FORTH-ICS /TR-328 November 2003

Performance Evaluation of Variable Packet Size Buffered Crossbar Switches

Georgios Passas

ABSTRACT

The crossbar is very popular for the implementation of switches with a moderate number of ports. Unbuffered crossbars (IQ - architecture) require complex switch matrix scheduling algorithms that operate on fixed-size cells. On the other hand, buffered crossbars (CICQ - architecture), use small buffers at the crosspoints and provide excellent features; scheduling is simplified and variable-size packets can be directly switched. In this report we present the performance evaluation of a buffered crossbar directly switching variable sized packets. We describe in detail the simulator developed for the experimental analysis, and our set of experiments showing that the buffered crossbar switching variable-size packets outperforms most of the existing packet switch architectures.

Performance Evaluation of Variable Packet Size Buffered Crossbar Switches

Georgios A. Passas[1,2]

Computer Architecture & VLSI Systems (CARV) Laboratory,
Institute of Computer Science(ICS)
Foundation for Research and Technology — Hellas(FORTH)
Science and Technology Park of Crete
P.O. Box 1385, Heraklion, Crete, GR-711-10 Greece
email: passas@ics.forth.gr

Technical Report FORTH-ICS/TR-328 — November 2003

Work Performed as Diploma Thesis at the Depart. of Computer Science, Univ. of Crete, under the supervision of prof. Manolis Katevenis

Keywords: Variable Packet Size Buffered Crossbar, Performance Evaluation, Simulation, Traffic Generators, Delay, Throughput

[1]ICS-FORTH, P.O. Box 1385, GR-711-10 Heraklion, Crete, Greece. E-mail: passas@ics.forth.gr
[2]Department of Computer Science, University of Crete, Heraklion, Crete, Greece.
E-mail: passas@csd.uch.gr

Abstract

The crossbar is very popular for the implementation of switches with a moderate number of ports. Unbuffered crossbars (IQ - architecture) require complex switch matrix scheduling algorithms that operate on fixed-size cells. On the other hand, buffered crossbars (CICQ - architecture), use small buffers at the crosspoints and provide excellent features; scheduling is simplified and variable-size packets can be directly switched. In this report we present the performance evaluation of a buffered crossbar directly switching variable sized packets. We describe in detail the simulator developed for the experimental analysis, and our set of experiments showing that the buffered crossbar switching variable-size packets outperforms most of the existing packet switch architectures.

Acknowledgments

I would like to thank Nikos Chrysos for supervising my work and for providing the architeture of the simulator. Professor Manolis Katevenis is thanked for his guidance and discussions throughout this work. I would also like to thank Georgios Sapunjis and the members of the switch architecture group for the helpful discussions, as well as my friends and family for their support. Lastly, ICS-FORTH is thanked for providing technical support.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, packet switches are the basic building blocks for constructing high-speed networks that employ point-to-point links. As the demand for network bandwidth is continuously increasing, attention is turning to cost-effective switch scalability. In that direction, crossbars are very popular for the implementation of switches with low to modest numbers of ports (4×4 to 128×128). Additionally, crossbar switches are used in the core of switching fabrics with greater number of ports.

Although, crossbar fabrics suffer from inherent drawbacks, such as O(N^2) complexity-cost and the need for input queues (Virtual Output Queues-VOQs), due to the well-known Head Of Line blocking (HOL) [1] they offer many advantages: they are simple, internally non-blocking and rather available in marketing.

Most crossbars in research and industry are unbuffered ones. An unbuffered crossbar switch is configured using a centralized scheduler and uses fixed sized cell as a transfer unit. The scheduler implements a switch matrix scheduling algorithm [1] [2] [3], based on parallel and iterative grant-request-accept cycles. Variable-size packets are segmented at input ports, transfered across the switch and finally reassembled into packets at output ports. This operation introduces many inefficiencies. First, when the cell time is short, the centralized scheduler cannot practically achieve both high throughput and quality of service (QoS) guarantees; this coupled with the overhead due to packet segmentation[2] brings the need for internal speed-up [3] - usually a factor of two to four to commercial products [4]. Second, the segmentation at inputs brings the need for reassembly at outputs, which requires significant egress buffering (CIOQ - Combined Input-Output Queuing).

On the other hand, buffered crossbars [5] [6] benefit by using small buffers at the crosspoints (CICQ - Combined Input-Crosspoint Queuing). For buffered

[1] Under certain conditions HOL limits the maximum achievable throughput to $2 - \sqrt{2} \approx 58.6\%$.

[2] The worst case situation is a stream of back-to-back packets of length 1 byte greater than the cell size. For this case we need a speed-up near two

[3] Speed-up is defined to be the ratio of the line card's bandwidth into/from the switch core to the link speed.

crossbars we do not need a centralized scheduler because packets don't have to be switched to the outputs right away but they can be buffered at the crosspoints [7]. Thus the scheduling is distributed at the distinct inputs and outputs: input transmissions are independent of each other and of the output ones, resulting to a simplified scheduling. In addition, buffered crossbars can directly switch variable size packets; there is no need for segmentation and reassembly. Thus, no speed-up and no egress buffering is needed. Speed-up and buffering are the main contributors to cost in switch design, so buffered crossbars have the potential to reduce the cost of packet switches.

The buffered crossbar concept dates at least as far back as 1987 [8]. However the lack of the suitable technology for a single chip buffered crossbar made the buffered crossbar idea not implementable. Recently, with the current million transistor technology many groups [6] [9] [10] have studied the buffered crossbar architecture. However little work has been done up to now for one of its main advantages : the direct operation on variable sized packets. This key feature and more precisely the performance evaluation of a buffered crossbar natively supporting variable sized packets, is studied in detail in this report.

We have to mention that this report does not examine QoS and priority level issues. Instead, the main purpose is to show that the architecture we describe in session 2, performs very well directly switching variable sized packets. So, we consider only best-effort traffic.

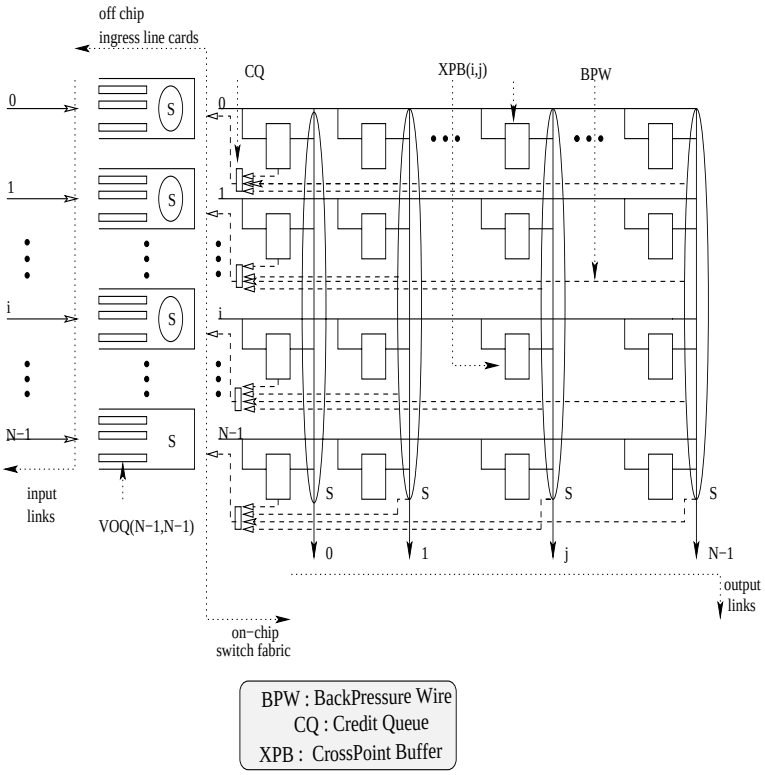

Figure 1: Generic Architecture.

2 Generic Architecture

The proposed architecture uses a crossbar for interconnecting the cards on which ports reside. As shown in fig.1, we assume buffers at the two potential contention points: inputs and crosspoints (thus the name Combined Input Crosspoint Queuing-CICQ). At each input i, i $\in$(0, N-1) there are N large off-chip memories (e.g. DRAMs), Virtual Output Queues(VOQs), identified by an input-output pair (i, j), j$\in$(0, $N-1$). At each crosspoint (i,j), there is a small on-chip buffer (SRAM), dedicated to the input VOQ (i,j). Each input i also maintains a scheduler serving the N competitive VOQs, by selecting one among them and forwarding its HOL packet to its dedicated crosspoint buffer. At each output j, resides a scheduler su-

pervising the N crosspoint buffers(i, j). It keeps selecting the next crosspoint buffer from which the head packet will be forwarded to the respective output j. A simple, credit-based flow control ensures that the small crosspoint buffers will never overflow.

Concerning the flow control scheme, we assume that each input i maintains N credit counters: one for each VOQ. Initially, the value of these counters equals the crosspoint buffer size. Whenever an input scheduler forwards a packet from a VOQ, it decrements its respective credit counter by the size of the forwarded packet. Whenever an output scheduler forwards a packet from a crosspoint buffer (i,j), it sends a credit to the VOQ (i,j) [4]. The credit needs only to contain information for the VOQ's input-output pair. Each input maintains N simple, small FIFO queues, one per VOQ, in order to "remember" the size of those packets send but still not acknowledged. So, if an input receives a credit for VOQ (i,j), it knows that it has to increment VOQ's credit counter by the size, stored at the head of the j-th FIFO queue. Via the above optimization, we reduce the backpressure overhead, since credits do not have to carry information for the packet's size.

An input port scheduler selects among that input's VOQs the one from which the next packet will be forwarded to the crossbar chip. The selection is performed in round-robin (RR) manner, among the eligible VOQs; a VOQ is eligible if and only if : (i)it is not empty and (ii)it will not cause its corresponding crosspoint buffer to overflow, that is the value of its credit counter is greater or equal to the size of its head packet. Identically, an output scheduler serves in RR discipline the next eligible crosspoint buffer. Note that a crosspoint buffer is eligible if and only if it is not empty.

In the proposed architecture, at every queue we employ cut-through operation: whenever the packet's header reaches a VOQ, input scheduler can start transmitting it (if its VOQ is eligible); whenever the packet's header reaches a crosspoint buffer, output scheduler can start forwarding it to the output port.

We have to mention that we decided to place input schedulers off-chip, at input line cards, cause :(i)these schedulers would add to the cost of the crossbar chip,

[4]The credit is forwarded to the input as soon as the output starts transmitting a packet.

(ii)ingress line cards would have to communicate to the crossbar chip the head packet of each VOQ, (iii)the scheduler's decision would need to travel to the line card, increasing the scheduler's latency.

More details about the architecture and the hardware implementation appear in [7].

3 Simulator Description

This section concerns a description for the simulator developed toward the performance evaluation purpose. We preferred to create our own simulator instead of using standard libraries (e.g. CSIM), considering it is a more flexible solution. Thus, we developed a software model in $C++$ which matches the hardware of the proposed architecture at a system level.

3.1 Simulator Architecture/Entities/Events

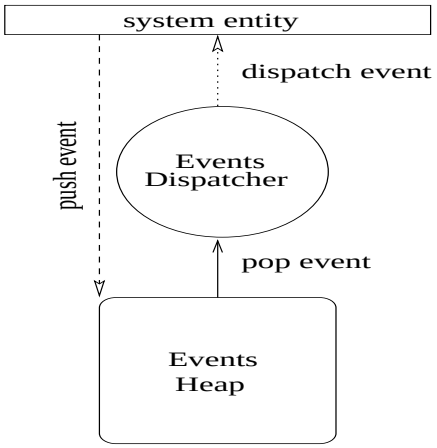

Figure 2: The core event-based architecture for the simulator.

Specifically, the software model was "built" around an event-based architecture, resulting to an event-driven simulator[5]: the system consists of entities which communicate with each other via events. An event is another entity that holds information about the time of the event's birth and its type. The events are stored in a heap data structure, which serves as an "event-list" - holds the events sorted by time of birth order. A dispatcher keeps pulling out events from the heap and delivering them to the system's entities; the entity receiving an event, handles it and depending on its type, generates a new one that stores (pushes) into the heap. The events

[5]We also assume a single execution thread.

are popped (by the dispatcher) from the heap in increasing time of birth order and the simulation time is procedurally advanced to the point when the popped event occurred. We mention here that the operation of the simulator in a time-slotted manner could not be feasible for the IP switching case. Time-slotted simulation could work well only when modeling (fixed size - ATM) cell switching.

Below, we describe the basic entities forming the simulator and the main event types they communicate to each other. We basically assume 7 entities and 7 event-types.

First, the *packet* entity models the IP packets arriving at the switch ingress line cards. It contains fields which specify the discrete packet's id, the input port it reaches, the output port it is destined to, its arrival/departure time to/from the switch and its size (payload plus header size). As described in session 2 packets are buffered in FIFO queues. We model them with the *queue* entity, which maintains a pointer to the head packet and supplies enqueue/dequeue operations.

```
class pkt{
private:
  int pkt_id; //the packet's id
  int input_id,output_id; //packet's input-output port
  int priority; //packet's priority
  int size;    // packet's size
  class pkt *nxt,*prev;  //next/previous packet in queue
  double arr_time,dep_time; //packet's arrival/departure time
public:
  pkt();   //constructs a new packet
  ~{}pkt(); //destructs a packet
  void  set_attr(int pkt_id,int input_id,int output_id,int priority,
                 int size , double arr_t,double);  //sets the packets attributes
  void print();  //for debugging purpose
}
```

```
class entity_queue{
private:
  int size;          // queue's size
  pkt *head,*tail;  //pointer to the head and tail packet
public:
  entity_queue();             //constructs an empty queue
  ~entity_queue();           //destructs a queue
  int enqueue(pkt *packet); //enqueues packet
  pkt * dequeue();         //dequeues packet
  int not_empty();        //checks for empty queue
  void print();          //for debugging purpose

};
```

The whole input line card is modeled with the *input controller* entity. Each *input controller* is identified by the input it resides and maintains a pointer to a table of queues, corresponding to its input's VOQs. In addition, *input controller* implements scheduling operations, in order to specify the VOQ to be served; the scheduling is performed in WRR discipline but in our simulations we assume that the weights of all flows are equal.

```
class input_controller{
 private:
  int input_id; //controller's input port
  queue **VOQs; // N array of pointers to queues (the VOQs)
  int enqueue_packet(pkt *packet,int VOQ_id); //enqueues packet to VOQ
     //VOQ_id of input input_id
  pkt *dequeue_packet(int VOQ_id); //dequeues a packet from VOQ
          //VOQ_id of input input_id
  int *credits; //credits for each VOQ
```

```
  int exists_eligible_flow(); //checks if there is an eligible flow at input input
  double *vtime;  //virtual time for WRR
  void scheduling_discipline_for_priority(int priority_id) //implies WRR and
                                                    //finds VOQ_to_be_served
  int VOQ_to_be_served;  //the VOQ to be served by the scheduler
  int start_scheduling_expired; // variables
  int new_transfer; //   for the scheduler's
  int state;      // state machine
 public:
  void set_input_id(int); //sets the input id for the controller
  input_controller();    //constructs an input line card module
  ~input_controller();  //destructs an input line card module
  void handle_event(event *ev, HEAP *); //checks the type of the ev
                                        //and pushes a new one to the heap
  void goto_schedule_state_or_remain( double time, HEAP * events_heap); //checks i
                                                  //and performs scheduling or chang
  void print(TIME msec); //for debbuging purpose
};
```

Similarly, *output controller* is specified by the output port it resides. It maintains a pointer to a table of queues, corresponding to the crosspoint buffers of that output. It performs WRR scheduling, thus specifying the next crosspoint buffer from which the next packet will be forwarded to the output.

```
class output_controller{
private:
  int output_id;
  queue *priority_level; //N array of pointers to queues (the xpoint buffers)
  int enqueue_packet(PACKET Packet,int priority_id,int xpoint_buffer_id); //enqueu
                                          //packet to VOQ VOQ_id of input input_
```

```
  pkt *dequeue_packet(int xpoint_buffer_id); //dequeues
                  //a packet from xpoint buffer xpoint_bufer_id of output output_
  int exists_eligible_flow(); //checks if exists non empty xpoint buffer
  double *vtime;      //virtual time for WRR
  void scheduling_discipline(int priority_id); //implies WRR and
                               //finds xpoint_buffer_to_be_served
  int xpoint_buffer_to_be_served;       //the xpoint buffer to be served by
                                       //the output scheduler
  credits_controller ** credit_controllers; // used to pass credits with
   //on-chip zero delay
  int start_scheduling_expired;          //variables for the schedulers state mach
  int new_transfer;
  int state;
public:
  void set_output_id(int);  //specifies the output port for the module
  output_controller();     //constructs an output module
  ~output_controller();   //destructs an output module
  void set_credits_controllers(credits_controller **);
  void handle_event(event_link, HEAP * events_heap);     //similarly to input cont
  void goto_schedule_state_or_remain(  TIME , HEAP * ); //similarly to input contr
  void print();       //for debugging purpose
};
```

We assume that the schedulers at input and output ports operate in accordance to a specified finite state machine. For the description of that state machine refer to [11].

Statistics monitor is the module that collects the statistics for the simulation. It computes the packets' average/max delay, the delay's standard deviation, the switch throughput, as well as when to stop simulation (see section 3.2). It prints

the results to a specified file.

```
class statistics_monitor{
 private:
  int nofpackets_counter;    //total packets left the switch during the simulatio
  int packets_per_exp;      //total packets left the switch during the experiment
  double aggregate_delay;  //cumulative delay for all packets
  double average_delay;
  double max_delay;
  double aggregate_bits;  //cumulative bits for packets left the
                         //switch during the experiment
  double throughput;
  double sample_variance; //variables for the stop-simulation decision
  double sample_variance_updated;
  double sample_mean;
  double sample_mean_updated;
  double delay_i;        //variables for computing delay's standard deviation
  double delay_i_s;
  double sum_delay_i;
  double sum_delay_i_s;
  double standard_deviation;
 public:
  statistics_monitor();       //construct a monitor
  ~statistics_monitor();     //destruct the monitor
  int collect_statistics(event_link Event);  //collects the new statistics
  int compute_confidence_interval();  //decides when to stop simulation
  void print(FILE *fp);    //prints the statistics to file fp.
};
```

Lastly, *traffic controller* generates the packets that arrive at the switch ingress

line cards. It provides various traffic patterns (see section 5) and destination distribution options.

PACKET_AT_INPUT event is generated by the traffic generator when a new packet is generated [6] . PACKET_HEADER_REACHES_CROSSPOINT event is generated by the input controller notifying that the header of the packet has reached the crosspoint. INPUT_SCHEDULING_STARTED and INPUT_SCHEDULING_COMPLETED events are generated by input controller according to its state machine and point the start/completion of a scheduling operation. OUTPUT_SCHEDULING_STARTED and OUTPUT_
SCHEDULING_COMPLETED events are similarly generated by output controller. PACKET_TRANSFER_TO_OUTPUT_COMPLETED event is generated by output controller when the packet has been totally transmitted to output. CREDITS_AT_INPUT event is generated by credit controller notifying the arrival of a credit at input.

For example when traffic generator generates a packet a new event PACKE T_AT_INPUT is created and pushed to the heap. Then the events dispatcher passes the event to input controller; input controller enqueues the packet to the corresponding VOQ and handles the event by changing its state. If scheduler is idle it makes a scheduling decision,dequeues a packet from a VOQ and creates a new event PACKET_HEADER_REACHES_CROSSPOINT.

Before starting the simulation we have to specify the parameters for the system. Particularly we have to specify the switch size, the rates of external, internal and credit lines, the crosspoint buffer size, the propagation/memory access/scheduling time values, as well as the parameters for the traffic generator. That is the parameters defining the traffic pattern to be generated and the distribution for the packets' destinations.

Lastly, we have to report that in order to compare the proposed architecture to the CIOQ one and to the ideal output queuing we have also developed software models simulating the operation of the iSLIP switch and the switch employing output queuing. For these models we assume the same event-based architecture. Details appear in the Appendix.

[6] We assume that a packet is generated at zero time and concurrently reaches the input port

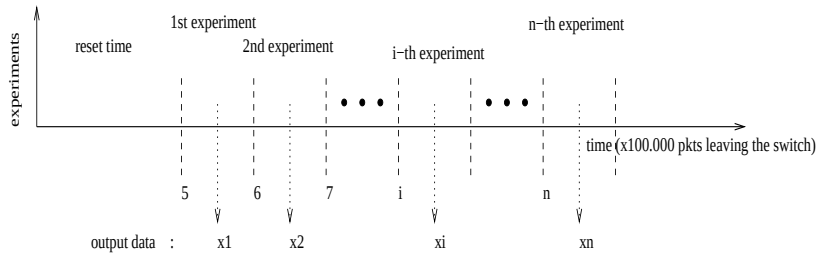

Figure 3: Simulation's Evolution.

3.2 Deciding When to Stop Simulation

We adopt the method described in [12] in order to determine when to stop simulation. We briefly discuss it and then fit it to our special case.

In general, when simulating a system we are interested in measuring a quantity X, which is the output data of a simulation. We assume that a simulation consists of multiple simulation runs (or experiments). Usually, X is a random variable and we are interested in its average value θ. In order to compute θ, we consider a variable-estimator $\bar{x}$ of θ: $\bar{x}$= $\sum_{i=1}^{n} \frac{x_i}{n}$, where x_i is the output of the simulation's i^{th} run. The problem of finding when to stop simulation actually matches to the one of finding the value for n, such as $\bar{x}$ is a good estimator of θ; that is $\bar{x}$ converges to θ.

To examine when $\bar{x}$ converges to θ, we need another metric: "mean square error" (mse), specified by the following formula: $mse = \frac{\sigma^2}{n}$, where σ is the standard deviation for $\bar{x}$. The difficulty now is that we do not know the value for σ^2. Thus, as we did for $\bar{x}$, we consider an estimator for σ^2 : $S^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$, measured during the simulation's evolution. We can now specify the algorithm for the simulation-stop decision:

1. Specify a value d for the $\bar{x}$'s standard deviation.
2. Generate at least 30 values x_i (i.e. run the simulation at least 30 times and collect the output data).
3. Continue generating values x_i until $\frac{S}{\sqrt{k}} < d$, where k the number of x_i values generated.

For instance (see fig. 3), when measuring e.g. the average delay for the packets going through the switch, we start the simulation and wait until 500.000 packets have left the switch; this concerns the *reset time*, which is used in order to skip the initial/transient phenomena, before the switch stabilizes. Then we consider n successive experiments $ex_1, ex_2, ..., ex_i, ..., ex_n$ with a duration (for each of them) specified by the time required for 200.000 packets to leave the switch. In each experiment ex_i we measure the average value x_i for the packets' average delay. In order to determine n's value (i.e. when to stop simulation), we apply the algorithm described above. Setting $d = 0.04$, we get that we can stop the simulation when $n >= 30$ and $\frac{S}{\sqrt{n}} < 0.04$. When the latter is achieved, $\bar{x}$= $\sum_{i=1}^{n} \frac{x_i}{n}$ well-approaches the value for packets' average delay, that is we can stop simulation.

4 Traffic Generators Description

We present the traffic generators used for the experimental performance analysis.

4.1 Poisson Arrivals-Pareto Size(PoissPar) Generator

The **PoissPar** traffic generator generates packets of size given by the pareto distribution (subexponential family). We use the pareto distribution because it is heavy tailed, approaching the distribution of packet sizes in Internet: most packets are small and rarely appear large packets. To be more precise we consider the bounded pareto distribution with a minimum value 40 (the minimum packet size, $MinPktSize$) and maximum value 1500 (the maximum packet size, $MaxPktSize$). We set its expected value to 370; that is the mean packet size ($MeanPktSize$) is 370 bytes. For packet interarrival time we used the exponential distribution so as the packet birth is a poisson process (see fig. 4). In order to vary the load for the generated

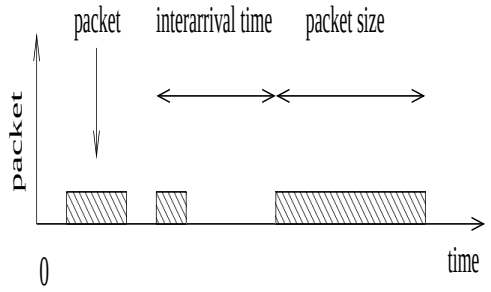

Figure 4: Poisson Arrivals-Pareto Size Traffic Model.

traffic, we vary the mean interarrival time ($MeanInterArrTime$) value. The load generated by the **PoissPar** generator is given by the formula:

$$load = \frac{2 \cdot MeanPktSize}{MeanInterArrTime + \frac{MeanPktSize}{ExtLineRate}} \quad (1)$$

where $ExtLineRate$ stands for the capacity of the link feeding a switch port[7].

[7] In our simulations the link's capacity is 10Gbps.

4.2 Synthetic-Backbone(SynthBackb) Traffic Generator

Trying to represent as much as possible the Internet backbone traffic, we consider that it is dominated by two groups of application level "conversations"[8]: bulk and interactive conversations [13]. The bulk conversations include applications such as http page responses or FTP transfers while the interactive conversations include TELNET-like applications and TCP acknowledgments. We also claim that the traffic going through a backbone router, results by multiplexing the above conversations.

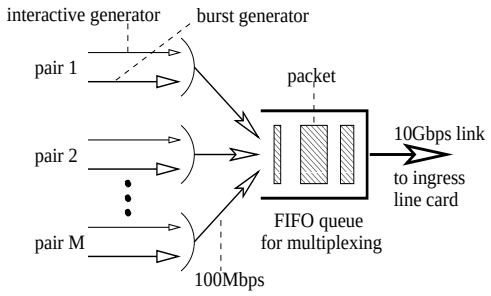

Figure 5: SynthBackb Generator. The aggregate load of a pair of sources is 100Mbps. For $M \times 100$Mbps load we multiplex M pairs.

So, we developed two traffic generators (sources) that generate the aforementioned conversations: interactive generator (– IG, generating interactive conversations) and burst generator (– BG, generating bulk conversations). In fig 5 we demonstrate how we use these generators to feed the switch with realistic traffic: the traffic arriving at each ingress line-card is generated by multiplexing M distinct pairs of sources in a FIFO queue. Each pair consists of an IG and a BG.

The synthetic traffic model consists of two layers, namely session and packet as illustrated in fig 6.

A session contains several packets depending on the application it belongs to.

[8]We define a conversation to be a stream of packets traveling between the end points of an association.

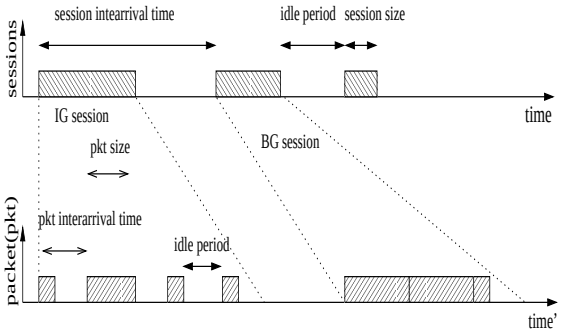

Figure 6: Synthetic Traffic Model. BG sessions emulate bulk transfers. IG sessions emulate interactive conversations.

For instance in a www browsing, a (BG) session corresponds to the downloading or uploading of a www document; the packets within this session form a bursty sequence, i.e. are coming back-to-back. On the other hand, a (IG) session corresponding to an application like TELNET consists of small packets which are delimited by idle periods. Idle periods also delimit sessions and emulate the thinking time of the application user, that is the thinking time during a TELNET "conversation" or the time after an http document retrieval [14].

To be more precise, the duration of a session generated by IG and BG follows the pareto distribution with mean value 125 packets and 8 Kbyte respectively [15]. All sessions are generated according to a Poisson process. Packets within IG sessions vary from 40 to 44 bytes and their interarrival time follows the exponential distribution. A BG session consists of packets, having the same size –1500 (x%) or 552 (y%) or 576(z%) bytes– except for the last one; x, y, z [9] and the ratio of rates between IG and BG are selected so that 60% of all generated packets have size between 40-44 bytes, 18% 552 or 576 bytes, and 18% 1500 bytes. These analogies seem to characterize the Internet traffic, as many sources of statistics and many studies on the nature of Internet traffic indicate. Representative studies can be found in [16]. In **SynthBackb** generator, each pair of sources has aggregate rate

[9] x=72.66, y=13.39, z=13.95; refer to Appendix for details.

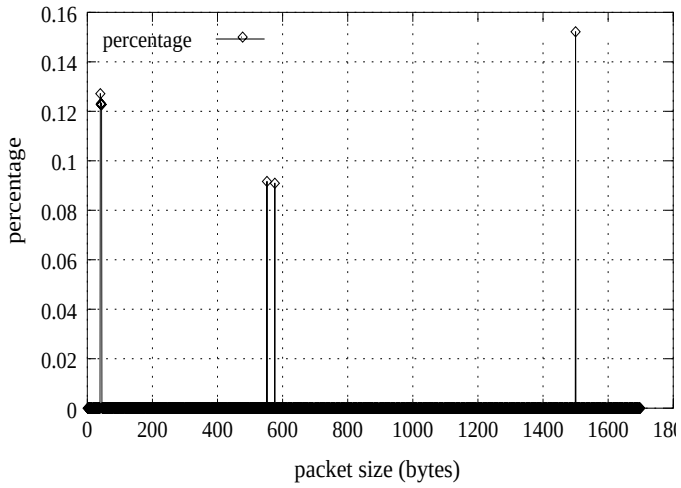

Figure 7: Packet Size Distributions. Percentage.

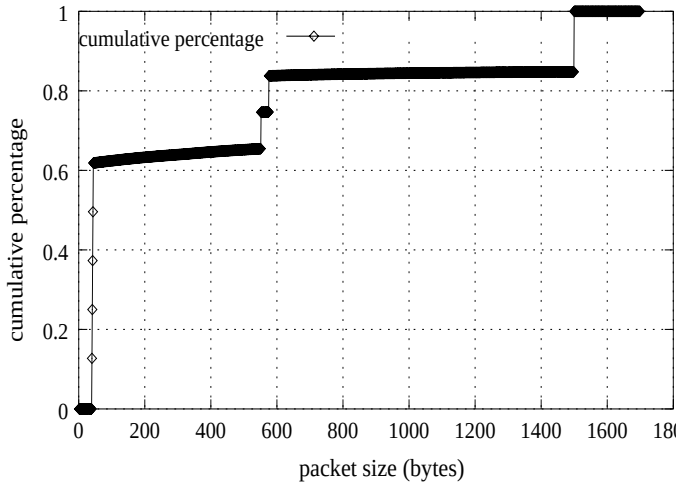

Figure 8: Packet Size Distributions. Cumulative Percentage. Observe the knees at the discrete packet sizes 40, 41, 42, 43, 44 and 552, 576 and 1500 bytes.

100 Mbps; so for a M * 100 Mbps load we multiplex M pairs.

Table 1 briefly shows the mathematical distributions while table 2 shows the values for the parameters of the traffic model. In fig. 7 and 8 we demonstrate re-

spectively the percentage and cumulative percentage for the sizes of the packets belonging to the stream generated by **SynthBackb**.

Quantity	Distribution
Session Size (pkts)	Pareto Ditribution
Session Interarrival Time (usec)	Exponential Ditribution
Packet Size (bytes)	Pareto Ditribution
Packet Interrarival Time (usec)	Exponential Distribution

Table 1: Distributions for the Synthetic Traffic Model

4.3 Burst Generator (BurstGen)

In order to evaluate the performance of the switch under extreme conditions we developed the burst generator (**BurstGen**). Actually, we isolated the BG of the SynthBackb generator described in the previous section and considered a single BG feeding each of the switch ports. In order to vary the load of **BurstGen** we vary the session interarrival times of BG. The load of **BurstGen** is equal to:

$$load = \frac{2 \cdot MeanBurstSize}{MeanInterArrTime + \frac{MeanBurstSize}{ExtLineRate}} \quad (2)$$

where ExtLineRate stands for the capacity of the link feeding a switch port and MeanBurstSize is equal to 8KBytes.

Parameter	Value
avg IG pkt size	42 bytes
avg BG pkt size	1032 bytes
avg IG session duration	125 pkts
avg BG session duration	8 Kbytes
IG's load	6.7 Mbps
BG's load	93.3 Mbps
IG's capacity	100Mbps
BG's capacity	100Mbps
avg IG session interarr. time	128 usec
avg BG session interarr. time	126 usec
avg IG pkt interarr. time	49 usec

Table 2: Parameters' values for the Synthetic Traffic Model

5 Simulation Results

In this section we present the performance evaluation of the proposed architecture via simulation. First, we explicitly study the influence of the Round Trip Time (RTT) and the buffer size on performance and then we present some experiments where we measure the delay and throughput of packets crossing the switch, using the traffic patterns described in session 4. We compare the proposed architecture to output queuing (OQ), which is the ideal system, to the iSLIP switch, a representative and efficient example of the unbuffered crossbar family, as well as to buffered crossbar switching fixed sized cells.

5.1 Simulation Parameters

In all experiments we assumed 32×32 switch with port speed of 10Gbps(Line_Rate). For simplicity we considered no internal packet header overhead and thus no internal speed-up. Our input line-cards and crosspoint buffers implement cut-through operation while the credit line rate is such that the duration of a credit transmission equals a minimum packet transmission time [17]. Credits destined to the same input line-card are sent in FIFO order.

Concerning the RTT between input line cards and switch fabric (see fig. 9), it results as the sum:

$$RTT = RTT_i + RTT_j = (MD + ISD + PPD) + (OSD + CPD + CTD) \quad (3)$$

where MD stands for VOQ memory access delay, ISD for input scheduling delay, PPD for packet propagation delay, OSD for output scheduling delay, CPD for credit propagation delay and CTD for credit transmission delay.

Setting the above parameters to the values shown in table 3 we get that the RTT in our system equals to 500 byte times, corresponding to 400 ns at 10 Gbps line rate.

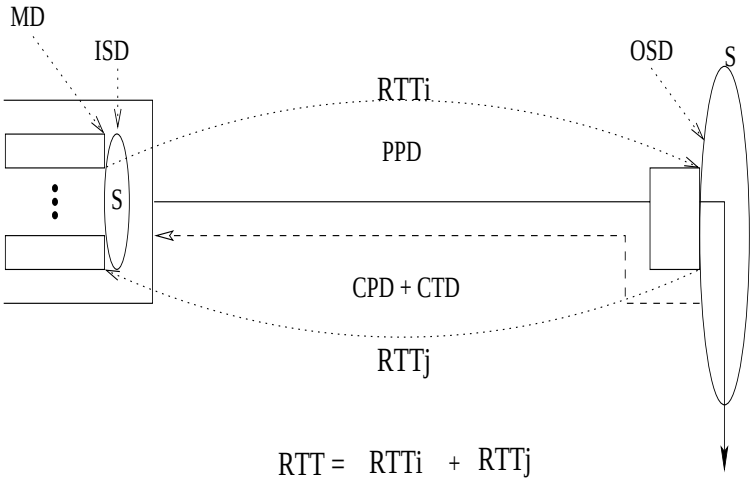

Figure 9: $RTT = RTT_i + RTT_j$

5.2 On Crosspoint Buffer Size Requirements

In this section we present why the use of crosspoint buffer size equal to $MaxPktSize+ RTT \times Line\_Rate$ is a reasonable choice.

5.2.1 Crosspoint Buffer Size less than $MaxPktSize + RTT \times Line\_Rate$

We assume a single, persistent flow, i.e. load is 10 Gbps. For crosspoint buffer size (B) varying from 1400 to 2400 bytes we measure the output utilization as a fraction of 10Gbps. We repeat the experiment for RTT values varying from 250 to 700 byte times. First, (realistic case), we let the packets of the flow being generated by **SynthBackb**. Next, we experiment with a worst case scenario where we continuously alternate between packets $p1$ and $p2$ with sizes $s1$, equal to $MaxPktSize$ and $s2$, equal to max(B-($MaxPktSize-1$), $MinPktSize$) bytes; $s1$ and $s2$ have been selected so that *(a)* $s2$ is as small as possible, while *(b)* $p2$ is able to block $p1$ at the input. Condition *(b)* creates the necessary condition for underutilization, and *(a)* maximizes the duration of this possible underutilization.

Fig. 10 shows that output underutilization occurs for every B less than $MaxPktSize+$

Parameter	Value (nsec)
MD	80
ISD	30
PPD	114
OSD	30
CPD	114
CTD	32

Table 3: System Parameters

$RTT \times Line\_Rate$; however, by employing a crosspoint buffer size equal to $MaxPktSize + RTT \times Line\_Rate$ full output utilization is achieved. This happens because if B equals $MaxPktSize + RTT \times Line\_Rate$, we impose that $p1$ will be blocked at the input only if $s2$ is greater than $RTT \times Line\_Rate$. But in this case, when $p1$ will be ready for transmission at the output after receiving $p2$'s credit (i.e. RTT times after starting transmitting $p2$), the output will still be busy transmitting $p2$, because its size is greater than $RTT \times Line\_Rate$. So, with this buffer size, full output utilization is guaranteed.

Under **SynthBackb** arrivals the knee at crosspoint buffer size $MaxPktSize + RTT \times Line\_Rate$ is also observable (fig. 10), but not as strongly as with the aforementioned worst-case scenario.

The conclusion is that we have to place crosspoint buffers of at least $MaxPktSize + RTT \times Line\_Rate$ if we want full output utilization to be achieved. In the proposed architecture and in our simulations we assume 2KBytes buffer size (resulting as $MaxPktSize + RTT \times Line\_Rate$, where MaxPktSize=1500 bytes and RTT=500 byte times).

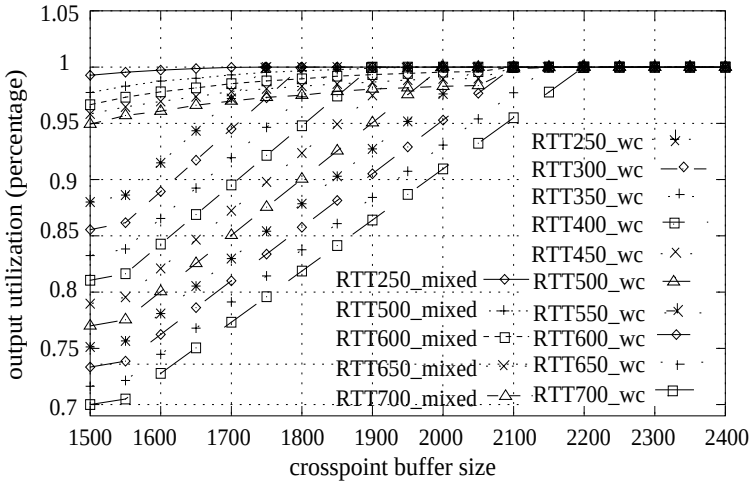

Figure 10: Buffer Size Less than $MaxPktSize + RTT \times Line\_Rate$. For buffer size less than 1500 bytes, all measurements for output utilization are zero. Upper curves stand for **SynthBackb** traffic, lower curves for worst case.

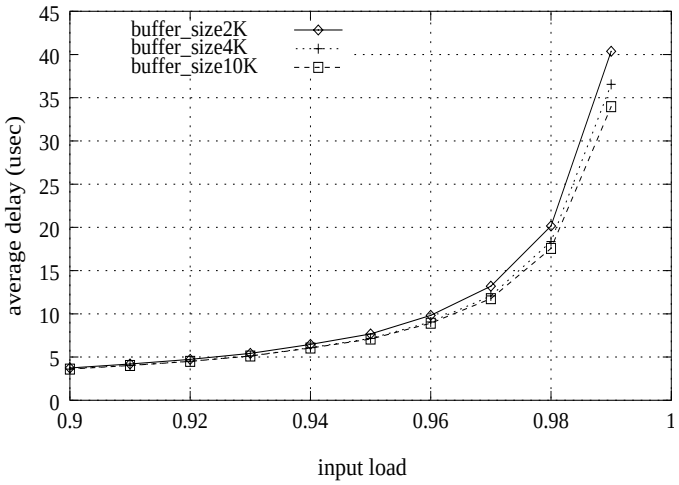

Figure 11: Buffer Size Greater than $MaxPktSize + RTT \times Line\_Rate$. Packets' average delay under **PoissPar** traffic.

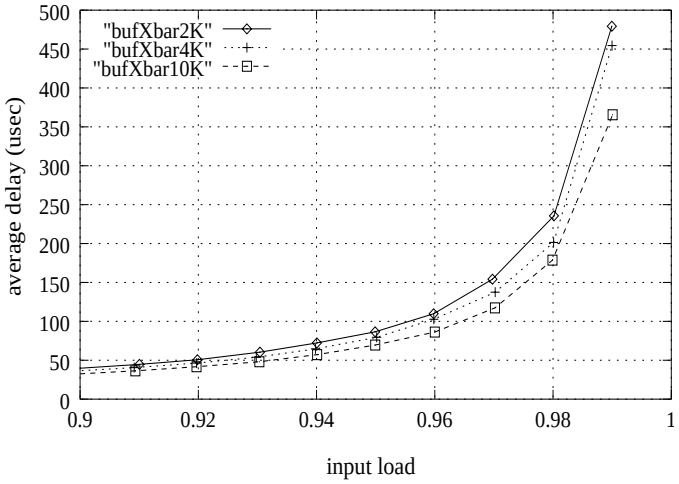

Figure 12: Buffer Size Greater than $MaxPktSize + RTT \times Line\_Rate$. Packets' average delay under **BurstGen** traffic.

5.2.2 Crosspoint Buffer Size greater than $MaxPktSize + RTT \times Line\_Rate$

Next, we experimented for buffer size greater than $MaxPktSize + RTT \times Line\_Rate$ using the **PoissPar** and **BurstGen** Generator, assuming all flows are active and measuring the average delay of packets that cross the switch. Fig 11 and fig 12 show the results. It is observable that under the **PoissPar** traffic generator we actually don't benefit enough in performance by increasing the buffer size. However, under the extreme conditions of bursty traffic (**BurstGen**), the performance is better when using crosspoint buffers greater than 2Kbytes, especially for loads greater than 0.94. For instance under load 0.98 we have a reduction in average delay of approximately 24% when using 10Kbytes buffers rather than 2Kbytes. Additionally, in non-uniform traffic we also benefit by increasing the crosspoint buffer size. See below the experiment for unbalanced traffic (fig. 18).

5.3 Delay Experiments

5.3.1 Uniform Traffic

Under uniform traffic, the destination port of each session is chosen uniformly; all packets in a session have the same destination port. The reported delay is the time between the packet's first byte exit-time minus the packet's first byte enter-time, averaged over all packets; This way we measure only the queuing delay of the packets. The results are compared to output queuing (OQ) and to the CIOQ using iSLIP. For iSLIP we consider one iteration, 64 byte segments and various speedup factors.

We experimented with all traffic generators described in session 4. Fig 14 shows the results we get using the **PoissPar** generator, while fig. 15 and 16 show the results for traffic generated by **SynthBackb** and **BurstGen**. Observe that for all traffic patterns the performance of the proposed architecture with a crosspoint buffer of 2KB is very close to OQ; iSLIP with speedup 2 also performs close to the ideal system. Especially for the most "interesting" of the traffic patterns, the **SynthBackb**, the iSLIP switch with no speedup saturates at input load 0.65; for speedup equal to 1.2 it saturates near load 0.8.

Fig. 13 presents the packets' maximum delay for the proposed architecture, iSLIP and output queuing, under the synthetic traffic pattern. For load 0.9, the maximum delay is 1 msec for buffered crossbar, 3.5msec for 1-SLIP with speed up equal to 1.6 and 0.9 msec for OQ.

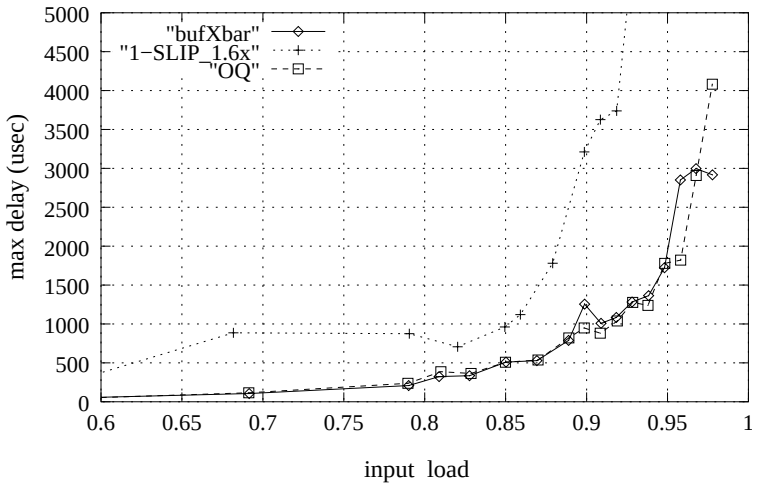

Figure 13: Maximum delay that packets suffer under SynthBackb traffic. Comparative results for buffered crossbar, 1-SLIP with speed-up equal to 1.6 and output queuing

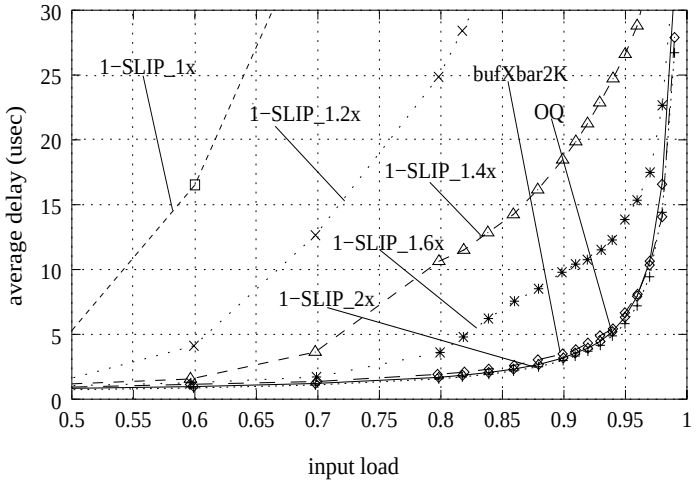

Figure 14: Poisson Arrivals-Pareto Pkt Size Traffic (PoissPar) -Uniformly Selected Outputs. 32×32 switch.

Finally, we measured the standard deviation of delay for the various architectures that we examine, under uniform synthetic traffic. In fig. 17 we observe that the standard deviation of the dealy for the proposed architecture is always less that the respective one for i-SLIP, even with a speed-up equal to 2, while it closely approximates the one for output queuing.

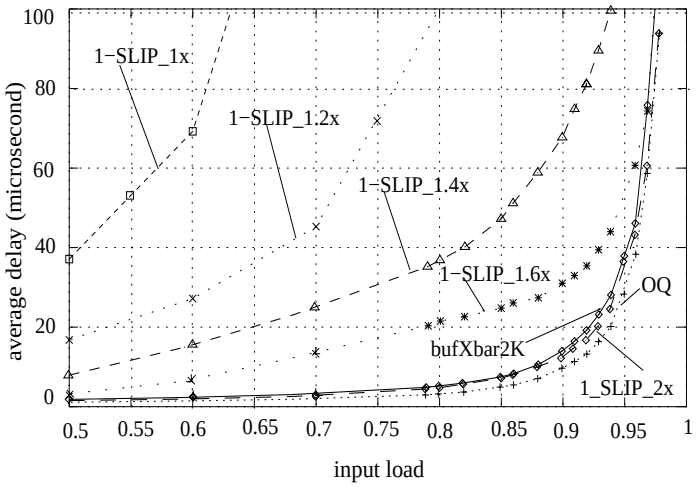

Figure 15: Synthetic Traffic (SynthBackb) -Uniformly Selected Outputs. 32×32 switch.

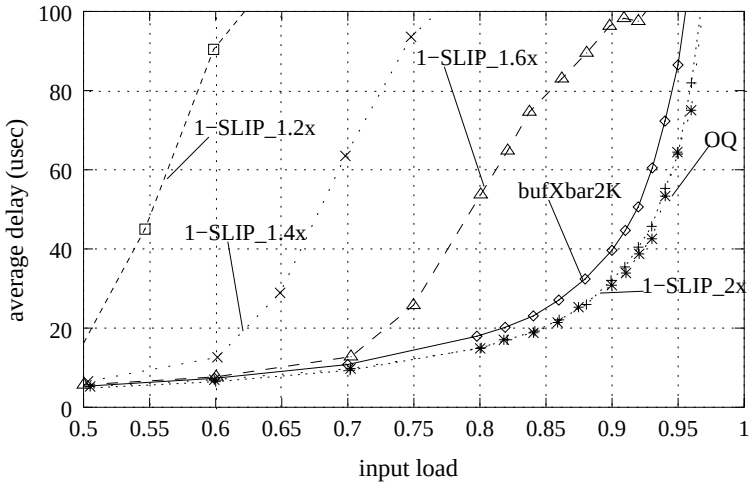

Figure 16: Bursty Traffic (BurstGen) - Uniformly Selected Outputs. 32×32 switch.

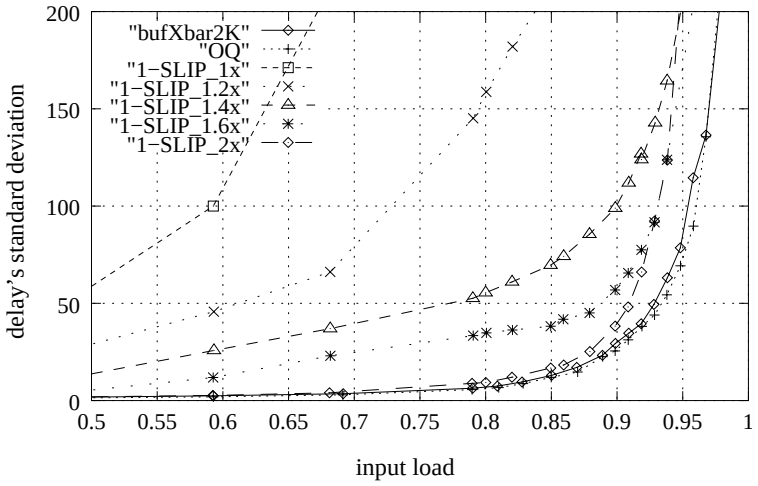

Figure 17: Delay's Standard Deviation. Comparative results for buffered crossbar, 1-SLIP with various speed-up factors and output queuing.

5.3.2 Non-Uniform Traffic

First we experimented with unbalanced traffic, considering the traffic scenario described in [18]. According to this scenario, there is an unbalance factor f, such that input i sends to output i with probability f, and to all other outputs uniformly with collective probability $1 - f$.

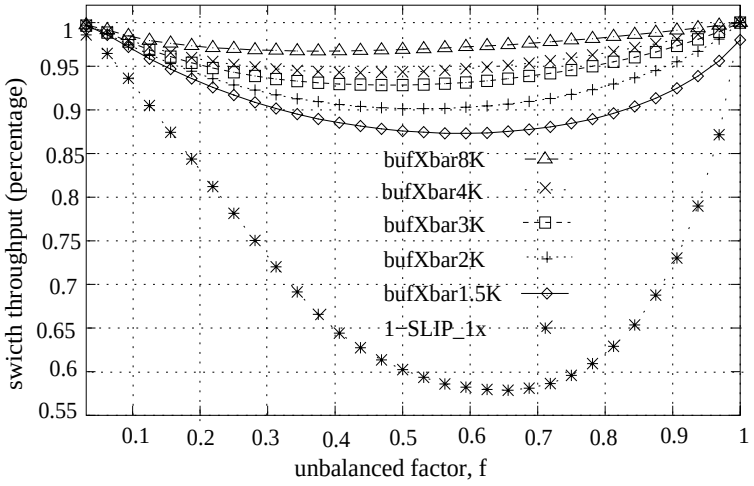

Figure 18: Unbalanced Traffic. 32×32 switch. Load is 100%. For crossbar, buffer size is 1.5K, 2K, 3K, 4K and 8K.

In this experiment we use **PoisPar** arrivals model and we assume all flows are active. We measure the switch throughput as a fraction of the maximum possible one (320 Gbps). For iSLIP (1 iteration, 64 bytes segment, speed-up equal to 1.0) packets have sizes equal to $k \times 64bytes$ (k integer), so as to eliminate segmentation overheads.

Even under this assumption, we find (see fig. 18) that the CICQ architecture with variable-size packets considerably outperforms the CIOQ (iSLIP) switch. With 2KB crosspoint buffer size, the worst switch throughput under this scenario, for the buffered crossbar is 0.90 versus 0.58 for the iSLIP switch.

Next, we experimented under hotspot traffic, using the traffic generator **SynthBackb**. For hotspot traffic, each destination belonging to a designated set of

"hotspots" receives traffic at 100% collective load, uniformly from all sources; the rest of the destinations receive uniform traffic. Without loss of generality, we assume that the hotspots are ports 0, 1, 2 and 3 [19]. The reported delay is the time between the packet's first byte exit-time minus the packet's first byte enter-time, averaged over all packets destined to non-hotspot outputs.

Fig. 19 shows the performance evaluation for buffered crossbar, versus iSLIP and output queuing while fig. 20 shows the comparative results for uniform and hotspot traffic.Under hotspot traffic, in the buffered-crossbar system, we observe that non-hotspot traffic stays unaffected by the presence of hotspots (the uniform and the hotspot plots actually match), due to the isolation/protection that is provided to flows (input/output pairs) by the crossbar/queuing architecture. On the other hand, when we apply hotspot traffic to the iSLIP switch, all flows' performance degrades considerably due to the absence of any flow control.

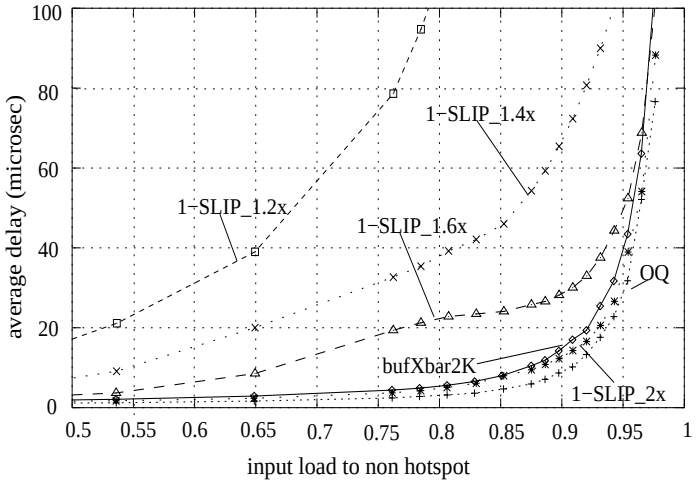

Figure 19: Hot-spotted SynthBackb traffic. Comparative results for buffered crossbar, 1-SLIP with various speed-up factors and output queuing.

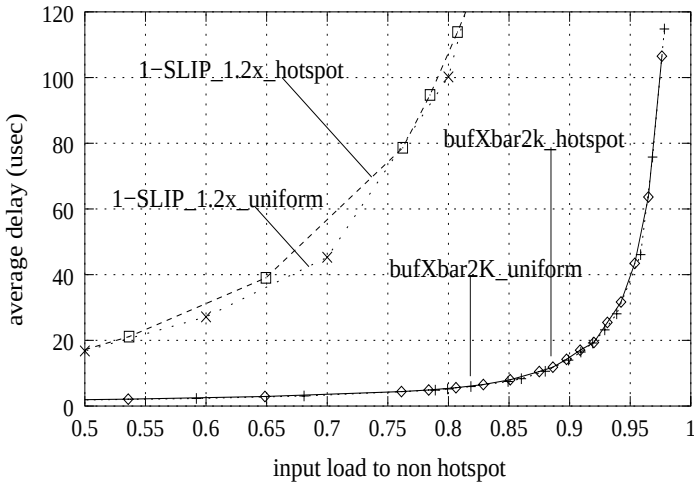

Figure 20: Comparative results for uniform and hot-spotted SynthBackb traffic.

5.4 v-bufXbar vs. s-bufXbar

We compare the buffered crossbar switch natively supporting variable sized packets (v-bufXbar) to the one switching fixed sized cells (s-bufXbar). For s-bufXbar we considered 64-byte segments. Fig 21 shows the average delay that packets suffer in v-bufXbar and s-bufXbar under **SynthBackb** traffic. S-bufXbar needs speed-up, due to segmentation overheads, to perform as v-bufXbar. This speed-up is at least 1.1.

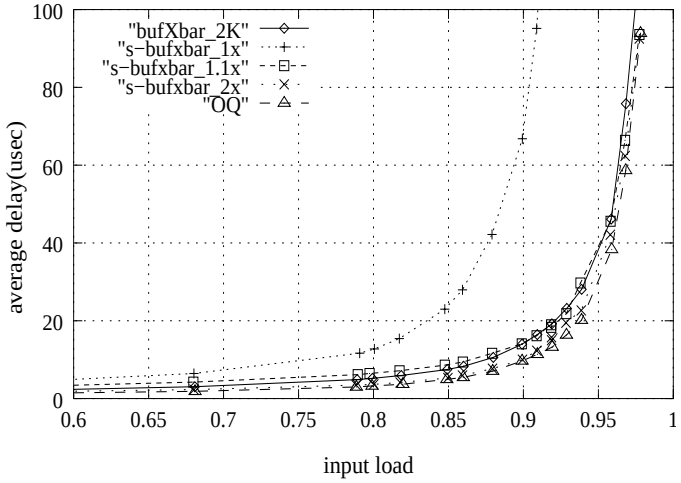

Figure 21: s-bufXbar vs. v-bufXbar. s-bufXbar needs speed-up 1.1.

5.5 Fabric Size Dependence of Performance

In switch design it is desirable for the performance not to degrade when we increase the fabric size. We run simulations for various switch sizes and we concluded that for the proposed architecture the performance is not affected by the fabric size. As fig. 22 shows, for fabric sizes 16×16, 32×32, 64×64 and 128×128 the respective delay plots match. On the contrary, for the iSLIP switch the performance degrades when increasing the switch size, for loads greater that 0.7; we can observe this phenomeno in fig. 22 where we consider speed-up equal to 1.6 and one iteration.

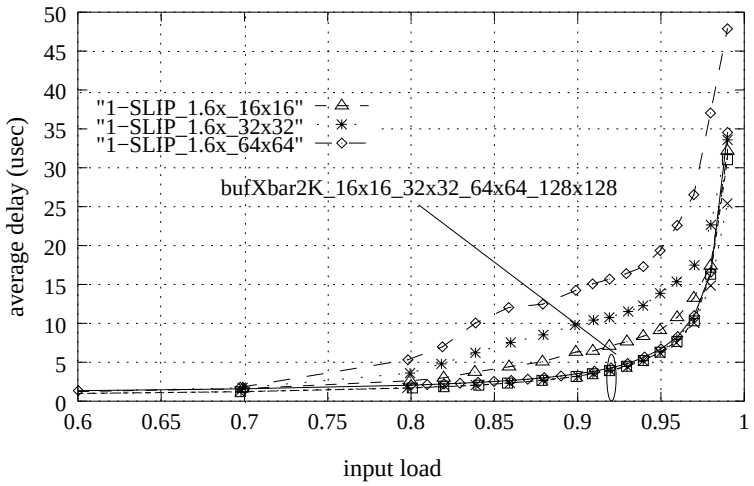

Figure 22: Fabric Size Dependence of Performance. For buffered crossbar the plots match.

In fact, iSLIP normally requires $\log N$ iterations, where N is the switch size, to produce near full matches. So, in the case we have a 64×64 switch we need 6 iterations, which translates to scheduling time approximately equal to 60 nsec[10], obviously unfeasible for the 10 Gbps link speed.

[10] 1 iteration can be performed in approximately 10 nsec [20]

6 Conclusion

We presented an extensive performance evaluation via simulation of a buffered crossbar directly operating on variable sized packets. We showed for various traffic patterns that the buffered crossbar architecture outperforms the iSLIP switch with speed-up less than two and performs very close to the ideal output queuing. Concerning the buffered crossbar architecture we presented why we need a crosspoint buffer size of at least $MaxPktSize + RTT \times Line\_Rate$ bytes and the general influence of buffer size on performance.

References

[1] T. Anderson, S. Owicki, J. Saxe, C. Thacker: “High-Speed Switch Scheduling for Local-Area Networks”, *ACM Trans. on Computer Systems*, vol. 11, no. 4, Nov. 1993, pp. 319-352.

[2] R. LaMaire, D. Serpanos: “Two-Dimensional Round-Robin Schedulers for Packet Switches with Multiple Input Queues”, *IEEE/ACM Trans. on Networking*, vol. 2, no. 5, Oct. 1994, pp. 471-482.

[3] N. McKeown: “The iSLIP Scheduling Algorithm for Input-Queued Switches”, *IEEE/ACM Trans. on Networking*, vol. 7, no. 2, April 1999, pp. 188-201; http://tiny-tera.stanford.edu/∼nickm/papers/ ToN_April_99.pdf

[4] Cyriel Minkenberg, e.a.: “Current Issues in Packet Switch Design”, *HOTNETS*, NJ, USA, October 2002.

[5] D. Stephens, H. Zhang: “Implementing Distributed Packet Fair Queueing in a scalable switch architecture”, *Proc. INFOCOM’98 Conf.*, San Francisco, CA, March 1998, pp. 282-290.

[6] N. Chrysos, M. Katevenis: “Weighted Fairness in Buffered Crossbar Scheduling”, *Proc. IEEE Workshop High Perf. Switching & Routing (HPSR 2003)*, Torino, Italy, June 2003, pp. 17-22; http://archvlsi .ics.forth.gr/bufxbar/

[7] M. Katevenis, G. Passas, D. Simos, Y. Papaeystathioy, N Chrysos: “Variable Packet Size Buffered Crossbar Switches” , ICS-FORTH, September 2003. http://archvlsi.ics.forth.gr/bufxbar

[8] S. Nojima, E. Tsutio, H. Fukuda, M. Hashimoto: “Integrated Services Packet Network Using Bus Matrix Switch”, *IEEE J. Sel. Areas in Communications*, vol. 5, no. 8, October 1987, pp. 1284-1292.

[9] M. Nabeshima: “Performance Evaluation of a Combined Input and Crosspoint Queued Switch”, *IEICE Trans. Commun.*, vol. E83-B, no. 3, Mar. 2000, pp. 737-741.

[10] T. Javidi, R. Magill, and T. Hrabik: “A High-Throughput Scheduling Algorithm for a Buffered Crossbar Switch Fabric” *Proc. IEEE Int. Conf. on Communications (ICC’2001)*, Helsinki, Finland, June 2001, vol. 5, pp. 1586-1591.

[11] N. Chrysos: ”Design Issues of Variable-Packet-Size, Multiple-Priority Buffered Crossbars”, *Technical Report FORTH-ICS/TR-325, Inst. of Computer Science, FORTH,* Heraklion, Crete, Greece, October 2003. http://archvlsi.ics.forth.gr/bufxbar

[12] Ross, S. “Simulation”. Academic Press, San Diego, California, second edition, 1997.

[13] Ramon Cáceres e.a. : “Characteristics of Wide-Arrea TCT-IP Conversations” *ACM SIGCOM, 1991*

[14] Trung Nguyen e.a. : “Computer Modelling of 3G Cellural Traffic” *Telecommunications and Microelectronics Centre, Victoria University*

[15] Bruce A. Mah: “An emprical Model of HTTP Network Traffic”, *INFOCOM’97*

[16] “Cooperative Association for Internet Data Analysis”; http://www.caida.org

[17] Ferdinand Gramsamer e.a.: “Flow Control Scheduling” , revised and extended version of *ICCCN 2002*, Oct. 14-16, Miami, FL, pp438-443

[18] R. Rojas-Cessa, E. Oki, and H. Jonathan Chao: “CIXOB-k: Combined Input-Crosspoint-Output Buffered Switch”, *Proc. IEEE GLOBECOM*, 2001, vol. 4, pp. 2654-2660.

[19] G. Sapountzis, M. Katevenis: “Benes Switching Fabrics with O(N)-Complexity Internal Backpressure”, *Proc. IEEE Workshop High Perf. Switching & Routing (HPSR 2003)*, Torino, Italy, June 2003, pp. 11-16; http://archvlsi.ics.forth.gr/bpbenes/

[20] F. Abel, C. Minkenberg, R. Luijten, M. Gusat, I. Iliadis: “A Four-Terabit Packet Switch Supporting Long Round-Trip Times”, *IEEE Micro Magazine*, vol. 23, no. 1, Jan./Feb. 2003, pp. 10-24.

[21] N. Ni, L. N. Bhuyan: “Fair scheduling for Input Buffered Switches”, citeseer.nj.nec.com/482342.html

[22] P. Krishna, N. Patel, A. Charny, R. Simcoe: “On the Speedup Required for Work-Conserving Crossbar Switches”, *IEEE J. Sel. Areas in Communications*, vol. 17, no. 6, June 1999, pp. 1057-1066.

[23] M. Katevenis: “Fast Switching and Fair Control of Congested Flow in Broad-Band Networks”, *IEEE J. Sel. Areas in Communications*, vol. 5, no. 8, October 1987, pp. 1315-1326.

[24] K. Yoshigoe, K. Christensen: “A Parallel-Polled Virtual Output Queued Switch with a Buffered Crossbar”, *Proc. IEEE Workshop High Perf. Switching & Routing (HPSR 2001)*, Dallas, TX, USA, May 2001, pp. 271-275; http://www.csee.usf.edu/∼christen/hpsr01.pdf

[25] N. Chrysos, M. Katevenis: “Multiple Priorities in a Two-Lane Buffered Crossbar” , ICS-FORTH, September 2003. http://archvlsi.ics.forth.gr/bufxbar

[26] Quantum Flow Control (QFC) Alliance: *“Quantum Flow Control: A cell-relay protocol supporting an Available Bit Rate Service”*, version 2.0, July 1995; originally at http://www.qfc.org but no longer there –copy at http://archvlsi.ics.forth.gr/∼kateveni/534/qfc/

[27] G. Kornaros e.a.: “ATLAS I: Implementing a Single-Chip ATM Switch with Backpressure”, *IEEE Micro*, vol. 19, no. 1, Jan/Feb. 1999, pp. 30-41; http://archvlsi.ics.forth.gr/atlasI/hoti98/

Distributions Used

(a) Pareto Distribution.

$$PDF : f(x) = \frac{a}{b}(\frac{b}{x})^{a+1} \quad (4)$$

$$CDF : F(x) = 1 - (\frac{b}{x})^{a} \quad (5)$$

$$Mean : E(x) = \frac{ab}{a-1} \quad (6)$$

$$Variance : Var(x) = \frac{ab^2}{(a-1)^2(a-2)} \quad (7)$$

$$(8)$$

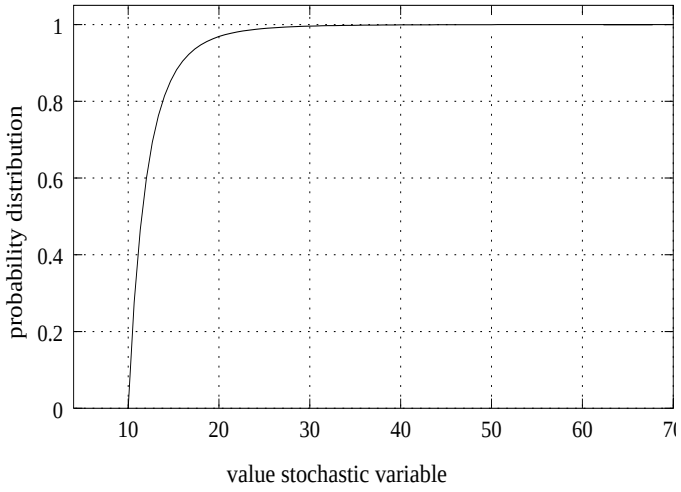

Figure 23: Pareto Distribution. b=10, a=5.

(b) Exponential Distribution.

$$PDF : f(x) = ae^{-ax} \tag{9}$$

$$CDF : f(x) = 1 - e^{-ax} \tag{10}$$

$$Mean : E(x) = \frac{1}{a} \tag{11}$$

$$Variance : Var(x) = \frac{1}{a^2} \tag{12}$$

$$\tag{13}$$

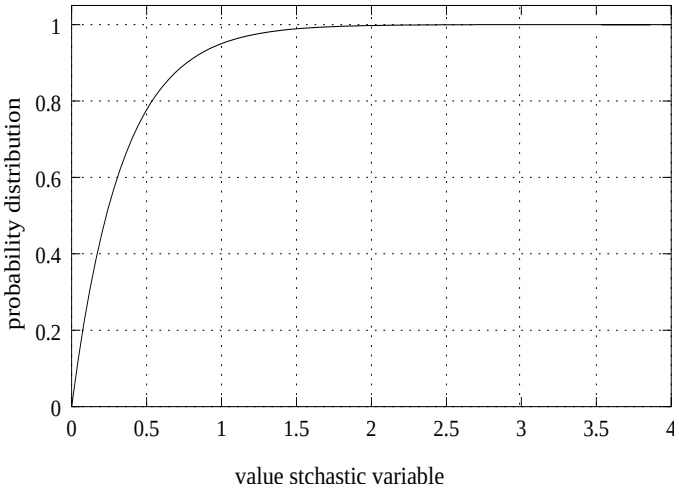

Figure 24: Exponential Distribution. a=3.

Generating Discrete Random Variables

We adopt the Inverse Transform Method to generate a discrete random variable, having a specific distribution function F. According to this method, we generate a random number u, uniformly distributed over $(0, 1)$ and then compute the value $F^{-1}(u)$, which is the desirable random variable. We mention here that for the random number generation we used the nrand48 function from the math.h standard C library.

Applying the method, described above, we get that a Pareto random variable is obtained by the formula $b(\frac{1}{1-u(0,1)})^{\frac{1}{a}}$ while an Exponential Random Variable is obtained by $-\frac{lnu(0,1)}{a}$.

Computing the Traffic Parameters

Suppose that IG's and BG's load is x, y Mbps respectively. Then

$$x + y = 100 \tag{14}$$

In a time interval equal to 1sec, IG has generated on average $\frac{x \cdot 10^6}{42 \cdot 8}$ packets, since its average packet size is 42 bytes, and let k be the number of packets generated on average by BG. Then approximately $k/2$ packets have size 1500, $k/4$ 552 and $k/4$ 576 bytes. It holds that:

$$(\frac{k}{2} \cdot 1500 + \frac{k}{4} \cdot 552 + \frac{k}{4} \cdot 576) \cdot 8 = y \cdot 10^6 \Rightarrow \tag{15}$$

$$k = \frac{y \cdot 10^6}{8256} \tag{16}$$

It also holds that the number of packets having size 40-44 bytes is near 64/18 times greater than the respective one for packets with size 1500 bytes:

$$\frac{\frac{x \cdot 10^6}{42 \cdot 8}}{\frac{k}{2}} = \frac{64}{18} \tag{17}$$

From eq. 16 and eq. 17 we get that:

$$\frac{x}{y} = 0.072 \tag{18}$$

Eq. 14 and eq. 19 yield that $x = 6.7Mbps$ and $y = 93.3Mbps$.

IG transmits on average 125 packets every approximately

$$125 \cdot \overline{p\_interarr\_t} + \overline{s\_interarr\_t} \tag{19}$$

times, resulting to a transmission rate

$$\frac{125 \cdot 42 \cdot 8}{125 \cdot \overline{p\_interarr\_t} + \overline{s\_interarr\_t}} \tag{20}$$

which must equal 6.7 Mbps; $\overline{p\_interarr\_t}$ denotes the mean IG's packet interarrival time and $\overline{s\_interarr\_t}$ denotes the mean IG's session interarrival time. Setting $\overline{p\_interarr\_t}$ to 49 usec, it follows that $\overline{s\_interarr\_time}$ equals 128usec.

BG's sessions are on average of size 8KB. Suppose that under probability x%, y%, z% the transfer is fragmented to packets with size 1500, 552, 576 bytes respectively. As a result there are generated $5 \cdot x$, $14 \cdot y$, $13 \cdot z$ packets of size 1500, 552, 576 bytes respectively. It must hold that :

$$\begin{cases} x + y + z = 100 \\ 5 \cdot x = 2 \cdot 14 \cdot y \\ 5 \cdot x = 2 \cdot 13 \cdot z \end{cases}$$

so, $x = 72.66$, $y = 13.39$ and $z = 13.95$.

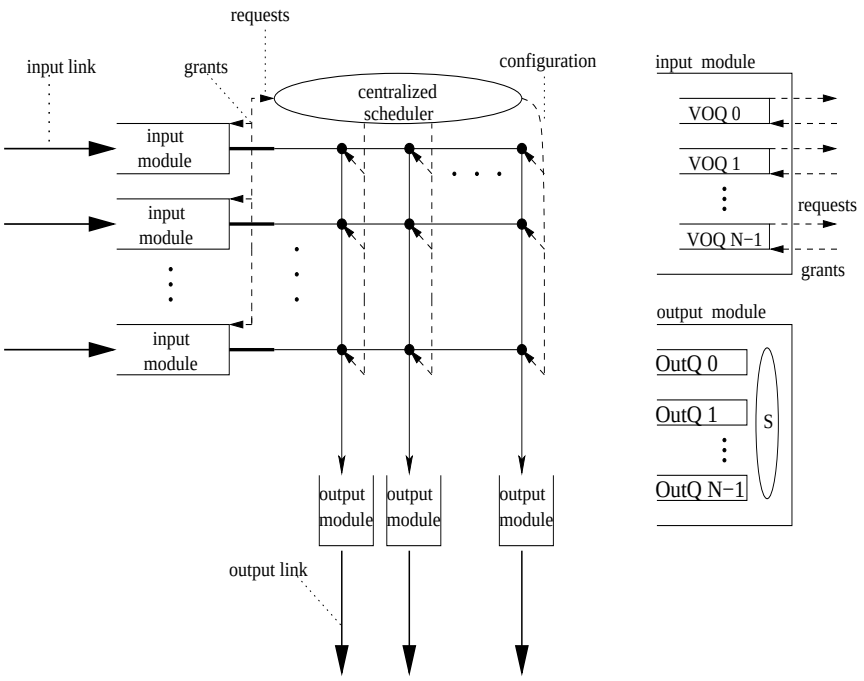

Figure 25: CIOQ architecture using the iSLIP centralized scheduler.

The CIOQ Architecture Using iSLIP

Fig. 25 indicates the simulated CIOQ architecture: Virtual Output Queuing is adopted at inputs, memories are placed at outputs and a centralized scheduler finds the configuration matrix by performing the iSLIP scheduling algorithm. Variable sized packets are transfered from inputs to outputs in fixed-size segments.

Particularly, we consider that when a packet arrives at an input port, it is fragmented to 64-byte segments which are enqueued to the corresponding VOQ [11]. Whenever a VOQ (i,j) is not empty it sends a request to the centralized scheduler, which selects among the VOQs the ones to be granted and sends the grants to these VOQs, thus specifying the configuration matrix. Each granted VOQ(i,j) is dequeued and its head packet is forwarded to the queue i of output j. We mention

[11] This operation introduces no delay.

here that in the simulated system, the output memories are partitioned in N queues (OutQ i, $i \in (0, N-1)$) of infinite size and each OutQ i of the output j is assigned to the VOQ(i,j), $j \in (0, N-1)$. These queues serve as reassemble buffers.

At each output j resides a scheduler; it keeps checking for reassembled packets at the queues of the output j and selects in RR way between that queues the next reassembled packet to be sent over the output link.

Concerning the centralized scheduler, it performs the iSLIP scheduling algorithm with one iteration. We have taken its description from [3] and we repeat it below:

Step 1. Request. Each input sends a request to every output for which it has a queued cell.

Step 2. Grant. If an output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the granted input if and only if the grant is accepted in Step 3.

Step 3. Accept. If an input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output.

The simulator for the iSLIP switch is based in the same event-based architecture described in section 3. However, we had to alter the events used for the buffered crossbar simulator. In the case of the iSLIP switch, the basic event types are:

- PACKET_AT_INPUT : notifies that a packet has arrived at an input port.
- INPUT_SCHEDULING_COMPLETED : signals the end of the scheduling operation of the centralized scheduler.
- INPUT_START_SCHEDULING : signals the beginning of the scheduling operation of the centralized scheduler.

- SEGMENT_TRANSFER_TO_OUTPUT_COMPLETED : notifies that a segment has been transfered to a reassemble buffer.

- REASSEMBLY_COMPLETED : notifies that all the segments of a packet have been transfered to a reassemble buffer.

- TOTAL_TRANSFER_COMPLETED : notifies that a (reassembled) packet has totally left the switch.

Note that for the request/grant operation we do not consider any events. Instead we assume that the request/grant dealy is included in the scheduling delay.

Concerning the states machines of the schedulers, we present them below:

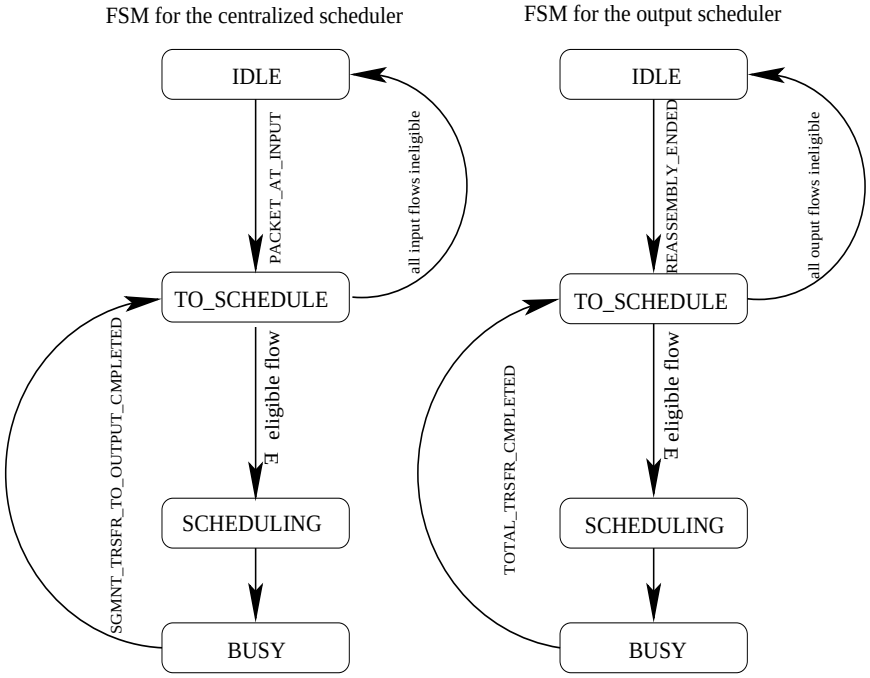

Figure 26: The FSMs for the schedulers in the simulated CIOQ architecture.

For the input scheduler a flow is eligible if it is non empty. For the output scheduler a flow is eligible if it contains at least one reassembled packet.