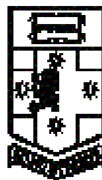
Design and Performance Analysis of CBCS^{WFQ} Packet Scheduling Algorithm

Fariza Sabrina and Sanjay Jha
School of Computer Science and Engineering
The University of New South Wales, NSW 2052, Australia
Email: {farizas;Sjha}@cse.unsw.edu.au

UNSW-CSE-TR-0306
May 2003

THE UNIVERSITY OF
NEW SOUTH WALES



SYDNEY · AUSTRALIA

Abstract

In active and programmable networks, packet processing could be accomplished in the router within the data path. For efficient resource allocation in such networks, the packet scheduling schemes should consider multiple resources such as CPU and memory in addition to the bandwidth to improve overall performance. The dynamic nature of network load and the inherent unpredictability of processing times of active packets pose a significant challenge in CPU scheduling. It has been identified that unlike bandwidth scheduling, prior estimation of CPU requirements of a packet is very difficult since it is platform dependent and it also depends on processing load at the time of execution and operating system scheduling etc. This paper presents a new composite scheduling algorithm called $CBCS^{WFQ}$ which is based on Weighted Fair Queuing (WFQ) and is designed for scheduling both bandwidth and CPU resources adaptively, fairly and efficiently. $CBCS^{WFQ}$ uses an adaptive prediction technique for estimating the processing requirements of active flows efficiently and accurately. Through simulation and analysis works we show the improved performance of our scheduling algorithm in achieving better delay guarantees compared to WFQ if used separately for CPU and Bandwidth scheduling.

1. INTRODUCTION

Active and programmable networks facilitate fast deployment of new services and protocols by allowing execution of packets in routers and switches. They also allow customized computation on packets flowing through them [1, 11, 12]. To use this technology safely and efficiently, individual nodes must understand the varying resource requirements for specific network traffic and must manage their resources efficiently.

Network resources in traditional networks mainly refer to bandwidth. Service discipline such as Fair Queuing (FQ) provides perfect fairness among contending network flows. However, traditional notion of fair queuing that specifies a resource allocation constraint for a single resource does not directly extend to active and programmable networks since allocation of resources in these networks involves more than one resource such as, CPU, bandwidth, and memory. Moreover, the allocation of these resources is interdependent and maintaining fairness in allocating one resource does not necessarily entail fairness in allocating other resource(s) [3, 8]. It is apparent that for large-scale deployment of active and programmable networks, researchers must address the issues of managing multiple resources within a node.

Today's Internet consist not only the best effort flow but also the QoS flows which is sensitive to delay, delay jitter etc. For QoS flows ensuring equal (fair) resource allocation for every flow would not be enough. It has to ensure that all the flows get its reserved resource and QoS is also maintained. To ensure this there has to be mechanism to give guaranteed bandwidth and computational resources to incoming flows. Guarantees in these two dimensions means that a flow always gets its reserved share except when the flow requires computational or link bandwidth more than what they have reserved.

QoS guarantees for multimedia traffic are difficult because first, when the traffic is processed in real-time (for interactive application) it is impossible to predict precisely what the actual behavior will be. Second, the traffic itself may be highly variable. For these reasons efficient resource allocation along with good control of QoS, is a challenging problem.

In an environment, where we want to be able to give service guarantees to data flows, it is typically necessary to explicitly reserve resources for that flow. This should be done during the flow setup and allows the network to route a new flow in such a fashion that enough bandwidth is available on the chosen path.

To make the resource reservation and admission control mechanism effective, it is important to have the knowledge of the resource requirement for every flow. Although knowing the bandwidth requirement for each flow is easy but the inherent unpredictability of execution time of arbitrary execution code poses a significant challenge in providing QoS guarantees for data flows that compete for such processing resource in the network.

Pappu et al. [7] presented a processor scheduling algorithm called Estimation-based Fair Queuing (EFQ) that estimated the execution times of various applications on packet of given lengths and then scheduled the processing resources based on the estimation. Their estimation process depends on the application specific estimation parameters that need to be determined through off-line experimentation of processing some packets using the application, and therefore, it does not support packets referencing any new application for which the parameters were not determined. They claimed that this algorithm provided better delay guarantees than processor scheduling algorithms that did not take packet execution times into consideration.

Galtier *et al.* [4] proposed a scheme to predict the CPU requirements to execute a specific code on a variety of platforms. In this work, a code can be executed on a specific platform off-line and the CPU requirement on this platform can be used to predict the CPU requirement on other platforms. They calculated the CPU requirement based on the relative calibration performance of the local and reference nodes. In reality, the predicted CPU requirement on a platform can differ significantly from the actual requirement. Therefore, it would be very difficult to implement the scheme on a busy node where allocation of proper share to competing flows was crucial.

In our previous work [8], we have developed a framework that has the capability of determining the CPU requirements of active packets on-the-fly. This system stores the CPU requirement of an active packet referencing a specific code/program in a node database once the referenced code is executed first time in the node. The database information and a scaling method are used to determine the processing requirements of the incoming packets (having varying packet sizes) referencing the same code/program.

To resolve the issues of efficiently and fairly managing both the processing and bandwidth resources in programmable and active networks and providing better QoS guarantees to the competing flows, this research work has been focused on developing a composite scheduler based on Weighted Fair Queuing (WFQ) which is able to predict the CPU requirements of the packets with reasonable accuracy and can schedule both the bandwidth and CPU resources adaptively, fairly, and efficiently. We have investigated the alternative techniques for predicting the processing requirements and have examined the performance of these prediction techniques in order to propose the solution that could be efficiently used for our composite scheduler $CBCS^{WFQ}$. The scheduler also uses the framework developed in our previous work [8] for managing the node resources. Through some simulation analysis we show how $CBCS^{WFQ}$ provides better delay bounds compared to using separate WFQ schedulers for CPU and bandwidth.

2. RESOURCE RESERVATIONS AND ADMISSION CONTROL

We assume the existence of a reservation protocol that the end system could use to communicate their resource requirement to the network. Evaluation of resource requirement and reservations are done according to the framework described in our previous work [8]. For each packet entering the node, the framework determines both the processing and bandwidth resource requirements for the packet and the information then used by the composite scheduler for admission control and scheduling of the packet. It may be noted that the framework takes the packet size into account for determining CPU requirements if the referenced code / program were classified as a payload processing application in the node database. It has been identified that for payload processing applications (such as packet compression, packet content transcoding) the processing requirements depend on the packet size, whereas the processing complexity for header processing applications (such as IP forwarding, QoS routing etc.) is in general independent of the size of the packet. It may also be noted that processing of packets on an active or programmable node can also affect the size of packets after processing is completed. To take this packet size change into account for bandwidth consumption, our framework described in [8] calculates and uses an Expansion Factor that is the packet size after processing divided by the packet size before processing.

When admitting a new flow, the admission control of this system decides whether the flow get the service requested, and it also decides whether admitting the flow will prevent the node from keeping its prior commitments. The admission control is used to limit the packet-loss probability to a known value. The basic idea of admission control is that a host must probe the path to the receiver before sending the actual data. It accepts the packet of the probe is received with no or at most a moderate amount of loss. The aim of the admission control is to establish a reliable upper bound on the packet-loss probability in the network. Admission control in our system uses the following policies:

2.1 Reserved (QoS) flows

If an incoming flow α requests guaranteed service, the admission control algorithm:

1. Denies the request if summation of the reserved bandwidth rate of all the reserved flows and the current flow's requested bandwidth rate exceed the targeted link utilization level. I.e., deny accepting if –

$$r_{bw}^{\alpha} + \sum_{i=1}^{N_r} r_{bw}^{i} < \mu_{bw} R_{bw} \qquad (1)$$

2. Denies the request if admitting the new flow could violate the delay bound, of the same priority level.
3. Denies the request if summation of the reserved CPU rate of all the reserved flows and the current flow's requested CPU rate exceed the targeted CPU utilization level. I.e., deny accepting if –

$$r_{cpu}^{\alpha} + \sum_{i=1}^{N_r} r_{cpu}^{i} < \mu_{cpu} R_{cpu} \qquad (2)$$

Where in (1) and (2),

r_{bw}^{i}, r_{cpu}^{i} = reserved bandwidth and CPU rates for flow i.
N_r = number of registered reserved flows.
r_{bw}^{α}, r_{cpu}^{α} = requested bandwidth and CPU rates.
μ_{bw}, μ_{cpu} = targeted bandwidth and CPU utilization factors.
R_{bw} = transmission link rate.
R_{cpu} = processing rate of the node.

2.2 Best effort flows

The system equally distributes the remaining bandwidth and CPU resources to all the competing best effort flows. While registering a new best effort flow and also after registering any reserved flow, the system re-calculates the allowable bandwidth and CPU rates for all the best effort flows as follows:

$$r_{bw}^{be} = \frac{\left(\mu_{bw} R_{bw} - \sum_{i=1}^{N_r} r_{bw}^{i} \right)}{N_{be}} \qquad (3)$$

$$r_{cpu}^{be} = \frac{\left(\mu_{cpu} R_{cpu} - \sum_{i=1}^{N_r} r_{cpu}^{i} \right)}{N_{be}} \qquad (4)$$

Where,
r_{bw}^{be} = Allocated bandwidth rate for a best effort flow.
r_{cpu}^{be} = Allocated CPU rate for a best effort flow.
N_{be} = Number of best effort flows in the system.

The system also compares the number of best effort flows against a pre-defined threshold and denies accepting a new best effort flow if the number exceeds the threshold. It also checks the queue size of the individual best effort flows and denies accepting packet if the queue size gets bigger than another pre-defined threshold.

3. PREDICTING PROCESSING REQUIREMENTS

3.1 Application Types

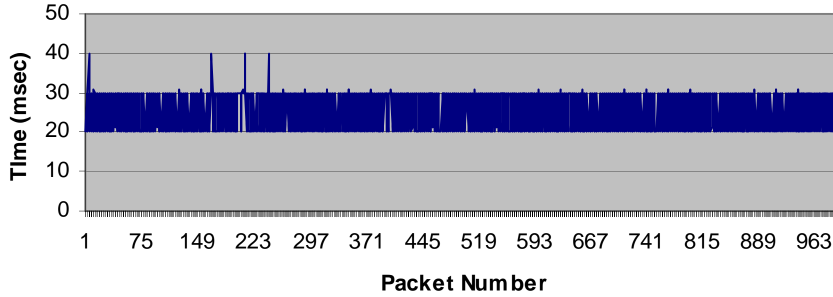
Applications that can process packets on a programmable or active node can be classified into two categories: header processing applications and payload processing applications. Header processing applications only perform read and write operations in the header of the packet and therefore, the processing complexity is in general independent of the size of the packets. Examples of the header processing applications include IP forwarding, transport layer classification, and QoS routing. Whereas the payload processing applications performs read and write operations to all the data in the packet, in particular the payload of the packet, and therefore, the processing complexity here strongly correlates to the packet size [7]. Examples of payload processing applications are IPSec Encryption, packet compression, packet content transcoding (e.g. image format transcoding) etc.

3.2 Need for Adaptive Predictions

We have performed some experiments to investigate the impacts of processing load and operating system scheduling of a node on the packet processing times. It has been identified that the packet processing times significantly varies in different number of executions by both the header processing applications and payload processing applications even though the packet size remained the same for all executions.

Figure 1 shows the processing times consumed by MPEG2 encoder code for 1000 repetitive executions on a MPEG2 data packet having a packet size of 24576 bytes. The processing times varied between 20 – 40 milliseconds. The experiments were performed on a machine having Pentium 4, 1.8 GH CPU and running on Linux (RedHat 7.2) operating system. The results indicate that processing time consumed measured through off-line experimentations by a particular application to process a packet cannot be readily used for estimating processing times for new incoming packets referencing the same application for a scheduler (such as WFQ) that schedules packets based on estimated finish time of the packets. Therefore, we need an adaptive prediction process that can accommodate the variation in packet size and the effect of processing load and operating system scheduling.

Figure 1: Variation in processing times for MPEG2 data packets having same packet size



3.3 Evaluating Alternative Prediction Techniques

In this section we present two alternative smoothing methods that we have implemented to predict the CPU requirements of data packets. Experiments have been performed (again on a Pentium 4, 1.8 GH processor under Linux operating system) to analyze the performance and suitability of the methods in the context of using them within a CPU scheduler.

3.3.1 Single Exponential Smoothing

Single Exponential Smoothing (SES) is computationally simple and an attractive method of forecasting. Researchers have used this method to forecast the display cycle time (which include decompression time plus rendering time) for compressed video data packets [15]. SES uses the following equation to calculate new forecasted value.

$$F_{t+1} = \alpha X_t + (1-\alpha)F_t \qquad \text{where } 0 \le \alpha \le 1 \qquad (5)$$

here,

F_t = Forecasted value for the tth period
X_t = Current actual value
A = Parameter chosen by the user.
F_{t+1} = Forecasted value for (t+1) period, e.g., next forecasted value.

3.3.2 Linear Least Square Regression

Researchers [7] have demonstrated that the processing times of payload processing applications are highly dependent on the packet size. They have formulated the following equation of packet processing cost in terms of a fixed overhead and a variable cost that depends on the packet size.

$$C = \alpha_a + \beta_a . l, \qquad (6)$$

Where,

C= the processing cost of a packet of length l.
α_a = per packet fixed processing cost of application a.
β_a.=per byte processing cost of application a.

The researchers in [7] have determined the values of α and β for different applications through off-line experimentations and suggested that the parameters could be estimated on-line using Linear Least Square Regression (LLSR) techniques. As the packets are processed, the router can keep tracks of the cost and length of the packets being processed and the value of those parameters could be determined using the following relationships.

$$\beta_a = \frac{\sum_n c_i . l_i - \sum_n c_i . \sum_n l_i / n}{\sum_n l_i^2 - \sum_n l_i . \sum_n l_i / n} \qquad (7)$$

$$\alpha_a = \frac{\sum_n c_i - \beta_a . \sum_n l_i}{n} \qquad (8)$$

3.3.3 Performance Evaluation of the Prediction Techniques

It may be noted that the online estimation of α and β using the equations (7) and (8) and using those for predicting processing cost of a packet within a flow can only de done if the flow contains packets of variable lengths. In other words, this technique cannot be used to predict processing costs of packets within a flow having all the packets with same size. Moreover, as the equations suggest, the overhead for estimations using LLSR is significantly higher than that using SES.

We have compared the performance of both prediction techniques on processing an MPEG packet flow where the flow contained packets of varying lengths e.g. 24576, 13824, 6144, and 1536 bytes. The packets with varying lengths were generated randomly and fed to the MPEG2 encoder code for encoding. We measured the actual processing times elapsed for 1000 packets and the estimated times predicted by both prediction schemes. The processing times elapsed varied between 10 and 40 milliseconds. Figure 2 shows the actual processing times elapsed vs. estimated times predicted by SES technique, whereas figure 3 shows the actual processing times elapsed vs. estimated times predicted by LLSR technique. Figure 4 shows the deviation of actual processing times from the estimated times for both the schemes. For SES, the value of α was set to 0.5.

Figure 2: Actual processing time and predicted processing times using SES

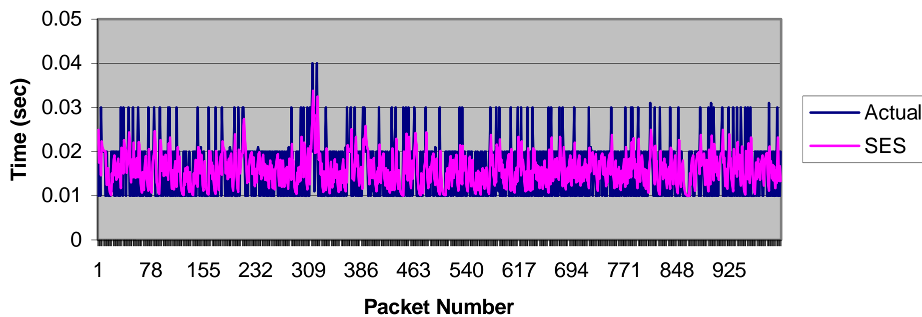


Figure 2 – 4 show that both the estimation techniques produced estimated results that could be quite acceptable for a CPU scheduler. In both cases the patterns of the predicted values resembled the patterns of the actual processing times. Figures show that LLSR technique can produce a-bit better prediction results with the expense of extra processing overheads. It also may be mentioned that we have used both techniques within our composite scheduler CBCS^{WFQ} to analyze the difference in delay behaviors, and found that both technique produced comparable delays. The delay results are shown in section 5.

Figure 3: Actual processing time and predicted processing times using LLSR

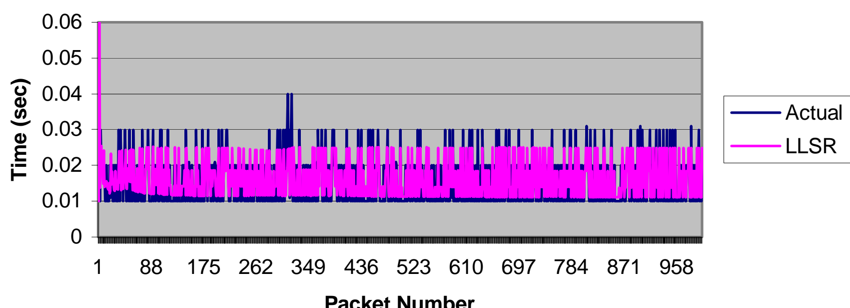
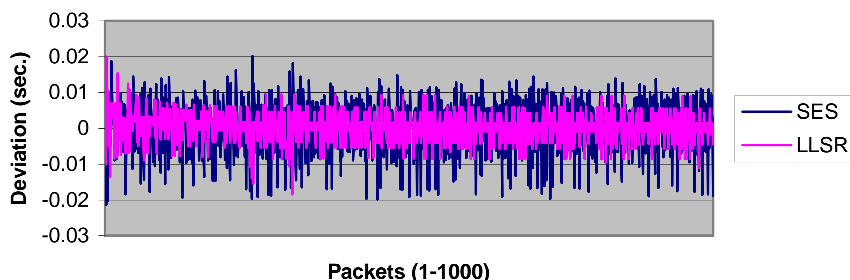


Figure 4: Deviation of actual processing times from predicted times using SES and LLSR



3.4 Adaptive Prediction Scheme used in CBCS^{WFQ}

SES technique is simple and it requires in-significant processing overhead for maintaining the state variables required for the prediction process. It can produce results comparable to LLSR and it can also be used for predicting processing requirements of flows having packets with fixed sizes. SES technique would be perfect for estimating CPU requirements for any header processing applications and also for payload processing applications for processing packet flow having minor variation in packet sizes within the flow. It would also produce quite good estimates for payload processing applications for processing flows having variations in packet sizes. The SES prediction process can quickly react to a new actual processing time caused by changes in packet size and or processing load / operating system scheduling and can produce a new good estimate for the next packet.

For the simplicity and the benefits of the SES technique as discussed above, we have opted to use SES technique and the framework presented in [8] to predict the CPU requirements of the incoming packets within our composite scheduler CBCS^{WFQ}. The framework in [8] has a node

database to store the information of different applications (including the CPU time requirement for processing a packet of a specific size, application type etc.) referenced by packets in the node. The system stores the CPU requirement of an active packet referencing a specific code/program in a node database once the referenced code is executed first time in the node. The database information can be used for admission control and reserving resources etc. The $CBCS^{WFQ}$ uses the database information as the starting point for predicting the processing requirements of the packets, i.e. the processing time indicated in the database is used as estimated processing time for packets in a flow until a packet in that flow is processed by the scheduler and then SES is used for updating the predictions.

4. $CBCS^{WFQ}$ –THE COMPOSITE BANDWIDTH AND CPU SCHEDULER

The architecture of the $CBCS^{WFQ}$, a composite bandwidth and CPU scheduler based on Weighted Fair Queuing, is depicted in Figure 5. Primary goal of the scheduler is to manage both the bandwidth and CPU resources fairly and efficiently and to provide better delay guarantees.

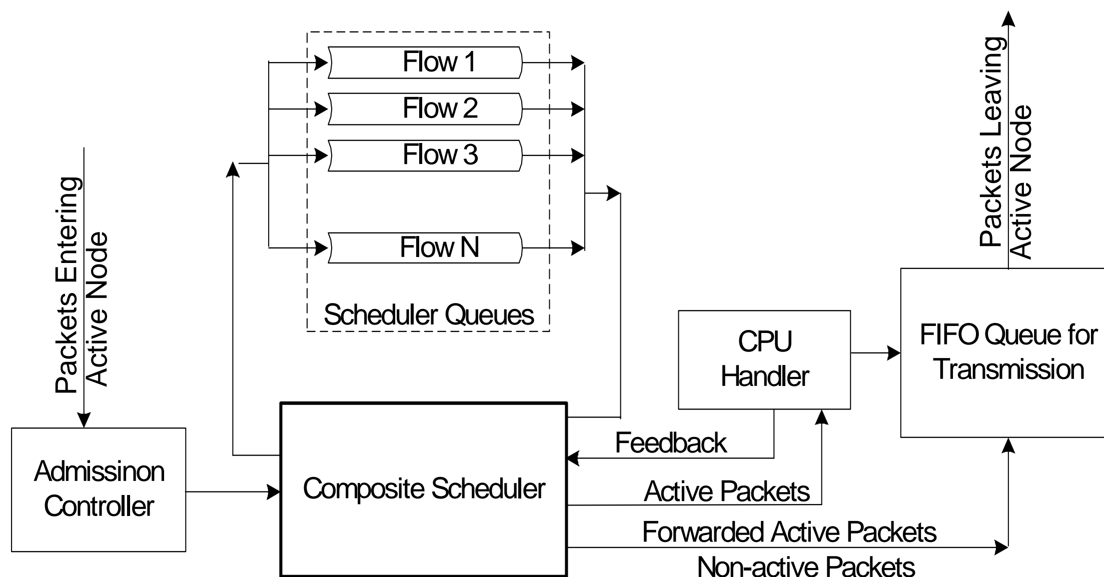


Figure 5 Composite Scheduler Architecture

The Admission controller controls the flow registration and setup (including setting up weights for the flows based on the reserved rates of the bandwidth and processing resources) and admission of each individual packet. The scheduler estimates the processing times of the incoming packets and en-queues them in the corresponding flow queues and de-queues packets using its composite scheduling algorithm $CBCS^{WFQ}$ which takes both the estimated processing time and transmission time of packets into account to decide which packet to de-queue. After de-queuing a packet, the scheduler hands the packet to the CPU handler for processing if the packet needed any processing. CPU handler object notifies the scheduler 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, which is served by a separate thread for sending the packets to their next destination. The packets that do not require processing (e.g. non-active packets and active packets already processed in up-stream nodes)

enters directly into the transmission queue after the scheduler de-queues them from their flow queues. The scheduler algorithm is discussed below.

4.1 Composite Scheduling Algorithm

CBCS^{WFQ} enforces fairness in resources allocation using the Weighted Fair Queuing (WFQ) Principle [5]. Most Packet Fair Queuing (PFQ) algorithms such as WFQ, WF^{2}Q+ (Worst Case Fair Weighted Fair Queuing +) [6], SFQ (Start time Weighted Fair Queuing) [13] are based on approximating Generalized Processor Sharing (GPS) [2] scheme which is an ideal scheduling discipline based on a fluid flow model in which the traffic is infinitely divisible and each session is serviced simultaneously according to its weight. In these algorithms, packets from different sessions are stamped with virtual finishing times and selected to schedule based on increasing order of virtual finishing times. 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 algorithms use the packet size and a system virtual time (which is updated each time a new packets 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 new backlogged sessions in order to equalize the normalized services of these sessions with current backlogged sessions. A flow is *backlogged* during the time interval ($t_2 - t_1$) if the queue for the flow is never empty during the interval. The differences in defining the system virtual function in Packet Fair Queuing (PFQ) algorithms result in different implementations complexities, fairness measurement and delay behaviors.

The traditional PFQ algorithms (including WFQ and WF^{2}Q) are mainly used for packet scheduling i.e. for allocating bandwidth resource only and they do not directly extend for allocating multiple resources. In CBCS^{WFQ}, new system virtual function and finish time calculation equations have been formulated in order to schedule both bandwidth and CPU resources from a single composite scheduler. The following parameters and equations are used.

$$sum_w(t) = \sum_{i \varepsilon B(t)} \left(\phi^i_{cpu} + \phi^i_{bw} \right) \quad (9)$$

$$V(t+\tau) = V(t) + \frac{\tau}{sum_w(t)} \quad (10)$$

$$S_i^k = \max\left\{ F_i^{k-1}, V(a_i^k) \right\} \quad (11)$$

$$F_i^k = S_i^k + \frac{P_i^k}{\varphi^i_{cpu}} + \frac{L_i^k * 8 * \gamma^i}{\varphi^i_{bw} * BW * 10^6} \quad (12)$$

Where,

ϕ^i_{cpu}, ϕ^i_{bw} = Weight for CPU and bandwidth for flow i.

$B(t)$ = Set of backlogged flows at time t.

$sum_w(t)$ = Summation of the weights (both CPU and bandwidth) of all the active flows at time t.

$V(t)$ = System virtual time at time t.

τ = 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 packet k of flow i.

F_i^k, F_i^{k-1} = Virtual finish time of packets k and K-1 of flow i.

$P_i^k =$ Estimated processing cost of packet k of flow I in sec.
$L_i^k =$ Length of packet k of flow i in Bytes.
$\gamma^i =$ Expansion factor of the packets in flow i.
BW = Transmission link bandwidth in Mbps.

At the start of the scheduler $V(t)$ is set to zero. With the arrival of every new packet, admission controller determines the estimated processing cost i.e. P_i^k and the expansion factor i.e. γ^i for the packet using equation (5) and the framework in [8] and hands the packet to the composite scheduler specifying the values for P_i^k, and γ^i. The scheduler updates $V(t)$, and calculates the virtual finish time of the packet (F_i^k) using (9) – (12). The scheduler then stores the packet in the corresponding flow queue.

While any packet exists within the queues, the scheduler algorithm (i.e. the de-queue process of the scheduler) checks the finish times of the packets at the head of all active flow queues and de-queues the packet (from the head of the queue) having minimum finish time. It then hands the packet to the CPU handler object that in turn process the packet and then en-queues it in the FIFO transmission queue.

The pseudo-code for the en-queue and de-queue methods are given below.

Pseudo-code for en-queue

```
Read the packet header and determine the flow id;
Instantiate a queue item object and set the packet reference;
Find the current time;
//Update V(t)
If (total number of packets in all queues is zero)
{
  for each flow i
  {
       F[i]=0;
       B[i]=0;
  }
    V(t)=0;
    sum_w(t)=0;
    LastVtUpdate =current time;
}
else
{
  V(t) = V(t)+(current time -LastVtUpdate) / sum_w(t);
  LastVtUpdate =current time;
}

//Update B and sum
if(B[flow id] is zero)
{
    B[flow id] = 1; //mark it as active or backlogged flow
    sum_w(t) += weight[flow id];
}

//Calculate Finish time
Calculate start time using equation (7) i.e. start time = max {F[flow id], V(t)};
```

Calculate finish time using equation (8) i.e. update F[flow id];
Set the value of the f_time member of the queue item to F[flow id];
//Update total number of packets
Total number of packets += 1;

//Store the packet
En-queue the queue item in the individual queue corresponding to the flow id;

Pseudo-code for de-queue

```
While (true)
{
   if (total number of packets > 0)
   {
       Access the queue item from head-of-queue of all the active flow queues and read the
       f_time member;
       Locate the queue i having the queue item with minimum f_time;
       De-queue the packet from queue i;
       Hand the packet to the CPU Handler;

       //Update total number of packets
       Total number of packets -= 1;  //same variable used by en-queue
      //Update B and sum if necessary
       if(length of queue i is zero)
      {
          B[i] = 0; //mark as non active flow
          sum_w(t)  -= weight[i];
       }
   }

}
```

4.2 Delay Guarantees and Analysis

We use the concept of *delay guarantee* that is defined in [10] to formulate and compare the worst-case delay of $CBCS^{WFQ}$ with that of WFQ when separately used for CPU and bandwidth scheduling. WFQ algorithm when used for bandwidth or CPU scheduling is able to guarantee a delay to a session flow based on maximum normalized transmission cost or maximum normalized processing cost of a packet within the flow. Also the worst-case delay is influenced by the maximum bandwidth or CPU cost of a packet within all the backlogged flows. Table 1 shows delay guarantees of WFQ algorithm.

Table1. Delay Guarantees of WFQ

Resource scheduled	Delay Guarantees
CPU	$\frac{P_i^{\max}}{r_{cpu}^i} + \frac{P_{\max}}{R_{cpu}}$
Bandwidth	$\frac{L_i^{\max}}{r_{bw}^i} + \frac{L_{\max}}{R_{bw}}$

4.2.1 Delay Guarantees with Separate WFQ CPU and Bandwidth Schedulers

As the WFQ delay guarantees given in table 1, to service active packets in programmable / active node if we use WFQ separately for scheduling CPU and bandwidth resources, the delay guarantees for any flow i becomes:

$$\delta_i^{wfq} = \left(\frac{P_i^{\max}}{r_{cpu}^i} + \frac{P_{\max}}{R_{cpu}} \right) + \left(\frac{L_i^{\max}}{r_{bw}^i} + \frac{L_{\max}}{R_{bw}} \right) \tag{13}$$

Where,

δ_i^{wfq} = Delay guarantee for flow i when WFQ is used separately for CPU and bandwidth scheduling.

$P_i^{\max}$ = The maximum processing cost of a packet within flow i.

$P_{\max}$ = The maximum processing cost of a packet within all the backlogged flows.

$L_i^{\max}$ = The maximum length of a packet within flow i.

$L_{\max}$ = The maximum length of a packet within all the backlogged flows.

$\frac{L_i^{\max}}{r_{bw}^i}$ = Maximum normalized transmission cost of a packet within flow i.

$\frac{P_i^{\max}}{r_{cpu}^i}$ = Maximum normalized processing cost of a packet within flow i.

4.2.2 Delay Guarantees of CBCS^{WFQ}

While scheduling both the CPU and bandwidth resources using CBCS^{WFQ}, the scheduler considers both the processing cost and the transmission cost (i.e. total cost) of the packets in order to calculate the finish times of the packets. Therefore, CBCS^{WFQ}, is able to guarantee a delay to a session flow based on flow's properties such as its reserved rates for bandwidth and CPU and the maximum total cost of a packet within the flow. Also the worst-case delay is influenced by the maximum total cost of a packet within all the backlogged flows. The delay guarantee of the CBCS^{WFQ} for any flow i can be derived as:

$$\delta_i^{cbcs} = \left(\frac{P_i^k}{r_{cpu}^i} + \frac{L_i^k}{r_{bw}^i} \right)^{\max} + \max\left\{ \left(\frac{P_1^k}{R_{cpu}} + \frac{L_1^k}{R_{bw}} \right)^{\max}, \ldots\ldots, \left(\frac{P_n^k}{R_{cpu}} + \frac{L_n^k}{R_{bw}} \right)^{\max} \right\} \tag{14}$$

Where,

δ_i^{cbcs} = delay guarantee for flow i using CBCS^{WFQ}

P_i^k = processing cost of packet k in flow i.

L_i^k = length of packet k in flow i.

$\left(\frac{P_i^k}{r_{cpu}^i} + \frac{L_i^k}{r_{bw}^i} \right)^{\max}$ = maximum normalized total cost of a packet within flow i.

$\left(\frac{P_n^k}{R_{cpu}} + \frac{L_n^k}{R_{bw}}\right)^{max}$ = maximum total cost of a packet within flow n.

4.2.3 Delay Analysis

Analysis of (13) and (14) shows that CBCS^{WFQ} can provide better delay guarantee than WFQ when used separately for bandwidth and CPU scheduling. We will take an example to analyze the delay provided by (13) and (14).

Let assume that a node is serving 2 flows. In flow 1, each packet length is 10 Bytes and requires 30 CPU cycles to process. In flow 2, each packet length is 30 Bytes and requires 10 CPU cycles to process. The node processing capability (i.e., R_{cpu}) is 1000 Cycles per second, and the bandwidth of the transmission link (i.e., R_{bw}) is 1000 Bytes per second. Flow 1 has reserved 75% of the CPU (i.e. r_{cpu}^1 = 750 cycles / second) and 25 % of the bandwidth (i.e., r_{bw}^1 = 250 Bytes / sec). Flow2 has reserved 25% of the CPU (i.e. r_{cpu}^2 = 250 cycles / second) and 75 % of the bandwidth (i.e., r_{bw}^2 = 750 Bytes / sec). Let's consider that packet arrival rates from both flow 1 and flow 2 are such that both flows are just saturating their reserved rate.
Table 2 shows the calculated delays using (13) and (14), and shows that CBCS^{WFQ} provides better delay guarantees for this example.

Table 2 An example of delay comparison

Flow	P_i	L_i	r_{cpu}^i	r_{bw}^i	δ_i^{wfq}	δ_i^{cbcs}
1	30 cycles	10 Bytes	750 cycles / s	250 Bytes/ s	140 msec	120 msec
2	10 cycles	30 Bytes	250 cycles / s	750 Bytes/ s	140 msec	120 msec

5. SIMULATION RESULTS

In order to demonstrate the characteristics of CBCS^{WFQ} compared with WFQ, we have modified the NS2 network simulator [14] to implement the components described in Fig. 5 to simulate an active / programmable network. The simulation was performed on a PC having a 1.8 GHZ Pentium 4 processor and 384 MB memory and running under the Linux operating system (RedHat 7.2). We have also implemented both the SES and LLSR estimation techniques (either one is selectable through a configuration parameter) within our scheduler in order to analyse the differences in delay influenced by these estimation techniques. The experimental results achieved using CBCS^{WFQ} are compared with the results using WFQ for scheduling CPU and bandwidth independently.

5.1 Simulation Settings

We used 10 hosts to generate network traffics for an active / programmable node, where each host generated CBR traffic corresponding to a flow. Hosts #1, #3, and #5 generated active packets containing MPEG2 video data of four different sizes (e.g., 1536, 6144, 13824, and 24576) and referencing MPEG2 encoder code. The other hosts generated active packets with different packet sizes and referencing different codes. MPEG2 encoder code was implemented

in C++ within the NS2 environment and the framework in [8] was used to evaluate the CPU requirement and expansion factor of MPEG2 data packet the first time one arrived at the active node. For the other hosts, the packets generated from a given host carried the same code reference, and the framework evaluated the CPU requirements and expansion factor for the code (when the first packet arrived in the active node). The output link capacity was set to 5 Mbps. The simulation settings of the individual flows are given below.

Table 3: Settings for individual flows

Flow#	1, 3, 5	2	4	6	7	8	9	10
Packet size (Bytes)	1536 - 24576	3	3	3	3	3	3	3
CPU Req. (msec)	10 - 40	10	8	4	3	2	2	2
Expansion Factor	0.11 – 0.36	1	1	1	1	1	1	1
ϕ_{cpu}^{i}	3	3	2	2	1	1	1	1
ϕ_{bw}^{i}	3	1	2	2	1	1	3	3

The simulations were run for 50 seconds and measurements were taken at 5-second intervals. Packet generation rates for all the flows were adjusted such that all the flows required 97% of the CPU resources and 97% of the bandwidth resources (i.e., resource utilizations were just below 100%) so that the measured delays are because of scheduling and not because of queuing backlog. Simulations were re-run using both estimation techniques and the results achieved using $CBCS^{WFQ}$ is compared with that achieved using WFQ for scheduling CPU and bandwidth independently. It may be noted that we have also implemented separate WFQ CPU scheduler (that also estimates the processing time of packets using the same schemes) and bandwidth scheduler within NS2 environment.

Each simulated active packet header contained some additional fields (or parameters) to signify the packet as an active packet and to facilitate the handling of the packet by the active node. In reality, the communication between active nodes could be done by exchanging messages, similar to the PANTS [13] architecture. However, for our simulation in NS2 simulator we added some parameters in the common header (i.e., hdr_cmn structure) of the NS2 packet.

5.2 Fairness Measurements

We have measured the both utilized CPU rates and bandwidth (BW) rates by all the flows and compared that with the reserved rates (based on the weights used in table 3) and found that fairness achieved in utilizing a flow's share of CPU and bandwidth of $CBCS^{WFQ}$ is similar to that achieved by WFQ. As an example, the results measured at 45th seconds (while using SES prediction) are shown in table 4. Total bandwidth utilization was 97.39% using WFQ compared to 97.17% in $CBCS^{WFQ}$. Total CPU utilization in both cases was 97.1%. Both schedulers allowed a flow to over-utilize its share of CPU and /or bandwidth when other flows were not sending packets to saturate their shares.

Table 4 Fairness measurement

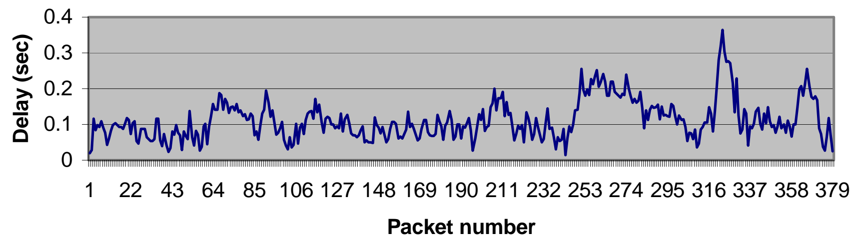
Flow	r^i_{cpu}	r^i_{bw}	Utilized CPU rates		Utilized BW rates	
			CBCS	WFQ	CBCS	WFQ
1	0.15	0.1364	0.125578	0.125123	0.007004	0.006947
2	0.15	0.0454	0.144871	0.146934	0.080814	0.082037
3	0.15	0.1364	0.125578	0.124665	0.007004	0.006927
4	0.10	0.0909	0.117728	0.117181	0.082223	0.081781
5	0.15	0.1364	0.125578	0.124665	0.007004	0.006927
6	0.10	0.0909	0.088342	0.088161	0.123462	0.123055
7	0.05	0.0454	0.073671	0.075802	0.137166	0.141072
8	0.05	0.0454	0.051494	0.051450	0.143954	0.143628
9	0.05	0.1364	0.073603	0.073055	0.205685	0.203941
10	0.05	0.1364	0.073557	0.072964	0.205685	0.203686

5.3 Delays Measurements

Figure 6 – 9 shows the delays measured at 45th seconds for the MPEG flow 1 using CBCS^{WFQ} and WFQ while using SES and LLSR as the estimation techniques.

The figures show that LLSR can provide a-bit better delays compared to SES. However, the delays achieved using SES is quite convincing compared to that using LLSR given the fact that SES is significantly less computationally expensive and simple compared to LLSR.

Figure 6: Delay for MPEG flow 1 using CBCS^{WFQ} composite scheduler and SES estimation



Delay results show that CBCS^{WFQ} achieved much superior delay guarantees compared to WFQ in both estimations. While using CBCS^{WFQ} the worst case delays measured were 0.3649 sec and 0.296 sec for SES and LLSR respectively. While using WFQ the worst case delays measured were 0.6693 sec and 0.5273 sec for SES and LLSR respectively. Therefore, the results show that when WFQ was used for scheduling CPU and bandwidth independently, the worst case delays increased by 83% and 78% compared to using CBCS^{WFQ}.

Figure 7: Delay for MPEG flow 1 using CBCS^{WFQ} composite scheduler and LLSR estimation

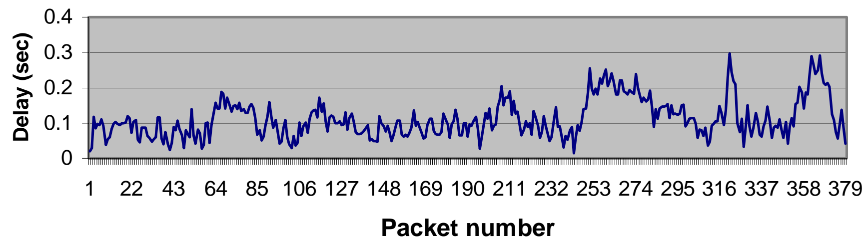


Figure 8: Delay for MPEG flow 1 using WFQ scheduler and using SES estimation

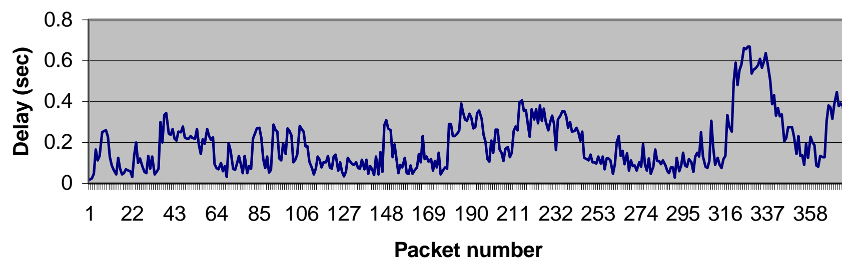
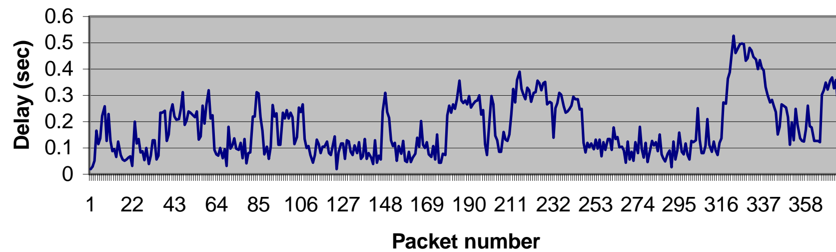


Figure 9: Delay for MPEG Flow 1 using WFQ scheduler and LLSR estimation



5.4 Throughput Measurements

Figure 10 – 13 shows the CPU and bandwidth utilizations measured at every 5th second during simulation. The results show that both the CBCS^{WFQ} and WFQ achieved similar or comparable throughput. Also the results show that CPU and bandwidth utilizations in both cases were not influenced by SES or LLSR.

Figure 10: CPU Utilization Using CBCS^{WFQ} Composite Scheduler

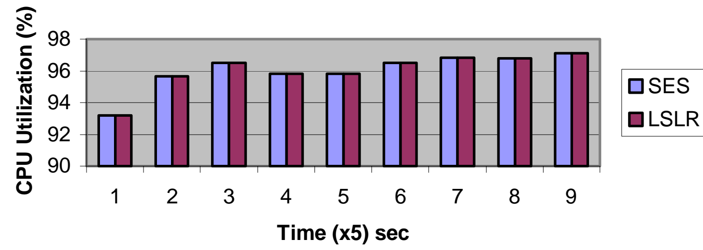


Figure 11: CPU Utilization using WFQ

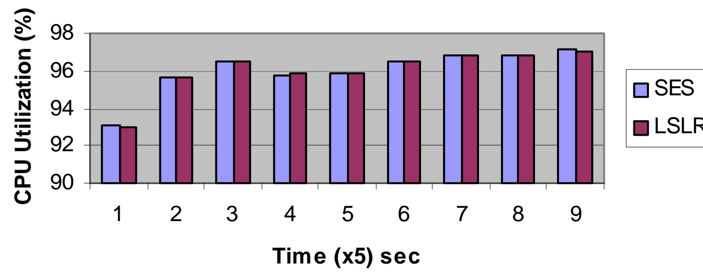


Figure 12: Bandwidth utilization using CBCS^{WFQ}

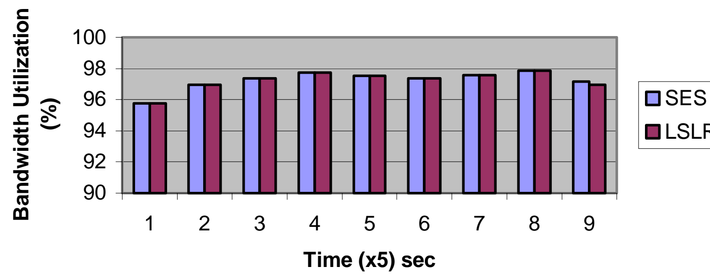
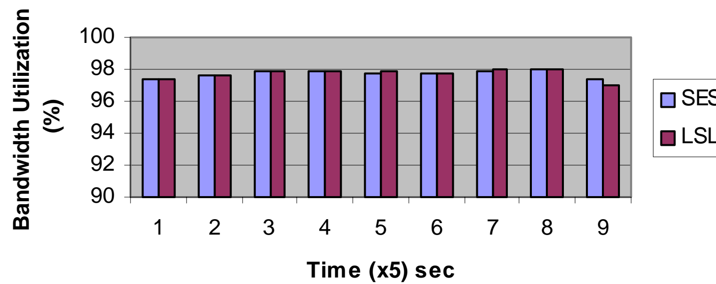


Figure 13: Bandwidth Utilization Using WFQ



6. CONCLUSION

In this work, we have presented an approach for providing QoS guarantees for flows that are processed on nodes in the network. We have investigated the alternative methods for estimating the processing requirements for active flows and provided a solution that can be used for predicting processing requirements of packets flowing through active and programmable networks. The suggested estimation process can be utilized for both the header processing and the payload processing applications. We have evaluated the performance of our combined scheduling algorithm for fair service among all the competing flows through simulation. The simulation result and also the analysis work shows that our composite scheduling algorithm offers better delay guarantees than the traditional WFQ algorithm when used separately for CPU and bandwidth scheduling. It also offers similar fairness and throughput compared to WFQ. $CBCS^{WFQ}$ would be very attractive for an active / programmable node where the node needs to efficiently manage active packet flows competing for both CPU and bandwidth resources. Especially, the $CBCS^{WFQ}$ would provide much superior delay guarantees under highly dynamic environment where even each flow can carry packets with varying sizes and varying CPU requirements.

ACKNOWLEDGEMENT

This research is supported by Co-Operative Research Centre (CRC) for smart Internet Technology, Australia.

REFERENCES

1. Campbell, H.D. Meer, M. Kounavis, K. Miki, J. Vicente and D. Villela, “A survey of programmable networks,” SIGCOMM Computer Communications Review, vol. 29, no. 2, April 1999
2. A.K. Parekh and R.G. Gallagher. “A generalized processor sharing approach in integrated services networks”, INFOCOMM ’93, 1993.
3. Ramachandran, R. Pandey and S.-H. Chan, “Fair Resource Allocation in Active Networks,” Proceedings of the IEEE International Conference on Computer Communications and Networks (ICCCN), pp. 468–475, Las Vegas, Nevada, Oct 16–18, 2000.
4. Galtier, K. Mills and Y. Carlinet, “Predicting and Controlling Resource Usage in a Heterogeneous Active Network (2001),” National Institute of Standards 2001.
5. A. Demers, S. Keshav and S. Shenkar, “Analysis and simulation of a fair queueing algorithm”, Internetworking: Research and Experience, Vol.1, no.1, pp.3-26, 1990.
6. J. C. R. Bennett and H. Zhang. WF Q: Worst-case Fair Weighted Fair Queueing. In Proceedings of the IEEE INFOCOM, San Francisco, March 1996.
7. P. Pappu and T. Wolf, “Scheduling Processing Resources in Programmable Routers”, Department of computer science, IEEE INFOCOM 2002, New York, NY, June 2002.
8. F. Sabrina and Sanjay Jha, “A Novel Architecture for Resource Management in Active Networks Using a Directory Service” ICT03, February 2003.
9. V.Y Ramachandran and R. Pandey, “Resource allocation in active networks”, Technical report TR-99-10, Parallel and Distributed Computing Laboratory, Computer Science Department, University of California, Davis, CA 95616, August 28, 2000.
10. P.Goyal, H.M. Vin, 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.
11. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall and G.J. Minden, “A Survey of active network research,” IEEE communications Magazine, pp.80–86, January 1997.
12. Wetherall, U. Legedza and J. Guttang, “Introducing new internet services: why and how,” IEEE Network Magazine, July/August 1998.
13. Fernando and B Kummerfeld, “A New Dynamic Architecture for an Active Network”, IEEE Openarch 2000.
14. NS2 Simulator, The LBLN Network simulator, University of California, Berkley.
15. "Playout Management of Interactive Video - an Adaptive Approach" - Sanjay K. Jha, Peter A. Wright, Michael Fry , IWQOS, 1997