Asynchronous Operation of Bufferless Crossbars

Georgios Passas[1] and Manolis Katevenis[1]
Foundation for Research and Technology - Hellas, Inst. of Computer Science - member of HiPEAC
FORTH-ICS, P.O. Box 1385, Vassilika Vouton, Heraklion, Crete, GR-711-10 Greece

***Abstract*— It is widely believed that bufferless crossbar switches with virtual-output queues (VOQ) at their inputs can only operate when their input-output connections are reconfigured in synchrony, i.e. only under fixed-size cell traffic. Packet-mode scheduling has been studied, but, again, assuming that all packets consist of an integer number of cells, where the scheduling time coincides with the cell time. We show that bufferless crossbars can operate directly on variable-size packets, with input-output connections being made and torn down asynchronously with respect to each other. Although such operation can initially be thought of as an extension of packet-mode scheduling, the critical difference is that now the scheduling time is much longer than packet-size granularity. We study a transformation of the well-known iSLIP scheduling algorithm to asynchronous mode of operation, and we show by simulation that it can be adapted to yield throughput close to 100% under a range of workloads. The overall result is an efficient scheduling operation, with the added advantages of eliminating (a) packet fragmentation overhead (no partially filled cells), and (b) packet reassembly in the egress datapath.**

I. Introduction

High-performance packet switches of moderate scale are popularly architected with a bufferless crossbar fabric, Virtual Output Queueing (VOQ), and fixed-size switching units, known as *cells* [1], [2], [3], [4]. External packets are segmented at the crossbar inputs into cells and queued in VOQs; the cells are switched through the crossbar to the outputs, where the original packets are reassembled and are afterwards transmitted on the line to the network.

The crossbar configuration is decided by a central scheduler running a parallel matching algorithm [1], [2]. During each cell time, each input requests the outputs corresponding to non-empty VOQs, each output grants a single request, and each input, in turn, accepts a single grant. When the three steps are completed, each output is matched to at most one input and vice-versa[2]. The crossbar is configured accordingly in the subsequent cell time.

PIM [3] and iSLIP [4] are two well-known paradigms of three-phase matching scheduling algorithms. PIM, iSLIP, and the majority of the switch scheduling algorithms known in the literature schedule cells ignoring which cell belongs to which packet; packet transmission on the output line cannot, hence, start at least until the output port has received the last cell of the packet. Given that scheduler decisions cannot be predicted, reassembly buffers are required at the crossbar outputs and *store-and-forward* operation is enforced.

In order to avoid the complexity of packet reassembly, *packet-mode* scheduling has been proposed in [5], [6]: the crossbar configuration changes again *synchronously* but an input-output connection is maintained until all cells of the associated packet are forwarded to the switch output. Since the switch output knows that all cells of a packet arrive consecutively in time, *(a)* it needs no reassembly buffer, and *(b)* it may start transmitting the packet right away, i.e. *cut-through* is allowed. The advantages are particularly important in systems requiring low latency and when traffic includes large packets (e.g. jumbo frames [7]).

Packet-mode scheduling assumes cell granularity for packet size. The cell size is specified by the time needed by the scheduler circuit to run the matching steps, with a typical size being 64 Bytes [8]. As a result, when a packet occupies a non-integral multiple of cells, its last cell is padded with useless bytes. In order to switch the excessive information, *internal speedup*[3] is needed, which is around two in the worst case - e.g. with 65-Byte packets in a 64-Byte cell switch. Internal speedup increases power consumption and limits port and line-rate scalability.

We study bufferless crossbar switches operating directly on external variable-size packets without prior segmentation to fixed-size cells. We show how the scheduling task can be carried out so that input-output connections are made and torn-down *asynchronously* with respect to each other. In the resulting system packet size granularity is independent of the scheduling time, and it can be infinitely small[4]. The proposed operation, hence, improves upon synchronous packet-mode operation by eliminating the need for internal speedup to accommodate cell-padding overheads.

Our study was motivated by our previous work on variable-size packet switching in *buffered crossbars* [9], [10], [11]. Crosspoint buffering allows for temporarily conflicting input-output matchings and, this, in turn, renders asynchronous operation straightforward. Buffered crossbars can perfectly well operate directly on variable-size packets, but could bufferless crossbars do so as well?

The remainder of the paper is organized as follows. In Section II, we describe the proposed switch operation. Section III examines the fairness properties of this operation;

[1] The authors are also with the University of Crete, Dept. of Computer Science, Heraklion, Crete, Greece.

[2] The scheduling algorithm solves a bipartite graph matching problem.

[3] The switch core operates faster than the external lines.

[4] Packet size granularity is actually quantized by the crossbar datapath width.

we indicate scenarios leading to starvation, and we propose a method for starvation-free operation. In Section IV, we describe observations from simulations, and in Section V we present numerical results. We show that the proposed scheme may be adapted to yield throughput close to 100% under a range of workloads. We also show that, factoring-out padding overheads, our scheme performs virtually as good as packet-mode scheduling. Finally, Section VI is a conclusion.

II. ASYNCHRONOUS OPERATION

In this section, we describe how to configure a bufferless crossbar asynchronously for variable-size packets. Specifically, like in the original iSLIP [4], per-input and per-output round-robin arbiters compute bipartite matchings between crossbar inputs and outputs in three steps – *request*, *grant*, and *accept*. The critical differentiation from iSLIP is that the arbiters make scheduling decisions without being synchronized with respect to each other, and new input-output matchings emerge asynchronously.

We first describe some notation and assumptions that will be used in the discussion to follow. If N is the size of the switch, for $0 \leq i, j < N$, VOQ_{ij} denotes the VOQ corresponding to output j at input i. Each input arbiter (or input for brevity's sake) submits requests to outputs corresponding to non-empty VOQs at this input; req_{ij} denotes a request signal from input i to output j. Each output arbiter (or output) arbitrates between received requests, grants a single request, and informs the corresponding input; $grnt_{ij}$ denotes a grant signal from output j to input i. Each input arbitrates between received grants, selects a single grant, and informs the corresponding output; $acpt_{ij}$ ($rjct_{ij}$) denotes an accept (reject) signal from input i to output j.

The assertion of a request, grant, or accept signal is considered a zero-time asynchronous event. An arbitration is triggered by a raised request or grant signal, and it is completed in time T. We consider the beginning of an arbitration period an event having the lowest precedence among concurrently occurring events. We call the interval $2T$ a scheduling (time) window because an output and an input arbitration are needed for a matching to be found. The scheduling window bounds from below the supported packet size; the maximum packet size on the other hand is unbounded.

We outline the switch operation describing an input and an output *finite state machine (FSM)*.

Input i FSM ($0 \leq j < N$)

Idle State. As soon as a VOQ_{ij} becomes non-empty, input i raises the request signal req_{ij}. Requests req_{ij} remain raised until at least one of them is granted; then input i lowers all req_{ij} and transitions to the *accept* state - see Fig. 1.

Accept State. In this state, input i arbitrates among raised grant signals $grnt_{ij}$ ($grnt_{11}$ and $grnt_{12}$ in Fig. 1). A single grant is accepted dependent on the position of an accept round-robin pointer, as in the original iSLIP; the pointer is updated similarly to iSLIP. If $grnt_{ij}$ is accepted, input i raises the accept signal $acpt_{ij}$; if $grnt_{ij}$ is rejected, input i raises the reject signal $rjct_{ij}$. Grant signals $grnt_{ij}$ received in the middle of the arbitration ($grnt_{13}$ in Fig. 1) are registered and they are rejected at the end of the arbitration. Input i transitions to the *transmit* state when the arbitration is completed.

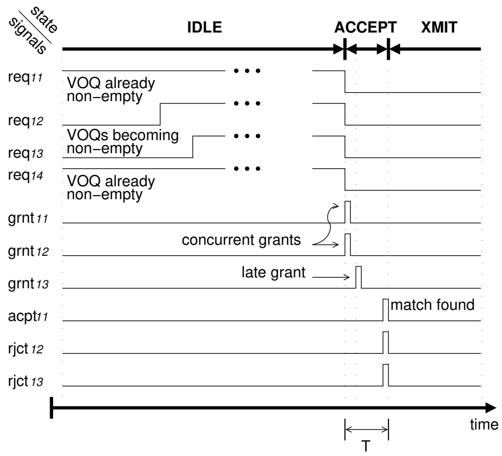

Fig. 1. Example of the input state transitions in asynchronous scheduling.

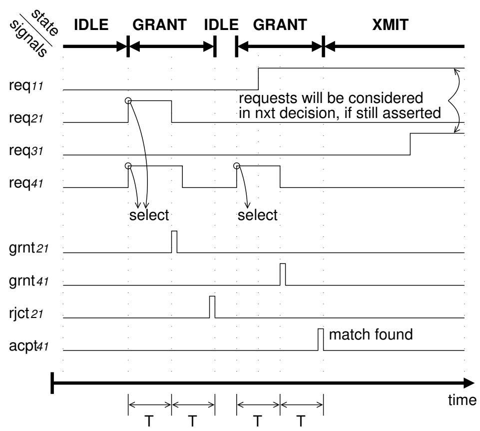

Fig. 2. Example of the output state transitions in asynchronous scheduling.

Transmit State. If $grnt_{ij}$ was accepted in the *accept* state, input i starts forwarding the head packet of VOQ_{ij}. Input i raises a request signal req_{ij} for each non-empty VOQ_{ij} and transitions to the *idle* state one scheduling window before the completion of the forwarding,

Output j FSM ($0 \leq i < N$)

Idle State. Output j remains in the current state until a request signal req_{ij} is raised; then it transitions to the *grant* state – see Fig. 2.

Grant State. In this state, output j arbitrates among raised requests req_{ij}. A single request is granted dependent on the position of a grant round-robin pointer, as in the original iSLIP; the pointer is updated similarly to iSLIP. If req_{ij} is granted, output j raises the grant signal $grnt_{ij}$. The output remains in the *grant* state until signal $acpt_{ij}$ or $rjct_{ij}$ is raised. It transitions to the *transmit* state in the former case and to the *idle* state in the latter

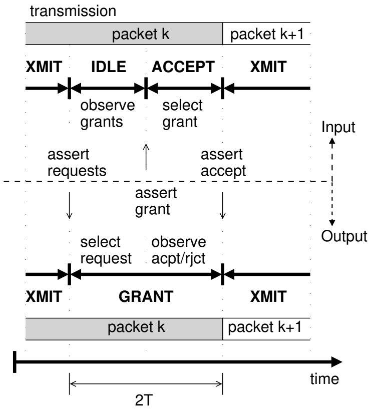

Fig. 3. Example of an input-output pair jointly passing through the FSM states in asynchronous scheduling.

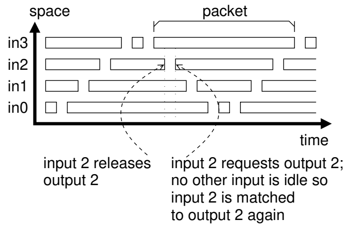

Fig. 4. Example of malicious traffic causing an asynchronous packet scheduler to lock the crossbar in a fixed configuration. Assume a 4×4 switch. Input i is initially connected to output i (for all i in [0,3]) and never switches output in the future. The figure shows the 4 flows that are constantly served; other flows with non-empty queues exist, but are never served.

one.

Transmit State. Output j waits in this state until one scheduling window remains for the completion of the current packet forwarding; then it transitions to the *idle* state.

Figure 3 depicts the state transitions of both an input and an output in a typical case of successful matching on first trial. Note that an input receives no grants in the *transmit* state because it aborts all its requests when it transitions to *accept* and it issues no request in the *transmit* state. With this design choice we save switch throughput because an input can accept no grants in the *transmit* state. Otherwise, two connections from the same input to two distinct outputs would have to be configured. Also notice that since outputs do not synchronize their arbitration, when some of them grant the same input, it is likely that some grants will be raised while the arbitration at the input has already been triggered. The input, hence, receives grants in the *accept* state although, being already arbitrating, it cannot accept them. Because of causality (the outcome of an arbitration cannot be known in the beginning of the arbitration) we cannot prevent the late grants from being raised. A similar problem appears when pipelining the decisions of traditional synchronous schedulers [12].

III. FAIRNESS

Assuming all inputs have queued packets for all outputs, an output sees requests of and arbitrates among only those inputs which are neither in the middle of an arbitration, nor in the middle of a packet forwarding. It may happen that each time an output starts an arbitration (transitions to the *grant* state), a fixed subset of the inputs have already started their arbitration (already in the *accept* state), or they are in the middle of a packet forwarding (*transmit* state). The output, then, sees no requests from these inputs, and flows from these inputs to this output starve. Figure 4 shows an example traffic pattern causing the above situation. Symmetrically, an input may never accept grants from a fixed subset of the outputs.

A synchronous packet-mode scheduler similarly suffers from starvation because, during each cell-time, it considers candidacies only for those inputs and outputs which are not in the middle of a packet-mode forwarding [5], [6]. The traffic pattern shown in Fig. 4 would similarly lead a packet-mode scheduler to lock the crossbar in a fixed configuration – consider time is now slotted and packets are integral multiples of cells. Comparing our scheme to packet-mode scheduling, our scheme imposes a single additional constraint: when an output (input) starts an arbitration, it further excludes candidacies for those inputs (outputs) which are in the middle of an arbitration at that time. This constraint is meaningless in packet-mode scheduling because inputs and outputs arbitrate in syncrhony.

Starvation-free operation can be guaranteed in a brute-force way. Enforcing an output to remain idle until active flows not served for a long time are served, we guarantee no flow is left indefinitely unserved. If the inter-service time interval indicating starvation is long enough, we expect the throughput loss from the idling output to be negligible. Similar techniques have been investigated in [13]. Due to space constraints, in this paper we restrict our study to the throughput asynchronous scheduling may offer and to a comparison between asynchronous scheduling and synchronous packet-mode scheduling. Issues related to starvation will be studied in more depth in future work[5].

IV. OBSERVATIONS FROM SIMULATION

This section describes observations from simulation. We present the simulation environment and the numerical results in the next Section V. In the discussion to follow, we assume uniform traffic and we distinguish between two cases: (i) average packet size is many scheduling windows, and (ii) average packet size is few scheduling windows.

In both of the above cases, under light load outputs are idle most of the time, and inputs are connected to outputs almost as soon as they request them. Under heavy load, however, contention becomes heavy, and the operation of the switch depends on the packet size distribution of the workload.

When average packet size is large, connections emerge incrementally until all inputs are eventually connected with

[5]We mention that, as was shown in [11], buffered crossbars do not suffer from this starvation situation because input-output matchings are not necessarily bipartite. More clever solutions to the starvation problem could be based on this observation.

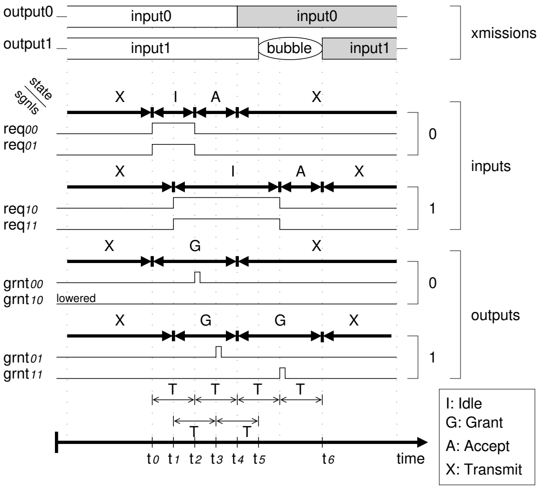

Fig. 5. Example of conflicting output decisions in asynchronous scheduling when a pair of connections are torn down close in time with respect to each other. We show a pair of connections; we assume the rest of the inputs and outputs remain connected in the shown time interval. The connection from input 0 to output 0 terminates at time t_4; the connection from input 1 to output 1 terminates at time t_5, with $0 < t_5 - t_4 < T$. Inputs 0 and 1 raise requests to both outputs at time $t_0 = t_4 - 2T$ and $t_1 = t_5 - 2T > t_0$ respectively; we assume packets for both outputs are queued at the inputs, which is realistic under heavy load and uniform traffic. Output 0 starts arbitrating at time t_0 seeing only the request from input 0, and grants again input 0 at time $t_2 = t_0 + T$. Output 1, on the other hand, starts arbitrating at time t_1 seeing the requests from both inputs 0 and 1. Output 1 had previously granted input 1, thus, it grants input 0 at time $t_3 = t_1 + T > t_2$. Input 0 accepts output 0, since the grant from output 0 was asserted first, and rejects output 1 at time $t_4 = t_2 + T$. A new connection from input 0 to output 0 is configured at that time. Output 1 remains idle until time $t_6 = t_4 + 2T$. When packet size is large, this problem affects performance only under the heaviest of the loads, because the time an output remains idle is negligible compared to the packet time. Note that in synchronous operation, packets would be padded, and connections would terminate concurrently; outputs 0 and 1 would start their arbitration concurrently, they would both see requests from both inputs, and they would switch inputs with no throughput loss. Of course, there is throughput loss due to padding bytes; in our evaluation (Section V)we factor this loss out.

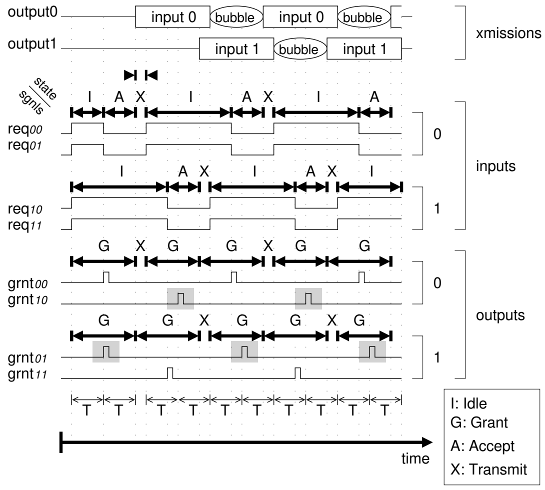

Fig. 6. Example of grant pointers moving in lock step in asynchronous scheduling. Assume a 2×2 switch, packet size is $1.025T \times R$, and all inputs request all outputs. If the grant pointers of both outputs point to input 0 at some point in time, they keep moving in lock-step and throughput is lost. $grnt_{01}$ is always raised later than $grnt_{00}$, it is, hence, always rejected, and input 0 cannot be connected to output 1 (the rejected grants are shaded); $grnt_{10}$ is always raised later than $grnt_{11}$, it is always rejected, and input 1 cannot be connected to output 0. The grant pointers keep moving in lock step until VOQ_{00} or VOQ_{11} drains. Notice that due to this behavior the VOQs grow up non-uniformly, and the grant pointers are prevented from desynchronizing as in the original iSLIP. Also notice that if packets had the minimum size, the output arbiters would always start their arbitration concurrently, and their grant pointers would desynchronize as in the original iSLIP.

an output. In the general case, two or more packet transmissions rarely finish concurrently. Instead, (a) within one scheduling window before, or after, the completion of a packet forwarding, no other forwarding is completed[6], or, (b) few other transmissions are completed. In case (a), if there are packets queued in the corresponding VOQ, the same input-output connection is re-configured as was shown in Fig. 4 and Fig. 3. In case (b), the released outputs attempt to switch inputs, but they fail, throughput is lost due to conflicting output decisions, and the same connections are re-configured with high probability. We explain how in Fig. 5. Under heavy load, it is, thus, likely that asynchronous scheduling reconfigures the same connection for as long as there are packets queued in the corresponding VOQ. Asynchronous scheduling degenerates to an *exhaustive service* discipline [13]. When the VOQ drains, the corresponding input and output remain idle for a short period of time, and switch to the output and input that finish their packet forwarding first. In the next Section V, we show that the throughput loss occurring when the outputs fail to switch inputs, results to an increase in packet delay at loads greater than 97%. We also show that, as a result of the degeneration described above, the reconfiguration frequency of the crossbar is significantly reduced when packet size is large.

When average packet size is comparable to the scheduling time window, on the other hand, the grant pointers tend to move in lock-step, and performance degrades. We explain how in Fig. 6, which extends the scenario of Fig. 5. In the next section, we show that when average packet size is close to the minimum, the saturation throughput of asynchronous scheduling and of plain synchronous *round-robin matching – RRM* [4] are almost the same.

V. PERFORMANCE EVALUATION

We used event-driven simulation to model our scheme, which we compare to packet-mode scheduling. We modeled the packet-mode scheduler described in [5] using slotted-time simulation.

In our workload, packet size granularity was 0.025 scheduling windows (2T) for asynchronous scheduling[7]; packets were integral multiples of cells for synchronous scheduling. We factored-out the padding overheads incurred by synchronous operation in order to compare pure scheduling efficiency; the latter assumption clearly favors the synchronous system.

[6]This is also the case for packet-mode scheduling.

[7]Minimum IP packet size is 40 bytes. Then, with 2T=40 byte-times, we have 1-byte granularity.

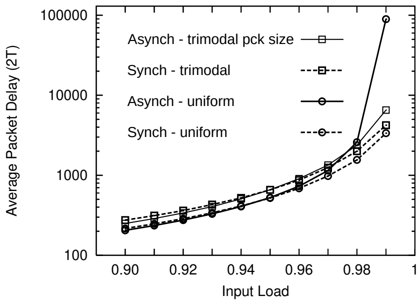

Fig. 7. Packet delay of asynchronous and synchronous scheduling under uniform traffic and trimodal, or uniform packet size.

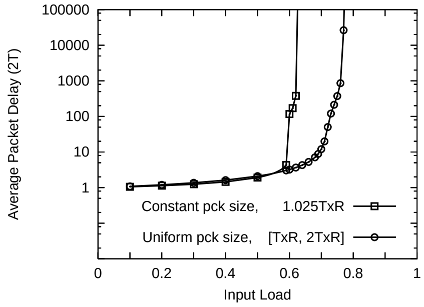

Fig. 8. Saturation throughput of asynchronous scheduling under uniform traffic and pathological packet size conditions.

We run all simulations until the simulated time became greater than at least $5 \times 10^6 \times N$ scheduling windows, where N is the switch size. We assumed one scheduling time window coincides with one cell time. We considered the switch saturated when the total backlog at any input port became greater than $25 \times 10^3 \times N$ scheduling windows. Fabric size was always 16×16.

First, we studied performance under uniform traffic in terms of average packet delay. We defined average packet delay as the time interval between the moment the first byte of a packet starts entering the switch to the moment its first byte starts departing averaged over the number of packets. We report packet delay in number of scheduling windows. When the switch is saturated, packet delay is considered infinite.

A. Uniform traffic

1) Minimum-size packets: We considered Poisson arrivals of minimum-size packets for asynchronous scheduling and i.i.d Bernoulli arrivals of cells for synchronous scheduling. We found that asynchronous scheduling is marginally more efficient under loads less than 0.5, while at heavier loads the delay curves of the two systems match. (see the *SLIP* curve in Fig. 5 of reference [4].)

2) Variable-size packets: We repeated the previous experiment assuming that external packets have variable size. For asynchronous scheduling we considered Poisson packet arrivals, while for synchronous scheduling we assumed packet arrivals are decided by an ON-OFF model, as in [6]. We experimented with trimodal and uniform packet size. In the first case, 60% of the packets had size $2T \times R$ (1 cell for synchronous scheduling), 20% $28.8T \times R$ (15 cells) and 20% $75T \times R$ (38 cells); this packet size distribution resembles the respective one in real-world IP networks [14]. In the latter case, packet size was uniformly distributed between $2T \times R$ and $75T \times R$. Figure 7 shows the simulation results. We observe that for loads up to 97% the delay curves of the two systems almost match. For greater loads, however, packet delay increases significantly in our system. The reason was explained in Fig. 5.

3) Corner packet-size conditions: Fig. 8 plots the saturation throughput of asynchronous scheduling when packet size is constant $1.025T \times R$, or uniform between $T \times R$ and $2T \times R$. We observe that the saturation throughput now drops to 63% of the link capacity in the first case, and to 78% in the second case. The reason for the performance degradation is the failure of the grant pointers to desynchronize, as we explained in Fig. 6.

4) Virtually large packets: We can improve performance by virtually increasing the size of the external packets. Switching multiple small packets each time when an input-output connection has been configured[8] we get saturation throughput close to 100%, as in Section V-A.2 – average packet size was well above 10 scheduling windows in the experiment of Section V-A.2. The method we propose is similar to the method proposed in [15], [10]: when the occupancy of a VOQ exceeds a threshold value $Q \times (2T \times R)$, Q is an integer greater than zero, the packets at the head of the queue whose cumulative size exceeds the threshold value appear in the scheduler circuit as a single external packet with size equal to the cumulative size of these packets; if the occupancy of a VOQ is below the threshold value, the boundaries of the internal packets coincide with the boundaries of the external packets. Simulation results for various values of Q are shown in Fig. 9; packet size is $1.025T \times R$. Note that for $Q = 1$ we get the baseline system. Also observe that for values of Q greater or equal to 8, we get switch throughput near 100%. Delay increases with Q because as Q increases, a greater backlog is required for the switch to start operating efficiently.

5) Crossbar reconfiguration probability: We measured the crossbar reconfiguration probability as the probability an input switches output in its next scheduling decision. Figure 10 shows the crossbar reconfiguration probability for asynchronous and synchronous scheduling under minimum or trimodal packet size. When load is low, VOQs are empty, and the probability equals the probability a packet arriving at an input is destined to an output different than the output the previous packet at this input was destined to. As load increases, VOQs become backlogged, and the probability depends on the packet-size distribution. When packet size is minimum, both asynchronous and synchronous scheduling reduce to the original iSLIP. Under heavy load, iSLIP degenerates to time

[8]One could argue that then the padding overheads are negligible, and, hence, our scheme loses its advantage over synchronous scheduling. This is correct only when traffic is uniform. Consider the scenario described in [15] (Section II), where many light flows are never backlogged, and steal bandwidth through padding bytes from a heavy flow.

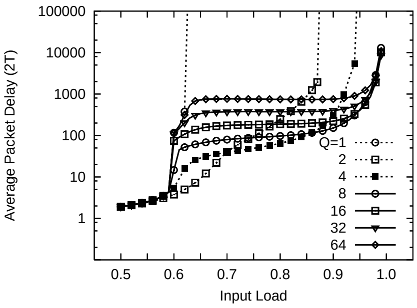

Fig. 9. Saturation throughput of asynchronous scheduling as a function of the threshold value. Traffic is uniform and packet size is $1.025T \times R$.

division multiplexing, and, hence, the crossbar reconfiguration probability equals 1. When packet size is trimodal, the reconfiguration probability is significantly reduced and appears almost the same for both asynchronous and synchronous scheduling. This is reasonable since for both schemes the same connections are maintained for multiple scheduling windows as explained in sections III, IV. The probability is further reduced applying the method described in Section V-A.4.

B. Unbalanced Traffic

We modelled unbalanced traffic based on the Zipf's law, as suggested in [16]. Figure 11 plots the attainable switch throughput as a function of the Zipf order k. Asynchronous scheduling reaches the same throughput with synchronous scheduling. Throughput increases with the threshold value Q because the crossbar reconfiguration frequency is further reduced as Q increases, and the scheduling task is further simplified.

VI. Conclusion

We proposed and evaluated bufferless crossbar switches operating directly on variable-size packets under the arbitration of an asynchronous crossbar scheduler. The operation eliminates packet reassembly and allows for cut-through, like the synchronous packet-mode operation does, while at the same time, as opposed to the packet-mode operation, it relieves from cell-padding overheads and the associated crossbar speedup requirements. The proposed operation may yield throughput close to 100% under a range of traffic patterns, and has common performance characteristics with the packet-mode operation.

Acknowledgement

This work was supported by the European Commission in the context of the SARC (Scalable Computer Architecture) integrated project #27648 (FP6), and the HiPEAC network of excellence.

References

[1] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick, and M. Horowitz, "The Tiny Tera: A packet switch core," in *Hot Interconnects V*, August 1996.

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

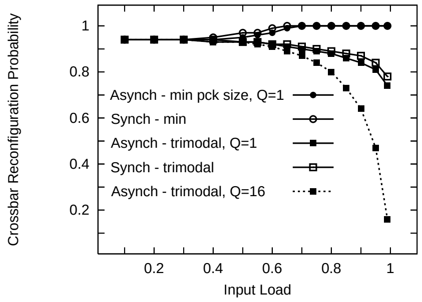

Fig. 10. Crossbar reconfiguration frequency with asynchronous and synchronous scheduling under uniform traffic and minimum, or trimodal packet size.

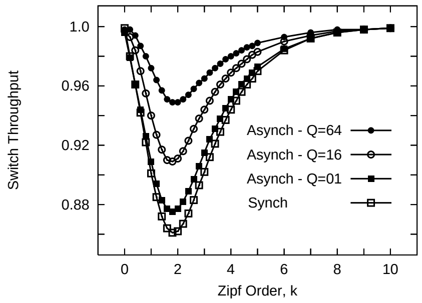

Fig. 11. Switch throughput of asynchronous scheduling under unbalanced (Zipf) traffic. Packet size is trimodal.

[3] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High-speed switch scheduling for local-area networks," *ACM Trans. on Computer Systems*, vol. 11, no. 4, pp. 319–352, Nov. 1993.

[4] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[5] S. H. Moon and D. K. Sung, "High-performance variable-length packet scheduling algorithm for IP traffic," in *IEEE GLOBECOM*, November 2001.

[6] A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, "Packet-mode scheduling in input-queued cell-based switches," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, April 2002.

[7] J. Hurwitz and W. Feng, "Initial end-to-end performance evaluation of 10-gigabit ethernet," in *IEEE Hot Interconnects XI*, August 2003.

[8] S. Iyer, R. Kompella, and N. McKeown, "Analysis of a memory architecture for fast packet buffers," in *Proceedings of IEEE HPSR*, May 2001.

[9] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos, "Variable packet size buffered crossbar (CICQ) switches," in *IEEE ICC*, June 2004.

[10] M. Katevenis and G. Passas, "Variable-size multipacket segments in buffered crossbar (CICQ) architectures," in *IEEE ICC*, May 2005.

[11] G. Passas and M. Katevenis, "Packet-mode scheduling in buffered crossbar (CICQ) switches," in *IEEE HPSR*, June 2006.

[12] P. Gupta and N. McKeown, "Design and implementation of a fast crossbar scheduler," *IEEE Micro*, vol. 19, no. 1, pp. 20–28, Jan.-Feb. 1999.

[13] Y. Li, S. Panwar, and H. J. Chao, "Performance analysis of a dual round robin matching switch with exhaustive service," in *IEEE Globecom*, November 2002.

[14] *Cooperative Association for Internet Data Analysis*, http://www.caida.org.

[15] K. Kar, T. Lakshman, D. Stiliadis, and L.Tassiulas, "Reduced complexity input buffered switches," in *Hot Interconnects VIII*, 2000.

[16] I. Elhanany, D. Chiou, V. Tabatabaee, R. Noro, and A. Poursepanj, "The network processing forum switch fabric benchmark specifications: An overview," *IEEE Network*, vol. 19, no. 2, 2005.

