PACKET MODE SCHEDULING IN BUFFERED CROSSBAR SWITCHES

MASTER OF SCIENCE THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF THE UNIVERSITY OF CRETE

Georgios A. Passas

April 2006

Packet Mode Scheduling in
Buffered Crossbar Switches

by

Georgios A. Passas

Abstract

Buffered crossbars have emerged as an advantageous switch architecture mainly because of their scheduling efficiency and capacity for variable size packets. Buffered crossbars directly operating on variable size packets require at least one maximum packet worth buffer per crosspoint. When we cannot afford such large buffer sizes, we are forced to segment packets.

This thesis studies efficient methods for packet segmentation, taking into consideration hardware implementation issues. We propose a hybrid SRAM/DRAM ingress packet buffer organization which is capable of accomodating large packet backlogs during periods of congestion, while at the same time converting external packets to small internal segments. The required crosspoint buffer size becomes significantly smaller than and independent of the maximum external packet size. Because packet segmentation at the switch ingress path implies the need for packet reassembly at the switch egress path, we apply packet mode scheduling to buffered crossbars in order to reduce the reassembly delay or even eliminate the reassembly buffers: the switch is scheduled so that the resulting input-output port pairings are maintained until all the segments of the packet are forwarded to the switch output. Using behavioral simulation, we show that our system performs virtually as well as buffered crossbars which use no segmentation and require large crosspoint buffers.

Χρονοπρογραμματισμός ενός Buffered Crossbar Μεταγωγέα με Γνώμονα τα Όρια των Πακέτων

Γεώργιος Α. Πασσάς

Περίληψη

Οι μεταγωγείς buffered crossbar (μεταγωγείς διασταυρούμενων ράβδων με έναν ενταμιευτή σε κάθε διασταύρωση) έχουν αναδειχθεί ως μία προνομιούχος αρχιτεκτονική, κυρίως λόγω του εύκολου χρονοπρογραμματισμού τους και της δυνατότητάς τους να λειτουργούν απ' ευθείας με πακέτα μεταβλητού μεγέθους (ασύγχρονη λειτουργία). Για λειτουργία με πακέτα μεταβλητού μεγέθους, σε κάθε διασταύρωση ράβδων απαιτείται ένας ενταμιευτής ικανός να αποθηκεύσει ένα μέγιστο εξωτερικό πακέτο. Τεχνολογικοί περιορισμοί δύνανται να καταστήσουν αυτήν την απαίτηση προβληματική, ιδίως όταν το μέγιστο εξωτερικό πακέτο είναι πολύ μεγάλο.

Η παρούσα εργασία μελετά αποδοτική κατάτμηση πακέτων σε μικρότερα εσωτερικά τεμάχια ώστε να μειωθεί η απαίτηση για μεγάλους ενταμιευτές στις διασταυρώσεις. Δίνουμε το σκαρίφημα ενός υβριδικού ενταμιευτή πακέτων για τις εισόδους του μεταγωγέα. Ο ενταμιευτής είναι ικανός να αποθηκεύει μεγάλες συσσωρεύσεις πακέτων κατά τη διάρκεια περιόδων συμφόρησης, ενώ παράλληλα μετατρέπει εξωτερικά πακέτα σε μικρότερα εσωτερικά τεμάχια. Το μειονέκτημα του προκύπτοντος συστήματος είναι η ανάγκη για επανασύνθεση των πακέτων από τα εσωτερικά τεμάχια στις εξόδους του μεταγωγέα. Για να μειώσουμε την καθυστέρηση που συνεπάγεται η επανασύνθεση ή/και να απαλείψουμε τους ενταμιευτές που αυτή απαιτεί, χρονοπρογραμματίζουμε τον μεταγωγέα ώστε κάθε ζευγάρωμα εισόδου-εξόδου να διατηρείται έως ότου προωθηθεί ολόκληρο το πακέτο. Με προσομοίωση στο επίπεδο του αλγορίθμου δείχνουμε οτι το τελικό σύστημα είναι σχεδόν το ίδιο αποδοτικό με buffered crossbars χωρίς κατάτμηση πακέτων και μεγάλους ενταμιευτές στις διασταυρώσεις.

Acknowledgements

During the last two years many people helped me with my work. Among them, i would mostly like to thank my family for being so close to me even though they are many miles away and my friends, especially Theokliti, for tolerating my testiness and some times my bad company.

I also thank prof. Manolis Katevenis for the opportunity he gave me to work on high performance switching. Particularly concerning this study, i thank him for suggesting the line card organization (chapter 3) and the application of packet mode scheduling in buffered crossbars (chapter 4).

The Scalable Systems and Networks group offered computational resources in their PC cluster, allowing for detailed and long simulations. I thank them all and mostly Michail Flouris.

This study was founded by the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH). The first part (chapter 3) was presented in IEEE International Conference on Communications (ICC 2005), Seoul, Korea, May 2005 [27] while the second one is going to appear in IEEE Workshop on High Performance Switching and Routing (HPSR 2006), Poznan, Poland, June 2006.

Contents

Acknowledgements **v**

1 Introduction **1**

2 The Crossbar Scheduling Problem **5**

2.1 Scheduling in a bufferless crossbar 5

2.2 Scheduling in a buffered crossbar 7

2.3 The reference architecture 8

3 Packet Segmentation in Buffered Crossbars **13**

3.1 Ingress line card queueing architecture 14

3.2 Segment size adaptivity 16

3.3 Crosspoint buffer size 17

3.4 Egress queueing and reassembly 17

3.5 Related work 17

4 Packet Mode Scheduling in Buffered Crossbars **21**

4.1 Probabilistic packet mode scheduling 24

4.1.1 Detecting pairings and entering packet mode 24

4.1.2 Piggybacking control signals 26
4.1.3 Correctness 26
4.1.4 Starvation 28
4.1.5 Performance 29
4.1.6 Egress buffering 29
4.2 Deterministic packet mode scheduling 31
4.2.1 Scheduling operation 31
4.2.2 Contrast to the traditional three-phase matching algorithms 34
4.3 Packet mode scheduling connected with ingress memory management 36
4.4 Packet mode scheduling in buffered vs. bufferless crossbars 37

5 Method **41**

5.1 Traffic models 43
5.2 Performance metrics 44
5.3 Interval estimates of the measurements 45

6 Results **47**

6.1 Segment size adaptivity 47
6.2 Segment size dependence of performance 48
6.3 Probabilistic packet mode scheduling 49
6.3.1 Comparison to counterpart systems 49
6.3.2 Effect of the synchronization distance on performance 50
6.4 Deterministic packet mode scheduling 52
6.4.1 Comparison to counterpart systems 52

6.4.2 Effect of the lock scheduling policy on performance 54

6.4.3 Effect of the masking operation on performance 54

6.4.4 Effect of the crosspoint buffer size on performance 56

6.5 Unbalanced traffic 56

7 Conclusion and Future Work **59**

Bibliography **61**

A Simulation Running Time **65**

List of Tables

2.1 RTT decomposition at 10Gbps line rate 10

5.1 Default Simulation Parameters 42

List of Figures

2.1 Finding bipartite matches with 3-phase algorithms. Four inputs and four outputs are assumed. 6

2.2 The bufferless (left) and buffered (right) versions of the crossbar. Two inputs and two outputs are shown. 8

2.3 A buffered crossbar directly operating on variable size packets. A 3×3 switch is assumed. We only show one linecard and its associated buffers in the core; imagine two more linecard bubbles in the vertical direction and horizontally add two more rows in the crossbar to get the full system. 9

2.4 (a) Backpressure RTT in a buffered crossbar switch (b) Internal link under-utilization when dimensioning the crosspoint buffer ignoring the backpressure RTT. 11

3.1 DRAM block layout. In this example, we consider four DRAM banks and 512-byte blocks. Each block is laid out as four 128-byte sub-blocks, with each sub-block residing in a separate DRAM bank. 14

3.2 VOQ datapath in the ingress line card. Memory management uses fixed-size blocks, while traffic is forwarded to buffered crossbar in variable-size segments. 15

3.3 Overall datapath architecture. SRAM is used in the ingress path to streamline external traffic in order to optimize DRAM throughput utilization. SRAM is also used in the egress path for packet reassembly. 16

3.4 The proposed segmentation method (d) compared to already known schemes (a)-(c); (d) can be considered as the combination of (b) and (c). 18

4.1 Packet mode versus segment (cell) mode switch scheduling 22

4.2 Input-output port pairings in a bufferless (left) and a buffered crossbar (right). Pairings in a buffered crossbar are "inexact". 23

4.3 Maximum allowable synchronization distance. 25

4.4 A traffic scenario which displays that the maximum packet interleaving can occur with probabilistic packet mode scheduling. The interleaved packets are labelled P_A, P_B. 30

4.5 The egress buffer - switch throughput tradeoff. Four inputs and three outputs are shown. Inputs 0 and 3 have reserved the reassembly slots of output 1, preventing other inputs to transmit to this output. 31

4.6 Proposed (asynchronous) crossbar scheduling. The lock acquisition phase is involved only when the output scheduler attempts to serve a flow whose head packet is partly stored in the crosspoint buffer. 33

4.7 Updating the round-robin pointers of input and output schedulers. When a packet-mode pairing is configured (from input 0 to output 0 in the figure), the pointers are not updated and remain locked until the whole packet is forwarded. 33

4.8 Packet mode scheduling connected with memory management. Comparison with the line card organization proposed in [18]. 37

4.9 A staircase-like traffic pattern that leads a packet-mode scheduler for a bufferless crossbar to "lock" in a fixed configuration. Assume a 4×4 switch; the figure shows the 4 flows that are constantly served; other flows with non-empty queues exist, but are never served. 39

5.1 Load distribution in a 4×4 switch according to the Zipf law, $k = 2$. Each input favors a single output and recursively the same is true for the rest of the outputs. Each input favors an output so that no output is overloaded. 44

6.1 Buffered crossbar with blind SAR (SM model). Saturation throughput with uni- and multi- packet segments. Header size is 4 bytes. Packet size is *constant(40)* and traffic is uniformly destined. 48

6.2 Buffered crossbar with blind SAR (SM model). Switch throughput under unbalanced (Zipf) traffic with multi- and uni- packet segments. Crosspoint buffer size is 512 or 1024 bytes. Packet size is trimodal. 48

6.3 Buffered crossbar with blind SAR (SM model) and variable size multipacket segments. Crosspoint buffer size equals maximum segment size. Maximum segment size is 128, 256, 512 or 1024B. Round-trip time equals maximum segment time. Header size is 4 bytes. Packet size is trimodal and traffic is uniformly destined. The reported results were obtained from a single run. . 49

6.4 Buffered crossbar with blind SAR (SM model) and multipacket segments. The plot shows the percentage of delay in reassembly buffers over total queueing delay. Traffic is uniform and packet size is bimodal, uniform or constant(8KB). 50

6.5 Deterministic packet mode (DPM) scheduling under uniformly destined traffic. Comparison to probabilistic scheduling - PPM (upper row), (full) variable-packet-size scheduling in buffered crossbar - VPS (middle row), packet mode scheduling in Input Queueing - IQ (bottom row). 51

6.6 Deterministic packet mode (DPM) scheduling under uniformly destined traffic. Comparison to probabilistic scheduling - PPM (upper row), (full) variable-packet-size scheduling in buffered crossbar - VPS (middle row), packet mode scheduling in Input Queueing - IQ (bottom row). 53

6.7 Performance of deterministic packet mode scheduling (DPM) with three different policies for granting locks: round-robin, strict priority and random. Traffic is uniformly destined. The reported results were obtained from a single run. 54

6.8 Performance of deterministic packet mode scheduling (DPM) with (PUSH) and without (PULL) the masking operation. Traffic is uniformly destined. The reported results were obtained from a single run. 55

6.9 Buffered crossbar with blind SAR (SM model) and multipacket segments. The plot shows the percentage of delay in reassembly buffers over total queueing delay. Traffic is uniform and packet size is bimodal, uniform or constant(8KB). 56

6.10 Switch throughput under unbalanced (Zipf) traffic. 57

A.1 Simulated real time as a function of input load. The reported values correspond to the VPS system under bimodal packet size and uniform destinations. 66

A.2 Slotted-time vs. event-driven simulator performance. The reported values concern the VPS, SM and IQ system with packet mode scheduling. The packet size is bimodal, uniform or constant(8KB) and the traffic is uniform. 66

A.3 Performance of event-driven simulation for the models of buffered crossbar with packet segmentation. The packet size is bimodal, uniform or constant(8KB) and the traffic is uniform. 67

Chapter 1

Introduction

The majority of modern, high-speed networks - WANs, MANs, LANs or SANs - transfer information between end users in *variable size packets*. However, traditional packet switches - which form the core of these networks - operate on *fixed size cells*: variable-size packets are segmented into cells at the switch inputs, the cells are switched and the original packets are transmitted on the line after they have been reassembled from the internal cells at the switch outputs.

Most packet switches with a moderate port count use a crossbar to deliver packets from their input to their output ports, because crossbars are simple and non-blocking. The crossbar is configured by a (usually central) scheduler [1][2], which changes the crossbar configuration pairing input to output ports subject to the maximization of the switch throughput or/and the satisfaction of certain quality of service (QoS) criteria [3]-[5]. The crossbar configuration changes synchronously, with a period specified by the scheduling time, and hence the traditional cell operation.

Unfortunately though, external packets often occupy an non-integral multiple of internal switch cells and the last cell of a packet is padded with useless bytes. To cope with the cell-padding overhead, internal speedup (core overspeed) is used and it is around two in the worst case (e.g. with 65-Byte packets in a 64-Byte-cell switch). Speedup is also needed to account for the inefficiencies of the scheduler, hence crossbar switches use a core speedup of around 2 or greater. Internal speedup increases power consumption and limits port and line-rate scalability [6]. Moreover, output buffering is required, not only for the packet reassembly, but also because the crossbar operates faster than the output lines.

Recently, *buffered crossbars* have emerged as an advantageous architecture; they contain

small buffers at their crosspoints, and use backpressure to the inputs to prevent these crosspoint buffers from overflowing. The first observation about buffered crossbars concerned the simplicity and high efficiency of their scheduling [7] - [13]; no internal speedup is needed to compensate for scheduler inefficiencies, thus allowing the increase of port speed. A subsequent observation was that buffered crossbars can directly switch *variable-size* packets (asynchronous operation) [7][9]. Doing so, without any segmentation, eliminates the need for speedup to cope with the segmentation overheads; in turn, the lack of speedup eliminates egress queueing, and the lack of segmentation eliminates reassembly buffers, thus reducing cost [14].

However, buffered crossbars directly operating on variable size packets have limitations. First, for switch operation without segmentation and reassembly (SAR), the required buffer size per crosspoint, as shown in [14], is at least one maximum-size packet plus one round-trip-time (RTT) worth of data. Reference [14] assumed 1500-byte maximum packets (old ethernet limit) and RTT $\leq$ 500 B, thus using 2 KByte buffers. For a 32×32 switch, the resulting silicon area is expensive in current technologies; hence, for the high port-count switches of the next few years, it is desirable to limit buffer size per crosspoint to probably 1024 bytes or less. Second, when very large packets (e.g. "jumbo frames", ten or more KBytes per packet [15]) are to be switched without SAR, the required buffer size is very large, thus limiting port count below a dozen or so. Third, when multiple flows per switch input-output port pair are supported, the on-chip buffer size requirements rise because multiples queues per crosspoint are needed for flow isolation. Fourth, large buffer space is typically needed at the ingress line cards of a switch to store packets during periods of heavy congestion, so DRAM is typically used as a packet buffer [17][18]; DRAM memory management is almost impossible unless fixed size blocks are used.

To overcome these limitations, SAR has to be re-introduced in buffered crossbars. This paper examines SAR in buffered crossbars. We propose a novel method for segmenting packets: *(i)* the segment size is variable, thus eliminating padding overhead; *(ii)* the maximum segment size is much smaller than the maximum packet size, thus drastically reducing crosspoint buffer size; and *(iii)* multiple (small) packets can be placed in a same segment, thus avoiding small segments so as to reduce relative header overhead. We describe our segmentation scheme in chapter 3. We show that it is well adapted to DRAM memory management and we sketch a line card architecture.

In order to avoid reassembly buffers and/or delays, we introduce *packet mode scheduling* in buffered crossbars. Packet mode scheduling had only been studied for bufferless cross-

bar switches: when the central switch scheduler establishes an input-output port pairing, the pairing is maintained until all cells of the packet are forwarded [19]-[21]. The extension to buffered crossbars is not trivial because the scheduling process does not necessarily determine such pairings. We introduce two schemes for buffered crossbars: *probabilistic* and *deterministic* packet mode scheduling; we describe these schemes in chapter 4. The probabilistic case assumes independent crossbar output schedulers, but requires reassembly buffers. Using simulation, we show that it reduces packet latency compared to the system with pure SAR; for a representative traffic pattern the average packet latency is improved by more than 80% for loads up to 50%. Deterministic packet mode scheduling sacrifices some scheduler independence in order to eliminate the reassembly buffers; based on our simulations, it performs very close to buffered crossbars without segmentation (which use very large crosspoint buffers), and it also performs always better than packet mode scheduling in bufferless crossbars. The probabilistic scheme performs similar to the deterministic scheme for some traffic patterns, and better than it for other traffic patterns.

Our simulation method is described in chapter 5 and our experimentation in chapter 6. We start by briefly giving some background on the crossbar scheduling problem in the subsequent chapter.

Chapter 2

The Crossbar Scheduling Problem

2.1 Scheduling in a bufferless crossbar

Crossbars have replaced the buses in old-time, general-purpose computers, which were being used as switches [22], and they form the interconnection network for the linecards providing multiple parallel paths in the communication between them[1]. However, buffers (queues) are needed to store packets arriving at distinct crossbar inputs and contenting for the same crossbar output, mainly due to the statistical nature of the traffic.

The placement of the buffers - a core architectural choice - was driven by the requirement for low buffer memory access rate [23]. By placing a buffer at each crossbar output - *output queueing architecture* - we achieve ideal performance and we can easily provide quality of service guarantees, but in the case of a switch with N inputs the memory write access rate needs to be at least N times the link rate. Even worse, using a centralized memory - *shared memory architecture* - we require at least N times higher read rate too. Thus, queueing has converged to be distributed at inputs - *input queueing architecture*; then, both the read and write access rate match the link rate.

However, scheduling is needed in order to decide which packet to serve, i.e. which input to connect to which output, when output contention arises. In a paper fundamental for switch architecture, Karol an Hluchyj show that with input queueing, and for any scheduling policy, the input links "saturate at a utilization of around $2 - \sqrt{2} = 0.586$"

[1] Although modern VLSI technology permits the integration of thousands of switches into a single chip, the number of pins in a chip cannot be greater than some hundreds and thus for large port counts multistage fabrics are used as the interconnection network. In this paper, we only consider switches with a medium number of ports.

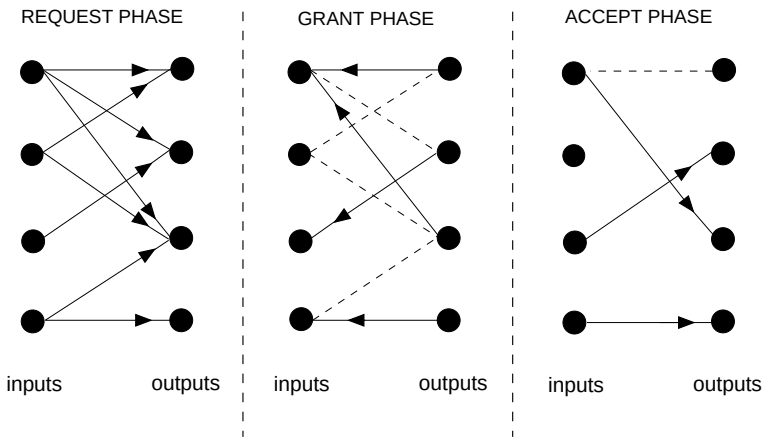


Figure 2.1: Finding bipartite matches with 3-phase algorithms. Four inputs and four outputs are assumed.

(when N is large) due to the *head-of-line (HOL) blocking* phenomenon: a congested head packet in an input queue prevents the next packets in the queue from being served, even though their destinations are uncongested [24][25]. Tamir and Frazier [26] proposed a simple queueing discipline - usually referred to as *Virtual Output Queueing (VOQ)* - which solves the HOL blocking problem: each input provides one fifo queue per output, for a total of N^2 input queues. The problem altered in scheduling packet transfers from N^2 input queues to N outputs, which is essentially equivalent to finding an $N^2 \times N$ bipartite graph.

Significant effort has been pushed in researching bipartite matching algorithms for scheduling in input queued packet switches (e.g. [3][4][28]). These algorithms should be *efficient*, providing high switch throughput, and at the same time *simple*, in order to be implemented in fast hardware. They find a match between crossbar input and output ports, usually operating in the following three phases (see fig 2.1):

1. *Request.* Each unmatched input sends a request to every output for which it has queued cells.

2. *Grant.* Each unmatched output receiving requests selects among them the one to grant.

3. *Accept.* Each input receiving grants selects among them the one to accept.

The policy in which inputs issue requests or accept grants and the outputs grant requests is crucial in finding *maximal* matches. The PIM (Parallel Iterative Matching) algorithm [3] uses randomization and multiple iterations. In each iteration only unmatched inputs and outputs participate in the matching process, so the size of the match is increased after each

iteration. The famous SLIP scheduling algorithm [4] *desynchronizes* the outputs so that each one grants a distinct input. Desynchronization is enforced by a simple modification of round-robin scheduling (a grant pointer is updated if and only if the grant is accepted) but assuming *all input queues are backlogged*. This is true under *uniformly destined traffic*, the algorithm degenerates into time division multiplexing and provides 100% throughput with a single iteration. However, under unbalanced traffic some queues drain, outputs synchronize and the switch throughput is lowered. Hence, either multiple iterations or internal speedup is required. The first is difficult to implement at high link rates, while the second one limits switch scalability - a switch with a speedup of two is equivalent to a switch with 0.50 maximum link utilization.

Refering, for the shake of presentation, to the three-phase matching algorithms (see fig. 2.1), an input cannot start transmitting to an output unless its request is granted; symmetrically, an output cannot start reading from an input unless its grant is accepted. So, the crossbar configuration can only change in fixed-length time intervals - *time slots*, synchronous operation - whose length is specified by the scheduling time. Typically, the time slots worth 64 or 128 bytes, hence crossbars operate on fixed length transfer units, named *cells*, with a typicall cell size being 64 or 128 bytes [18].

2.2 Scheduling in a buffered crossbar

A buffered crossbar is a special version of the crossbar with a small buffer at each crosspoint (fig. 2.2). Notice that the read/write crosspoint memory access rate matches the line rate. The buffered crossbar architecture is not new (see for example [29]), but it had received very little attention up to the previous few years mainly because of the high cost of the crossbar chip, induced by the on-chip memory requirements (for N input and output ports we need N^2 crosspoint buffers). The advent of VLSI technology permits the fabrication of such crossbar chips today [13][14].

In a buffered crossbar, VOQs are held at the ingress line cards, as in the bufferless case, but their heads are extended inside the crossbar chip in small crosspoint buffers. Scheduling is carried out by schedulers which are independent and distributed at inputs and outputs: each input scheduler selects a non-empty VOQ_{ij} and sends a transfer unit to the crosspoint buffer (i, j); each output scheduler selects a non-empty crosspoint buffer and sends to the relative crossbar output. So simple. To prevent the crosspoint buffers from overflowing, a credit based flow control scheme [30][31] is used between the inputs and the crosspoints; a

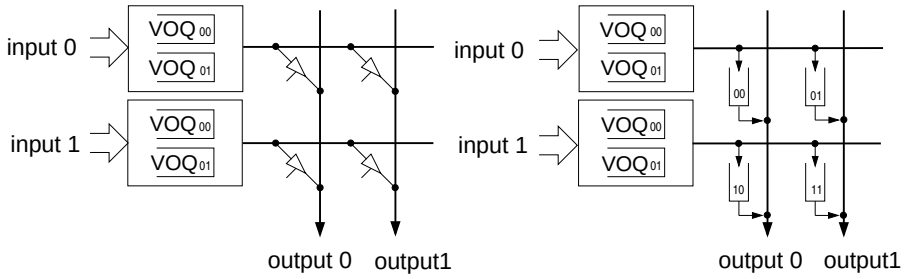


Figure 2.2: The bufferless (left) and buffered (right) versions of the crossbar. Two inputs and two outputs are shown.

VOQ is eligible for the input scheduler provided there are enough credits for the resulting transfer. Notice that the input schedulers are independent in the short-run but they are coordinated by the flow control protocol in the long-run.

In a crossbar with buffered crosspoints scheduling is simple because short-term input and output conflicts are allowed. Buffered crossbar scheduling finds an approximate match rather than a bipartite match and high throughput is easily provided - it has beeen shown that with round robin scheduling at the crossbar outputs and a *longest queue first discipline* at the inputs 100% throughput is achieved for any admissible traffic pattern[2] [10].

Owing to the scheduler independence, buffered crossbars can operate *asynchronously*: there is no need for the transfer units to have fixed size; they may have variable size, provided that the crosspoint buffer is at least one maximum internal transfer unit large. Below, we review how a variable-packet-size buffered crossbar works; this architecture [14] is the reference architecture in the current study.

2.3 The reference architecture

A buffered crossbar switch operating on variable size packets does not present significant differences compared to the fixed-size cell system. In essence, only the credit flow control needs to be ported in an appropriate manner[3].

In the cell system, a credit counter C_{ij} is associated with each VOQ_{ij} and it is initialized

[2]Probably what constitutes a "bug" in bufferless is a desirable feature in buffered crossbars: different input schedulers feeding the same output in a synchronized manner result in the crosspoint buffers being filled-up and thus high throughput is achieved at each crossbar output.

[3]Actually, this is true in an architectural perspective. Implementation has also to deal with memory management with variable size packets and other details. In the next chapter we will show that the memory management at the ingress cards is tricky in a variable packet size buffered crossbar. In [14], the implementation of the crossbar chip was shown to be very simple. Perhaps a shortcoming is that the packet size is not known until the packet is dequeued.

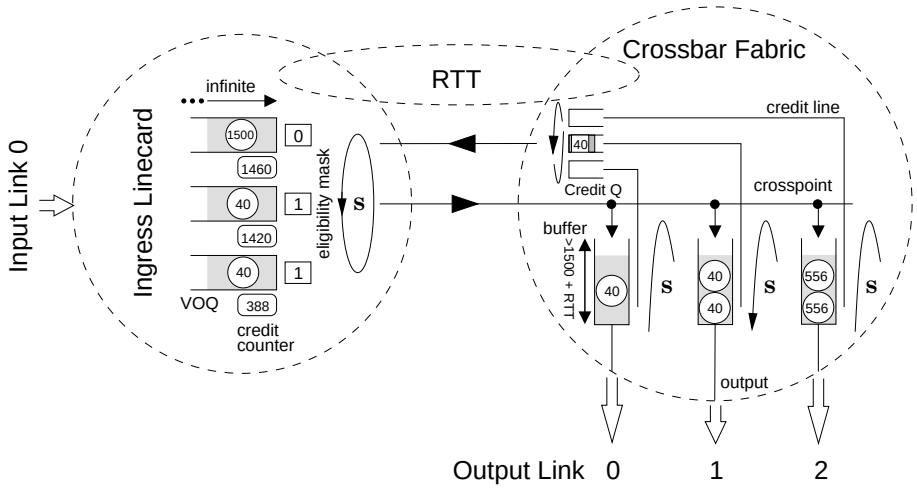


Figure 2.3: A buffered crossbar directly operating on variable size packets. A 3×3 switch is assumed. We only show one linecard and its associated buffers in the core; imagine two more linecard bubbles in the vertical direction and horizontally add two more rows in the crossbar to get the full system.

to a value which matches the number of cells that can be buffered at a crosspoint. Each time a cell is dequeued from VOQ_{ij} the credit counter is reduced by one. VOQ_{ij} is eligible if and only if it is non-empty (i.e. at least one cell is queued) and $C_{ij} > 0$. Each time a cell departs from crosspoint (i, j), a credit is sent to the ingress card i and upon its reception C_{ij} is increased by one. This is the classical (incremental) form of credit based flow control. In [30], an alternative method is described, which basically differs in providing resilience to corrupted credit messages. Throughout this study we assume incremental credit flow control. Intuitively, it is easy to see that buffer overflow cannot occur with credit based flow control; for a proof refer to [32]. In order to avoid buffer underflow (i.e. to sustain the link rate) the (crosspoint) buffer needs to be at least $RTT \times R$ cells large, where RTT is the round trip time between the fabric and the line cards and R the line rate [32].

In the variable packet size system, credits are connected to bytes rather than cells (refer to fig. 2.3 while reading). C_{ij} is initialized to a value indicating the number of bytes that can be buffered at a crosspoint. Each time a packet is dequeued from VOQ_{ij}, the credit counter is reduced by the size of the packet. VOQ_{ij} is eligible for the input scheduler if it is non empty (i.e. at least one packet is queued) and the size of the head packet (in bytes) is not larger than the available credits. Each time a packet starts departing from the crosspoint (i, j) a credit is sent to the ingress card i. The credit indicates the size (in bytes) of the packet just departed and upon its reception C_{ij} is increased by the relative value. Notice that the crosspoint buffer should be at least one maximum packet large. In fact, as we will describe next, it needs to be one RTT greater.

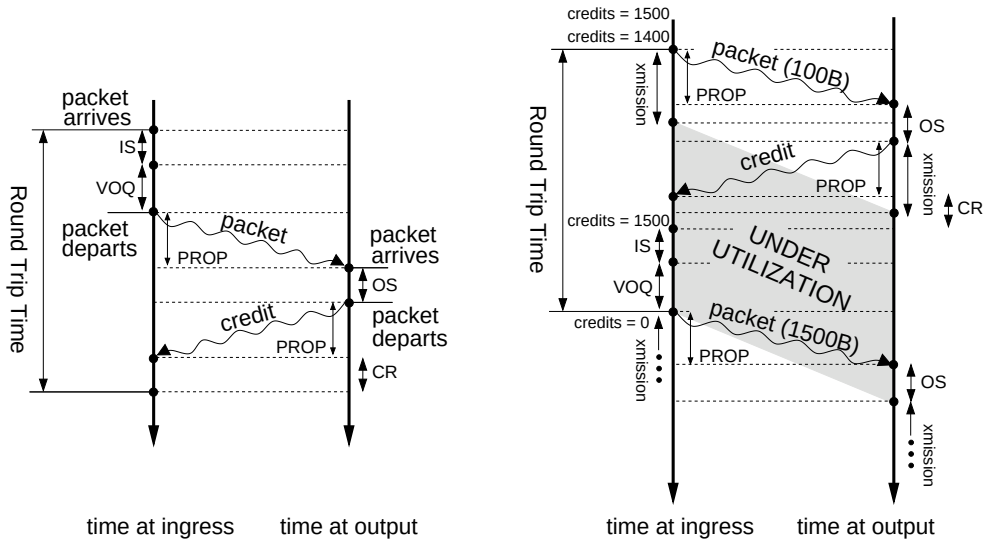
Time	Value (ns)
Input Scheduling (IS)	30
VOQ memory access (VOQ)	60
Packet propagation (PROP)	115
Output Scheduling (OS)	30
Credit transmission (CR)	30
Credit propagation (PROP)	115

Table 2.1: RTT decomposition at 10Gbps line rate

The RTT between the line cards and the fabric has become a significant factor in modern switch design mainly because in multi-rack router implementations the switch fabric is physically located far from the linecards [6]. In a buffered crossbar switch the RTT was defined in [14][33] as the sum of the delays shown in table 2.1; it corresponds to the time interval between the moment a packet arrives at a switch input port to the moment the input receives the credit that corresponds to the transmission of that packet at the output, assuming no output contention; see the space-time diagram in fig. 2.4(a); the horizontal direction represents space while the vertical one time, the arrows denote messages and the dots events [34]. A RTT value around 500ns at 10Gbps line rate is reasonable in realistic systems.

In order to sustain full line rate, the minimum crosspoint buffer size should be at least one maximum packet *plus* one RTT window large [14][33]. Dimensioning the crosspoint buffers ingoring the backpressure RTT results to output underutilization (fig. 2.4(b)). To see why, consider that the buffer worths only one maximum packet (e.g. 1500 bytes; in turn the maximum available credits are for 1500 bytes) and a small packet (e.g. 100 bytes) in the buffer blocks a maximum packet in the corresponding VOQ. When the small packet starts being transmitted to the output, a flow control credit is sent to the relative input in order for the next packet to follow. However, if the RTT window is larger than the small packet, there is a gap in time between the moment the small packet has been completely transmitted from the ingress card to the moment the large packet starts being transmitted. The bubble is propagated to the output link as well, if traffic from no other linecard is destined to the same output. Thus, output underutilization occurs. With 1500-byte buffers, a RTT of 400 byte times and alternating 40-byte, 1500-byte packets, output utilization drops to 80% when a single flow is active in the switch [14][33].

Fig. 2.3 shows that queues are used to buffer credits resulting from packets that at about the same time depart from a same crossbar row; contenting credits are served round-robin. Control lines, separate from the data lines and running from the crossbar to the linecards



(a) Round Trip Time (b) Link Underutilization

Figure 2.4: (a) Backpressure RTT in a buffered crossbar switch (b) Internal link underutilization when dimensioning the crosspoint buffer ignoring the backpressure RTT.

are used to carry the credits of the flow control protocol. Note that an inverse crossbar is formed by the control wires.

Chapter 3

Packet Segmentation in Buffered Crossbars

In this chapter, we describe how we can reconcile ingress memory management with variable size packets. We show how to segment external packets in order to optimize DRAM throughput utilization. The resulting segments, at the same time, form the switching units, so as to reduce the size of the crosspoint buffers in the fabric.

The Virtual Output Queues (VOQs) store the bulk of the backlog of the flows that cross an input queued switch and thus they may grow large during periods of heavy congestion. To accommodate their growth, DRAM is typically used in the ingress line cards, because DRAM is optimized for capacity. However, peak DRAM random access rate is rather low; for a modern DDR SDRAM a typical value for the cycle time is 60ns [32]. By contrast, SRAM provides much higher access rate - the cycle time is 2ns or less [32] - but the cost, as well as the power consumption, per unit of capacity is prohibitive.

In order to increase access rate, DRAM chips are partitioned in *banks* allowing accesses to all banks in parallel (*memory interleaving*). To avoid bank conflicts, i.e. accesses to a busy bank, the accesses have to be successfully scheduled, i.e. they are not truly random any more. One solution is to rearrange accesses in time in a way that no conflicts occur; this implies out-of-order execution and requires complex control hardware [16][17]. The other solution is to ensure that each memory access touches all DRAM banks. If this is true, then no conflicts occur. For example, accessing modern SDRAM memory in blocks of 512 bytes, we can achieve 45 Gb/s of aggregate (read/write) throughput; e.g. [35]: 128-bits wide $\times$ 200 MHz DDR gives 51.2 Gb/s minus 10% turn-around overhead, minus 2% refresh

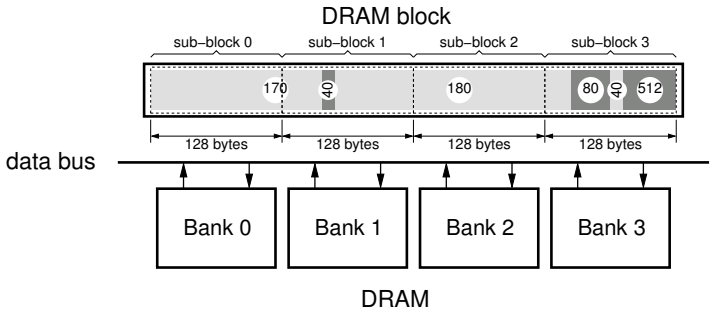


Figure 3.1: DRAM block layout. In this example, we consider four DRAM banks and 512-byte blocks. Each block is laid out as four 128-byte sub-blocks, with each sub-block residing in a separate DRAM bank.

overhead; each block is laid out as four 128-byte sub-blocks, with each sub-block residing in a separate DRAM bank; bank interleaving with 8-word (4-clock) bursts per bank provides peak throughput. We consider this second solution more attractive because it is easier to implement.

When the DRAM buffer is operated on variable size packets, two problems arise. First, short DRAM accesses are induced and thus we cannot guarantee that each DRAM access touches all DRAM banks. Second, fragmentation overheads are generated when the packets do not occupy an integral number of memory blocks; these overheads waste memory throughput in the same way that padded, fixed size cells waste throughput in the crossbar fabric.

3.1 Ingress line card queueing architecture

In order to expressly eliminate short DRAM write accesses, we introduce a small "smoothing" SRAM buffer in front of the DRAM one, which streamlines external memory traffic in order to ensure peak DRAM throughput utilization. The idea is to store arriving packets in SRAM until fixed-size and relatively large (e.g. 512-byte) DRAM memory blocks fill-up and afterwards transfer these blocks to DRAM. Then, peak throughput can be reached using memory interleaving. Below, we describe the operation of the packet buffer.

Arriving packets are classified and then written in the proper VOQ in SRAM. Packets in the same VOQ are written contiguously one after another and once the occupancy of a VOQ_{ij} in SRAM exceeds the DRAM block size, say 512 bytes, the first 512 bytes of VOQ_{ij} form a DRAM block and are ready to be transfered to DRAM. The resulting blocks may contain one or more packets or fragments thereof, as displayed in fig 3.1. Note that only packets belonging to the same flow are merged inside the same DRAM block.

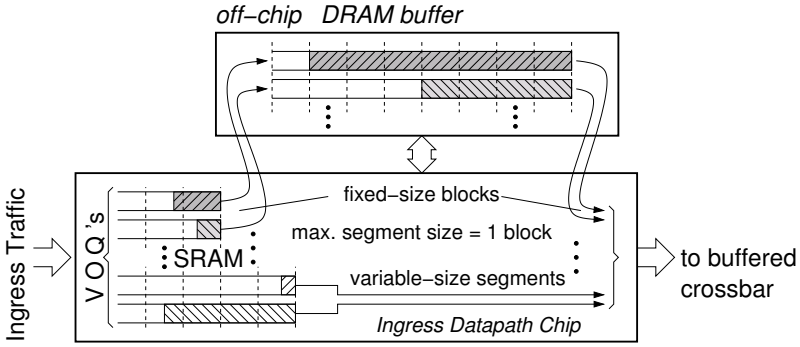


Figure 3.2: VOQ datapath in the ingress line card. Memory management uses fixed-size blocks, while traffic is forwarded to buffered crossbar in variable-size segments.

The VOQs are implemented as linked lists consisting of the fixed-size DRAM blocks. We will not describe here details concerning the management of the pointers, the free-list etc; these details can be found in [17].

While the SRAM buffer optimizes write accesses, read accesses are subject to a symmetrical problem when they are connected to variable size packets. Taking this into account, combined with our initial target, which is to reduce the crosspoint buffer size requirements, we propose that flows are being served in DRAM block granularity. 512-byte blocks are enough for sustaining peak DRAM throughput [35] while 512-byte crosspoint buffers are comfortable for modern technology [27]. Since all writes in DRAM are connected to large, fully-occupied, fixed-size DRAM blocks, all reads will also request large, fully-occupied, fixed-size DRAM blocks and thus peak read DRAM utilization is trivially guaranteed.

So, under heavy load queues grow and migrate to DRAM and then they are served in DRAM block granularity. On the other hand, under light load they may be fully accommodated in SRAM, bypassing the DRAM buffer. In this case, the occupancy of a VOQ may be much smaller than the DRAM block size. Then, we propose that the whole queue contents form a single, *variable-size* segment with no padding bytes and this segment is candidate to be forwarded to the buffered crossbar chip. Buffered crossbars cannot only operate on variable-size packets, but on variable size transfer units in general.

Fig. 3.2 shows the overall organization of the packet buffer. Notice that while memory management for the queues in the line card uses fixed-size blocks, traffic is forwarded to the crossbar in *variable-size* segments. Initially, we consider round-robin schedulers at both the switch inputs and outputs; the schedulers serve flows at segment granularity, ignoring the structure of the segment; in particular, the crossbar and the inputs do not know the location of packet boundaries, if any, inside the segment. Fig. 3.3, shows the overall datapath architecture. Since input and output schedulers are blind of packets, egress buffering is

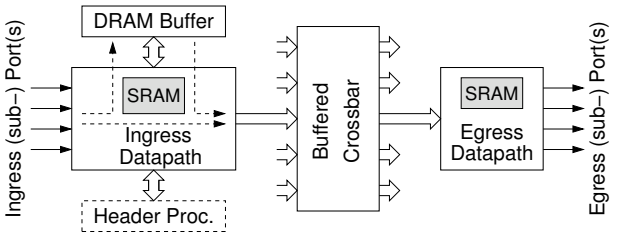


Figure 3.3: Overall datapath architecture. SRAM is used in the ingress path to streamline external traffic in order to optimize DRAM throughput utilization. SRAM is also used in the egress path for packet reassembly.

needed for packet reassembly. The egress module is described in section 3.4.

3.2 Segment size adaptivity

When a segment is forwarded to the crossbar, a crossbar-specific header is prepended to it, containing at least the crossbar output port ID and the segment length. The crossbar-specific per-segment header represents an undesirable overhead for crossbar throughput purposes. To minimize the effects of this overhead, we prefer most segments to have a large size relative to this header. If all segments had 40-byte (minimum-IP) size, the overhead would be a sizeable 10 percent with 4-byte internal headers; if most segments are above 200 bytes, the overhead is reduced below a comfortable 2 percent.

The segmentation scheme introduced in this paper has a nice *adaptivity* property with respect to the above overhead. Under light load, most segments contain a single packet, because packets are forwarded as soon as they arrive. Hence, small packets, which occur frequently in the Internet, generate small segments, which entail higher overhead; However, because the traffic is light, the increased overhead does not matter. Under heavy load, on the other hand, queue occupancy grows; when queue size exceeds one or two blocks, segment size is maximized. The reason is that all segments but the last two in a queue have a fixed size, equal to one DRAM block, hence the maximum segment size. Thus, under heavy traffic, the overhead is minimized and the crossbar operates very close to the maximum efficiency. In conclusion, the crossbar speedup that is needed for line-rate operation under uniform traffic[1] corresponds to the *minimum* value of the overhead, i.e. just 1 to 2 % with 512- or 256-byte maximum segments.

[1] unbalanced traffic is known to require additional speedup, for orthogonal reasons, though

3.3 Crosspoint buffer size

Buffered crossbars operating directly on variable size packets need a crosspoint buffer size of at least one round-trip time (RTT) window *plus* one maximum-size packet in order to achieve line-rate operation under single active flow and worst-case packet size conditions [14] (see section 2.3).

By contrast, the segmentation scheme introduced in this paper yields line-rate operation under single active flow and all packet size conditions with a crosspoint buffer size of just the *maximum of* one RTT worth of data or one maximum-size segment. The reason is that when queue occupancy grows under heavy load, the packets, *independent of their sizes*, are merged into maximum-sized segments, and the crossbar only sees a traffic consisting of such fixed-size units. Under such traffic, a buffer size of just one maximum-size segment suffices (provided it is also larger than one RTT worth of data).

3.4 Egress queueing and reassembly

The egress datapath chip collects segments until a full packet is reassembled for transmission. We assume that packet transmission to the egress port starts when the segment that contains the tail of the packet starts arriving from the crossbar; that is, we assume cut-through operation at the last-segment level. Cut-through operation, at the segment level, is also assumed inside the crossbar, as in [14].

For simplicity and economy, no flow control is needed from the egress line cards backwards: the output schedulers in the crossbars are assumed to operate at egress line rate[2]. A memory space of $N \times MaxPktSize$ per egress port, shared among the reassembly queues, ensures that reassembly buffers do not overflow; N is the number of ingress line cards.

3.5 Related work

In [18], Iyer e.a. study a hybrid SRAM/DRAM buffer organization. Their target is the design of a packet buffer as large as DRAM and as fast as SRAM. They assume that all VOQ's have their tail *and* their head blocks in SRAM, while their middle blocks migrate to DRAM. Two memory management algorithms were used to initiate transfers from SRAM to

[2] to support egress sub-ports, one needs per-subport VOQ's, and: either per-subport flow control from egress to ingress line card, or per-subport crosspoint buffers and output schedulers.

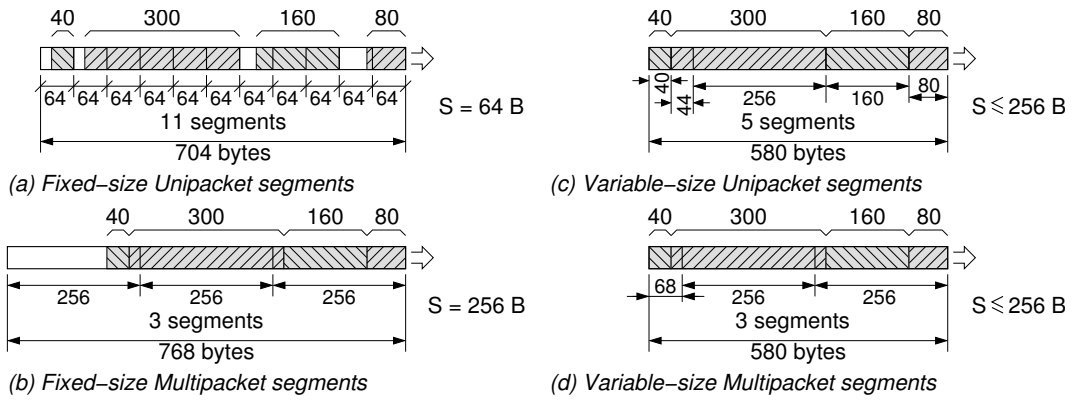


Figure 3.4: The proposed segmentation method (d) compared to already known schemes (a)-(c); (d) can be considered as the combination of (b) and (c).

DRAM and vice-versa in order to avoid SRAM buffer overflow and ensure that all requests from the crossbar arbiter are served from the head SRAM buffer. In essence the DRAM memory is transparent in [18]. This memory packet buffer architecture could be used itself at the ingress side of a variable-packet-size buffered crossbar.

By contrast, our SAR scheme allows the head blocks to remain in DRAM until ready to be switched, at which time they can move directly from DRAM to crossbar, as illustrated in fig. 3.2; this halves SRAM occupancy. Furthermore, we avoid the complexity of the memory management algorithm which transfers blocks from DRAM to SRAM; the design of this algorithm was the most critical issue in [18]. Another optimization relative to [18] is to allow low-occupancy VOQ's (such as the bottom queues in Fig. 3.2) to reside entirely in SRAM; this reduces DRAM throughput by twice the aggregate throughput of the flows that bypass DRAM.

Concerning already known segmentation methods, the traditional one is segmentation in fixed size segments (cells) (fig. 3.4(a)). Probably this is a relic of ATM and as we have already mentioned the disadvantage is that fragmentation overheads are incurred and core speedup is required. Stiliadis and Varma [36] proposed an alternative segmentation method for input queued switches, which resembles the scheme we propose in this paper: multiple packets or fragments thereof are packed inside an *envelope* and only fully occupied envelopes are switched through the (bufferless) crossbar (fig. 3.4(b)). The envelopes have large size and thus the peak scheduling rate and the crossbar reconfiguration frequency are reduced. These issues are critical in a bufferless crossbar architecture: the scheduling is hard - compared to the buffered crossbar architecture - and thus it is desirable to reduce scheduling rate; the crossbar configuration is usually changed by a central scheduler which is often placed on a separate chip, thus by reducing the crossbar reconfiguration frequency

we reduce power consumption. However, in [36] only fixed size cells can be switched and the authors propose various techniques to control the release of partially filled envelopes (e.g. partly filled envelopes are released upon a timer expiration). By contrast, partially filled "envelopes" are released at no cost in the buffered crossbar architecture. Moreover, the authors in [36] do not point out the connection of their segmentation method with the ingress memory management. Last, variable size segments have been used in [7] (fig. 3.4(c)). However, segments contain fragments of a single packet and thus the segmentation scheme has the shortcomings of switching variable size packets with respect to memory management and the inclusion of the backpressure window in dimensioning the crosspoint buffers. Moreover, the authors in [7] do not study the cost of segmentation and reassembly in buffered crossbars.

Chapter 4

Packet Mode Scheduling in Buffered Crossbars

In chapter 3, the switch was scheduled ignoring packet boundaries: we considered that flows are served at segment granularity, ignoring which segment belongs to which packet. In the resulted *segment-mode* operation (first part of fig. 4.1) the segments of a packet (say B) arrive at the egress line card interleaved in time with segments of other packets (A, C) arriving from other inputs. Packet transmission on the output line cannot start until the egress line card is certain of receiving the last segment of the packet. Given that scheduler decisions cannot be predicted, reassembly buffers are needed in the egress path in order to collect the packet segments; moreover, store-and-forward operation is enforced and cut-through cannot be used.

The problem is especially noticeable in systems requiring low latency (e.g. multiprocessor cluster interconnects), and when traffic includes large packets (e.g. jumbo frames [15]). In a 32×32 switch with 8 priority levels (flows) per switch input-output port pair and 8 Kbytes maximum packet size, we would need at least 2 Mbytes worth buffers per egress line card for the packet reassembly; the store and forward delay of a maximum packet is more than $6us$ at 10 Gb/s line rate.

Packet Mode Scheduling [19] - [38] is the alternative illustrated in the second part of fig. 4.1: when the switch makes a "connection" from an ingress to an egress line card for the first segment of a packet, that connection is maintained until all segments of the packet are switched. Since the egress line card knows that all segments of a packet will arrive consecutively in time, it needs no reassembly buffer, and it may start transmitting

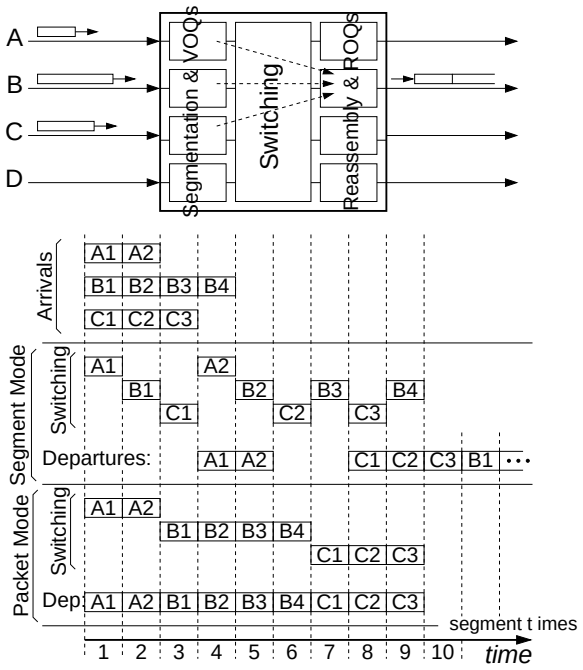


Figure 4.1: Packet mode versus segment (cell) mode switch scheduling

the packet right away, i.e. *cut-through* is allowed. Figure 4.1 illustrates that, under some circumstances, packet-mode scheduling reduces packet delay[1].

Packet mode scheduling had been studied in bufferless crossbar switches: a central scheduler determines input-output port pairings (connections) and once a connection is made, it keeps it until the last segment of the packet (fig. 4.2, left). Packet-mode scheduling in buffered crossbars is not a trivial extension of the bufferless case, because the scheduling process does not necessarily determine input-output port pairings. Scheduling is distributed and independent at switch input and output ports: each input selects a non-full crosspoint and sends to it; each output selects a non-empty crosspoint and reads from it (fig. 4.2, right). Two or more inputs (A and B in the figure) may send to the same column; two or more outputs (1 and 2 in the figure) may read from the same row. Crosspoint buffers allow periods of "inexact" pairings and that is why segment mode scheduling in buffered crossbars is superior to the bufferless case. On the other hand, packet mode scheduling requires exact pairings.

From another point of view, packet mode scheduling in buffered crossbars is a *synchronization* problem, resembling the traditional "producer-consumer problem" of operating systems [39]. The decision of a crossbar output port scheduler OS_j to operate in packet

[1] the timing diagrams were drawn ignoring the delay (assuming zero delay) of the line card and scheduler logic: a cell can be switched in the same time slot when it arrives; a packet departure may start in the same time slot when its last cell is known to have been scheduled.

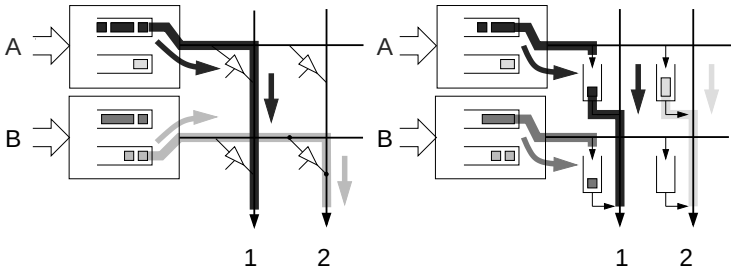


Figure 4.2: Input-output port pairings in a bufferless (left) and a buffered crossbar (right). Pairings in a buffered crossbar are "inexact".

mode for a packet of a flow f_{ij} presumes that the input counterpart scheduler IS_i keeps writing the rest of the packet to the crosspoint buffer (i, j) so that no buffer underflow occurs. Symmetrically, the decision of IS_i to operate in packet mode for a packet of flow f_{ij} presumes that OS_j keeps reading from the same crosspoint buffer (i, j) so that no buffer overflow occurs. Thus, input and output schedulers must synchronize before entering packet mode[2].

In this chapter we describe two methods for packet mode scheduling in buffered crossbars. The first one assumes distributed and independent scheduling at switch input and output ports, as in the classical buffered crossbar architecture. We synchronize the input and output schedulers so that whenever their independent decisions pair an input to an output port, that pairing is maintained for the lifetime of the corresponding packet transmission. We call this first scheme *probabilistic packet mode scheduling.* The second scheme, *deterministic packet mode scheduling*, obtains determinism - hence eliminates reassembly buffers - by reducing scheduling independence.

For the simplicity of presentation we will first assume that a packet with size S is fragmented to $\lfloor \frac{S}{U} \rfloor$ maximum segments with size U and a variable size segment with size $\frac{S}{U} - \lfloor \frac{S}{U} \rfloor$. We will explain how our methods operate with multipacket segments in section 4.3.

[2]of course, in case a packet is completely stored in a crosspoint buffer, or in case its last fragment is currently being written, OS_j can enter packet mode scheduling independent of IS_i. Similarly, IS_i can enter packet mode scheduling independent of OS_j when there is enough room in the crosspoint buffer for the entire part of the packet which is pending at the corresponding VOQ.

4.1 Probabilistic packet mode scheduling

4.1.1 Detecting pairings and entering packet mode

We say that the classical buffered crossbar scheduling process determines an (i, j) pairing when a segment of a packet P is being written to crosspoint buffer (i, j) by input i while the same or a previous segment of P is concurrently being read by output j (from the same crosspoint buffer). We propose that when such a pairing occurs - and under some appropriate timing constraints, which will be stated below - the input scheduler IS_i and the output scheduler OS_j keep serving the same flow f_{ij} until the whole packet P has been completely forwarded to the crossbar fabric and to the egress card respectively; then, the schedulers operate in *packet mode* for packet P.

OS_j can infer a pairing at the time of its occurrence: it just needs to observe both the read and write enable signals of the crosspoint buffer, while also checking that the same packet is being read and written. IS_i cannot observe the pairing sooner than half RTT from the moment it occurs. Since the input does not normally see the output decisions, IS_i also needs a *special notification* from the crossbar in order to know that a pairing occurs.

We claim that if an (i, j) pairing occurs while the crosspoint reads a segment s_k and writes a segment s_{k+m} of a packet P, it is safe for OS_j to enter packet mode scheduling for P if and only if

$$t_r - t_w \leq \Delta t_s - RTT, \tag{4.1}$$

where t_r denotes the time s_k starts being read from buffer (i, j), t_w denotes the time s_{k+m} starts being written to buffer (i, j) and Δt_s the maximum segment time. We name the time interval $t_r - t_w$ *synchronization distance* –SD; fig. 4.3 displays its maximum allowable value. So, if $SD \leq SD_{max}$, the pairing notification reaches the input before IS_i makes the next scheduling decision and IS_i enters packet mode scheduling *in time*: the subsequent segments of P depart from input i immediately after segment s_{k+m} and consecutively in time; therefore, crosspoint buffer (i, j) cannot underrun. Crosspoint buffer (i, j) can neither overflow because OS_j (being in packet mode) will keep reading the segments of packet P. On the other hand, if $SD > SD_{max}$, IS_i may possibly observe the pairing after making the next decision and thus it may start serving other flows while OS_j has already entered packet mode scheduling for packet P. A full proof of the crosspoint buffer *non-overflow/underflow property* appears in section 4.1.3.

Notice that when the segment size is smaller than the maximum, it contains the entire

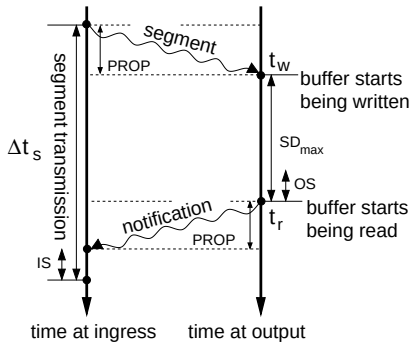


Figure 4.3: Maximum allowable synchronization distance.

tail of the packet and thus OS_j can enter packet mode scheduling independent of IS_i. Also note that when RTT is greater than the maximum segment time, no packet mode transmission occurs. Our method works only as long as RTT is smaller than the maximum segment time. We consider this to be achievable in usual, realistic systems, as e.g. with 512-byte maximum segment size at 10Gb/s and with RTT below 400ns [14].

Following the above discussion, in the proposed method, the input and output schedulers toggle between two modes. In the *segment mode*, they serve flows in a (weighted) round robin fashion, as in the classical buffered crossbar scheduling. In the *packet mode*, they keep serving the same flow until the whole packet has been forwarded. An output scheduler transits from segment to packet mode when a pairing occurs and the synchronization distance is smaller than the maximum. At the time it transits to packet mode, a pairing notification is sent to the input counterpart scheduler. An input scheduler transits from segment to packet mode when a pairing notification from the crossbar core arrives. Note that at most one pairing notification arrives at an input while the relative scheduler is in segment mode and no pairing notification arrives while in packet mode. Both input and output schedulers transit from packet to segment mode scheduling when the corresponding packet has been totally forwarded from the input and the crossbar output respectively.

Once the schedulers have entered packet mode scheduling, their decisions can be predicted and thus the packet can start being transmitted from the egress to the switch output port before it is completely stored at the reassembly buffer. What is needed is that the output scheduler communicates its mode of scheduling to the egress path.

4.1.2 Piggybacking control signals

A traditional buffered crossbar exchanges one control signal per data segment: a flow control credit is sent from the crossbar to an ingress line card for each data segment that departs from the crossbar to an egress line card. By contrast, our system may need up to three control signals per data segment:

(a) pairing notification to the ingress line card,

(b) packet mode notification to the egress for cut-through purposes, and

(c) credit to ingress.

Fortunately, we can piggyback the notifications (a) in the credit signals, and notifications (b) in the data segment headers.

Piggyback (a) has a penalty: we lose the opportunity to notify some pairings because when an input starts writing to an output after that output has started reading, the credit has already departed from the crossbar core[3]. To partly overcome this inefficiency, whenever the output scheduler fails to enter packet mode scheduling, due to the synchronization distance constraint, it "revisits" the same crosspoint during its subsequent decision slot and dequeues from it once more if a pairing occurs.

RTT should be increased to include the delay the credit may suffer in credit queues. If the credit scheduler always gives priority to credits carrying a pairing notification, then this delay is at most one credit time. Thus, the maximum allowable synchronization distance (equation 4.1) should be reduced by a credit time. Alternatively, the output scheduler should be notified that the credit was delayed and cancel its packet mode decision and the pairing notification to the input should be dropped.

4.1.3 Correctness

Theorem: *During a packet mode transmission of any packet P from input i to output j, there is no crosspoint buffer (i, j) overflow nor underflow.*

Proof: Let us denote with s_k the segment of packet P on whose the transmission at the output the schedulers synchronize and $s_n, n > k$ the segment carrying the last bytes

[3]We assume that a flow control credit is released at the time a segment starts being transmitted, as in [14].

of P. We will show that the credit c_i corresponding to segment s_i arrives at the input before the transmission of segment s_{i+1} starts, for all $i \in [k, n-1]$; then, because segments $[s_k, s_{n-1}]$ have fixed size –the maximum one, the transmission of any segment starts with the input having credits available at least for a maximum segment; hence, due to the properties of the credit flow control, no buffer overflow occurs. Equivalently, we will show that for all $i \in [k, n-1]$,

$$t_{ro,s_i} + RTT/2 \leq t_{ri,s_i} + \Delta t_s. \tag{4.2}$$

t_{ro,s_i}, t_{ri,s_i} denote the time the segment s_i starts being transmitted at the output and at the input respectively; (Remember that a credit is released at the time the relative segment starts being read at the output.) Δt_s denotes the maximum segment time. To show that the crosspoint buffer does not underrun, it suffices to show that

$$t_{ri,s_{i+1}} + RTT/2 \leq t_{ro,s_i} + \Delta t_s. \tag{4.3}$$

We will prove proposition 4.2 by induction on the number of packet segments which are being transmitted in packet mode. The proof of proposition 4.3 is symmetrical.

Base case: We will show that 4.2 holds for $i = k$. Because the packet mode transmission is initiated during the transmission of segment s_k, from equation 4.1 we get,

$$t_{ro,s_k} - t_{wo,s_k} \leq \Delta t_s - RTT. \tag{4.4}$$

Let t_{wo,s_k} denote the time segment s_k starts being written to buffer (i, j). Then:

$$t_{wo,s_k} = t_{ri,s_k} + RTT/2. \tag{4.5}$$

From equations 4.4, 4.5:

$$t_{ro,s_k} - (t_{ri,s_k} + RTT/2) \leq \Delta t_s - RTT \tag{4.6}$$

$$or, \qquad t_{ro,s_k} + RTT/2 \leq t_{ri,s_k} + \Delta t_s \tag{4.7}$$

Inductive step: We assume that credit c_m arrives at the input before segment s_{m+1} starts being transmitted. That is,

$$t_{ro,s_m} + RTT/2 \leq t_{ri,s_m} + \Delta t_s. \tag{4.8}$$

We will now prove that the same holds for credit c_{m+1} with respect to segment s_{m+2}. The transmission of segment s_{m+1} will start right after the transmission of segment s_m. Therefore,

$$t_{ri,s_{m+1}} = t_{ri,s_m} + \Delta t_s \tag{4.9}$$

$$and,$$

$$t_{wo,s_{m+1}} = t_{ri,s_m} + \Delta t_s + RTT/2. \tag{4.10}$$

It holds that

$$t_{ri,s_m} + RTT/2 \leq t_{ro,s_m}. \tag{4.11}$$

From equations 4.10, 4.11 we have

$$(t_{wo,s_{m+1}} - \Delta t_s - RTT/2) + RTT/2 \leq t_{ro,s_m} \tag{4.12}$$

$$or, \qquad t_{wo,s_{m+1}} \leq t_{ro,s_m} + \Delta t_s. \tag{4.13}$$

Therefore,

$$t_{ro,s_{m+1}} = t_{ro,s_m} + \Delta t_s. \tag{4.14}$$

From equations 4.8, 4.9, 4.14 we have

$$(t_{ro,s_{m+1}} - \Delta t_s) + RTT/2 \leq (t_{ri,s_{m+1}} - \Delta t_s) + \Delta t_s \tag{4.15}$$

$$or, \qquad t_{ro,s_{m+1}} + RTT/2 \leq t_{ri,s_{m+1}} + \Delta t_s. \tag{4.16}$$

4.1.4 Starvation

Scheduler synchronization during the transmission of segments belonging to different packets should be avoided in the face of the danger for *starvation*. If we allow an input scheduler i to synchronize with an output counterpart j while the first writes to output j a segment of a packet P_2 and the latter reads from the crosspoint (i, j) a segment of a previous packet P_1, it is possible that at the end of the packet mode transmission for P_2 a subsequent one will occur for the next packet P_3, again between input i and output j. This can easily happen when flow $f_{i,j}$ is the single active flow at input i. In this case flow $f_{i,j}$ monopolizes output j.

4.1.5 Performance

The proposed scheme is a combination of packet and segment mode scheduling. Under light loads, when input and output contention is rare, the buffered crossbar almost always pairs an input to an output port, packet mode transmissions are very likely to succeed and our method is very efficient. As input load increases and flows become congested, the schedulers are more likely not to synchronize, the percentage of packet mode transmissions decreases and our method's effectiveness degrades. Simulation results in section 6.3 confirmed these hypotheses. The contribution of the proposed method is the offered opportunity for cut-through transmission. Cut-through is beneficial under light loads since it greatly reduces packet latency.

4.1.6 Egress buffering

When the schedulers do not synchronize, some packets - or a part of them - are transmitted in segment mode, possibly interleaved with segments of other packets from/to other links. Thus, egress reassembly buffers are needed to collect these segments. Since no guarantee for packet mode transmission is provided, the reassembly buffers must be as large as the respective ones in segment mode scheduling: one maximum packet memory space per flow, per egress line card.

Fig. 4.4 shows why our method requires near the maximum egress buffering to account for the worst case. Two maximum sized packets P_A and P_B are queued at inputs A and B awaiting switching to outputs 2 and 1 respectively. Input B serves the first segment of packet P_A but output 1 does not because is already connected for a packet mode transmission with input A - hence the segment is buffered at the crosspoint (in the figure this corresponds to part (I); the thick lines display packet mode transmissions). Input B, after completing the segment transmission, connects to output 2 for a packet mode transmission (part II). When the packet mode transmission from input A to output 1 is completed (part III), output 1 serves the pending first segment of packet P_B, but it does not connect to input B because the segment is "stale" (the synchronization distance is long); the segment has to be queued at the corresponding reassembly buffer. Moreover, input A enqueues the first segment of packet P_A to the corresponding crosspoint buffer of output 2. Since that output is currently connected to a different input, it is impossible for input A to get in packet mode. However, after the segment transmission is completed, it does so with output 1 (part IV). When the packet mode transmission from input B to output 2 is completed, the first segment of

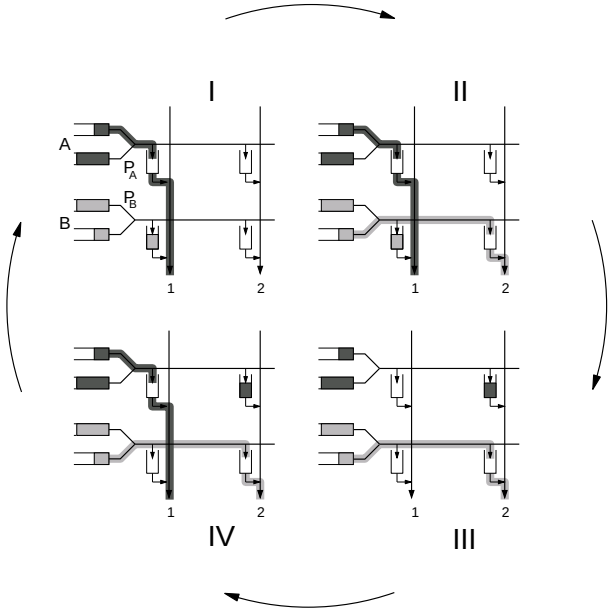


Figure 4.4: A traffic scenario which displays that the maximum packet interleaving can occur with probabilistic packet mode scheduling. The interleaved packets are labelled P_A, P_B.

packet P_A is transmitted to the egress and the second segment of P_B is enqueued to the relative crosspoint buffer. Again, no packet mode transmission can begin, so we recirculate to the initial state (part I). The described situation can be easily extended to show that if N flows step through the egress path, then we need at least $N-1$ buffers at the egress, one maximum packet worth each one.

Probabilistic packet scheduling reduces the packet interleaving in the switch core and thus also reduces the average occupancy in the reassembly buffers. Unfortunately, we cannot restrict the egress memory space based on the average case. We observed that a switch throughput - egress memory tradeoff appears. Suppose that $N = 4$ and two, instead of four, reassembly buffers (slots) are available at the egress. These buffers are allocated to the two flows whose packet segments arrive at the egress first. Other flows, to the same egress destination, are backpressured when they require packet reassembly because the buffers have been reserved. Consider that regarding egress 1 the two lucky flows are f_{01}, f_{31} as shown in fig. 4.5. Also consider that after these flows have been granted the reassembly buffers of egress 1, their inputs initiate a packet transmission to egress 0 and 2 respectively. Then, output 1 stays idle even if other flows destined to that output are active. Thus, by reducing egress memory we trade switch throughput. Early simulation results confirmed this trade-off, hence we abandoned this solution. As a matter of fact, the switch throughput degrades due to the *"faulty" decisions of the output schedulers*: each time a segment from input i is transmitted on segment mode to output j, that segment

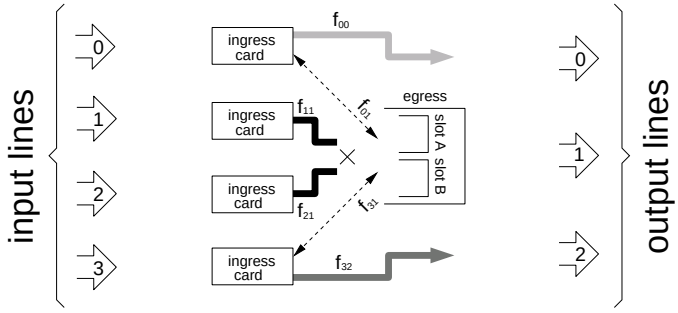


Figure 4.5: The egress buffer - switch throughput tradeoff. Four inputs and three outputs are shown. Inputs 0 and 3 have reserved the reassembly slots of output 1, preventing other inputs to transmit to this output.

reserves a reassembly buffer at output j at the risk that input i is afterwards connected for a packet mode transmission to a different output k.

4.2 Deterministic packet mode scheduling

In order to reduce or even eliminate the reassembly buffers we should avoid the "faulty" decisions of the output schedulers. To achieve this, one solution is to obtain determinism: an output scheduler should start serving a flow f_{ij} whose head packet is partly stored in the corresponding crosspoint memory, only if it is guaranteed that its input counterpart scheduler IS_i will start writing to buffer (i, j) no later than the time OS_j finishes transmitting the part of the packet already stored there.

4.2.1 Scheduling operation

Let ΔP denote the part of the packet stored at a crosspoint buffer, Δt the time interval within which the input scheduler responds with the continuation of the packet and R the line rate. If it is guaranteed that always:

$$\Delta t \times R \leq \Delta P \tag{4.17}$$

we can deterministically synchronize the input and output schedulers. As a result, each packet will be transmitted in packet mode and the reassembly buffers are not needed any more.

To guarantee that 4.17 always holds, the buffered crossbar is scheduled as follows.

1. A flow f_{ij} is eligible for the output scheduler OS_j if and only if a packet is completely

stored in the crosspoint buffer (i, j) or $\Delta P \geq (RTT + \Delta t_s) \times R$. In this inequality, Δt_s represents the max-size segment time: when input i receives the pairing request, it may delay honoring it for a time interval up to Δt_s, because it is busy transmitting a segment to another crosspoint. Each output scheduler serves the eligible flows round robin and asserts a flag in the flow control credit (as in the probabilistic method) each time it starts the transmission of a partly stored packet, requesting synchronization with the corresponding input.

2. Before an output scheduler OS_j starts the transmission of a partly stored packet at crosspoint (i, j) it must acquire a *lock* associated with input i. If that fails, because that input is currently connected to another output, the output scheduler proceeds serving the next eligible flow in the round robin schedule. Notice that the lock acquisition is an operation entirely *internal* to the buffered crossbar chip, and does not involve any transaction with the ingress line cards. As long as the lock for an input i has been acquired by an output scheduler OS_j all flows $f_{ik}, k \neq j$ having a partly stored packet at the crosspoint (i, k) are ineligible until the lock is released. A lock is released as soon as the last segment of the packet starts departing from the crossbar output.

3. An ingress line card receiving a synchronization request - inferred by the relative bit at the credit packet - gets synchronized to the requesting output right after its current segment transmission, if there is one, or right away if it is idle. If no synchronization request is received, the input scheduler serves flows in a round robin fashion.

With constraint (1) we impose that even if IS_i has initiated a transfer to output k at the time a request for synchronization with OS_j has arrived, it will be able to respond to OS_j in time. With constraint (2) we guarantee that there are no more than one requests arriving simultaneously at an input and that no requests arrive while an input is synchronized with an output.

Figures 4.6, 4.7 show an example of the proposed *asynchronoys* scheduling process. Figure 4.6 displays the status of the crosspoint buffers and how output schedulers are coordinated; fig. 4.7 displays how the round-robin pointers of input and output schedulers are updated. For simplicity, we assume that the RTT is zero. Each input selects an output to send to in a round robin manner. For example, the round-robin pointer of input 2 points to output 0. Output 0 is ineligible (because for example it is empty) but output 1 is eligible. Thus, input 2 sends to output 1 the first segment S_B of a packet P_B, and updates the round robin pointer to point to output 2. Likewise, input 0 concurrently sends the first segment

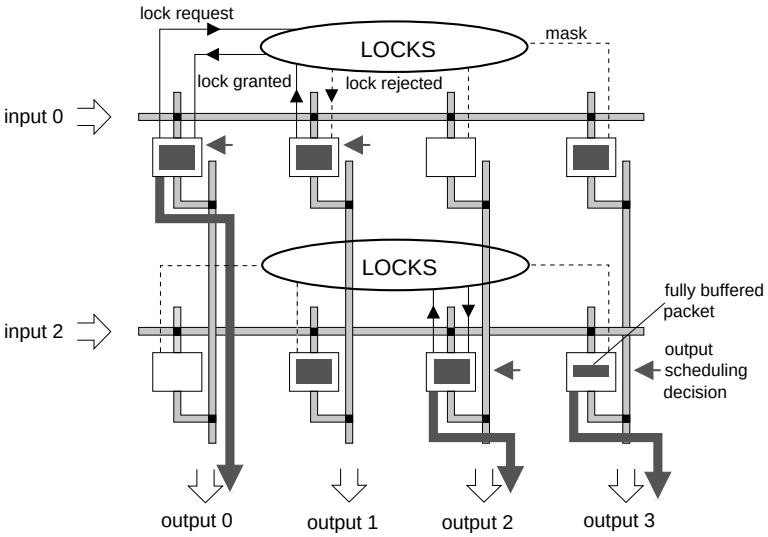


Figure 4.6: Proposed (asynchronous) crossbar scheduling. The lock acquisition phase is involved only when the output scheduler attempts to serve a flow whose head packet is partly stored in the crosspoint buffer.

of a packet to output 1 and updates its round robin pointer. Previously, it had sent to output 0 a segment S_A of another packet P_A, which was queued at the relative crosspoint buffer. Later on, outputs 0 and 1 finish their current transmissions at the same time and, since their round robin pointers both point to input 0, they decide to serve input 0. They both see that the queued segment - S_A for output 0 and S_B for output 1 - lacks the tail of the relative packet, so they request a lock for input 0. The lock is granted to output 1, so this output dequeues the segment S_B and locks its round robin pointer to input 0: it will keep serving the same input until the whole packet P_B is forwarded to the egress. Output

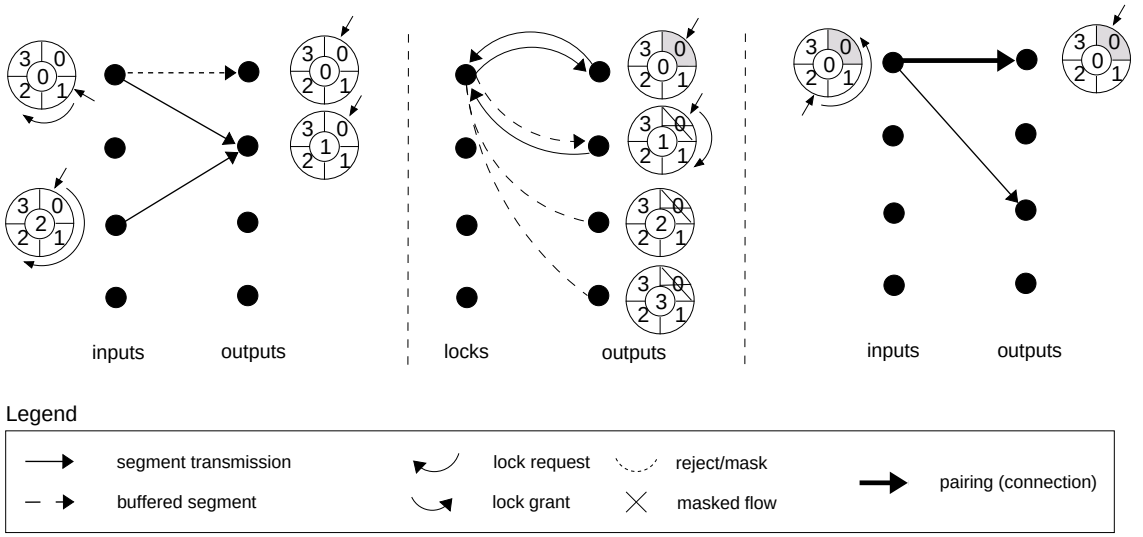


Figure 4.7: Updating the round-robin pointers of input and output schedulers. When a packet-mode pairing is configured (from input 0 to output 0 in the figure), the pointers are not updated and remain locked until the whole packet is forwarded.

1 asserts a packet mode notification inside the credit packet which corresponds to segment S_B. Output 0 receives a grant rejection and marks input 0 as ineligible. The rejection is also sent to the rest of the outputs. When input 0 receives the credit for segment S_A it sees the notification and locks its round robin pointer to output 0. Note that a packet mode pairing exists in parallel with a "segment mode pairing". After the current segment transmission is completed, input 0 will keep sending the subsequent segments of packet P_B to output 0. Input 0 will become eligible for the rest of the outputs when the last segment of P_B starts being transmitted to output 0. At that point in time, the lock is released and input 0 - output 0 get a divorce.

Note that the deterministic method does not impose a constraint on the RTT relative to the maximum segment size, like the probabilistic method does. However, each crosspoint buffer should be large enough in order for at least $(\Delta t_s + RTT) \times R$ bytes of the packet to fit there. In case the RTT is large, it is possible that an input has finished the transmission of the last fragment of the packet while an output requests synchronization for that packet. To avoid synchronization for different packet transmissions we associate an id with each credit, which is the same with the one of the corresponding packet. If an input scheduler receives a synchronization credit corresponding to a packet already departed from the VOQs, the scheduler simply discards the synchronization request.

In order to save switch throughput, a scheduling decision should be made before the current packet transmission has been completed. In [14], a 32×32 buffered crossbar was designed assuming link speed 10 Gb/s; the output scheduling time was 4 clock cycles, while a 40 byte packet transmission time was 10 clock cycles. Since in [14] the crosspoint buffers were 1 maximum packet large and the schedulers were completely independent, crosspoint buffer polling per switch output port was sufficient and efficient. Thus, in [14] output scheduling started 4 clock cycles before the completion of the packet transmission. In this paper, we assume that for a packet mode pairing (connection) to be established, an additional lock scheduling phase is required. Thus, we should start output scheduling by a lock acquisition time earlier than [14]. It is conservative to assume that the time for an output to learn whether the lock will be granted or not will take no longer than an output scheduling time, so this is feasible.

4.2.2 Contrast to the traditional three-phase matching algorithms

At first glance, deterministic packet mode scheduling resembles the classical request-grant-accept scheduling algorithm for bufferless crossbars. Resemblance concerns "large packet"

transmissions: the transmission of packet segments to the crossbar core can be considered as the request phase for packet mode transmissions, the output scheduling as the grant phase while the lock acquisition as the accept phase. The crucial observation that discriminates our scheme from the classical bufferless crossbar scheduling is that a *sufficient* first portion of a packet should be stored at a crosspoint in order for the corresponding flow to become eligible for the output scheduler. If P is the maximum packet size, S the maximum segment size and λ is the fraction of the maximum packet that can be stored in the crosspoint buffer, input scheduler IS_i is allowed to be "unfaithful" to output j, transmitting to one or more other outputs, for a time interval Δt from the time a synchronization notification arrives at the input.

$$\Delta t \leq \frac{k \times S}{R}, \qquad k = \lfloor \frac{\lambda \times P}{S} \rfloor \qquad and \qquad \frac{S + RTT \times R}{P} \leq \lambda \leq 1 \tag{4.18}$$

Furthermore, it can be easily shown that the maximum number NS of output schedulers which can synchronize with the same input at the same time is:

$$NS = \frac{1}{1-k}, \qquad k = \lfloor \frac{\lambda \times P}{S} \rfloor \qquad and \qquad \frac{S + RTT \times R}{P} \leq \lambda \leq 1 \tag{4.19}$$

In other words, up to NS output schedulers may hold a lock at the same time. Each input synchronized with $n \leq NS$ outputs keeps serving these outputs round robin at segment granularity. Thus, the greatest the part of the packet that can be stored at a crosspoint buffer, the greatest the decoupling of the schedulers. In the extreme case that the whole packet can be stored, the schedulers are completely independent. Notice however, that the buffer size requirements increase exponentially with the number of independent output schedulers.

With the proposed scheduling method the resulting switch operation is fully asynchronous. Concerning transmissions of packets that can be totally buffered at the crosspoints, scheduling reduces to the classical, asynchronous buffered crossbar scheduling [14], [27]. When large packets are involved, inputs need not synchronize before transmitting to outputs, because buffers at the crossbar outputs absorb the output conflicts; inputs synchronize in the long-run by the flow control protocol. Furthermore, outputs need not synchronize with inputs again because there is a buffered segment to read until the input is finally synchronized; inputs synchronize with outputs in the longer run, and particularly within the Δt time interval (equation 4.19). On the other hand, outputs need to be coordinated in order to eliminate input contention when large packets are involved, but this does not imply synchronous operation.

However, the proposed method induces a performance penalty compared to the buffered crossbar with one maximum packet worth crosspoint buffers: when large packets are involved, the matching capabilities of the buffered crossbar scheduling are limited because input contention is curtailed. Simulation results in section 6.4 indicate that indeed there is a cost, but it is rather low and perhaps it is negligible compared to the silicon savings.

4.3 Packet mode scheduling connected with ingress memory management

The scheduling methods proposed in this chapter can be easily connected with ingress memory management, i.e. they can be extended to handle multipacket segments (chapter 3).

Concerning the probabilistic case, we consider that input and output schedulers serve flows in segment granularity, with the segments being the ones aligned during the memory management of the DRAM packet buffer. Thus, the last segment in a packet mode transmission may contain packets (or pieces of them) which follow the packet being transmitted in packet mode, when this packet is not aligned with the segments. In this case, the bytes in the last segment, belonging to different packet/packets, get a free ride to the egress. For simplicity and in order to avoid starvation side-effects, if an input writes a segment to an output and that output reads this segment, the resulting pairing is not considered valid if the segment contains complete packets.

Concerning the deterministic scheduling, input schedulers write DRAM blocks in the crosspoint buffers, while the output schedulers read packets. Packets or fragments of packets that follow the packet being transmitted in packet mode will get a free ride to the crosspoint buffer when the packet is not aligned with the segments.

There is an interesting analogy between our methods and the scheduling proposed in [18] for the memory system of ingress line cards. In [18], the bulk of each VOQ is held in long-latency DRAM, while the head (and tail) of the VOQ is held in low-latency SRAM; a memory management controller (scheduler) tries to keep all VOQ head SRAM buffers non-empty, so that these can serve the (unpredictable) requests by the crossbar scheduler (see fig. 4.8, upper part; see also chapter 3.). The analogy is as follows: VOQ's in long-latency DRAM in [18] are analogous to our VOQ's in the ingress line cards; VOQ head SRAM buffers in [18] are analogous to our crosspoint buffers; the memory management controller

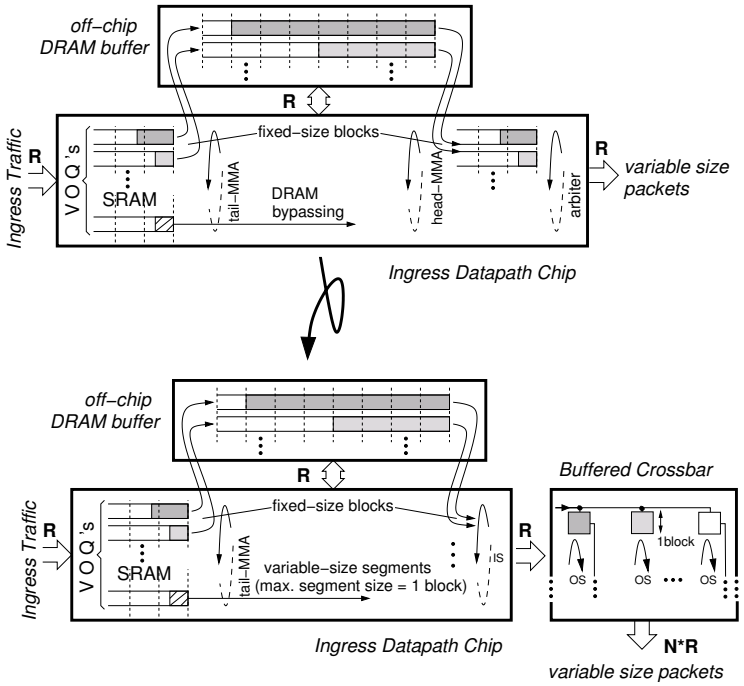


Figure 4.8: Packet mode scheduling connected with memory management. Comparison with the line card organization proposed in [18].

of each ingress line card in [18] is analogous to the input scheduler in each of our ingress line cards; the (central) crossbar scheduler in [18] is analogous to our output schedulers (see fig. 4.8).

In spite of the analogies between the resources to be scheduled, our case differs from that of [18]. In [18] the peak read access rate equals the switch line rate. In buffered crossbar scheduling the corresponding peak rate is N times larger, where N is the switch size, and it occurs when all the output schedulers decide to read from the same crossbar row. A study on the application of the algorithms in [18] to our problem gives rise to future work. Conversely, future work could include the application of our method to memory management scheduling.

4.4 Packet mode scheduling in buffered vs. bufferless crossbars

In this section, we compare qualitatively our scheduling schemes for buffered crossbars to the original packet mode policy for bufferless, input queued crossbar switches. A quantitative comparison appears in chapter 6.

The first difference concerns packet size granularity. Crosspoint buffers allow the input and output schedulers in the crossbar to operate independent of each other, thus eliminating the requirement for synchronized decisions and fixed size cells: buffered crossbars can directly switch variable size packets with fine size granularity[4] [14]. In this paper, we maintain scheduling independence, although to a reduced extent. As a result, our methods yield asynchronous operation and can switch variable size segments, with fine-grained segment size; padding overheads are eliminated.

By contrast, packet mode scheduling in bufferless crossbars [21] assumes cell granularity for packet size. The cell size is bounded below by the time needed for the crossbar scheduler to find a bipartite matching; this results in cell sizes usually in the range of 64 to 128 Bytes [18]. Fixed-size cells incur padding overhead, which requires internal crossbar *speedup*, by a factor of around 2 in the worst case[5].

Second, we point-out a shortcoming of packet mode scheduling in bufferless crossbars. Bufferless crossbars require *exact* pairings *at all times*. Consider a heavily loaded switch where all input ports are currently connected to one output each, and conversely all output ports are being fed by an input each. Connections are held for an entire packet duration. Consider a case where, when one of the connections is terminated –because the corresponding packet has been delivered in its entirety– *no other* connection happens to have terminated at the same time. Then, there is only a *single input* and a *single output* port that have become available for new pairing(s), hence the scheduler is forced to again pair the same input to the same output. Figure 4.9 shows an example of such a traffic pattern. Under such traffic, the scheduler is forced to maintain the crossbar configuration "locked" into a fixed set of connections (flows $f_{00}, f_{11}, f_{22}, f_{33}$ in the figure), thus starving all other flows.

By contrast, buffered crossbars allow *temporary* situations of "inexact" pairings, i.e. times when multiple inputs forward traffic to a same output or multiple outputs read packets from crosspoint buffers that had been fed by a same input at different times in the past. These periods of "inexact" input-output matchings, allow buffered crossbars to escape from the above "locked" configurations. Buffered crossbars with one maximum packet crosspoint buffers [14], or buffered crossbars scheduled with the proposed probabilistic method, allow such periods to occur and thus they do not lock in fixed configurations. On the other hand, deterministic packet mode scheduling requires exact pairings when the crosspoint buffer size is less than half the maximum packet size (see section 4.2.2). Thus, with this scheme and

[4] size granularity equals to the crossbar datapath width (4 Bytes in [14]).
[5] e.g. with 65-Byte packets in a 64-Byte-cell switch.

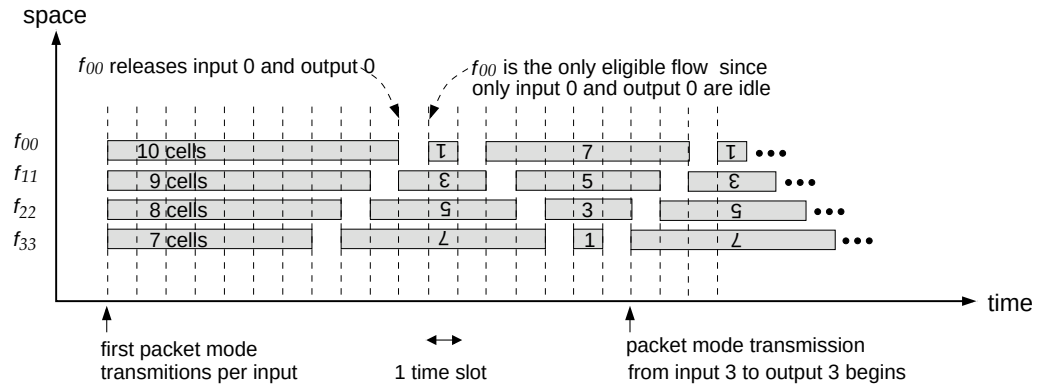


Figure 4.9: A staircase-like traffic pattern that leads a packet-mode scheduler for a bufferless crossbar to "lock" in a fixed configuration. Assume a 4×4 switch; the figure shows the 4 flows that are constantly served; other flows with non-empty queues exist, but are never served.

with less than half the maximum packet crosspoint buffer, the crossbar can still get locked. For example, the scenario shown in fig. 4.9 can lock a buffered crossbar if the packets shown are larger than the crosspoint buffer. The problem can be solved if the crosspoint buffers are at least half maximum packet large. Then, "inexact" pairings can still occur, as described in section 4.2.2, and the scheduling can escape from this bad situation.

Chapter 5

Method

We used computer simulation to study the performance of our methods. Our purpose was twofold: (i) study of the properties of our schemes and (ii) compare our systems to counterpart systems. In this chapter, we describe the methodology and the framework of our simulations; simulation results are presented in the next chapter.

We modelled the following systems:

1. *Buffered crossbar with probabilistic packet mode scheduling.* This scheme was described in chapter 4 and it is labelled *PPM*.

2. *Buffered crossbar with deterministic packet mode scheduling.* This scheme was described in chapter 4 as well, and it is labelled *DPM*.

3. *Buffered crossbar with full variable size packets*, labelled *VPS*. This is the system described in [14] and reviewed in chapter 2; it serves as the reference architecture.

4. *Buffered crossbar with blind SAR*, labelled *SM*. In this model, the input and output schedulers serve flows round-robin and forward segments ignoring packet boundaries. Two segmentation schemes are considered (see chapter 3): *(i)* segmentation to variable size, multipacket segments (fig. 3.4(d)) and *(ii)* segmentation to variable size, unipacket segments (fig. 3.4(c)). Concerning (i), we did not model the memory management functionality of the packet buffer, but instead we assumed that the queues are infinite and each dequeue operation gives a segment whose size (S) and contents

System	Xpoint BufSize(1)	Seg. Size(1)	RTT(2)	Credit Time(2)	Sched. Time(2)	$\frac{XbarRate}{Ext.LinkRate}$
PPM	1	[0.078-1]	0.95	0.033	0.031	1
DPM	3	[0.078-1]	0.95	0.033	0.031	1
SM	1	[0.078-1]	0.95	0.033	0.031	1
VPS	18	N/A	0.95	0.033	0.031	1
IQ	N/A	0.125	0	N/A	0.125	1

Table 5.1: Default Simulation Parameters

(1) in units of 512 Bytes, (2) in units of 512 Byte times. All switches are 16×16.

depend on the backlog of the queue (Q).

$$S = \begin{cases} MAX & \text{if } Q \geq MAX + MIN \\ MAX - MIN & \text{if } MAX < Q < MAX + MIN \\ Q & \text{if } Q \leq MAX \end{cases}$$

In the above equality, MAX and MIN are the maximum and minimum segment size. A segment with size S contains the head S bytes of the queue independent of packet boundaries. Concerning *(ii)*, if P is the part of the packet pending at a VOQ, then the head segment of the VOQ has size S,

$$S = \begin{cases} MAX & \text{if } P \geq MAX + MIN \\ MAX - MIN & \text{if } MAX < P < MAX + MIN \\ P & \text{if } P \leq MAX \end{cases}$$

and consists of the S first bytes of the pending part of the packet, i.e. the size of the segments depends only on the size of the packet.

5. *Input queueing* with packet (PM) or cell mode (CM) scheduling, labelled *IQ*. This is the classical input queueing architecture (with VOQ's and a central scheduler) and operates in cell-time granularity using packet or cell mode scheduling; the iSLIP scheduling algorithm is assumed. We simulated the packet mode modification of *iSLIP* from [19]; cell size is 64 Bytes and one iteration is assumed.

The models for the buffered crossbar were developed using an ad-hoc, event-driven simulator which is byte-time-accurate [41]. For the input queueing architecture we wrote a slotted-time simulator. Table 5.1 shows the default parameters for all the simulated systems.

5.1 Traffic models

We chose the traffic models in order to offer insight on the scheduler performance under some "clear and extreme" traffic circumstances, rather than using just a single, "real life" traffic model. Additionally, experiments with such traffic patterns are more easily repeatable.

For the buffered crossbar we assume minimum packet size 40 Bytes and maximum 8 KB with 1 byte packet size granularity. For the bufferless crossbar we consider minimum packet size 64B (1 cell) and maximum 8KB (128 cells), with one-cell packet size granularity. Although this integer-cell size granularity favors *IQ* (padding overhead is a serious disadvantage of *IQ* - section 4.4), we made this assumption in order to compare pure scheduling efficiency, factoring out padding overhead. Four packet size distributions were used:

- *bimodal* - 95% of the packets are minimum sized and 5% are maximum sized.
- *trimodal* - 64% of the packets are minimum sized, 18% have size 552 bytes and 18% 1500 bytes.
- *uniform* - packet size is uniformly distributed in the range between minimum and maximum.
- *constant(x)* - packet size is constant, x bytes.

Packet destinations are specified uniformly, i.e. with equal probability for all switch output ports, or in an unbalanced manner according to the Zipf law [42]: each switch input port sends to a switch output port with probability $Zipf(i)$.

$$Zipf(i) = \frac{i^{-k}}{\sum_{j=1}^{N} j^{-k}} \qquad N = \#ports \tag{5.1}$$

As k increases, a switch input port favors an output port and recursively the same is true with the rest of the outputs. For $k = 0$ the traffic is uniform while for $k \to \infty$ all traffic at an input is destined to a single output; for an intermediate value ($k = 2$) fig. 6.10 displays the load distribution. Each input has a fovored output so that the traffic is feasible (no output port is overloaded); for example, when each input i favors output i, no overloading occurs.

For the buffered crossbar we assumed Poisson packet arrivals, while for the bufferless crossbar we modelled packets as bursts of cells, following the same approach as [21]. The

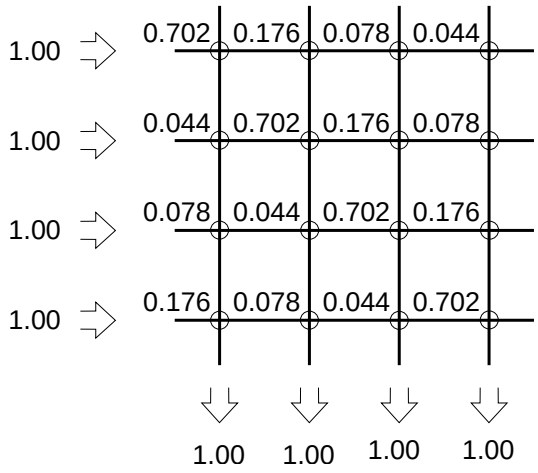


Figure 5.1: Load distribution in a 4×4 switch according to the Zipf law, $k = 2$. Each input favors a single output and recursively the same is true for the rest of the outputs. Each input favors an output so that no output is overloaded.

maximum packets were 8KB large in order to show that our methods efficiently switch very large packets.

5.2 Performance metrics

Switch throughput and *average queueing delay* were used as the performance metrics. The delay of a packet in a switch queue was defined as the time interval between the first byte of the packet arriving to the queue and its first byte departing from the queue. The total queueing delay of a packet in the switch was computed as the time interval between the first byte of the packet arriving to a VOQ and its first byte departing from the reassembly buffer, when the system requires egress reassembly buffers, or departing from the crossbar output port, when it does not. *Constant delays*, such as propagation and scheduling times, were subtracted. The delay was averaged over the number of packets with each packet delay contributing the same portion to the average (*average delay*). The reported delay values are in units of 512 byte times (the transmission time of a 512B segment). The switch throughput corresponds to the number of bytes departing from all switch output ports during the measured time interval, divided by that interval (in byte times) and by N, the size of the switch.

5.3 Interval estimates of the measurements

A simulation run is defined by the time needed for at least 25×10^3 packets per flow to depart from the switch egress ports. To effectively remove the cold-start bias, we start collecting statistics after at least 100 packets per flow have departed from the switch egress. The reported values are the average of multiple simulation runs and they differ from the real ones by less than 4% with probability greater than 95%. For the computation of the confidence intervals we follow the methodology described in [11]; however, the minimum number of simulation runs is 5 instead of 30. Few results are obtained from a single simulation run; we will distinguish these results by mentioning in the caption of the relative figure that they have been obtained from a single run.

Switch stability is checked as follows. The simulated time is divided into five intervals. If for all the intervals the backlog in a VOQ is greater than 10% of the backlog in the previous time interval, the switch is considered unstable.

Chapter 6

Results

6.1 Segment size adaptivity

We verify the adaptivity hypothesis described in section 3.2 and we show the benefits with respect to internal header overhead and crosspoint buffer size. We use the SM model with uni- and multi- packet segments.

First, we assume uniformly destined traffic and *constant(40)* packet size distribution. Figure 6.1 displays the saturation throughput with uni- and multi- packet segments when the switch header size is 4 bytes. The system saturates at a load of 91%(= 40/44) with unipacket segments while at a load of 99%(= 512/516) with multipacket segments. At a load about 91% the VOQs are backlogged, because the arrival rate exceeds the service rate. With unipacket segments, the header is added to minimum-size packets, so the backlog keeps increasing infinitely. With multipacket segments, from the point in time that the backlog becomes greater than 512 bytes, 512-byte segments are forwarded to the crossbar, hence the system saturates at a load of 99%. Notice that with multipacket segments some backlog is required to be created in order for the system to start operating efficiently and this is the reason for observing the "knee" on the curve.

Next, we assume trimodal packet size and unbalanced destinations. Figure 6.2 displays the switch throughput under a range of values for the zipf factor k. Observe that for large values of k (then the traffic becomes almost unidirected) using multipacket segments we achieve full line rate with just a maximum segment crosspoint buffers, while with unipacket segments the maximum throughput is slightly larger than 80%. With unipacket segments we lose throughput because alternating minimum-maximum segments induce output un-

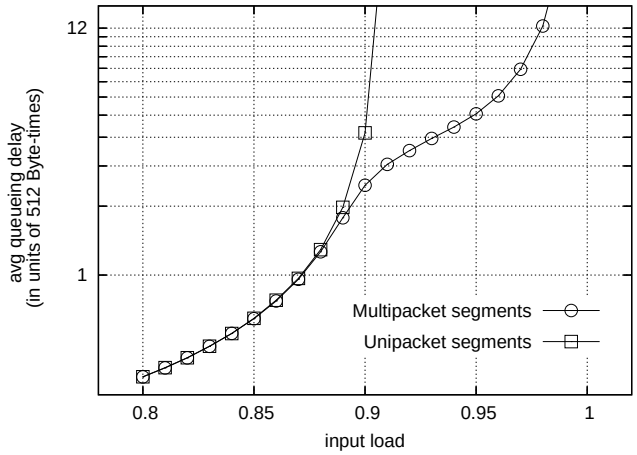


Figure 6.1: Buffered crossbar with blind SAR (SM model). Saturation throughput with uni- and multi-packet segments. Header size is 4 bytes. Packet size is *constant(40)* and traffic is uniformly destined.

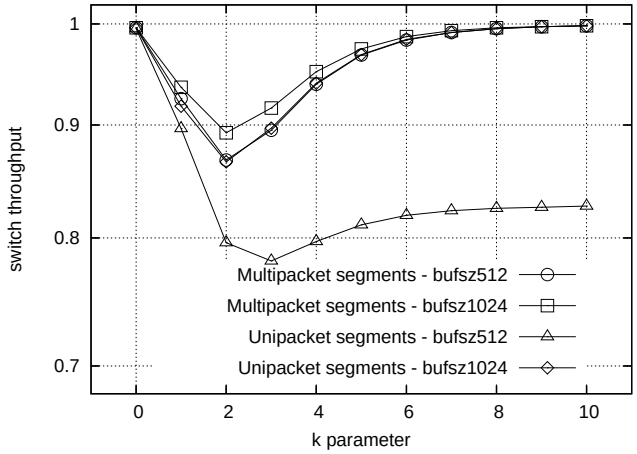


Figure 6.2: Buffered crossbar with blind SAR (SM model). Switch throughput under unbalanced (Zipf) traffic with multi- and uni- packet segments. Crosspoint buffer size is 512 or 1024 bytes. Packet size is trimodal.

derutilization as described in section 2.3; the problem is alleviated when 1 Kbyte crosspoint buffers are used (slightly greater than 1 max. packet + 1 RTT).

6.2 Segment size dependence of performance

Figure 6.3 shows the performance of buffered crossbar with blind segmentation and reassembly (SM model with multipacket segments) under a range of values for the maximum segment size. The crosspoint buffer always equals the maximum segment size. Header size is 4 bytes and packet size is trimodal. The egress can start transmitting a packet as soon as its last segment starts arriving, so with greater maximum segment size the delay at the egress is smaller and this is more pronounced under light loads. For heavy loads, delay is

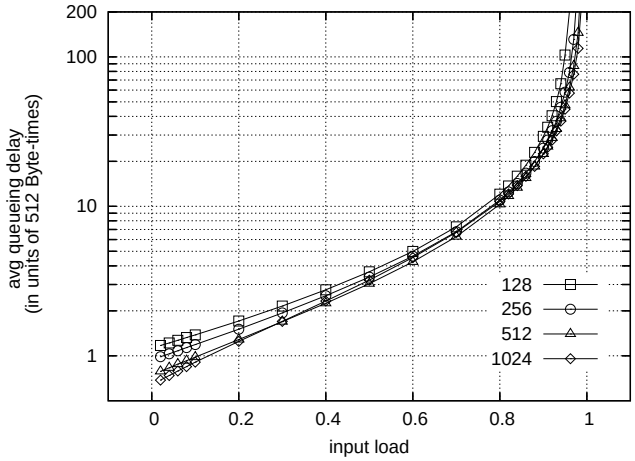


Figure 6.3: Buffered crossbar with blind SAR (SM model) and variable size multipacket segments. Crosspoint buffer size equals maximum segment size. Maximum segment size is 128, 256, 512 or 1024B. Round-trip time equals maximum segment time. Header size is 4 bytes. Packet size is trimodal and traffic is uniformly destined. The reported results were obtained from a single run.

again smaller when the maximum segment is large because the header overhead is reduced. (again 4-byte headers are assumed.)

6.3 Probabilistic packet mode scheduling

6.3.1 Comparison to counterpart systems

We compare probabilistic scheduling (PPM) to buffered crossbars with blind SAR (SM) and full size packets (VPS). Our goal is to show that PPM reduces delay in egress buffers. The SM model serves as the upper bound for reassembly delay, because the packets are loosely interleaved in the switch core and store&forward operation is enforced in the egress path (see fig. 4.1). VPS serves as the lower bound because no reassembly is required; furthermore the packets can cut-through directly from the crossbar core to the output.

We assume uniformly destined traffic and *bimodal*, *uniform* or *const(8KB)* packet size. We first report the results for bimodal packet size. In the buffered crossbar with blind SAR (SM), for loads up to 0.2, the large packets were delayed in the reassembly buffers for around one packet store time (16 segment times). In the rest of the load range, this delay increased significantly, due to the packet interleaving in the switch core. With *PPM*, the delay in the egress buffers was almost zero for loads up to 0.5, because cut-through transmissions were possible in the egress path and the packet interleaving was limited. PPM becomes less efficient for higher loads: egress delay reduction, compared to SM, goes down from almost 100% for light loads to 80% for loads around 0.7, and to only 30% for a load of 0.95.

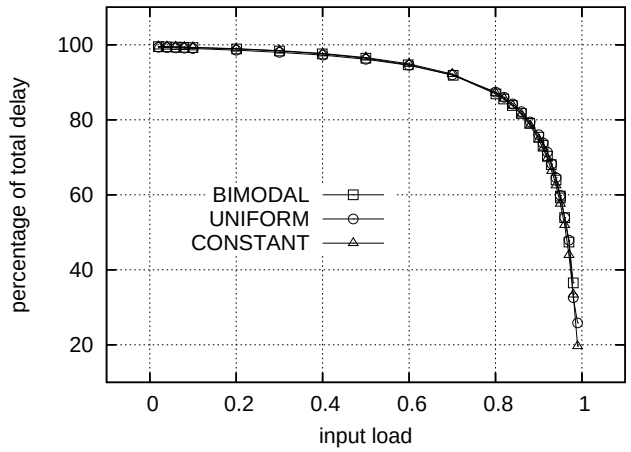


Figure 6.4: Buffered crossbar with blind SAR (SM model) and multipacket segments. The plot shows the percentage of delay in reassembly buffers over total queueing delay. Traffic is uniform and packet size is bimodal, uniform or constant(8KB).

The conclusions for $uniform$ and $const(8KB)$ traffic are similar with PPM being slightly worse as the average packet size increases.

Figure 6.5 shows the average (total) packet delay in SM, VPS and PPM buffered crossbar for three packet size distributions. The upper row of fig. 6.5 compares PPM to SM. The performance improvement is reflected on the average delay metric only when the percentage of large packets in the traffic is large enough, because only large packets benefit from our scheme; thus, average delay is greatly reduced with *uniform* and *const(8KB)* traffic but the reduction is less pronounced with bimodal traffic (95% of the packets are 40-byte packets). Note that, PPM is very close to VPS for all packet size distributions.

For loads greater than 95% the delay in the egress path contributes a significantly smaller percentage to the total packet delay. For such high loads, the VOQs start growing and the delay in the ingress overwhelms the delay in the egress path (see fig. 6.4).

6.3.2 Effect of the synchronization distance on performance

We run experiments varying the propagation time from the ingress cards to the fabric in order to study the effect of the synchronization distance on the performance of the probabilistic scheme. For propagation times of 64, 128, 256 and 460 byte times and under uniform traffic and bimodal packet size no variation on the average queueing delay was seen for the whole load range.

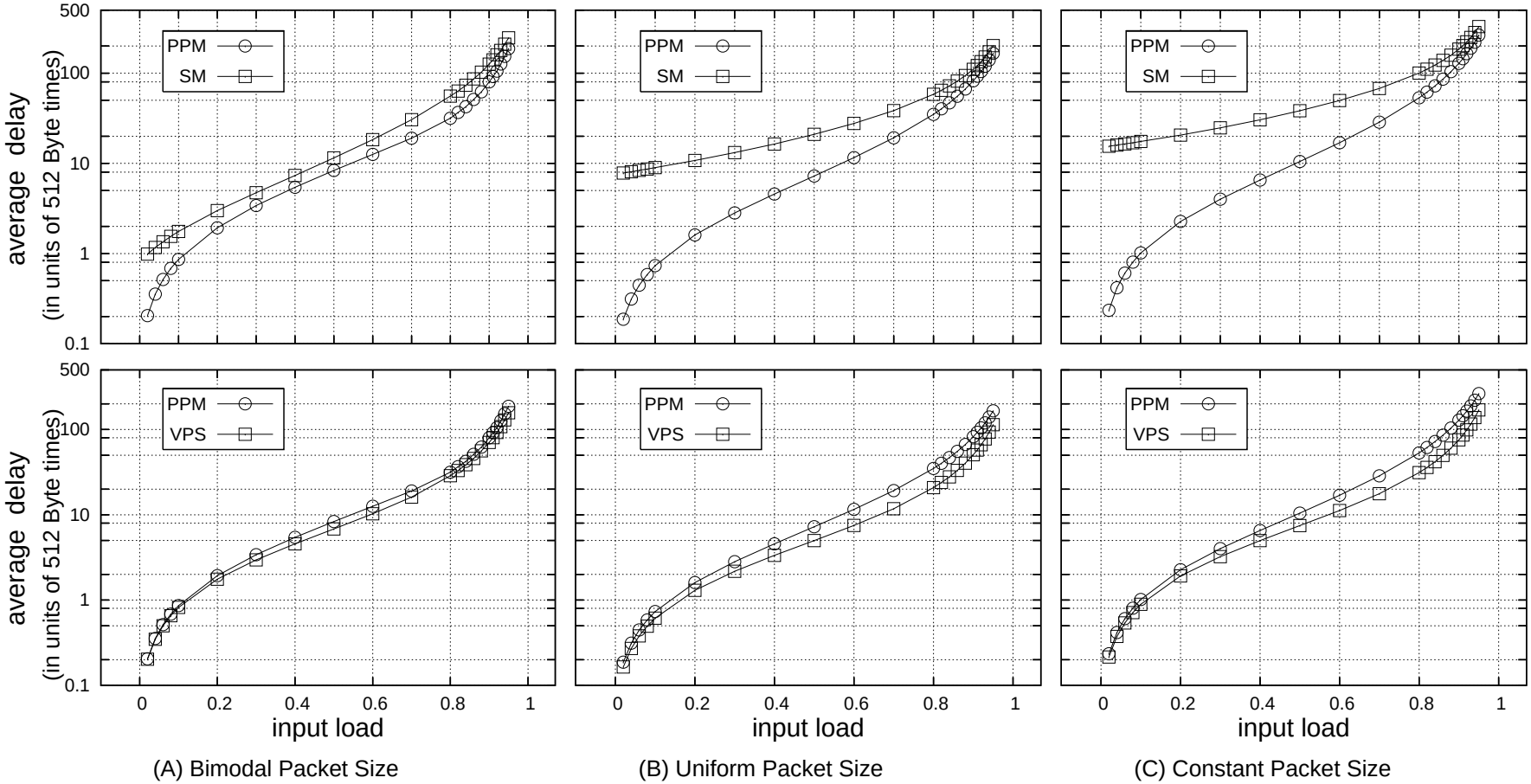


Figure 6.5: Deterministic packet mode (DPM) scheduling under uniformly destined traffic. Comparison to probabilistic scheduling - PPM (upper row), (full) variable-packet-size scheduling in buffered crossbar - VPS (middle row), packet mode scheduling in Input Queueing - IQ (bottom row).

6.4 Deterministic packet mode scheduling

6.4.1 Comparison to counterpart systems

We compare deterministic packet mode (DPM) scheduling to the probabilistic scheme (PPM), to (full) variable packet size scheduling (VPS) in buffered crossbars and to packet mode scheduling in bufferless, input queued switches (IQ). The comparison to the VPS system is unfair for DPM because VPS uses 18 times larger crosspoint buffers. The comparison to IQ is unfair for DPM as well, because we assume course-grained packet size granularity for IQ. Nevertheless, DPM appears virtually as good as buffered crossbars with very large crosspoint buffers and always better than IQ.

Comparison to probabilistic packet mode scheduling: Fig. 6.6(upper row) compares DPM to PPM. DPM imposes a crosspoint buffer delay and that is why it is slightly worse than PPM at light loads; this is more pronounced when the average packet size is large, i.e. for the uniform and constant packet size distribution. If desired, one can patch-up this inefficiency by combining DPM with PPM. For bimodal packet size, DPM is always (slightly) better than PPM; PPM's extra delay is due to the delay in reassembly buffers. For uniform and constant(8KB) packet size, the delay in reassembly buffers makes PPM slightly worse than DPM in the range of loads [0.5:0.95], but for loads greater than 0.95 PPM total delay is smaller than DPM even though it includes the delay in the egress buffers. This shows the *cost of the lock acquisition phase* in DPM: PPM exploits the full matching opportunities of the buffered crossbar scheduling, while DPM forbids input contention when "large" packets are involved. Note that under bimodal packet size lock acquisition was almost transparent.

Comparison to buffered crossbar with full size packets: Fig. 6.6 (middle row) shows the comparison between DPM and VPS. For bimodal packet size, the delay curves are almost indistinguishable. For uniform or constant packet size, VPS was superior again because DPM curtails the matching opportunities of the buffered crossbar scheduling. However, note that for the simulated configuration VPS uses 85% more memory in the crossbar chip.

Comparison to input queueing: Figure 6.6 (bottom row) compares packet mode scheduling in buffered and bufferless crossbars, in terms of average delay, and for all of the considered packet size distributions. The superiority of DPM is more pronounced when the average packet size is small because the efficiency of the buffered crossbar scheduling is better exploited then (less lock transactions are involved).

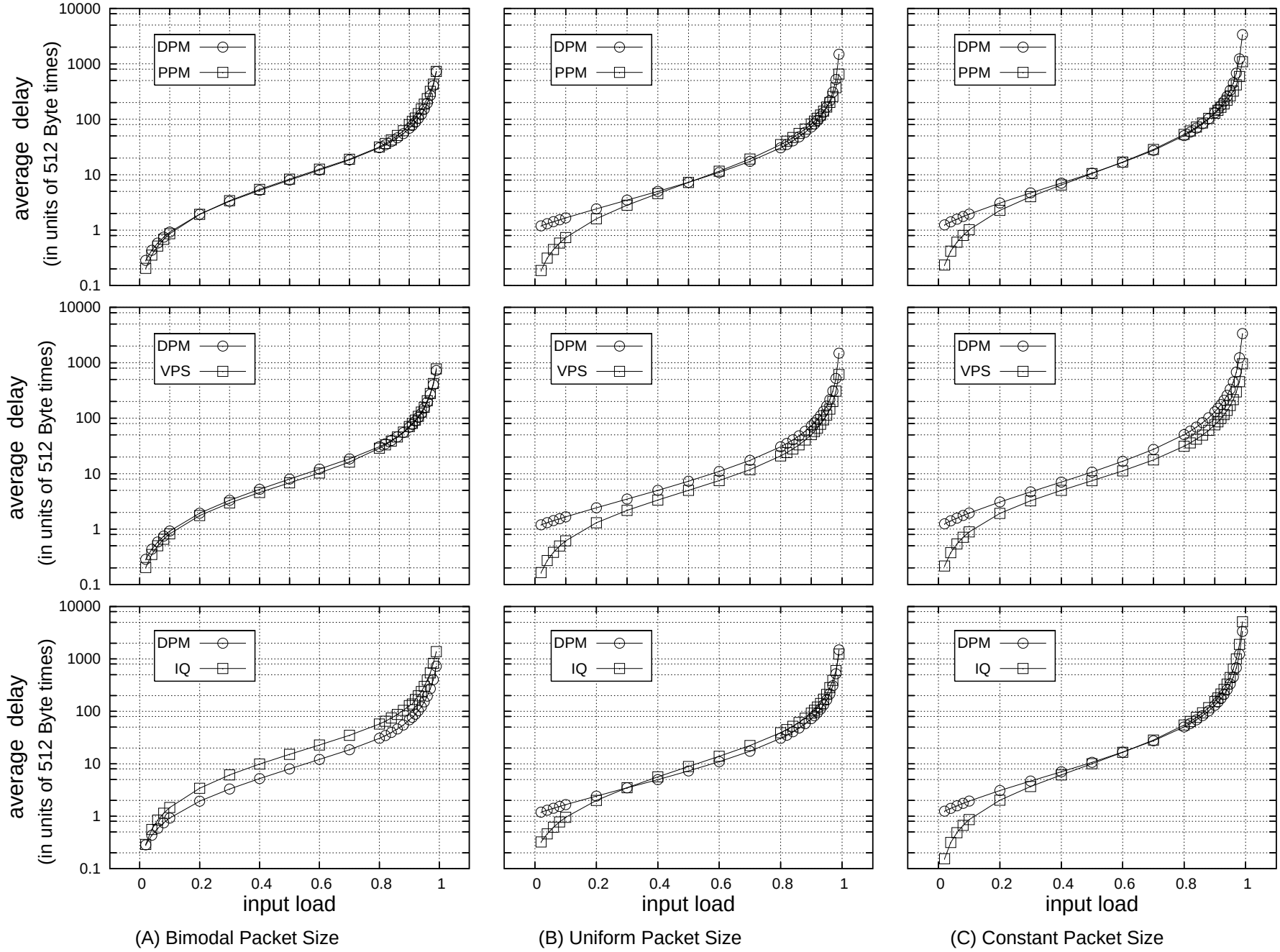


Figure 6.6: Deterministic packet mode (DPM) scheduling under uniformly destined traffic. Comparison to probabilistic scheduling - PPM (upper row), (full) variable-packet-size scheduling in buffered crossbar - VPS (middle row), packet mode scheduling in Input Queueing - IQ (bottom row).

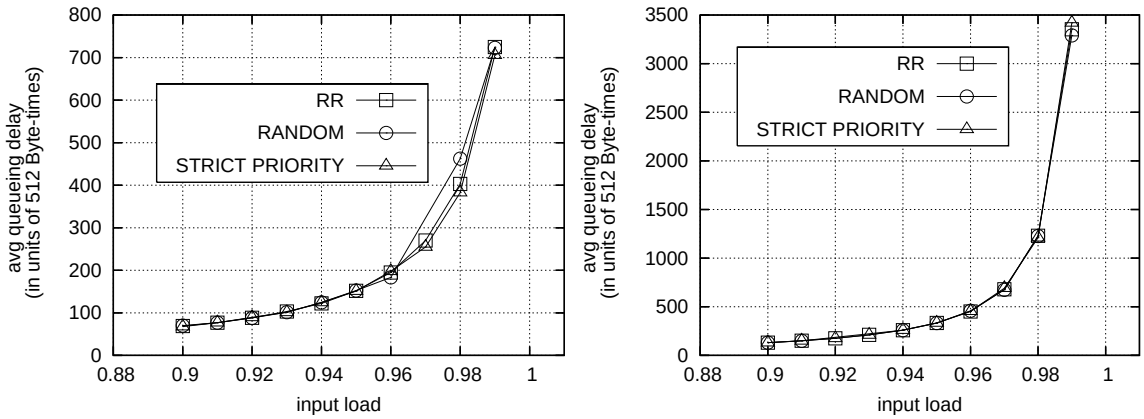


Figure 6.7: Performance of deterministic packet mode scheduling (DPM) with three different policies for granting locks: round-robin, strict priority and random. Traffic is uniformly destined. The reported results were obtained from a single run.

6.4.2 Effect of the lock scheduling policy on performance

By default, our *DPM* model assumes that locks are granted in a round-robin fashion to the requesting outputs. In this section, we describe our experimentation with two more lock scheduling policies: (a) the outputs have strict priorities and (b) the lock is granted to a randomly selected requesting output. Fig. 6.7 shows that for uniform traffic performance stays almost unaffected by the policy in which locks are granted. The reason is that, due to the masking operation, it is statistically rare that two or more outputs request the same lock at the same time. Actually, each requested lock is granted with great probability: for bimodal packet size and for loads greater than 0.95, the probability is greater than 98%; the respective one for uniform and constant packet size is 97% and 96% respectively.

6.4.3 Effect of the masking operation on performance

Deterministic packet mode scheduling introduces a lock acquisition phase: output schedulers cancel their decisions whenever they do not manage to have a lock granted and proceed in the rest of the round-robin schedule. On the other hand, when an output scheduler acquires a lock it starts the packet transmission and a masking operation is involved in order for the rest of the schedulers to know that the lock has been granted and thus do not attempt to acquire it at least until it is released; the relative input is marked as ineligible for these schedulers. The masking operation aims to improve performance by reducing the number of cancelled output scheduling decisions.

We experimented in order to find out how performance is improved with the masking

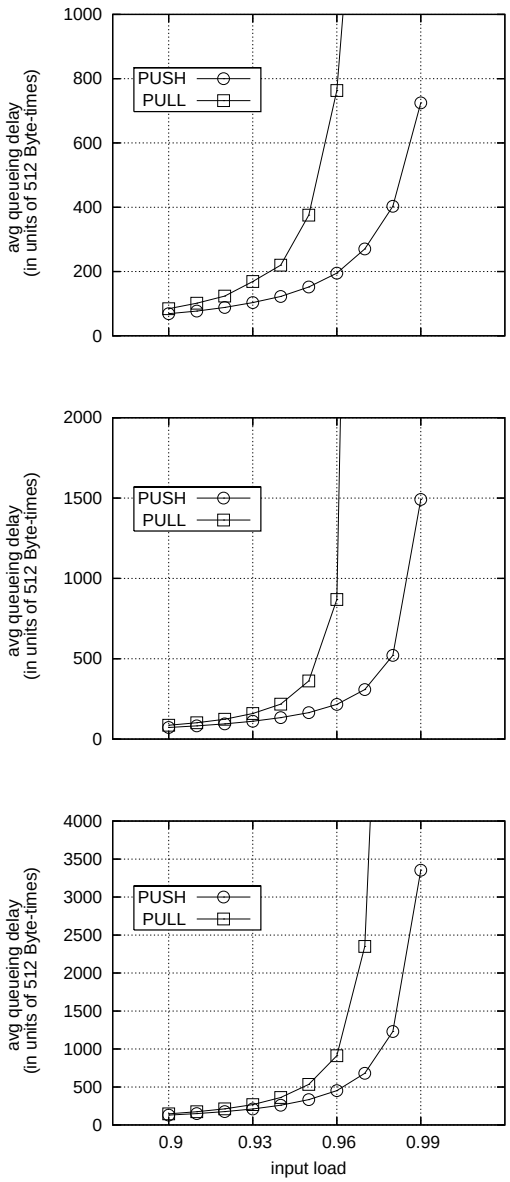


Figure 6.8: Performance of deterministic packet mode scheduling (DPM) with (PUSH) and without (PULL) the masking operation. Traffic is uniformly destined. The reported results were obtained from a single run.

operation. We modified the *DPM* model so that the masking operation is ommited and the output schedulers have to always request ("PULL") a lock even though the lock has been already granted to another output. *DPM* with the masking operation included is labeled "PUSH". Fig. 6.8 compares the *PULL* vs. *PUSH* version of *DPM* for *bimodal*, *uniform* and *const(8KB)* packet size and uniform destinations.

We observe that for loads up to 93% the masking operation is meaningless. However, without the masking operation the switch saturates earlier: under *bimodal* or *uniform* packet size the switch saturates at a load of 97% and under *const(8KB)* packet size it saturates at a load of 98%. By contrast, with the masking operation the switch is stable for all loads up

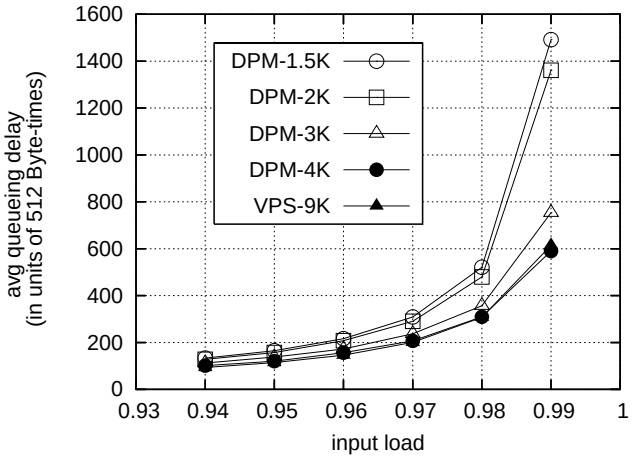


Figure 6.9: Buffered crossbar with blind SAR (SM model) and multipacket segments. The plot shows the percentage of delay in reassembly buffers over total queueing delay. Traffic is uniform and packet size is bimodal, uniform or constant(8KB).

to 99%.

6.4.4 Effect of the crosspoint buffer size on performance

Figure 6.9 shows the queueing delay in the DPM model as a function of input load and crosspoint buffer size. We experimented with 1.5, 2, 3 and 4KB crosspoint buffers and assuming uniform destinations and *uniform* packet size. The curve of the VPS model - which uses 9KB buffers - is also included in the plot. We observe that packet delay decreases as buffer size increases. With 4KB buffers the DPM matches the VPS curve.

6.5 Unbalanced traffic

We used the destination distribution model based on the Zipf law to study switch performance under unbalanced traffic. Results are shown in fig. 6.10 for three packet size distributions. All simulated models for the buffered crossbar offer almost the same throughput under unbalanced traffic; marginal differences exist though. These differences show that packet mode scheduling in buffered crossbars is superior to segment mode scheduling (marginally though) as is the case with the bufferless crossbar architecture. The worst-case throughput appears for $k \cong 2$ for all packet size distributions and it is always greater than 85%.

Note the the input queueing architecture with packet mode scheduling offers comparable throughput to the buffered crossbar. When operated on cell-mode the worst case throughput is poor.

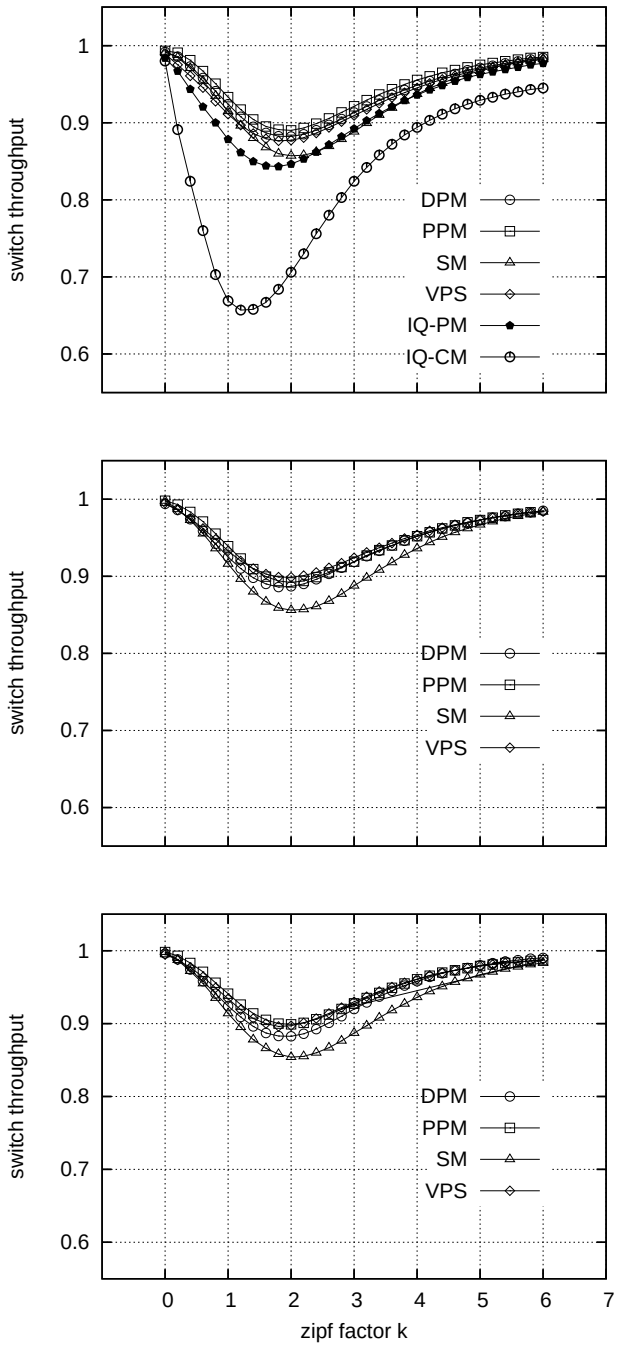


Figure 6.10: Switch throughput under unbalanced (Zipf) traffic.

Chapter 7

Conclusion and Future Work

We proposed and evaluated a *probabilistic* and a *deterministic* packet mode scheduling scheme for buffered crossbar switches. The deterministic scheme performs virtually as well as buffered crossbars that use no segmentation, and eliminates egress reassembly buffers like the latter systems do, while using crosspoint buffers whose size is only linked to the crossbar-ingress round-trip time –and not to the maximum packet size– hence can be much smaller than in buffered crossbars without segmentation. Performance is always better than bufferless crossbars with packet mode scheduling. Probabilistic packet mode scheduling performs even better than the deterministic scheme for some traffic patterns, while it performs similar for the rest; it allows independent output schedulers in the crossbar, but it does need the extra cost of egress reassembly buffers. Simulation results showed that for a traffic pattern with average packet size 550 Bytes the average packet delay (weighted with packet size) improves by more than 80% for loads up to 50%, compared to the system with blind segmentation and reassembly.

We also presented an innovative segmentation scheme for buffered crossbar switches. The scheme combines the known ability of buffered crossbars to directly operate on variable-size units and the aggregation of multiple packets or fragments of them into each segment. We showed that this scheme is well adapted to the use of DRAM for ingress buffering, and presented an overall system architecture.

Future work includes the following:

- study of implementation issues regarding the methods presented in chapter 4.
- Application of the deterministic packet mode scheduling, described in chapter 4, to

the hybrid SRAM/DRAM buffer organization proposed in [18].

- Study of possible solutions to the crossbar locking problem.
- Study of variations of the deterministic scheme in order for the crosspoint buffer size to become independent of the round-trip time.

Bibliography

[1] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick and M. Horowitz: "The Tiny Tera: A Packet Switch Core", Hot Interconnects V, Stanford University, August 1996. http://tiny-tera.stanford.edu/tiny-tera/

[2] Cisco 12000 Series Routers - Cisco 12008 and Cisco 12012 Routers. http://www.cisco.com/en/US/products/hw/routers/ps167

[3] 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. http://citeseer.ist.psu.edu/anderson93high.html

[4] N. McKeown: "iSLIP: A Scheduling Algorithm for Input-Queued Switches", IEEE/ACM Transactions on Networking, vol.7, no.2, pp.188201, April 1999. http://tiny-tera.stanford.edu/ nickm/papers/index.html

[5] D Stiliadis, A. Varma: "Providing Bandwidth Guarantees in an Input Buffered Crossbar Switch", IEEE Infocom95, pp. 960-968, April 1995.

[6] C. Minkenberg, R. Luijten, W. Denzel, and M. Gusat: "Current Issues in Packet Switch Design", HotNets-I, Princeton, NJ, 2002. http://citeseer.ist.psu.edu/minkenberg02current.html

[7] 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.

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

[9] K. Yoshigoe, K. Christensen: "A Parallel-Polled Virtual Output Queued Switch with a Buffered Crossbar", *Proc. IEEE Workshop High Perf. Switch-*

ing & Routing (HPSR 2001), Dallas, TX, USA, May 2001, pp. 271-275. http://www.csee.usf.edu/~christen/hpsr01.pdf

[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] 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.

[12] N. Chrysos, M. Katevenis: "Weighted Fairness in Buffered Crossbar Scheduling", *Proc. IEEE Workshop on High Performance Switching and Routing (HPSR 2003)*, Torino, Italy, June 2003, pp. 17-22.

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

[14] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches", *Proc. IEEE International Conference on Communications (ICC 2004)*, Paris, France, 20-24 June 2004, vol. 2, pp. 1090-1096.

[15] P. Dykstra: "Gigabit Ethernet Jumbo Frames". http://sd.wareonearth.com/ phil/jumbo.html

[16] S. Rixner, W. Dally, U. Kapasi, P. Matson, and D. Owens: Memory Access Scheduling, proc. ISCA-27, 2000.

[17] A. Nikologiannis, M. Katevenis: "Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques", Proc. IEEE Int. Conf. on Communications (ICC'2001), Helsinki, Finland, June 2001, pp. 2048-2052. http://archvlsi.ics.forth.gr/muqpro/queueMgt.html

[18] S. Iyer, R. Kompella, N. McKeown "Analysis of a Memory Architecture for Fast Packet Buffers", IEEE - High Performance Switching and Routing, Dallas, Texas, May 2001, pp. 368-373. http://tiny-tera.stanford.edu/ nickm/papers/index.html

[19] Sung-Ho Moon, Dan Keun Sung: "High-performance Variable-Length Packet Scheduling Algorithm for IP Traffic", *GLOBECOM 2001*, no. 1, Nov 2001 pp. 2666-2670

[20] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, F. Neri, "Packet Scheduling in Input-Queued Cell-Based Switches", *IEEE INFOCOM 2001*, no. 1, April 2001 pp. 1085-1094

[21] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, F. Neri: "Packet-Mode Scheduling in Input-Queued Cell-Based Switches", *IEEE/ACM Tr. on Networking*, vol. 10, no. 5, October 2002, pp. 666-678.

[22] S. Keshav: "An Engineering Approach to Computer Networking", Addison Wesley, 1997, ISBN 0-201-63442-2.

[23] R.R. Schaller: "Moore's Law: Past Present and Future", IEEE Spectrum, vol. 34, no. 6, June 1997, pp. 52-59.

[24] M. Karol, M. Hluchyj, S. Morgan: "Input versus Output Queueing on a Space-Division Packet Switch", IEEE Trans. on Communications, vol. 35, no. 12, Dec. 1987, pp. 1347-1356.

[25] M. Hluchyj, M. Karol: "Queueing in High-Performance Packet Switching", IEEE Journal on Sel. Areas in Commun. (JSAC), vol. 6, no. 9, Dec. 1988, pp. 1587-1597.

[26] Y. Tamir and G. Frazier: "High Performance Multiqueue Buffers for VLSI Communication Switches" International Symposium on Computer Architecture (ISCA), pages 343–354, May/June 1988.

[27] M. Katevenis, G. Passas: "Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures", Proc. IEEE International Conference on Communications (ICC 2005), Seoul, Korea, 16-20 May 2005, CR-ROM paper ID "09GC08-4", 6 pages.

[28] H. J. Chao and J.-S. Park: "Centralized Contention Resolution Schemes for a Large-Capacity Optical ATM Switch", IEEE ATM Workshop, Fairfax, VA, May 1998, pp. 11-16.

[29] S. Nojima, E. Tsutsui, H. Fukuda, and M. Hashimoto: "Integrated Services Packet Network using Bus-Matrix Switch", IEEE Journal on Selected Areas of Communications,vol.5,no.10, pp. 1284-1292, Oct. 1987.

[30] Quantum Flow Control (QFC) Alliance: "Quantum Flow Control: A Cell-Relay Protocol Supporting an Available Bit Rate Service", version 2.0, July 1995.

[31] M. Katevenis: "Fast Switching and Fair Control of Congested Flow in Broad-Band Networks", IEEE Journal on Selected Areas in Communications, vol. 5, no. 8, Oct. 1987, pp. 1315-1326.

[32] M. Katevenis: "Packet Switch Architecture", university course lectures. http://archvlsi.ics.forth.gr/ kateveni/534

[33] G. Passas: "Performance Evaluation of Variable Packet Size Buffered Crossbar Switches", Technical Report FORTH-ICS/TR-328, Inst. of Computer Science, FORTH, Heraklion, Crete, Greece; B.Sc. Thesis, Univ. of Crete; November 2003, 46 pages. http://archvlsi.ics.forth.gr/bufxbar

[34] L. Lamport: "Time, Clocks and the Ordering of Events in a Distributed System", Communications of the ACM, vol 21, no 7, 1978

[35] Micron DDR2 SDRAM product dcumentation, available at http://www.micron.com/products/dram/ddr2sdram/

[36] K. Kar, T. V. Lakshman, D. Stiliadis, and L. Tassiulas : "Reduced complexity Input Buffered Switches", *Proc. HOT Interconnects VIII*, Stanford University, Stanford, CA, August 2000.

[37] S. Mukherjee, F. Silla, P. Bannon, J. Emer, S. Lang, D. Webb: "A Comparative Study of Arbitration Algorithms for the Alpha 21364 Pipelined Router", *Proc. of the ACM ASPLOS-X Conf.*, San Jose, CA USA, Oct. 2002, pp. 223-234.

[38] X. Zhang, L. Bhuyan: "Deficit Round Robin Scheduling for Input-queued Switches" , *IEEE Journal of Selected Areas in Communications*, May 2003, pp. 584-594.

[39] Andrew S. Tanenbaum: "Operating Systems: Design & Implementation", 2nd Ed., 1997, Prentice-Hall, ISBN 0-13-630195-9.

[40] 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.

[41] Sheldon M. Ross: "Simulation", Academic Press, 3rd Edition, 2001, ISBN 0125980531

[42] Network Processing Forum (NPF): "Fabric Benchmarking Traffic Models", available from http://www.npforum.org/benchmarking/fabric_bm.shtml

Appendix A

Simulation Running Time

Our evaluation platform was a dual AMD Opteron 242 CPUs with 1 GByte of RAM running x86-based Linux.

Figure A.1 shows the simulated real time as a function of input load; the values correspond to a simulation run of the VPS system with bimodal packet size and uniform destinations. Since we terminate our simulations based on the number of packets that depart from the switch output ports, the simulated real time decreases as load increases. For the lightest of the loads the real time reaches 10 sec while for the heaviest ones it drops to around 200msec.

Figure A.2 compares event-driven to slotted-time simulation. We plot the ratio of running time to simulated real time as a function of input load for the (event-driven) simulator of the VPS and SM buffered crossbar and the (slotted-time) simulator of the bufferless crossbar with packet mode scheduling (IQ). We assume *bimodal*, *uniform* and *const(8KB)* packet size and uniform destinations. We observe that the performance of slotted time simulation is independent of input load and packet size distribution, while the performance of event-driven simulation scales with load and packet size. Event-driven simulation is much faster for loads smaller than 0.5 and when the average packet size is large enough. For the SM and VPS model the simulation is slower at heavy loads mainly because of the increased modelled complexity.

Figure A.3 compares the performance of the event-driven simulator for the models of buffered crossbar with packet segmentation. PPM is faster than SM which in turn is faster than DPM.

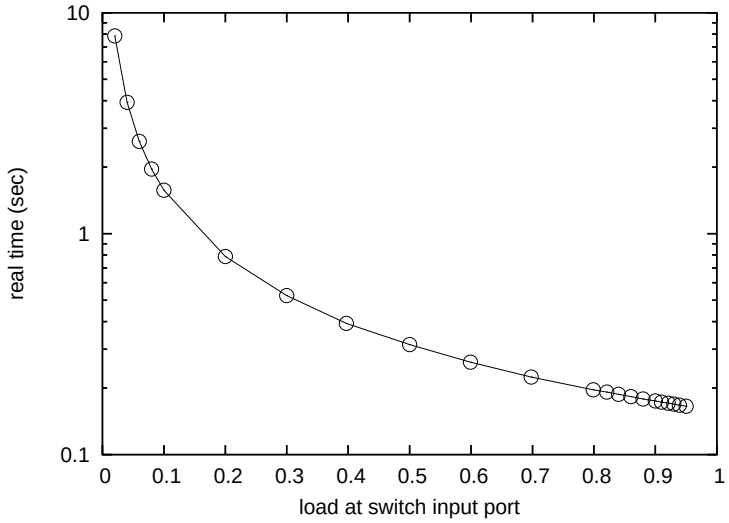


Figure A.1: Simulated real time as a function of input load. The reported values correspond to the VPS system under bimodal packet size and uniform destinations.

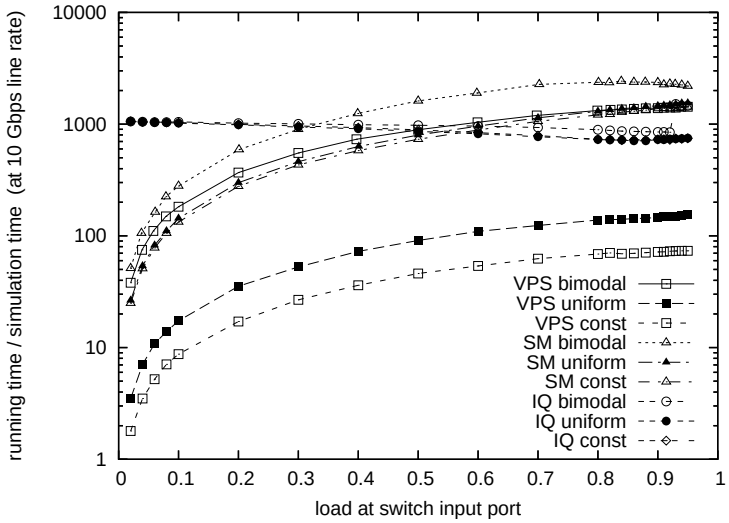


Figure A.2: Slotted-time vs. event-driven simulator performance. The reported values concern the VPS, SM and IQ system with packet mode scheduling. The packet size is bimodal, uniform or constant(8KB) and the traffic is uniform.

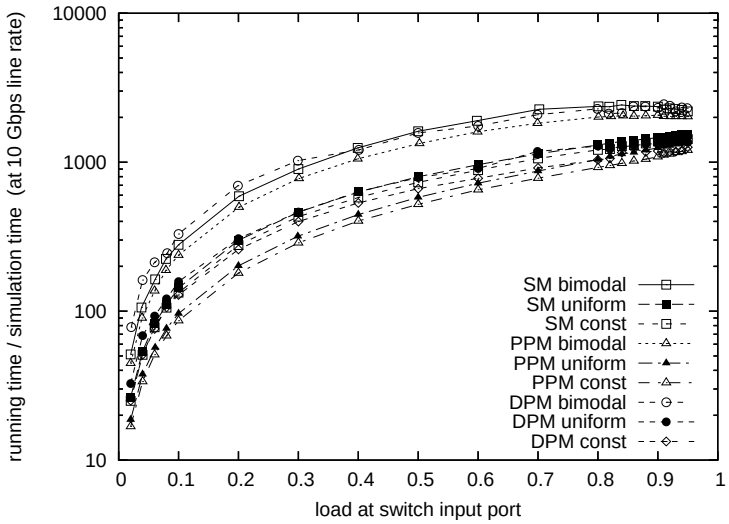


Figure A.3: Performance of event-driven simulation for the models of buffered crossbar with packet segmentation. The packet size is bimodal, uniform or constant(8KB) and the traffic is uniform.